Pine Script v6 User Manual

Print edition, mirrored from TradingView on 2026-10-29.

78 pages of text and code. The 827 figures are not in this file; the HTML edition has all of them. Source: https://www.tradingview.com/pine-script-docs/

Content © TradingView, Inc. Not affiliated with or endorsed by TradingView.

Contents

1. Alerts (concepts/alerts)
2. Bar states (concepts/bar-states)
3. Chart information (concepts/chart-information)
4. Inputs (concepts/inputs)
5. Libraries (concepts/libraries)
6. Non-standard charts data (concepts/non-standard-charts-data)
7. Other timeframes and data (concepts/other-timeframes-and-data)
8. Repainting (concepts/repainting)
9. Sessions (concepts/sessions)
10. Strategies (concepts/strategies)
11. Strings (concepts/strings)
12. Time (concepts/time)
13. Timeframes (concepts/timeframes)
14. Errors and warnings / CE10101 (errors/CE10101)
15. Errors and warnings / CE10117 (errors/CE10117)
16. Errors and warnings / CW10003 (errors/CW10003)
17. Errors and warnings / CW10026 (errors/CW10026)
18. Errors and warnings / RE10139 (errors/RE10139)
19. Errors and warnings / RE10143 (errors/RE10143)
20. Overview (errors/overview)
21. Alerts (faq/alerts)
22. Data structures (faq/data-structures)
23. Functions (faq/functions)
24. FAQ (faq/general)
25. Indicators (faq/indicators)
26. Other data and timeframes (faq/other-data-and-timeframes)
27. Programming (faq/programming)
28. Strategies (faq/strategies)
29. Strings and formatting (faq/strings-and-formatting)
30. Techniques (faq/techniques)
31. Times, dates, and sessions (faq/times-dates-and-sessions)
32. Variables and operators (faq/variables-and-operators)
33. Visuals (faq/visuals)

34. Arrays (language/arrays)
35. Built-ins (language/built-ins)
36. Conditional structures (language/conditional-structures)
37. Declaration statements (language/declaration-statements)
38. Enums (language/enums)
39. Execution model (language/execution-model)
40. Identifiers (language/identifiers)
41. Loops (language/loops)
42. Maps (language/maps)
43. Matrices (language/matrices)
44. Methods (language/methods)
45. Objects (language/objects)
46. Operators (language/operators)
47. Script structure (language/script-structure)
48. Type system (language/type-system)
49. User-defined functions (language/user-defined-functions)
50. Variable declarations (language/variable-declarations)
51. Overview (migration-guides/overview)
52. To Pine Script® version 2 (migration-guides/to-pine-version-2)
53. To Pine Script® version 3 (migration-guides/to-pine-version-3)
54. To Pine Script® version 4 (migration-guides/to-pine-version-4)
55. To Pine Script® version 5 (migration-guides/to-pine-version-5)
56. To Pine Script® version 6 (migration-guides/to-pine-version-6)
57. First indicator (primer/first-indicator)
58. First steps (primer/first-steps)
59. Next steps (primer/next-steps)
60. Release notes (release-notes)
61. Backgrounds (visuals/backgrounds)
62. Bar coloring (visuals/bar-coloring)
63. Bar plotting (visuals/bar-plotting)
64. Colors (visuals/colors)
65. Fills (visuals/fills)
66. Levels (visuals/levels)
67. Lines and boxes (visuals/lines-and-boxes)
68. Overview (visuals/overview)
69. Plots (visuals/plots)
70. Tables (visuals/tables)
71. Text and shapes (visuals/text-and-shapes)
72. Welcome to Pine Script® v6 (welcome)
73. Where can I get more information? (where-can-i-get-more-information)
74. Debugging (writing/debugging)
75. Limitations (writing/limitations)
76. Profiling and optimization (writing/profiling-and-optimization)
77. Publishing scripts (writing/publishing)

78. Style guide (writing/style-guide)

Alerts

Introduction

TradingView alerts run 24x7 on our servers and do not require users to be logged in to execute. Alerts are created from the charts user interface (*UI*). You will find all the information necessary to understand how alerts work and how to create them from the charts UI in the Help Center's About TradingView alerts page.

Some of the alert types available on TradingView (*generic alerts*, *drawing alerts* and *script alerts* on order fill events) are created from symbols or scripts loaded on the chart and do not require specific coding. Any user can create these types of alerts from the charts UI.

Other types of alerts (*script alerts* triggering on *alert() function calls*, and *alertcondition() alerts*) require specific Pine Script® code to be present in a script to create an *alert event* before script users can create alerts from them using the charts UI. Additionally, while script users can create *script alerts* triggering on *order fill events* from the charts UI on any strategy loaded on their chart, Programmers can specify explicit order fill alert messages in their script for each type of order filled by the broker emulator.

This page covers the different ways Pine Script programmers can code their scripts to create alert events from which script users will in turn be able to create alerts from the charts UI. We will cover:

- How to use the alert() function to *alert() function calls* in indicators or strategies, which can then be included in *script alerts* created from the charts UI.
- How to add custom alert messages to be included in *script alerts* triggering on the *order fill events* of strategies.
- How to use the alertcondition() function to generate, in indicators only, *alertcondition() events* which can then be used to create *alertcondition() alerts* from the charts UI.

Keep in mind that:

- No alert-related Pine Script code can create a running alert in the charts UI; it merely creates alert events which can then be used by script users to create running alerts from the charts UI.
- Alerts only trigger in the realtime bar. The operational scope of Pine Script code dealing with any type of alert is therefore restricted to realtime bars only.
- When an alert is created in the charts UI, TradingView saves a mirror image of the script and its inputs, along with the chart's main symbol and timeframe to run the alert on its servers. Subsequent changes to your script's inputs or the chart will thus not affect running alerts previously created from them. If you want any changes to your context to be reflected in a running alert's behavior, you will need to delete the alert and create a new one in the new context.

Background

The different methods Pine programmers can use today to create alert events in their script are the result of successive enhancements deployed throughout Pine Script's evolution. The alertcondition() function, which works in indicators only, was the first feature allowing Pine Script programmers to create alert events. Then came order fill alerts for strategies, which trigger when the broker emulator creates *order fill events*. *Order fill events* require no special code for script users to create alerts on them, but by way of the `alert_message` parameter for order-generating `strategy.*()` functions, programmers can customize the message of alerts triggering on *order fill events* by defining a distinct alert message for any number of order fulfillment events.

The alert() function is the most recent addition to Pine Script. It more or less supersedes alertcondition(), and when used in strategies, provides a useful complement to alerts on *order fill events*.

Which type of alert is best?

For Pine Script programmers, the alert() function will generally be easier and more flexible to work with. Contrary to alertcondition(), it allows for dynamic alert messages, works in both indicators and strategies and the programmer decides on the frequency of alert() events.

While alert() calls can be generated on any logic programmable in Pine, including when orders are **sent** to the broker emulator in strategies, they cannot be coded to trigger when orders are **executed** (or *filled*) because after orders are sent to the broker emulator, the emulator controls their execution and does not report fill events back to the script directly.

When a script user wants to generate an alert on a strategy's order fill events, he must include those events when creating a *script alert* on the strategy in the “Create Alert” dialog box. No special code is required in scripts for users to be able to do this. The message sent with order fill events can, however, be customized by programmers through use of the `alert_message` parameter in order-generating `strategy.*()` function calls. A combination of alert() calls and the use of custom `alert_message` arguments in order-generating `strategy.*()` calls should allow programmers to generate alert events on most conditions occurring in their script's execution.

The alertcondition() function remains in Pine Script for backward compatibility, but it can also be used advantageously to generate distinct alerts available for selection as individual items in the “Create Alert” dialog box's “Condition” field.

Script alerts

When a script user creates a *script alert* using the “Create Alert” dialog box, the events able to trigger the alert will vary depending on whether the alert is created from an indicator or a strategy.

A *script alert* created from an **indicator** will trigger when:

- The indicator contains alert() calls.
- The code's logic allows a specific alert() call to execute.
- The frequency specified in the alert() call allows the alert to trigger.

A *script alert* created from a **strategy** can trigger on *alert() function calls*, on *order fill events*, or both. The script user creating an alert on a strategy decides which type of events he wishes to include in his *script alert*. While users can create a *script alert* on *order fill events* without the need for a strategy to include special code, it must contain alert() calls for users to include *alert() function calls* in their *script alert*.

`alert()` function events

The alert() function has the following signature:

```
alert(message, freq)
```

`message`

A “series string” representing the message text sent when the alert triggers. Because this argument allows “series” values, it can be generated at runtime and differ bar to bar, making it dynamic.

`freq`

An “input string” specifying the triggering frequency of the alert. Valid arguments are:

- `alert.freq_once_per_bar`: Only the first call per realtime bar triggers the alert (default value).
- `alert.freq_once_per_bar_close`: An alert is only triggered when the realtime bar closes and an alert() call is executed during that script iteration.
- `alert.freq_all`: All calls during the realtime bar trigger the alert.

The alert() function can be used in both indicators and strategies. For an alert() call to trigger a *script alert* configured on *alert() function calls*, the script's logic must allow the alert() call to execute, **and** the frequency determined by the `freq` parameter must allow the alert to trigger.

Note that by default, strategies are recalculated at the bar's close, so if the alert() function with the frequency `alert.freq_all` or `alert.freq_once_per_bar` is used in a strategy, then it will be called no more often than once at the bar's close. In order to enable the alert() function to be called during the bar construction process, you need to enable the `calc_on_every_tick` option.

Using all `alert()` calls

Let's look at an example where we detect crosses of the RSI centerline:

//@version=6 indicator("All `alert()` calls") r = ta.rsi(close, 20) // Detect crosses. xUp = ta.crossover(r, 50) xDn = ta.crossunder(r, 50) // Trigger an alert on crosses. if xUp alert("Go long (RSI is " + str.tostring(r, "#.00")) else if xDn alert("Go short (RSI is " + str.tostring(r, "#.00")) plotchar(xUp, "Go Long", "▲", location.bottom, color.lime, size = size.tiny) plotchar(xDn, "Go Short", "▼", location.top, color.red, size = size.tiny) hline(50) plot(r)

If a *script alert* is created from this script:

- When RSI crosses the centerline up, the *script alert* will trigger with the “Go long...” message. When RSI crosses the centerline down, the *script alert* will trigger with the “Go short...” message.
- Because no argument is specified for the `freq` parameter in the alert() call, the default value of `alert.freq_once_per_bar` will be used, so the alert will only trigger the first time each of the alert() calls is executed during the realtime bar.
- The message sent with the alert is composed of two parts: a constant string and then the result of the str.tostring() call which will include the value of RSI at the moment where the alert() call is executed by the script. An alert message for a cross up would look like: “Go long (RSI is 53.41)”.
- Because a *script alert* always triggers on any occurrence of a call to alert(), as long as the frequency used in the call allows for it, this particular script does not allow a script user to restrict his *script alert* to longs only, for example.

Note that:

- Contrary to an alertcondition() call which is always placed at column 0 (in the script's global scope), the alert() call is placed in the local scope of an if branch so it only executes when our triggering condition is met. If an alert() call was placed in the script's global scope at column 0, it would execute on all bars, which would likely not be the desired behavior.

- An alertcondition() could not accept the same string we use for our alert's message because of its use of the str.tostring() call. alertcondition() messages must be constant strings.

Lastly, because alert() messages can be constructed dynamically at runtime, we could have used the following code to generate our alert events:

// Trigger an alert on crosses. if xUp or xDn firstPart = (xUp ? "Go long" : "Go short") + " (RSI is " alert(firstPart + str.tostring(r, "#.00"))

Using selective `alert()` calls

When users create a *script alert* on *alert() function calls*, the alert will trigger on any call the script makes to the alert() function, provided its frequency constraints are met. If you want to allow your script's users to select which alert() function call in your script will trigger a *script alert*, you will need to provide them with the means to indicate their preference in your script's inputs, and code the appropriate logic in your script. This way, script users will be able to create multiple *script alerts* from a single script, each behaving differently as per the choices made in the script's inputs prior to creating the alert in the charts UI.

Suppose, for our next example, that we want to provide the option of triggering alerts on only longs, only shorts, or both. You could code your script like this:

//@version=6 indicator("Selective `alert()` calls") detectLongsInput = input.bool(true, "Detect Longs") detectShortsInput = input.bool(true, "Detect Shorts") repaintInput = input.bool(false, "Allow Repainting") r = ta.rsi(close, 20) // Detect crosses. xUp = ta.crossover(r, 50) xDn = ta.crossunder(r, 50) // Only generate entries when the trade's direction is allowed in inputs. enterLong = detectLongsInput and xUp and (repaintInput or barstate.isconfirmed) enterShort = detectShortsInput and xDn and (repaintInput or barstate.isconfirmed) // Trigger the alerts only when the compound condition is met. if enterLong alert("Go long (RSI is " + str.tostring(r, "#.00")) else if enterShort alert("Go short (RSI is " + str.tostring(r, "#.00")) plotchar(enterLong, "Go Long", "▲", location.bottom, color.lime, size = size.tiny) plotchar(enterShort, "Go Short", "▼", location.top, color.red, size = size.tiny) hline(50) plot(r)

Note how:

- We create a compound condition that is met only when the user's selection allows for an entry in that direction. A long entry on a crossover of the centerline only triggers the alert when long entries have been enabled in the script's Inputs.
- We offer the user to indicate his repainting preference. When he does not allow the calculations to repaint, we wait until the bar's confirmation to trigger the compound condition. This way, the alert and the marker only appear at the end of the realtime bar.
- If a user of this script wanted to create two distinct script alerts from this script, i.e., one triggering only on longs, and one only on shorts, then he would need to:
    - Select only “Detect Longs” in the inputs and create a first *script alert* on the script.
    - Select only “Detect Shorts” in the Inputs and create another *script alert* on the script.

In strategies

alert() function calls can be used in strategies also, with the provision that strategies, by default, only execute on the close of realtime bars. Unless `calc_on_every_tick = true` is used in the strategy() declaration statement, all alert() calls will use the `alert.freq_once_per_bar_close` frequency, regardless of the argument used for `freq`.

While *script alerts* on strategies will use *order fill events* to trigger alerts when the broker emulator fills orders, alert() can be used advantageously to generate other alert events in strategies.

This strategy creates *alert() function calls* when RSI moves against the trade for three consecutive bars:

//@version=6 strategy("Strategy with selective `alert()` calls") r = ta.rsi(close, 20) // Detect crosses. xUp = ta.crossover(r, 50) xDn = ta.crossunder(r, 50) // Place orders on crosses. if xUp strategy.entry("Long", strategy.long) else if xDn strategy.entry("Short", strategy.short) // Trigger an alert when RSI diverges from our trade's direction. divInLongTrade = strategy.position_size > 0 and ta.falling(r, 3) divInShortTrade = strategy.position_size < 0 and ta.rising(r, 3) if divInLongTrade alert("WARNING: Falling RSI", alert.freq_once_per_bar_close) if divInShortTrade alert("WARNING: Rising RSI", alert.freq_once_per_bar_close) plotchar(xUp, "Go Long", "▲", location.bottom, color.lime, size = size.tiny) plotchar(xDn, "Go Short", "▼", location.top, color.red, size = size.tiny) plotchar(divInLongTrade, "WARNING: Falling RSI", "•", location.top, color.red, size = size.tiny) plotchar(divInShortTrade, "WARNING: Rising RSI", "•", location.bottom, color.lime, size = size.tiny) hline(50) plot(r)

If a user created a *script alert* from this strategy and included both *order fill events* and *alert() function calls* in his alert, the alert would trigger whenever an order is executed, or when one of the alert() calls was executed by the script on the realtime bar's closing iteration, i.e., when barstate.isrealtime and barstate.isconfirmed are both true. The *alert() function events* in the script would only trigger the alert when the realtime bar closes because `alert.freq_once_per_bar_close` is the argument used for the `freq` parameter in the alert() calls.

Order fill events

When a *script alert* is created from an indicator, it can only trigger on *alert() function calls*. However, when a *script alert* is created from a strategy, the user can specify that *order fill events* also trigger the *script alert*. An *order fill event* is any event generated by the broker emulator which causes a simulated order to be executed. It is the equivalent of a trade order being filled by a broker/exchange. Orders are not necessarily executed when they are placed. In a strategy, the execution of orders can only be detected indirectly and after the fact, by analyzing changes in built-in variables such as strategy.opentrades or strategy.position_size. *Script alerts* configured on *order fill events* are thus useful in that they allow the triggering of alerts at the precise moment of an order's execution, before a script's logic can detect it.

Pine Script programmers can customize the alert message sent when specific orders are executed. While this is not a pre-requisite for *order fill events* to trigger, custom alert messages can be useful because they allow custom syntax to be included with alerts in order to route actual orders to a third-party execution engine, for example. Specifying custom alert messages for specific *order fill events* is done by means of the `alert_message` parameter in functions which can generate orders: strategy.close(), strategy.entry(), strategy.exit() and strategy.order().

The argument used for the `alert_message` parameter is a “series string”, so it can be constructed dynamically using any variable available to the script, as long as it is converted to string format.

Let's look at a strategy where we use the `alert_message` parameter in both our strategy.entry() calls:

//@version=6 strategy("Strategy using `alert_message`") r = ta.rsi(close, 20) // Detect crosses. xUp = ta.crossover(r, 50) xDn = ta.crossunder(r, 50) // Place order on crosses using a custom alert message for each. if xUp strategy.entry("Long", strategy.long, stop = high, alert_message = "Stop-buy executed (stop was " + str.tostring(high) + ")") else if xDn strategy.entry("Short", strategy.short, stop = low, alert_message =

"Stop-sell executed (stop was " + str.tostring(low) + ")") plotchar(xUp, "Go Long", "▲", location.bottom, color.lime, size = size.tiny) plotchar(xDn, "Go Short", "▼", location.top, color.red, size = size.tiny) hline(50) plot(r)

Note that:

- We use the `stop` parameter in our strategy.entry() calls, which creates stop-buy and stop-sell orders. This entails that buy orders will only execute once price is higher than the `high` on the bar where the order is placed, and sell orders will only execute once price is lower than the [low] on the bar where the order is placed.
- The up/down arrows which we plot with plotchar() are plotted when orders are **placed**. Any number of bars may elapse before the order is actually executed, and in some cases the order will never be executed because price does not meet the required condition.
- Because we use the same `id` argument for all buy orders, any new buy order placed before a previous order's condition is met will replace that order. The same applies to sell orders.
- Variables included in the `alert_message` argument are evaluated when the order is executed, so when the alert triggers.

When the `alert_message` parameter is used in a strategy's order-generating `strategy.*()` function calls, script users must include the `{{strategy.order.alert_message}}` placeholder in the “Create Alert” dialog box's “Message” field when creating *script alerts* on *order fill events*. This is required so the `alert_message` argument used in the order-generating `strategy.*()` function calls is used in the message of alerts triggering on each *order fill event*. When only using the `{{strategy.order.alert_message}}` placeholder in the “Message” field and the `alert_message` parameter is present in only some of the order-generating `strategy.*()` function calls in your strategy, an empty string will replace the placeholder in the message of alerts triggered by any order-generating `strategy.*()` function call not using the `alert_message` parameter.

While other placeholders can be used in the “Create Alert” dialog box's “Message” field by users creating alerts on *order fill events*, they cannot be used in the argument of `alert_message`.

`alertcondition()` events

The alertcondition() function allows programmers to create individual *alertcondition events* in their indicators. One indicator may contain more than one alertcondition() call. Each call to alertcondition() in a script will create a corresponding alert selectable in the “Condition” dropdown menu of the “Create Alert” dialog box.

While the presence of alertcondition() calls in a **strategy** script will not cause a compilation error, alerts cannot be created from them.

The alertcondition() function has the following signature:

```
alertcondition(condition, title, message)
```

`condition`

A “series bool” value (`true` or `false`) which determines when the alert will trigger. It is a required argument. When the value is `true` the alert will trigger. When the value is `false` the alert will not trigger. Contrary to alert() function calls, alertcondition() calls must start at column zero of a line, so cannot be placed in conditional blocks.

`title`

A “const string” optional argument that sets the name of the alert condition as it will appear in the “Create Alert” dialog box's “Condition” field in the charts UI. If no argument is supplied, “Alert” will be used.

`message`

A “const string” optional argument that specifies the text message to display when the alert triggers. The text will appear in the “Message” field of the “Create Alert” dialog box, from where script users can then modify it when creating an alert. **As this argument must be a “const string”, it must be known at compilation time and thus cannot vary bar to bar.** It can, however, contain placeholders which will be replaced at runtime by dynamic values that may change bar to bar. See this page's Placeholders section for a list.

The alertcondition() function does not include a `freq` parameter. The frequency of *alertcondition() alerts* is determined by users in the “Create Alert” dialog box.

Using one condition

Here is an example of code creating *alertcondition() events*:

//@version=6 indicator("`alertcondition()` on single condition") r = ta.rsi(close, 20) xUp = ta.crossover(r, 50) xDn = ta.crossunder(r, 50) plot(r, "RSI") hline(50) plotchar(xUp, "Long", "▲", location.bottom, color.lime, size = size.tiny) plotchar(xDn, "Short", "▼", location.top, color.red, size = size.tiny) alertcondition(xUp, "Long Alert", "Go long") alertcondition(xDn, "Short Alert", "Go short ")

Because we have two alertcondition() calls in our script, two different alerts will be available in the “Create Alert” dialog box's “Condition” field: “Long Alert” and “Short Alert”.

If we wanted to include the value of RSI when the cross occurs, we could not simply add its value to the `message` string using `str.tostring(r)`, as we could in an alert() call or in an `alert_message` argument in a strategy. We can, however, include it using a placeholder. This shows two alternatives:

alertcondition(xUp, "Long Alert", "Go long. RSI is {{plot_0}}") alertcondition(xDn, "Short Alert", 'Go short. RSI is {{plot("RSI")}}')

Note that:

- The first line uses the `{{plot_0}}` placeholder, where the plot number corresponds to the order of the plot in the script.
- The second line uses the `{{plot("[plot_title]")}}` type of placeholder, which must include the `title` of the plot() call used in our script to plot RSI. Double quotes are used to wrap the plot's title inside the `{{plot("RSI")}}` placeholder. This requires that we use single quotes to wrap the `message` string.
- Using one of these methods, we can include any numeric value that is plotted by our indicator, but as strings cannot be plotted, no string variable can be used.

Using compound conditions

If we want to offer script users the possiblity of creating a single alert from an indicator using multiple alertcondition() calls, we will need to provide options in the script's inputs through which users will indicate the conditions they want to trigger their alert before creating it.

This script demonstrates one way to do it:

//@version=6 indicator("`alertcondition()` on multiple conditions") detectLongsInput = input.bool(true, "Detect Longs") detectShortsInput = input.bool(true, "Detect Shorts") r = ta.rsi(close, 20) // Detect crosses. xUp = ta.crossover(r, 50) xDn = ta.crossunder(r, 50) // Only generate entries when the trade's direction is allowed in inputs. enterLong = detectLongsInput and xUp enterShort = detectShortsInput and xDn plot(r) plotchar(enterLong, "Go Long", "▲", location.bottom, color.lime, size = size.tiny) plotchar(enterShort, "Go Short", "▼", location.top, color.red, size = size.tiny) hline(50) // Trigger the alert when one of the conditions is met. alertcondition(enterLong or enterShort, "Compound alert", "Entry")

Note how the alertcondition() call is allowed to trigger on one of two conditions. Each condition can only trigger the alert if the user enables it in the script's inputs before creating the alert.

Placeholders

These placeholders can be used in the `message` argument of alertcondition() calls. They will be replaced with dynamic values when the alert triggers. They are the only way to include dynamic values (values that can vary bar to bar) in alertcondition() messages.

Note that users creating *alertcondition() alerts* from the “Create Alert” dialog box in the charts UI are also able to use these placeholders in the dialog box's “Message” field.

`{{exchange}}`

Exchange of the symbol used in the alert (NASDAQ, NYSE, MOEX, etc.). Note that for delayed symbols, the exchange will end with “_DL” or “_DLY.” For example, “NYMEX_DL.”

`{{interval}}`

Returns the timeframe of the chart the alert is created on. Note that Range charts are calculated based on 1m data, so the placeholder will always return “1” on any alert created on a Range chart.

`{{open}}`, `{{high}}`, `{{low}}`, `{{close}}`, `{{volume}}`

Corresponding values of the bar on which the alert has been triggered.

`{{plot_0}}`, `{{plot_1}}`, [...], `{{plot_19}}`

Value of the corresponding plot number. Plots are numbered from zero to 19 in order of appearance in the script, so only one of the first 20 plots can be used. For example, the built-in “Volume” indicator has two output series: Volume and Volume MA, so you could use the following:

alertcondition(volume > ta.sma(volume,20), "Volume alert", "Volume ({{plot_0}}) > average ({{plot_1}})")

`{{plot("[plot_title]")}}`

This placeholder can be used when one needs to refer to a plot using the `title` argument used in a plot() call. Note that double quotation marks (") **must** be used inside the placeholder to wrap the `title` argument. This requires that a single quotation mark (') be used to wrap the `message` string:

//@version=6 indicator("") r = ta.rsi(close, 14) xUp = ta.crossover(r, 50) plot(r, "RSI", display = display.none) alertcondition(xUp, "xUp alert", message = 'RSI is bullish at: {{plot("RSI")}}')

`{{ticker}}`

Ticker of the symbol used in the alert (AAPL, BTCUSD, etc.).

`{{time}}`

Returns the time at the beginning of the bar. Time is UTC, formatted as `yyyy-MM-ddTHH:mm:ssZ`, so for example: `2019-08-27T09:56:00Z`.

`{{timenow}}`

Current time when the alert triggers, formatted in the same way as `{{time}}`. The precision is to the nearest second, regardless of the chart's timeframe.

Avoiding repainting with alerts

The most common instances of repainting traders want to avoid with alerts are ones where they must prevent an alert from triggering at some point during the realtime bar when it would **not** have triggered at its close. This can happen when these conditions are met:

- The calculations used in the condition triggering the alert can vary during the realtime bar. This will be the case with any calculation using `high`, `low` or `close`, for example, which includes almost all built-in indicators. It will also be the case with the result of any request.security() call using a higher timeframe than the chart's, when the higher timeframe's current bar has not closed yet.
- The alert can trigger before the close of the realtime bar, so with any frequency other than “Once Per Bar Close”.

The simplest way to avoid this type of repainting is to configure the triggering frequency of alerts so they only trigger on the close of the realtime bar. There is no panacea; avoiding this type of repainting **always** entails waiting for confirmed information, which means the trader must sacrifice immediacy to achieve reliability.

Note that other types of repainting such as those documented in our Repainting section may not be preventable by simply triggering alerts on the close of realtime bars.

Bar states

Introduction

A set of built-in variables in the `barstate` namespace allow your script to detect different properties of the bar on which the script is currently executing.

These states can be used to restrict the execution or the logic of your code to specific bars.

Some built-ins return information on the trading session the current bar belongs to. They are explained in the Session states section.

Bar state built-in variables

Note that while indicators and libraries run on all price or volume updates in real time, strategies not using `calc_on_every_tick` will not; they will only execute when the realtime bar closes. This will affect the detection of bar states in that type of script. On open markets, for example, this code will not display a background until the realtime closes because that is when the strategy runs:

//@version=6 strategy("S") bgcolor(barstate.islast ? color.silver : na)

`barstate.isfirst`

barstate.isfirst is only `true` on the dataset's first bar, i.e., when bar_index is zero.

It can be useful to initialize variables on the first bar only, e.g.:

// Declare array and set its values on the first bar only. FILL_COLOR = color.green var fillColors = array.new<color>(0) if barstate.isfirst // Initialize the array elements with progressively lighter shades of the fill color. array.push(fillColors, color.new(FILL_COLOR, 70)) array.push(fillColors, color.new(FILL_COLOR, 75)) array.push(fillColors, color.new(FILL_COLOR, 80)) array.push(fillColors, color.new(FILL_COLOR, 85)) array.push(fillColors, color.new(FILL_COLOR, 90))

`barstate.islast`

barstate.islast is `true` if the current bar is the last one on the chart, whether that bar is a realtime bar or not.

It can be used to restrict the execution of code to the chart's last bar, which is often useful when drawing lines, labels or tables. Here, we use it to determine when to update a label which we want to appear only on the last bar. We create the label only once and then update its properties using `label.set_*()` functions because it is more efficient:

//@version=6 indicator("", "", true) // Create label on the first bar only. var label hiLabel = label.new(na, na, "") // Update the label's position and text on the last bar, // including on all realtime bar updates. if barstate.islast label.set_xy(hiLabel, bar_index, high) label.set_text(hiLabel, str.tostring(high, format.mintick))

`barstate.ishistory`

barstate.ishistory is `true` on all historical bars. It can never be `true` on a bar when barstate.isrealtime is also `true`, and it does not become `true` on a realtime bar's closing update, when barstate.isconfirmed becomes

`true`. On closed markets, it can be `true` on the same bar where barstate.islast is also `true`.

`barstate.isrealtime`

barstate.isrealtime is `true` if the current data update is a real-time bar update, `false` otherwise (thus it is historical). Note that barstate.islast is also `true` on all realtime bars.

`barstate.isnew`

barstate.isnew is `true` on all historical bars and on the realtime bar's first (opening) update.

All historical bars are considered *new* bars because the Pine Script® runtime executes your script on each bar sequentially, from the chart's first bar in time, to the last. Each historical bar is thus *discovered* by your script as it executes, bar to bar.

barstate.isnew can be useful to reset varip variables when a new realtime bar comes in. The following code will reset `updateNo` to 1 on all historical bars and at the beginning of each realtime bar. It calculates the number of realtime updates during each realtime bar:

//@version=6 indicator("") updateNo() => varip int updateNo = na if barstate.isnew updateNo := 1 else updateNo += 1 plot(updateNo())

`barstate.isconfirmed`

barstate.isconfirmed is `true` on all historical bars and on the last (closing) update of a realtime bar.

It can be useful to avoid repainting by requiring the realtime bar to be closed before a condition can become `true`. We use it here to hold plotting of our RSI until the realtime bar closes and becomes an elapsed realtime bar. It will plot on historical bars because barstate.isconfirmed is always `true` on them:

//@version=6 indicator("") myRSI = ta.rsi(close, 20) plot(barstate.isconfirmed ? myRSI : na)

barstate.isconfirmed will not work when used in a request.security() call.

`barstate.islastconfirmedhistory`

barstate.islastconfirmedhistory is `true` if the script is executing on the dataset's last bar when the market is closed, or on the bar immediately preceding the realtime bar if the market is open.

It can be used to detect the first realtime bar with `barstate.islastconfirmedhistory[1]`, or to postpone server-intensive calculations until the last historical bar, which would otherwise be undetectable on open markets.

Example

Here is an example of a script using `barstate.*` variables:

//@version=6 indicator("Bar States", overlay = true, max_labels_count = 500) stateText() => string txt = "" txt += barstate.isfirst ? "isfirst\n" : "" txt += barstate.islast ? "islast\n" : "" txt += barstate.ishistory ? "ishistory\n" : "" txt += barstate.isrealtime ? "isrealtime\n" : "" txt += barstate.isnew ? "isnew\n" : "" txt += barstate.isconfirmed ? "isconfirmed\n" : "" txt += barstate.islastconfirmedhistory ? "islastconfirmedhistory\n" : "" labelColor = switch barstate.isfirst => color.fuchsia barstate.islastconfirmedhistory => color.gray

barstate.ishistory => color.silver barstate.isconfirmed => color.orange barstate.isnew => color.red => color.yellow label.new(bar_index, na, stateText(), yloc = yloc.abovebar, color = labelColor)

Note that:

- Each state's name will appear in the label's text when it is `true`.
- There are five possible colors for the label's background:
    - fuchsia on the first bar
    - silver on historical bars
    - gray on the last confirmed historical bar
    - orange when a realtime bar is confirmed (when it closes and becomes an elapsed realtime bar)
    - red on the realtime bar's first execution
    - yellow for other executions of the realtime bar

We begin by adding the indicator to the chart of an open market, but before any realtime update is received. Note how the last confirmed history bar is identified in #1, and how the last bar is identified as the last one, but is still considered a historical bar because no realtime updates have been received.

Let's look at what happens when realtime updates start coming in:

Note that:

- The realtime bar is red because it is its first execution, because `barstate.isnew` is `true` and `barstate.ishistory` is no longer `true`, so our switch structure determing our color uses the `barstate.isnew => color.red` branch. This will usually not last long because on the next update `barstate.isnew` will no longer be `true` so the label's color will turn yellow.
- The label of elapsed realtime bars is orange because those bars were not historical bars when they closed. Accordingly, the `barstate.ishistory => color.silver` branch in the switch structure was not executed, but the next one, `barstate.isconfirmed => color.orange` was.

This last example shows how the realtime bar's label will turn yellow after the first execution on the bar. This is the way the label will usually appear on realtime bars:

Chart information

Introduction

Scripts can retrieve multiple types of information about the current chart and its dataset by using a subset of built-in variables. The chart data that scripts can access using these variables includes the following:

- The available prices and volume
- The chart's timeframe
- The dataset's session information
- Symbol information
- Time series information
- The chart's type and color

The following sections on this page list the variables that can access chart information, along with examples demonstrating how to use them. To learn more about all the built-in variables available in Pine Script®, refer to the Built-ins page in this manual and the "Variables" section of our Reference Manual.

> NoteSeveral variables described on this page behave differently in *data requests*. For most of these variables, if a script uses them in the `expression` arguments of `request.*()` calls, or if the script is an indicator that includes a `timeframe` argument in its declaration statement, they represent information from the *requested dataset* rather than the current chart. For example, if a request.security() call includes `"NASDAQ:AAPL"` as its `symbol` argument, the value of the syminfo.prefix variable is `"NASDAQ"` inside that request's context, regardless of the symbol used by the current chart. Likewise, the close value inside the request refers to the share price for NASDAQ:AAPL stock. See the Other timeframes and data page to learn more.

Prices and volume

Most chart datasets include *OHLCV* (open, high, low, close, and volume) values for each available bar. The chart displays the *final* values for each *closed* bar, and the *developing* values for an *open realtime bar*. See the section The basics in the Execution model page to learn more about this behavior.

The variables that store final or developing OHLCV data for the current bar are as follows:

- open: Stores the current bar's opening price. The value does not fluctuate while the bar is open.
- high: Stores the current bar's highest price. If the bar is open, the value represents the bar's highest price as of the current tick.
- low: Stores the current bar's lowest price. If the bar is open, the value represents the bar's lowest price as of the current tick.
- close: Stores the current bar's final closing price, or the *latest* available price if the current bar is open.
- volume: Stores the trading volume reported for the current bar. If the bar is open, the value represents the total volume accumulated from the bar's opening tick to the current tick. The unit that the volume value uses varies by instrument. For example, the unit is typically shares for stocks, lots for Forex pairs, the base currency for cryptocurrency pairs, and contracts for futures and other derivatives.

Pine Script also includes multiple variables that store values *derived* from available OHLC data, including the following:

- hl2: Stores the average of the bar's high and low values (`(high + low) / 2`).
- hlc3: Stores the average of the bar's high, low, and close values (`(high + low + close) / 3`).
- ohlc4: Stores the average of the bar's open, high, low, and close values (`(open + high + low + close) / 4`).
- hlcc4: Stores a weighted average of the bar's high, low, and close values (`(high + low + close + close) / 4`).

> Note
>
> If a bar contains only a *single* price rather than complete OHLC prices, each of these price-based variables stores that value. If no volume data is available for a bar, the value of the volume variable is na.
>
> Some chart types do not *display* OHLC prices, but their datasets still *contain* those prices. For example, a line chart displays only one price per bar. However, the variables that access bar prices still use the available OHLC values from the underlying dataset.
>
> On *non-standard* charts, such as Heikin Ashi or Renko, variables that access price data store the chart's *synthetic* prices, not the instrument's *actual* prices. Therefore, logic that uses these variables can yield different results on these charts.

On tick charts that use the "1T" timeframe, scripts can also use the bid and ask variables to access the current *bid and ask* prices. The *bid* is the *highest* price that an active buyer is willing to pay for the instrument at its current value, and the *ask* is the *lowest* price that an active seller is willing to accept at the current value. On timeframes higher than "1T", the value of these variables is na.

All of these price and volume variables are of the "series float" qualified type, because they store floating-point values that can vary from bar to bar. Scripts can use the `[]` history-referencing operator to retrieve the past values of these variables from previous bars. For example, the expression `close[1]` retrieves the *previous bar's* closing price. Multiple built-in functions also access past values internally. For instance, the expression `ta.change(ohlc4, 20)` is equivalent to `ohlc4 - ohlc4[20]`; both expressions calculate the difference between the current ohlc4 value and the value from *20 bars back*.

The following example uses the prices and volume of current and previous bars on the chart to calculate a condition for a dynamic background color. The script colors the chart's background green only if the current values of the volume and close variables are greater than the previous values, and the current close value is greater than its 10-bar moving average. The script also plots the moving average for visual reference:

//@version=6 indicator("Price and volume variables demo", overlay = true, behind_chart = false) //@variable The 10-bar moving average of the `close` series. float ma = ta.sma(close, 10) //@variable Is `true` if the current volume is greater than the previous bar's volume, and `false` otherwise. bool risingVolume = volume > volume[1] //@variable Is `true` if the current price is greater than the closing price of the previous bar, and `false` otherwise. bool risingPrice = close > close[1] //@variable Is `true` if the current price is above the

current `ma` value, and `false` otherwise. bool closeAboveMA = close > ma // Plot the `ma` series on the chart. plot(ma, "10-bar MA", linewidth = 3) // Highlight the background in green when all three conditions are true. bgcolor(risingVolume and risingPrice and closeAboveMA ? #4caf5080 : na, title = "Condition highlight")

Chart timeframe

Scripts can retrieve the *timeframe* of the current chart by using the timeframe.period or timeframe.main_period variable. Both variables hold a “simple string” value representing the analyzed timeframe:

- The value of the timeframe.period variable represents the timeframe of a specific context. If used outside the `expression` argument of a `request.*()` call, the value represents the *chart's timeframe*, or the script's *main timeframe* if the script is an indicator whose declaration statement includes a `timeframe` argument. When used in the `expression` argument of a `request.*()` call, the value represents the timeframe of the *requested dataset*.
- The value of the timeframe.main_period variable *always* represents the chart's timeframe or the script's main timeframe, even if the script uses it inside a `request.*()` call. This behavior is often useful for nested requests that require information from the chart's timeframe in their logic.

The timeframe strings stored by these variables contain a number representing a *quantity (multiplier)* followed by a single letter representing the *time unit*. For all intraday timeframes that Pine expresses in *minutes*, the timeframe string contains a multiplier *without* a unit postfix. For example, `"1D"` represents the one-day timeframe, `"5"` represents the five-minute timeframe, `"60"` represents the one-hour (60-minute) timeframe, and `"3M"` represents the three-month timeframe. See the Timeframe string specifications section of the Timeframes page to learn more.

Multiple built-in functions feature a `timeframe` parameter that accepts a valid timeframe string. Scripts can pass the timeframe.period or timeframe.main_period variable to this parameter to use the chart's timeframe in the calculations.

For example, the following script uses the timeframe.period variable in calls to the `time()` and `time_close()` functions to retrieve the UNIX timestamps of the current bar's opening time and the previous bar's closing time, then measures the difference between the two timestamps to identify time gaps in the chart's bars. It also uses the variable in a call to timeframe.in_seconds() to retrieve the typical number of seconds represented by the timeframe, then uses the result to express the time difference as an approximate number of bars. Each time that the script detects a gap, it displays formatted text containing the gap's size in minutes and bars, the timeframe.period value, and the number of bars elapsed since the previous gap in a label at the current bar's high:

//@version=6 indicator("`timeframe.period` demo", overlay = true, behind_chart = false) //@variable The previous bar's closing UNIX timestamp. int prevCloseTime = time_close(timeframe = timeframe.period, timeframe_bars_back = 1) //@variable The current bar's opening UNIX timestamp. int currOpenTime = time(timeframe = timeframe.period) //@variable The number of seconds elapsed between the `prevCloseTime` and `currOpenTime` timestamps. int timeDiff = (currOpenTime - prevCloseTime) / 1000 //@variable The approximate span of the time difference in bars. int gapBarLength = int(timeDiff / timeframe.in_seconds(timeframe.period)) //@variable Is `true` if the difference is greater than zero. bool hasGap = timeDiff > 0 //@variable The number of bars since the `hasGap` value was last `true`. int barsSinceLastGap = ta.barssince(hasGap) // If a time gap occurs, display the gap in minutes, the current timeframe, and the bars since the previous gap // in a label at the current bar's high. if hasGap label.new(

bar_index, high, text = str.format("{0}-minute gap (~{1} bars) on "{2}" timeframe\nBars since previous gap: {3}", timeDiff / 60, gapBarLength, timeframe.period, barsSinceLastGap[1] + 1))

Note that:

- Programmers can also use an *empty string* ("") as a `timeframe` argument to specify the same timeframe as timeframe.period. For instance, our example script yields the same results if we use "" instead of the variable in the time(), time_close(), and timeframe.in_seconds() calls.

Scripts can use the timeframe.multiplier variable to retrieve a "simple int" value representing the *multiplier* of the timeframe referenced by timeframe.period. For example, if the timeframe is "2D", the timeframe.multiplier value is `2`. If the timeframe is "30S", the variable's value is `30`.

The following `timeframe.*` variables store "simple bool" values to indicate the *unit* of the timeframe referenced by timeframe.period:

- timeframe.isticks: Stores `true` if the current timeframe is a tick-based timeframe (e.g., `"10T"`), and `false` otherwise.
- timeframe.isseconds: Stores `true` if the current timeframe is a second-based timeframe (e.g., `"30S"`), and `false` otherwise.
- timeframe.isminutes: Stores `true` if the current timeframe is an intraday timeframe in minutes (`"1"` to `"1440"`), and `false` otherwise.
- timeframe.isintraday: Stores `true` if the current timeframe is any intraday timeframe (minutes, seconds, or ticks), and `false` otherwise.
- timeframe.isdaily: Stores `true` if the current timeframe is day-based (`"1D"` to `"365D"`), and `false` otherwise.
- timeframe.isweekly: Stores `true` if the current timeframe is week-based (`"1W"` to `"52W"`), and `false` otherwise.
- timeframe.ismonthly: Stores `true` if the current timeframe is month-based (`"1M"` to `"12M"`), and `false` otherwise.
- timeframe.isdwm: Stores `true` if the current timeframe is expressed in days, weeks, or months, and `false` otherwise.

The example below uses these variables to construct a custom representation of the chart's timeframe. On the first bar, the script uses multiple `timeframe.is*` variables in a switch statement to retrieve a string representing the chart timeframe's unit, then creates a formatted string using the result and the value of timeframe.multiplier. It displays the final text in a single-cell table in the chart's top-right corner:

//@version=6 indicator("Timeframe multiplier and unit variables demo", overlay = true, behind_chart = false) //@variable References a single-cell `table` object that displays the chart's timeframe in the top-right corner. var table displayTable = table.new(position.top_right, 1, 1, color.blue) if barstate.isfirst //@variable A string that describes the unit of the chart's timeframe. string unitsStr = switch timeframe.isticks => "tick" timeframe.isseconds => "second" timeframe.isminutes => "minute" timeframe.isdaily => "day" timeframe.isweekly => "week" => "month" //@variable A formatted string that contains the timeframe's multiplier and unit descriptor. string displayStr = str.format("Chart timeframe: {0,number,#} {1}{2}", timeframe.multiplier, unitsStr, timeframe.multiplier > 1 ? "s" : "") // Initialize the table's cell to display the text from the `displayStr` value. displayTable.cell(0, 0, displayStr, text_color = color.white, text_size = 24)

Refer to the Timeframes page to learn more about the `timeframe.*` built-ins and how to use them.

Session information

Pine Script includes multiple built-in variables that can retrieve information about an intraday dataset's *session*, which refers to the days and the times of day in which trading data is available. These variables represent session information for the current chart's dataset, or for a requested dataset if the script uses them in the `expression` argument of a `request.*()` function call.

Scripts can access the named session for the current chart's dataset by using the syminfo.session variable. The variable holds a "simple string" value representing the session's name. In most cases, the string matches the value of either of the following `session.*` constants by default:

- session.regular: Stores the string for the instrument's default trading session (`"regular"`). The default session varies with the instrument. For instance, on the charts for US equities, the string typically corresponds to regular trading hours (RTH). By contrast, on the charts for several futures contracts, it corresponds to electronic trading hours (ETH), because ETH is enabled by default. On timeframes higher than or equal to 1D, the values of syminfo.session and session.regular are always equal.
- session.extended: Stores the string for the instrument's extended session (`"extended"`), which includes data from pre- and post-market hours. Outside data requests, the syminfo.session variable holds this string only if the chart includes the option for extended sessions and the user selects that option in the chart's "Session" settings.

The syminfo.session variable can also hold *other* unique strings for specific subsessions defined by the exchange or data provider. For instance, the value is `"us_regular"` on a CME futures chart that uses the RTH session, and `"24h"` on an equities chart that includes *overnight (24-hour)* sessions. Refer to the Retrieving named sessions section of the Sessions page to learn more about named session strings.

Programmers can use the string from this variable to create session-specific logic in their scripts, or pass the string to the `session` parameter of the ticker.new() or ticker.modify() functions to create ticker identifiers for requesting data using the same session as the chart. See the Custom contexts section of the Other timeframes and data page for more information about these `ticker.*()` functions.

Additional variables in the `session` namespace hold "series bool" values that indicate the current market state or track the first and last bars in named sessions:

- session.ismarket: Stores `true` if the current bar belongs to the *regular* (default) session, and `false` otherwise. The value is typically always `true` on timeframes higher than or equal to 1D.
- session.ispremarket: Stores `true` if the current bar is a *pre-market* bar, and `false` otherwise. The value is always `false` on timeframes higher than or equal to 1D.
- session.ispostmarket: Stores `true` if the current bar is a *post-market* bar, and `false` otherwise. The value is always `false` on timeframes higher than or equal to 1D.
- session.isfirstbar: Stores `true` if the current bar is the first bar of the daily session, and `false` otherwise. If the dataset uses only the regular session, the value is `true` on the first bar in that session. If the dataset uses extended sessions, the value is `true` on the first *pre-market* bar. If the dataset uses 24-hour sessions, the value is `true` on the first *overnight* bar.
- session.isfirstbar_regular: Stores `true` if the current bar is the first bar of the instrument's *regular* session, and `false` otherwise, regardless of the dataset's session settings.
- session.islastbar: Stores `true` if the current bar is the last bar of the daily session, and `false` otherwise. If the dataset uses only regular trading hours, the value is `true` on the last bar in that

session. If the dataset uses extended or overnight sessions, the value is `true` on the last *post-market* bar.

- session.islastbar_regular: Stores `true` if the current bar is the last bar in the instrument's regular session, and `false` otherwise, regardless of the dataset's session settings.

The following example demonstrates the behavior of these variables. The script below calculates and plots the total volume for each subsession on an intraday chart that includes extended or overnight sessions. The script declares four persistent variables to store the total volume for regular, pre-market, post-market, and overnight hours. Then, inside the if structure, it uses session.ismarket, session.ispremarket, and session.ispostmarket as conditions for resetting or incrementing the value of each variable based on the current session state. The script also uses one of the `session.isfirstbar*` or `session.islastbar*` variables, depending on the selected inputs, as a condition to color the background of specific session bars. Additionally, the script checks the value of the syminfo.session variable to confirm that these calculations are compatible with the chart. It raises a custom *runtime error* if the value is not `"extended"` or `"24h"`, indicating that the chart is day-based or does not use the "Extended" or "24 hour" session setting:

//@version=6 indicator("`syminfo.session` and `session.*` demo", format = format.volume) // Create inputs for highlighting the first or last bar in a given session type. string firstLastInput = input.string("First", "Highlight", ["First", "Last"], inline = "0") string regularExtendInput = input.string("Regular Session", "Bar In ", ["Regular Session", "Extended/Overnight Hours"], inline = "0") // Raise an error if the current chart does not use an "Extended" or "24 hour" session. if not (syminfo.session == session.extended or syminfo.session == "24h") runtime.error("Open an intraday chart with an 'Extended' or '24 hour' session to use this script.") // Declare persistent variables to store the total volume for regular, pre-market, post-market, and overnight hours. var float regularTotal = 0.0 var float premarketTotal = 0.0 var float postmarketTotal = 0.0 var float overnightTotal = 0.0 // For each session, accumulate the current session's total and reset the previous session's total to 0. if session.ismarket regularTotal += volume premarketTotal := 0.0 else if session.ispremarket premarketTotal += volume postmarketTotal := 0.0 overnightTotal := 0.0 else if session.ispostmarket postmarketTotal += volume regularTotal := 0.0 else overnightTotal += volume postmarketTotal := 0.0 //@variable Is `true` for the first or last bar in the specified session type, and `false` for all others. bool highlightBar = if firstLastInput == "First" regularExtendInput == "Regular Session" ? session.isfirstbar_regular : session.isfirstbar else regularExtendInput == "Regular Session" ? session.islastbar_regular : session.islastbar // Plot the total volume for each session. plot(regularTotal == 0 ? na : regularTotal, "Regular total volume", #ff623b, style = plot.style_areabr) plot(premarketTotal == 0 ? na : premarketTotal, "Pre-market total volume", #ff9800, style = plot.style_areabr) plot(postmarketTotal == 0 ? na : postmarketTotal, "Post-market total volume", #2962ff, style = plot.style_areabr) plot(overnightTotal == 0 ? na : overnightTotal, "Overnight total volume", #d500f9, style = plot.style_areabr) // Highlight the background of the specified first/last bar in gray. bgcolor(highlightBar ? #787b8680 : na, title = "First/Last bar highlight")

Refer to the Sessions page to learn more about market sessions and the session-related built-ins.

Symbol information

The built-in variables in the `syminfo` namespace hold essential information about the chart's symbol and the underlying instrument. Most of these variables, excluding syminfo.main_tickerid, can also represent information relating to a requested dataset if a script uses them as the `expression` argument in a `request.*()` function call. Most `syminfo.*` variables have the "simple" type qualifier, because their values do not change after the first bar. However, the variables relating to analyst recommendations and targets have the "series" qualifier, because they store dynamic data that can change over time.

The available `syminfo.*` variables include the following:

- syminfo.ticker: Stores a string representing the dataset's symbol without the exchange prefix. For example, the value is `"AAPL"` for a NASDAQ:AAPL chart, `"BTCUSD"` for a BITSTAMP:BTCUSD chart, and `"ES1!"` for a CME_MINI_DL:ES1! chart.
- syminfo.prefix: Stores a string representing the symbol's broker/exchange identifier. For example, the value is `"NASDAQ"` for a NASDAQ:AAPL chart, `"BITSTAMP"` for a BITSTAMP:BTCUSD chart, and `"CME_MINI_DL"` for a CME_MINI_DL:ES1! chart.
- syminfo.root: Stores a string representing the instrument's *root code* if the symbol refers to a futures contract or another applicable derivative. Otherwise, it stores the same value as syminfo.ticker. For example, the value is `"ZW"` for wheat futures symbols such as ZW1! and ZWU2026.
- syminfo.tickerid: Stores a string representing the *ticker identifier* (ticker ID) of the chart's dataset, or of a requested dataset if the script uses it in the `expression` argument of a `request.*()` call. The ticker ID contains the dataset's symbol *with* the exchange prefix (e.g., `"NASDAQ:AAPL"`). The string can also contain information about dataset *modifiers*, such as extended hours, dividend adjustments, and currency conversion. Programmers can retrieve the dataset's ticker ID without modifiers by passing this variable to the ticker.standard() function.
- syminfo.main_tickerid: Stores a string representing the ticker ID of the *main dataset* on which the script runs. Unlike syminfo.tickerid, this variable does *not* store a different value when used in a `request.*()` call's `expression` argument. Therefore, scripts can use this variable to retrieve the current chart's ticker ID while executing data requests.
- syminfo.basecurrency: Stores a string representing the instrument's *base currency* if the current symbol refers to a Forex pair, a cryptocurrency pair, or a derivative instrument based on a currency pair. Otherwise, it stores an empty string. For example, the variable's value is `"EUR"` for any EURJPY pair, `"BTC"` for any BTCUSDT pair, `"CAD"` for CME:6C1! futures, and `""` for NASDAQ:AAPL stock.
- syminfo.currency: Stores a string representing the currency of the instrument's quoted prices, or `"NONE"` if the dataset's values do not represent currency amounts. For example, the value is `"JPY"` for a EURJPY pair, `"USD"` for NASDAQ:AAPL stock, and `"NONE"` for the TVC:US10Y bond yield dataset.
- syminfo.country: Stores a string representing the two-letter country code of the instrument's exchange or data provider, or an empty string if the exchange is not linked to a specific country. For example, the value is `"US"` for NASDAQ:AAPL, `"GB"` for LSE:AAPL, and `""` for BITSTAMP:BTCUSD.
- syminfo.timezone: Stores a string representing the dataset's *exchange time zone* in the IANA time zone database format. For example, the value is `"America/New_York"` for stocks traded on the NASDAQ exchange. See the Time zone strings section of the Time page to learn more about IANA identifiers.
- syminfo.session: Stores a string representing the dataset's session setting. See the Session information section above for more information.
- syminfo.current_contract: Stores the "string" ticker identifier of the underlying contract if the current symbol refers to a continuous futures dataset, and an empty string otherwise.
- syminfo.description: Stores a string containing the description or extended name of the current instrument or dataset.
- syminfo.employees: Stores the issuing company's reported number of employees if the current symbol refers to a stock, and na otherwise.
- syminfo.shareholders: Stores an "int" value representing the total number of reported shareholders if the current instrument is a stock, and na otherwise.
- syminfo.shares_outstanding_float: Stores the total reported number of outstanding shares, excluding any restricted shares, if the symbol refers to a stock. For other symbols, the value is na.

- syminfo.shares_outstanding_total: Stores the total reported number of outstanding shares, including restricted shares held by insiders, major shareholders, and employees, if the symbol refers to a stock. For other symbols, the value is na.
- syminfo.expiration_date: Stores an “int” UNIX timestamp representing the start of the last day of the current contract if the symbol refers to a non-continuous futures dataset. On other datasets, the value is na.
- syminfo.isin: Holds a string representing the International Securities Identification Number (ISIN) for the underlying instrument, or an empty string if no ISIN information is available for the instrument.
- syminfo.mincontract: Stores a “float” value representing the minimum number of contracts/lots/shares/units required for a trade, as set by the exchange. For many instruments, the value is 1. Additionally, note that the value is 1 if the symbol does not refer to a tradable instrument.
- syminfo.minmove: Stores a whole number for calculating the smallest increment by which the instrument’s prices change. It is the *numerator* of the syminfo.mintick formula: `syminfo.mintick = syminfo.minmove / syminfo.pricescale`.
- syminfo.pricescale: Stores a whole number for calculating the smallest increment by which the instrument’s prices change. It is the *denominator* of the syminfo.mintick formula: `syminfo.mintick = syminfo.minmove / syminfo.pricescale`.
- syminfo.mintick: Stores the dataset’s minimum tick size, i.e., the smallest increment by which the instrument’s recorded prices change. For example, the value is `0.00001` for the OANDA:EURUSD pair, `0.01` for NASDAQ:AAPL stock, and `0.25` for CME_MINI_DL:ES1! futures.
- syminfo.pointvalue: Stores the instrument’s point value, which represents a multiplier of the instrument’s price for determining the value of a single contract. For most instruments, the variable’s value is typically 1, which means the instrument’s price directly represents the value of a contract, share, etc. However, for some futures instruments, the price on the chart represents the value per *index point* or *unit* of the underlying *commodity*. For example, the point value for COMEX:GC1! futures is 100, because the standard size of a single contract is *100 troy ounces* of gold. Therefore, the cost of purchasing one contract is *100 times* the price on the chart.
- syminfo.type: Stores a string representing the instrument’s type. Possible values include `"stock"`, `"futures"`, `"index"`, `"forex"`, `"crypto"`, `"fund"`, `"dr"`, `"cfd"`, `"bond"`, `"warrant"`, `"structured"`, and `"right"`.
- syminfo.volumetype: Stores a string indicating the type of volume reported for the instrument. Possible values include `"base"`, `"quote"`, `"tick"`, and `"n/a"`.
- syminfo.sector: Stores a string representing the sector associated with the underlying instrument, or an empty string if there is no associated sector. The sector refers to a broad classification of the associated economy. For example, the value is `"Electronic Technology"` for NASDAQ:AAPL, `"Technology Services"` for NYSE:IBM, and `"Miscellaneous"` for AMEX:SPY.
- syminfo.industry: Stores a string representing the industry associated with the underlying instrument, or an empty string if there is no associated industry. The industry associated with an instrument refers to a subset of the corresponding sector. For example, the value is `"Telecommunications Equipment"` for NASDAQ:AAPL, `"Information Technology Services"` for NYSE:IBM, `"Investment Trusts/Mutual Funds"` for AMEX:SPY, and `""` for SP:SPX.
- syminfo.recommendations_buy: Stores the total number of analysts who gave the current instrument a “Buy” rating.
- syminfo.recommendations_buy_strong: Stores the total number of analysts who gave the current instrument a “Strong Buy” rating.
- syminfo.recommendations_sell: Stores the total number of analysts who gave the current instrument a “Sell” rating.

- syminfo.recommendations_sell_strong: Stores the total number of analysts who gave the current instrument a “Strong Sell” rating.
- syminfo.recommendations_hold: Stores the total number of analysts who gave the current instrument a “Hold” rating.
- syminfo.recommendations_total: Stores the total number of recommendations for the current instrument.
- syminfo.recommendations_date: Stores an “int” UNIX timestamp representing the starting date of the latest set of recommendations for the current instrument.
- syminfo.target_price_average: Stores the average of the last yearly analyst price targets for the instrument.
- syminfo.target_price_high: Stores the last highest yearly analyst price target for the instrument.
- syminfo.target_price_low: Stores the last lowest yearly analyst price target for the instrument.
- syminfo.target_price_median: Stores the median of the last yearly analyst price targets for the instrument.
- syminfo.target_price_date: Stores an “int” UNIX timestamp representing the starting date of the last analyst price target prediction for the current instrument.
- syminfo.target_price_estimates: Stores the latest total number of analyst price target predictions for the current instrument.

The example script below displays a table containing a simple summary of symbol and instrument information from the chart. On the first bar, the script creates two “string” arrays using the array.from function. The first array contains titles for the table’s first column. The second array contains corresponding strings from multiple `syminfo.*` variables for the second column. The script iterates through the arrays and populates the cells on each table row within a for loop:

//@version=6 indicator("`syminfo.*` variables demo", overlay = true, behind_chart = false) if barstate.isfirst //@variable References a `table` object that displays symbol and instrument information on the chart. table displayTable = table.new(position.middle_right, 2, 9, border_color = chart.fg_color, border_width = 1) //@variable References an array of "string" titles for the table's first column. array<string> titles = array.from("Ticker ID", "Symbol", "Prefix", "Type", "Description", "ISIN", "Currency", "Tick size", "Point value") //@variable References an array of "string" representations of `syminfo.*` values for the second column. array<string> values = array.from(syminfo.tickerid, syminfo.ticker, syminfo.prefix, syminfo.type, syminfo.description, syminfo.isin, syminfo.currency, str.tostring(syminfo.mintick), str.tostring(syminfo.pointvalue)) // Loop through the arrays and populate the rows with the corresponding `titles` and `values` elements. for i = 0 to 8 displayTable.cell(0, i, titles.get(i), text_color = chart.fg_color) displayTable.cell(1, i, values.get(i), text_color = chart.fg_color)

Note that:

- The script initializes and populates the table only on the *first* bar because the values of the `syminfo.*` variables used in the code do not change from bar to bar. After the script creates the table and sets its cells on the first bar, the table’s output persists on the right side of the chart.
- The script uses the chart.fg_color variable to set the color of the table’s borders and text. The variable’s value changes based on the color of the chart’s *background*. See the Chart type and color section below for more information.

Time series information

Two built-in variables store information about the *bar indices* in the time series for the current chart, or for a requested dataset if used in the `expression` argument of a `request.*()` function call:

- bar_index: Stores a “series int” value representing the time series index for the current bar. The value is 0 on the first available bar, 1 on the second bar, and so on. The value on the last available bar is one less than the *total* number of bars.
- last_bar_index: Stores a “series int” value representing the time series index of the last available bar. The value is consistently one less than the total number of available bars, even while the script executes on the first bar.

Several variables in the `barstate` namespace hold “series bool” values to indicate the *states* of each bar in the chart’s dataset or a requested dataset. These variables include the following:

- barstate.isfirst: Stores `true` if the current bar is the first available bar, and `false` otherwise. The value is equivalent to the result of `bar_index == 0`.
- barstate.islast: Stores `true` if the current bar is the last available bar, and `false` otherwise. The value is equivalent to the result of `bar_index == last_bar_index`.
- barstate.isnew: Stores `true` on all historical bars and on the *first tick* of an open realtime bar. On subsequent ticks within an open bar, the value is `false`.
- barstate.isconfirmed: Stores `true` if the current bar is *closed* (confirmed), and `false` if the bar is *open*. The runtime system *commits (saves)* a script’s calculated data to the time series when the value is `true`.
- barstate.ishistory: Stores `true` if the current bar is historical, meaning that it closed *before* the script loaded on the dataset, and `false` otherwise.
- barstate.isrealtime: The *opposite* of barstate.ishistory. Stores `true` if the current bar is a realtime bar, which closes *after* the script loads, and `false` otherwise.
- barstate.islastconfirmedhistory: Stores `true` on the last available *historical* bar, and `false` otherwise.

Refer to the Bar states page to learn more about these variables and how they work. For detailed information about how scripts execute across historical and realtime bars, and how they manage data in the time series based on bar states, refer to the Execution model page.

The following example calculates a volume-weighted average price (VWAP) over periods spanning a specified number of bars. The script resets the VWAP calculation on each bar whose bar_index value is divisible by the specified period. For instance, with the default input value of 100, the calculation resets on bar 0, 100, 200, and so on. The script plots the VWAP series and highlights the background of each bar on which the reset occurs. Additionally, the script uses the bar_index, last_bar_index, and barstate.ishistory variables to calculate the total number of historical bars, realtime bars, and completed periods, then displays the results in a single-cell table on the last bar:

//@version=6 indicator("`*bar_index` and `barstate.*` demo", overlay = true, behind_chart = false) //@variable The total number of bars in each VWAP calculation. int periodInput = input.int(100, "VWAP period", 2) //@variable Is `true` once every `periodInput` bars, starting from bar 0, and `false` otherwise. bool resetVWAP = bar_index % periodInput == 0 //@variable The VWAP for the current period. The calculation resets each time the specified number of bars elapses. float vwap = ta.vwap(hlc3, resetVWAP) // This `if` structure's scope executes on the last bar. // The condition `bar_index == last_bar_index` is equivalent to `barstate.islast`. if bar_index == last_bar_index //@variable References a single-cell table that displays bar and period information in the top-right corner. var table infoTable = table.new(position.top_right, 1, 1) //@variable The initial total bars if the last bar is historical, and one less than the total otherwise. var int

historicalBars = barstate.ishistory ? bar_index + 1 : bar_index //@variable A formatted string containing bar and VWAP period information. string displayText = str.format("Historical bars: {0}\nRealtime bars: {1}\nVWAP period: {2} bars\nComplete periods on chart: {3}", historicalBars, bar_index + 1 - historicalBars, periodInput, int(bar_index / periodInput)) // Display the text from the `displayText` string in the table's cell. infoTable.cell(0, 0, displayText, text_color = #000000, bgcolor = #2196f3) // Plot the `vwap` series and highlight the background on each calculation reset. plot(vwap, "Periodic VWAP", linewidth = 3) bgcolor(resetVWAP ? #ff98004d : na, title = "VWAP reset highlight")

Note that:

- On the first bar where the bar_index and last_bar_index values are equal, the script checks the value of the barstate.ishistory variable to determine whether that bar is historical. If the value is `true`, the total number of historical bars is one greater than the bar index on that bar. Otherwise, the number of historical bars equals the bar index. As new bars become available, the script counts the number of realtime bars by subtracting the historical total from the value of `bar_index + 1`.
- The script counts the total number of completed VWAP periods by dividing the latest bar_index value by the input period, then rounding the result down to the nearest integer.

Pine Script also features several built-in variables that access *time* information for the bars on the chart or a requested dataset:

- The time and time_close variables hold UNIX timestamps representing the current bar's opening and closing times, respectively.
- The last_bar_time variable stores a UNIX timestamp representing the opening time of the last available bar.
- The time_tradingday variable holds a UNIX timestamp representing the starting time of the trading day to which the current bar belongs.
- The timenow variable stores the UNIX timestamp of the script's latest execution.
- The year, month, weekofyear, dayofmonth, dayofweek, hour, minute, and second variables store calendar-based values derived from the current bar's opening time. The values are expressed in the exchange time zone.
- The chart.left_visible_bar_time and chart.right_visible_bar_time variables store UNIX timestamps representing the opening times of the leftmost and rightmost visible chart bars.

Refer to the Time page for detailed information about these variables and examples of how they work.

Chart type and color

Multiple built-in variables in the `chart` namespace hold "simple bool" values to indicate the type of chart on which the script runs. These variables can also indicate a requested chart dataset's type when used in the `expression` argument of a `request.*()` function call:

- chart.is_standard: Stores `true` if the chart is any of the *standard* types, including line charts, bar charts, candlestick charts, and other chart types that use the instrument's actual OHLC prices, as opposed to calculated prices. Otherwise, the value is `false`.
- chart.is_heikinashi: Stores `true` if the chart type is Heikin Ashi, and `false` otherwise.
- chart.is_linebreak: Stores `true` if the chart type is line break, and `false` otherwise.
- chart.is_pnf: Stores `true` if the chart type is point & figure, and `false` otherwise.
- chart.is_kagi: Stores `true` if the chart type is Kagi, and `false` otherwise.

- chart.is_range: Stores `true` if the chart type is range, and `false` otherwise.
- chart.is_renko: Stores `true` if the chart type is Renko, and `false` otherwise.

These `chart.is_*` variables are typically useful when a script's logic must respond differently on non-standard charts. For example, the following script demonstrates a simple strategy that places market orders to enter trades based on the crossing of two moving averages. On a non-standard chart, these orders can generate *misleading* results, because Pine's broker emulator fills them at the chart's *calculated* prices rather than using the instrument's actual prices. To prevent such results, the script allows orders only on standard chart types by using chart.is_standard in the conditions that control the strategy.entry() commands. As shown below, if the script runs on a non-standard chart, it does not generate any orders or display performance data in the strategy report:

//@version=6 strategy("`chart.is_standard` demo", overlay = true, behind_chart = false) // Calculate two moving averages for the cross signal. float ma1 = ta.sma(close, 10) float ma2 = ta.sma(close, 20) if chart.is_standard // This logic executes only on standard charts, preventing misleading trade data on non-standard charts. // Close any short position and open a new long position if the first MA crosses over the second MA. if ta.crossover(ma1, ma2) strategy.entry("Buy", strategy.long) // Close any long position and open a new short position if the first MA crosses under the second MA. if ta.crossunder(ma1, ma2) strategy.entry("Sell", strategy.short) // Plot the moving averages on the chart. plot(ma1, "Fast MA", color.orange, 2) plot(ma2, "Slow MA", color.blue, 3) // Color the background translucent red if the chart is not a standard type. bgcolor(not chart.is_standard ? color.new(color.red, 60) : na, title = "Non-standard chart highlight")

Note that:

- An alternative way to avoid misleading trade prices on *Heikin Ashi* charts is to include `fill_orders_on_standard_ohlc = true` in the strategy() declaration statement. This argument configures the broker emulator to fill orders using *standard* chart prices by default. See the Strategies page to learn more about strategy scripts.

The `chart` namespace also features the following variables that store "input color" values based on the background color defined in the chart's settings:

- chart.bg_color: Stores the value of the chart's background color, as defined by the "Background" inputs in the "Canvas" tab of the chart's settings.
- chart.fg_color: Stores a grayscale color that provides high contrast with most chart background colors. For dark backgrounds, the color is `#dbdbdb`. For light backgrounds, the color is `#0f0f0f`.

The following script creates a single-cell table to indicate whether the chart's background is light or dark, based on the value of chart.fg_color. If the value is `#0f0f0f`, the table's text states that the background is considered light. Otherwise, it states that the background is considered dark. The script colors the table's background using the foreground color, and it sets the text color using the value of chart.bg_color. The script also sets the table's frame color using the middle value of a gradient from the background color to the foreground color:

//@version=6 indicator("`chart.bg_color` and `chart.fg_color` demo") if barstate.isfirst //@variable The color from the midpoint of a gradient using the chart's background and foreground colors. color middleColor = color.from_gradient(0.5, 0, 1, chart.bg_color, chart.fg_color) //@variable References a table with `chart.fg_color` as the background color and `middleColor` as the frame color. table displayTable = table.new(position.middle_center, 1, 1, chart.fg_color, middleColor, 2) //@variable Is `true` if the chart's

foreground color is `#0f0f0f`, indicating a light background. bool isLight = chart.fg_color == #0f0f0f //@variable A string indicating whether the chart's background is considered light or dark. string displayText = "The chart's background is considered " + (isLight ? "light." : "dark.") // Initialize the table's cell to display the string's text colored using the chart's background color. displayTable.cell(0, 0, displayText, text_color = chart.bg_color, text_size = 30)

Inputs

Introduction

Inputs receive values that users can change from a script's “Settings/Inputs” tab. By utilizing inputs, programmers can write scripts that users can more easily adapt to their preferences.

The following script plots a 20-bar simple moving average (SMA) using a call to the ta.sma() function. While it is straightforward to write, the code is not very *flexible* because the function call uses specific `source` and `length` arguments that users cannot change without modifying the code:

//@version=6 indicator("MA", "", true) plot(ta.sma(close, 20))

If we write our script this way instead, it becomes much more flexible, as users can select the `source` and the `length` values they want to use from the “Settings/Inputs” tab without changing the source code:

//@version=6 indicator("MA", "", true) sourceInput = input(close, "Source") lengthInput = input(20, "Length") plot(ta.sma(sourceInput, lengthInput))

Inputs are only accessible while a script runs on a chart. Users can access script inputs from the “Settings” dialog box. To open this dialog, users can:

- Double-click on the name of an on-chart indicator
- Right-click on the script's name and choose the “Settings” item from the dropdown menu
- Choose the “Settings” item from the “More” menu icon (three dots) that appears when hovering over the indicator's name on the chart
- Double-click on the indicator's name from the Data Window (fourth icon down to the right of the chart)

The “Settings” dialog always contains the “Style” and “Visibility” tabs, which allow users to specify their preferences about the script's visuals and the chart timeframes that can display its outputs.

When a script contains calls to `input.*()` functions, an “Inputs” tab also appears in the “Settings” dialog box.

Scripts process inputs when users add them to the chart or change the values in the script's “Settings/Inputs” tab. Any changes to a script's inputs prompt it to re-execute across all available data using the new specified values.

Input functions

Pine Script® features the following input functions:

- input()
- input.int()
- input.float()
- input.bool()
- input.color()
- input.string()
- input.text_area()
- input.timeframe()
- input.symbol()

- input.source()
- input.session()
- input.time()
- input.price()
- input.enum()

Scripts create input *widgets* in the “Inputs” tab that accept different types of inputs based on their `input.*()` function calls. By default, each input appears on a new line of the “Inputs” tab in the order of the `input.*()` calls. Programmers can also organize inputs in different ways by using the `input.*()` functions' `group` and `inline` parameters. See this section below for more information.

Our Style guide recommends placing `input.*()` calls at the beginning of the script.

Input functions typically contain several parameters that allow programmers to define their default values, value limits, their organization in the “Inputs” tab, and other properties.

Since an `input.*()` call is simply another function call in Pine Script, programmers can combine them with arithmetic, comparison, logical, and ternary operators to assign expressions to variables. This simple script compares the result from a call to input.string() to the “On” string and assigns the result to the `plotDisplayInput` variable. This variable is of the “input bool” type because the == operator returns a “bool” value:

//@version=6 indicator("Input in an expression`", "", true) bool plotDisplayInput = input.string("On", "Plot Display", options = ["On", "Off"]) == "On" plot(plotDisplayInput ? close : na)

All values returned by `input.*()` functions except “source” ones are “input” qualified values. See our User Manual's section on type qualifiers for more information.

Input function parameters

The parameters common to all input functions are: `defval`, `title`, `tooltip`, `inline`, `group`, `display`, and `active`. Some input functions also include other parameters: `options`, `minval`, `maxval`, `step` and `confirm`.

Most input parameters require “const” arguments. However, two parameters allow values with stronger qualifiers: the `active` parameter of all `input*()` functions accepts an “input bool” value, and the `defval` parameter of input.source() accepts a “series float” value.

The parameters that require “const” arguments *cannot* use dynamic values or results from other `input*()` calls as arguments, because “input” and other qualifiers are *stronger* than the “const” qualifier. See the Type system page for more information.

Let’s examine each parameter:

`defval`

The default value assigned to the input variable, and the initial value that appears in the input widget. It is the first parameter of all `input*()` functions. The required type for a `defval` argument depends on the input function type, e.g., an “int” `defval` argument for input.int(), a “string” `defval` argument for input.string(), etc. The generic input function infers its input type based on the `defval` argument used in the input() call.

`title`

The input field's label in the "Inputs" tab. If the function call does not specify a `title` string, the input variable's name appears as the label.

`tooltip`

An optional string that offers more information about the input. Using a `tooltip` argument displays a question mark icon to the right of the input field, which shows the tooltip's text when users hover over it. The `tooltip` string supports newline (`\n`) characters.

Note that if multiple input widgets appear on the same line (using `inline`), the tooltip always appears to the right of the *rightmost* field and displays the text of the *last* `tooltip` argument specified in the line.

`inline`

Using the same `inline` argument in multiple `input*()` calls displays their input widgets on the *same line* in the "Inputs" tab. The tab's width limits the amount of input widgets that can fit on one line; longer lines are automatically wrapped. The `inline` string is case-sensitive, so `input*()` calls must use the same characters and letter case in their `inline` arguments to appear in the same line. Unlike the `group` heading, the `inline` string does not appear in the "Inputs" tab.

For consistent input alignment in the "Settings/Inputs" tab, specify a title for only the first input in any one line. For more information, see the page for the warning The inputs with the inline argument "X" might not align correctly in the "Settings/Inputs" tab.

`group`

Using the same `group` argument in any number of `input*()` calls groups the inputs in an *organized section* in the "Inputs" tab. The string used as the `group` argument becomes the section's heading. The `group` string is case-sensitive, so `input*()` calls must use the same characters and letter case in their `group` arguments to appear in the same section.

`display`

Controls whether the input value appears next to the script title in the status line and Data Window. It accepts the following values: display.all, display.status_line, display.data_window, or display.none. The default is display.all for all input types except "bool" and "color" inputs, which use display.none by default.

Note that the input value always appears in the "Inputs" tab, regardless of the `display` argument.

`active`

Controls whether users can change the input value in the "Inputs" tab; it is `true` by default. If `false`, the input field appears dimmed and users cannot change its value. This parameter accepts an "input bool" argument, so an input's `active` state can depend on the value of *other* inputs.

For example, if a script uses a "bool" `showAverageInput` toggle to show or hide an average line, we can use `active = showAverageInput` in other inputs related to the average, such as `averageLengthInput` or `averageColorInput`, to enable them only when users select the "Show average" checkbox.

`options`

A list specifying the possible values that this input can have. This parameter accepts a tuple, which is a comma-separated list of elements enclosed in square brackets (e.g., `["ON", "OFF"]`, `[1, 2, 3]`, `[myEnum.On,`

`myEnum.Off`]). These elements appear in a dropdown widget, from which users can select only one value at a time. If an input uses the `options` parameter, the `defval` value must be one of the list's elements.

`minval`

The minimum valid value for the input field in an integer input or float input.

`maxval`

The maximum valid value for the input field in an integer input or float input.

`step`

The increment by which the field's value changes when clicking the up/down arrows in an integer input or float input widget. The default `step` value is 1.

`confirm`

If `true`, the input widget appears in a "Confirm inputs" dialog box when users add the script to the chart, prompting them to configure the input value before the script executes. By default, this parameter's value is `false`. If more than one `input.*()` call uses `confirm = true` in the same script, multiple input widgets appear in the dialog box.

Using `confirm = true` for a time input or price input enables an interactive input mode where users can click on the chart to set time and price values.

The `minval`, `maxval`, and `step` parameters are only present in the second signatures of the input.int() and input.float() functions. Their first signatures use the `options` parameter instead. Function calls that use a `minval`, `maxval`, or `step` argument cannot also use an `options` argument.

Input types

The next sections explain what each input function does. As we proceed, we will explore the different ways you can use input functions and organize their display.

Generic input

input() is a simple, generic function that supports the fundamental Pine Script types: "int", "float", "bool", "color" and "string". It also supports "source" inputs, which are price-related values such as close, hl2, hlc3, and hlcc4, or which can be used to receive the output value of another script.

Its signature is:

```
input(defval, title, tooltip, inline, group, display, active) → input int/float
```

The function automatically detects the type of input by analyzing the type of the `defval` argument used in the function call. This script shows all the supported types and the qualified type returned by the function when used with `defval` arguments of different types:

//@version=6 indicator("`input()`", "", true) a = input(1, "input int") b = input(1.0, "input float") c = input(true, "input bool") d = input(color.orange, "input color") e = input("1", "input string") f = input(close, "series float") plot(na)

Integer input

Two signatures exist for the input.int() function; one when `options` is not used, the other when it is:

```
input.int(defval, title, minval, maxval, step, tooltip, inline, group, confirm,
input.int(defval, title, options, tooltip, inline, group, confirm, display, act
```

This call uses the `options` parameter to propose a pre-defined list of lengths for the MA:

//@version=6 indicator("MA", "", true) maLengthInput = input.int(10, options = [3, 5, 7, 10, 14, 20, 50, 100, 200]) ma = ta.sma(close, maLengthInput) plot(ma)

This one uses the `minval` parameter to limit the length:

//@version=6 indicator("MA", "", true) maLengthInput = input.int(10, minval = 2) ma = ta.sma(close, maLengthInput) plot(ma)

The version with the `options` list uses a dropdown menu for its widget. When the `options` parameter is not used, a simple input widget is used to enter the value:

Float input

Two signatures exist for the input.float() function; one when `options` is not used, the other when it is:

```
input.float(defval, title, minval, maxval, step, tooltip, inline, group, confir
input.float(defval, title, options, tooltip, inline, group, confirm, display, a
```

Here, we use a “float” input for the factor used to multiple the standard deviation, to calculate Bollinger Bands:

//@version=6 indicator("MA", "", true) maLengthInput = input.int(10, minval = 1) bbFactorInput = input.float(1.5, minval = 0, step = 0.5) ma = ta.sma(close, maLengthInput) bbWidth = ta.stdev(ma, maLengthInput) * bbFactorInput bbHi = ma + bbWidth bbLo = ma - bbWidth plot(ma) plot(bbHi, "BB Hi", color.gray) plot(bbLo, "BB Lo", color.gray)

The input widgets for floats are similar to the ones used for integer inputs:

Boolean input

Let's continue to develop our script further, this time by adding a boolean input to allow users to toggle the display of the BBs:

```
//@version=6 indicator("MA", "", true) maLengthInput = input.int(10, "MA length", minval = 1) bbFactorInput = input.float(1.5, "BB factor", inline = "01", minval = 0, step = 0.5) showBBInput = input.bool(true, "Show BB", inline = "01") ma = ta.sma(close, maLengthInput) bbWidth = ta.stdev(ma, maLengthInput) * bbFactorInput bbHi = ma + bbWidth bbLo = ma - bbWidth plot(ma, "MA", color.aqua) plot(showBBInput ? bbHi : na, "BB Hi", color.gray) plot(showBBInput ? bbLo : na, "BB Lo", color.gray)
```

Note that:

- We have added an input using input.bool() to set the value of `showBBInput`.
- We use the `inline` parameter in that input and in the one for `bbFactorInput` to bring them on the same line. We use `"01"` for its argument in both cases. That is how the Pine Script compiler recognizes that they belong on the same line. The particular string used as an argument is unimportant and does not appear anywhere in the “Inputs” tab; it is only used to identify which inputs go on the same line.
- We have vertically aligned the `title` arguments of our `input.*()` calls to make them easier to read.
- We use the `showBBInput` variable in our two plot() calls to plot conditionally. When the user unchecks the checkbox of the `showBBInput` input, the variable's value becomes `false`. When that happens, our plot() calls plot the na value, which displays nothing. We use `true` as the default value of the input, so the BBs plot by default.
- Because we use the `inline` parameter for the `bbFactorInput` variable, its input field in the “Inputs” tab does not align vertically with that of `maLengthInput`, which doesn't use `inline`.

Color input

As explained in this section of the Colors page, selecting the colors of a script's outputs via the “Settings/Style” tab is not always possible. In the case where one cannot choose colors from the “Style” tab, programmers can create color inputs with the input.color() function to allow color customization from the “Settings/Inputs” tab.

Suppose we wanted to plot our BBs with a lighter transparency when the high and low values are higher/lower than the BBs. We can use a code like this to create the colors:

```
bbHiColor = color.new(color.gray, high > bbHi ? 60 : 0) bbLoColor = color.new(color.gray, low < bbLo ? 60 : 0)
```

When using dynamic (“series”) color components like the `transp` arguments in the above code, the color widgets in the “Settings/Style” tab will no longer appear. Let's create our own input for color selection, which will appear in the “Settings/Inputs” tab:

```
//@version=6 indicator("MA", "", true) maLengthInput = input.int(10, "MA length", inline = "01", minval = 1) maColorInput = input.color(color.aqua, "", inline = "01") bbFactorInput = input.float(1.5, "BB factor", inline = "02", minval = 0, step = 0.5) bbColorInput = input.color(color.gray, "", inline = "02") showBBInput = input.bool(true, "Show BB", inline = "02") ma = ta.sma(close, maLengthInput) bbWidth = ta.stdev(ma, maLengthInput) * bbFactorInput bbHi = ma + bbWidth bbLo = ma - bbWidth bbHiColor = color.new(bbColorInput, high > bbHi ? 60 : 0) bbLoColor = color.new(bbColorInput, low < bbLo ? 60 : 0) plot(ma, "MA", maColorInput) plot(showBBInput ? bbHi : na, "BB Hi", bbHiColor, 2) plot(showBBInput ? bbLo : na, "BB Lo", bbLoColor, 2)
```

Note that:

- We have added two calls to input.color() to gather the values of the `maColorInput` and `bbColorInput` variables. We use `maColorInput` directly in the `plot(ma, "MA", maColorInput)` call, and we use `bbColorInput` to build the `bbHiColor` and `bbLoColor` variables, which modulate the transparency using the position of price relative to the BBs. We use a conditional value for the `transp` value we call color.new() with, to generate different transparencies of the same base color.
- We do not use a `title` argument for our new color inputs because they are on the same line as other inputs allowing users to understand to which plots they apply.
- We have reorganized our `inline` arguments so they reflect the fact we have inputs grouped on two distinct lines.

String input

The input.string() function creates a string input with either a single-line *text field* or a *dropdown menu* of predefined text options. Other `input.*()` functions also return "string" values. However, most of them are specialized for specific tasks, such as defining timeframes, symbols, and sessions.

If a call to the input.string() function includes an `options` argument, it creates a dropdown menu containing the listed options. Otherwise, the call creates a text field that parses user-input text into a "string" value.

Like the input.text_area() function, the input.string() text can contain up to 40,960 characters, including horizontal whitespaces. However, because the input's field in the "Settings/Inputs" tab is *narrow*, input.string() is best suited for defining small strings or for providing a quick set of input options for customizing calculations.

The simple script below contains two input.string() calls. The first call creates a text field for defining the `timezone` argument of two str.format_time() calls. It allows users to supply any text representing a time zone in *UTC-offset* or *IANA* formats. The second call creates a *dropdown* input with three preset options that determine the text shown in the drawn labels (`"Open time"`, `"Close time"`, or `"Both"`):

//@version=6 indicator("String input demo", overlay = true) //@variable A "string" specifying a UTC offset or IANA identifier for time zone specification. string timezoneInput = input.string("America/New_York", "Time zone") //@variable A "string" specifying whether the labels show opening times, closing times, or both. string displayModeInput = input.string("Both", "Display mode", ["Open time", "Close time", "Both"]) // Express the bar's `time` and `time_close` as formatted dates and times in the `timezoneInput` time zone. string openText = str.format_time(time, timezone = timezoneInput) string closeText = str.format_time(time_close, timezone = timezoneInput) //@variable A formatted "string" containing the `openText`, `closeText`, or both, based on the `displayModeInput`. string displayText = switch displayModeInput "Open time" => str.format("TZ: {0}\nOpen: {1}", timezoneInput, openText) "Close time" => str.format("TZ: {0}\nClose: {1}", timezoneInput, closeText) => str.format("TZ: {0}\nOpen: {1}\nClose: {2}", timezoneInput, openText, closeText) // Draw a label at the bar's `high` to show the `displayText`. label.new(bar_index, high, displayText)

Note that:

- An alternative way to provide a strict list of input options is to use an enum input, which constructs a dropdown menu based on the *members* of an enum type.
- In contrast to string declarations in code, the text field from a string input treats an input backslash (`\`) as a *literal character*. Therefore, the input.string() function *does not* parse input escape sequences such as `\n`.

Text area input

The input.text_area() function creates a text field for parsing user-specified text into a “string” value. The text field generated by this function is much larger than the field from input.string(). Additionally, it supports *multiline* text.

Programmers often use text area inputs for purposes such as alert customization and multi-parameter lists.

This example uses the value of a text area input to represent a comma-separated list of symbols. The script splits the parsed “string” value by its comma characters to construct an array of symbol substrings, then calls request.security() within a for…in loop on that array to dynamically retrieve the latest volume data for each specified symbol. On each loop iteration, the script converts the data to a “string” value with str.tostring() and displays the result in a table:

//@version=6 indicator("Text area input demo", overlay = true) //@variable A comma-separated list of symbol names with optional exchange prefixes. string symbolListInput = input.text_area("AAPL,GOOG,NVDA,MSFT", "Symbol list") //@variable An array of symbol substrings formed by splitting the `symbolListInput` by its commas. var array<string> symbols = str.split(symbolListInput, ",") if barstate.islast //@variable A table displaying requested volume data for each symbol in the `symbols` array. var table display = table.new(position.bottom_right, 2, symbols.size()) for [i, symbol] in symbols display.cell(0, i, symbol, text_color = chart.fg_color, text_size = 20) float vol = request.security(symbol, "", volume) display.cell(1, i, str.tostring(vol, format.volume), text_color = chart.fg_color, text_size = 20)

Note that:

- The script can use request.security() within a loop because dynamic requests are enabled by default.
- As with input.string(), the input.text_area() function’s text field treats backslashes (\) as literal characters. It cannot process escape sequences. However, the field automatically parses any line terminators and tab spaces in the specified text.
- Because text area inputs allow freeform, multiline text, it is often helpful to validate the input.text_area() function’s results to prevent erroneous user inputs. Refer to the Matching patterns section of the Strings page for an example that confirms an input symbol list using regular expressions.

Timeframe input

The input.timeframe() function creates a dropdown input containing *timeframe choices*. It returns a “string” value representing the selected timeframe in our specification format, which scripts can use in `request.*()` calls to retrieve data from user-selected timeframes.

The following script uses request.security() on each bar to fetch the value of a ta.sma() call from a user-specified higher timeframe, then plots the result on the chart:

//@version=6 indicator("Timeframe input demo", "MA", true) //@variable The timeframe of the requested data. string tfInput = input.timeframe("1D", "Timeframe") // Get the typical number of seconds in the chart's timeframe and the `tfInput` timeframe. int chartSeconds = timeframe.in_seconds() int tfSeconds = timeframe.in_seconds(tfInput) // Raise an error if the `tfInput` is a lower timeframe. if tfSeconds < chartSeconds runtime.error("The 'Timeframe' input must represent a timeframe higher than or equal to the chart's.") //@variable The offset of the requested expression. 1 when `tfInput` is a higher timeframe, 0 otherwise. int offset = chartSeconds == tfSeconds ? 0 : 1 //@variable The 20-bar SMA of `close` prices for the current symbol from the `tfInput` timeframe. float maHTF = request.security(syminfo.tickerid, tfInput,

ta.sma(close, 20)[offset], lookahead = barmerge.lookahead_on) // Plot the `maHTF` value. plot(maHTF, "MA", color.aqua)

Note that:

- By default, the input.timeframe() call's dropdown contains options for the chart's timeframe and all timeframes listed in the chart's "Time interval" menu. To restrict the available options to specific preset timeframes, pass a tuple of timeframe strings to the function's `options` parameter.
- This script calls runtime.error() to raise a custom runtime error if the timeframe.in_seconds() value for the `tfInput` timeframe is *less* than the number of seconds in the main timeframe, preventing it from requesting lower-timeframe data. See this section of the Other timeframes and data page to learn more.
- The request.security() call uses barmerge.lookahead_on as its `lookahead` argument, and it offsets the `expression` argument by one bar when the `tfInput` represents a *higher timeframe* to avoid repainting.

Symbol input

The input.symbol() function creates an input widget that mirrors the chart's "Symbol Search" widget. It returns a "string" *ticker identifier* representing the chosen symbol and exchange, which scripts can use in `request.*()` calls to retrieve data from other contexts.

The script below uses request.security() to retrieve the value of a ta.rsi() call evaluated on a user-specified symbol's prices. It plots the requested result on the chart in a separate pane:

//@version=6 indicator("Symbol input demo", "RSI") //@variable The ticker ID of the requested data. By default, it is an empty "string", which specifies the main symbol. string symbolInput = input.symbol("", "Symbol") //@variable The 14-bar RSI of `close` prices for the `symbolInput` symbol on the script's main timeframe. float symbolRSI = request.security(symbolInput, timeframe.period, ta.rsi(close, 14)) // Plot the `symbolRSI` value. plot(symbolRSI, "RSI", color.aqua)

Note that:

- The `defval` argument in the input.symbol() call is an empty "string". When the request.security() call in this example uses this default value as the `symbol` argument, it calculates the RSI using the *chart symbol's* data. If the user wants to revert to the chart's symbol after choosing another symbol, they can select "Reset settings" from the "Defaults" dropdown at the bottom of the "Settings" menu.

Session input

Session inputs are useful to gather start-stop values for periods of time. The input.session() built-in function creates an input widget allowing users to specify the beginning and end time of a session. Selections can be made using a dropdown menu, or by entering time values in "hh:mm" format.

The value returned by input.session() is a valid string in session format. See the manual's page on sessions for more information.

Session information can also contain information on the days where the session is valid. We use an input.string() function call here to input that day information:

//@version=6 indicator("Session input", "", true) string sessionInput = input.session("0600-1700", "Session") string daysInput = input.string("1234567", tooltip = "1 = Sunday, 7 = Saturday") sessionString = sessionInput

+ ":" + daysInput inSession = not na(time(timeframe.period, sessionString)) bgcolor(inSession ? color.silver : na)

Note that:

- This script proposes a default session of “0600-1700”.
- The input.string() call uses a tooltip to provide users with help on the format to use to enter day information.
- A complete session string is built by concatenating the two strings the script receives as inputs.
- We explicitly declare the type of our two inputs with the string keyword to make it clear those variables will contain a string.
- We detect if the chart bar is in the user-defined session by calling time() with the session string. If the current bar's time value (the time at the bar's open) is not in the session, time() returns na, so `inSession` will be `true` whenever time() returns a value that is not na.

Source input

Source inputs are useful to provide a selection of two types of sources:

- Price values, namely: open, high, low, close, hl2, hlc3, and ohlc4.
- The values plotted by other scripts on the chart. This can be useful to “link” two or more scripts together by sending the output of one as an input to another script.

This script simply plots the user's selection of source. We propose the high as the default value:

//@version=6 indicator("Source input", "", true) srcInput = input.source(high, "Source") plot(srcInput, "Src", color.new(color.purple, 70), 6)

This shows a chart where, in addition to our script, we have loaded an “Arnaud Legoux Moving Average” indicator. See here how we use our script's source input widget to select the output of the ALMA script as an input into our script. Because our script plots that source in a light-purple thick line, you see the plots from the two scripts overlap because they plot the same value:

Time input

The input.time() function creates a time input, which converts a user-specified date and time, in the chart’s time zone, into a time zone-agnostic UNIX timestamp. The timestamp represents the absolute number of *milliseconds* elapsed since 00:00:00 UTC on January 1, 1970. The input’s `defval` argument can be any “const int” value, including the value returned by the *single-argument* overload of the timestamp() function.

The input.time() function generates two fields: one for the *date* and the other for the *time of day*. Additionally, it adds a *vertical marker* to the chart. Users can change the input time either by moving this marker or by updating the value in the “Settings/Inputs” tab.

This simple script highlights the chart background for each bar whose opening time is past the date and time specified in a time input’s fields. This script defines the input.time() call’s default argument as the result of a timestamp() call that calculates the UNIX timestamp corresponding to December 27, 2024, at 09:30 in UTC-5:

//@version=6 indicator("Time input demo", overlay = true) //@variable A millisecond UNIX timestamp calculated from a specified date and time. // The input date and time are values in the chart's time zone, but

the resulting UNIX timestamp // is time zone-agnostic. int dateAndTimeInput = input.time(timestamp("27 Dec 2024 09:30 -0500"), "Date and time") //@variable Is `true` if the bar's opening time is beyond the input date and time; `false` otherwise. bool barIsLater = time > dateAndTimeInput // Highlight the background when `barIsLater` is `true`. bgcolor(barIsLater ? color.new(color.blue, 70) : na, title = "Bar opened later highlight")

Note that:

- The vertical line to the left of the background highlight is visible when selecting the script's status line or opening the "Settings" menu. Moving this line *changes* the input timestamp. Users can also change the time by choosing "Reset points" from the script's "More" menu and selecting a new point directly on the chart.
- Changing the time zone in the chart's settings can change the values shown in the input fields. However, the underlying UNIX timestamp does **not** change because it is unaffected by time zones.
- Users can *pair* time inputs with price inputs to create interactive chart points. See the next section to learn more.

Price input

The input.price() function creates a price input, which returns a specified floating-point value, similar to the input.float() function. Additionally, it adds a *horizontal marker* to the chart, allowing users to adjust the "float" value graphically, without opening the "Settings/Inputs" tab.

For example, this script calculates an RSI and plots the result with different colors based on the `thresholdInput` value. The plot is green if the RSI is above the value. Otherwise, it is red. Unlike a standard float input, users can set this script's input value by dragging the input's horizontal marker up or down on the chart:

//@version=6 indicator("Price input demo") //@variable The level at which the plot of the RSI changes color. // Users can adjust the value directly in the chart pane. float thresholdInput = input.price(50.0, "Threshold") //@variable The 14-bar RSI of `close` prices. float rsi = ta.rsi(close, 14) //@variable Is green if the `rsi` is above the `thresholdInput`; red otherwise. color rsiColor = rsi > thresholdInput ? color.green : color.red // Plot the `rsi` using the `rsiColor`. plot(rsi, "RSI", rsiColor, 3)

Programmers can also *pair* price inputs and time inputs to add *interactive points* for custom calculations or drawings. When a script creates pairs of time and price inputs that belong to the same group, and each pair has a unique, matching `inline` argument, it adds *point markers* on the chart instead of separate horizontal and vertical markers. Users can move these point markers to adjust input price and time values simultaneously.

This example creates four pairs of price and time inputs with distinct `inline` values. Each input includes `confirm = true`, meaning that users set the values when they add the script to a chart. The script prompts users to set four time-price points, then draws a closed polyline that passes through all the valid chart locations closest to the specified coordinates:

//@version=6 indicator("Price and time input demo", overlay = true) // Create price and time inputs with the same `inline` arguments to set them together on the chart. // Price and time for the first point. float price1Input = input.price(0, "Price 1", inline = "1", confirm = true) int time1Input = input.time(0, "Time 1", inline = "1", confirm = true) // Price and time for the second point. float price2Input = input.price(0, "Price 2", inline = "2", confirm = true) int time2Input = input.time(0, "Time 2", inline = "2", confirm = true) // Price and time for the third point. float price3Input = input.price(0, "Price 3", inline = "3", confirm = true) int time3Input =

input.time(0, "Time 3", inline = "3", confirm = true) // Price and time for the fourth point. float price4Input = input.price(0, "Price 4", inline = "4", confirm = true) int time4Input = input.time(0, "Time 4", inline = "4", confirm = true) //@variable An array of chart points created from the time and price inputs. var array<chart.point> points = array.from(chart.point.from_time(time1Input, price1Input), chart.point.from_time(time2Input, price2Input), chart.point.from_time(time3Input, price3Input), chart.point.from_time(time4Input, price4Input)) // Draw a closed, curved polyline connecting the points from the `points` array on the last bar. if barstate.islast var polyline shape = polyline.new(points, true, true, xloc.bar_time, color.purple, color.new(color.blue, 60))

Note that:

- Setting input times and prices together is possible only if there is exactly *one* input pair per `inline` value. If the inputs do not include `inline` arguments, or if more inputs have the same argument, the script sets times and prices separately.
- The script creates the drawing by constructing an array of chart points, then using that array in a polyline.new() call. Refer to the Polylines section of the Lines and boxes page to learn more about polyline drawings.

Enum input

The input.enum() function creates a dropdown input that displays *field titles* corresponding to distinct *members* (possible values) of an enum type. The function returns one of the unique, named values from a declared enum, which scripts can use in calculations and logic requiring more strict control over allowed values and operations. Supply a list of enum members to the `options` parameter to specify the members users can select from the dropdown. If one does not specify an enum field's title, its title is the “string” representation of its *name*.

This example declares a `SignalType` enum with four fields representing named signal display modes: `long`, `short`, `both`, and `none`. The script uses a member of this enum type as the `defval` argument in the input.enum() call to generate a dropdown in the “Inputs” tab, allowing users to select one of the enum's titles to control which signals it displays on the chart:

//@version=6 indicator("Enum input demo", overlay = true) //@enum An enumeration of named values representing signal display modes. //@field long Named value to specify that only long signals are allowed. //@field short Named value to specify that only short signals are allowed. //@field both Named value to specify that either signal type is allowed. //@field none Named value to specify that no signals are allowed. enum SignalType long = "Only long signals" short = "Only short signals" both = "Long and short signals" none //@variable An enumerator (member) of the `SignalType` enum. Controls the script's signals. SignalType sigInput = input.enum(SignalType.long, "Signal type") // Calculate moving averages. float ma1 = ta.sma(ohlc4, 10) float ma2 = ta.sma(ohlc4, 200) // Calculate cross signals. bool longCross = ta.crossover(close, math.max(ma1, ma2)) bool shortCross = ta.crossunder(close, math.min(ma1, ma2)) // Calculate long and short signals based on the selected `sigInput` value. bool longSignal = (sigInput == SignalType.long or sigInput == SignalType.both) and longCross bool shortSignal = (sigInput == SignalType.short or sigInput == SignalType.both) and shortCross // Plot shapes for the `longSignal` and `shortSignal`. plotshape(longSignal, "Long signal", shape.triangleup, location.belowbar, color.teal, size = size.normal) plotshape(shortSignal, "Short signal", shape.triangledown, location.abovebar, color.maroon, size = size.normal) // Plot the moving averages. plot(ma1, "Fast MA") plot(ma2, "Slow MA")

Note that:

- The `sigInput` value is the `SignalType` member whose field contains the selected title.
- Since we did not specify a title for the `none` field of the enum, its title is the “string” representation of its name (“none”), as we see in the above image of the enum input's dropdown.

By default, an enum input displays the titles of all an enum's members within its dropdown. If we supply an `options` argument to the input.enum() call, it will only allow users to select the members included in that list, e.g.:

SignalType sigInput = input.enum(SignalType.long, "Signal type", options = [SignalType.long, SignalType.short])

The above `options` argument specifies that users can only view and select the titles of the `long` and `short` fields from the `SignalType` enum. No other options are allowed:

Other features affecting inputs

Some parameters of the indicator() and strategy() functions populate a script's “Settings/Inputs” tab with additional inputs. These parameters are `timeframe`, `timeframe_gaps`, and `calc_bars_count`. For example:

//@version=6 indicator("MA", "", true, timeframe = "D", timeframe_gaps = false) plot(ta.vwma(close, 10))

Tips

The design of your script's inputs has an important impact on the usability of your scripts. Well-designed inputs are more intuitively usable and make for a better user experience:

- Choose clear and concise labels (your input's `title` argument).
- Choose your default values carefully.
- Provide `minval` and `maxval` values that will prevent your code from producing unexpected results, e.g., limit the minimal value of lengths to 1 or 2, depending on the type of MA you are using.
- Provide a `step` value that is congruent with the value you are capturing. Steps of 5 can be more useful on a 0-200 range, for example, or steps of 0.05 on a 0.0-1.0 scale.
- Group related inputs on the same line using `inline`; bull and bear colors for example, or the width and color of a line.
- When you have many inputs, group them into meaningful sections using `group`. Place the most important sections at the top.
- Do the same for individual inputs **within** sections.

It can be advantageous to vertically align different arguments of multiple `input.*()` calls in your code. When you need to make global changes, this will allow you to use the Editor's multi-cursor feature to operate on all the lines at once.

It is sometimes necessary to use Unicode spaces to achieve optimal alignment in inputs. This is an example:

//@version=6 indicator("Aligned inputs", "", true) var GRP1 = "Not aligned" ma1SourceInput = input(close, "MA source", inline = "11", group = GRP1) ma1LengthInput = input(close, "Length", inline = "11", group = GRP1) long1SourceInput = input(close, "Signal source", inline = "12", group = GRP1) long1LengthInput = input(close, "Length", inline = "12", group = GRP1) var GRP2 = "Aligned" // The three spaces after "MA source" are Unicode EN spaces (U+2002). ma2SourceInput = input(close, "MA source ", inline = "21",

group = GRP2) ma2LengthInput = input(close, "Length", inline = "21", group = GRP2) long2SourceInput = input(close, "Signal source", inline = "22", group = GRP2) long2LengthInput = input(close, "Length", inline = "22", group = GRP2) plot(ta.vwma(close, 10))

Note that:

- We use the `group` parameter to distinguish between the two sections of inputs. We use a constant to hold the name of the groups. This way, if we decide to change the name of the group, we only need to change it in one place.
- The first sections inputs widgets do not align vertically. We are using `inline`, which places the input widgets immediately to the right of the label. Because the labels for the `ma1SourceInput` and `long1SourceInput` inputs are of different lengths the labels are in different *y* positions.
- To make up for the misalignment, we pad the `title` argument in the `ma2SourceInput` line with three Unicode EN spaces (U+2002). Unicode spaces are necessary because ordinary spaces would be stripped from the label. You can achieve precise alignment by combining different quantities and types of Unicode spaces. See here for a list of Unicode spaces of different widths.

Libraries

Introduction

Pine Script® libraries are publications containing functions that can be reused in indicators, strategies, or in other libraries. They are useful to define frequently-used functions so their source code does not have to be included in every script where they are needed.

A library must be published (privately or publicly) before it can be used in another script. All libraries are published open-source. Public scripts can only use public libraries and they must be open-source. Private scripts or personal scripts saved in the Pine Script Editor can use public or private libraries. A library can use other libraries, or even previous versions of itself.

Library programmers should be familiar with Pine's typing nomenclature, scopes, and user-defined functions. For more information, see the User Manual's pages on the Type system and User-defined functions.

You can browse public library scripts in the Community Scripts feed.

Creating a library

A library is a special kind of script that begins with the library() declaration statement, rather than indicator() or strategy(). A library contains exportable function, method, UDT, and enum definitions, which constitute the only visible part of the library when imported by another script. Like other script types, libraries can also include Pine Script code in their global scopes. Programmers typically use a library's global code to demonstrate how other scripts can use its exported structures.

A library script has a structure like the following, which must include one or more exportable functions or types:

//@version=6 // @description <library_description> library(title, overlay) <script_code> //@type <type_description> //@field <field_name> <field_description> // ... export type <UDT_identifier> <field_type> <field_name>[= <value>] ... //@enum <enum_description> //@field <field_name> <field_description> // ... export enum <enum_name> <field_name>[= <field_title>] ... //@function <function_description> //@param <parameter> <parameter_description> //@returns <return_value_description> export <function_name> ([simple/series] <parameter_type> <parameter_name> [= <default_value>] [, ...]) => <function_code> <script_code>

Note that:

- The `//@description`, `//@enum`, `//@type`, `//@field`, `//@function`, `//@param`, and `//@returns` compiler annotations are optional but we highly recommend you use them. These annotations document the library's code and populate the default library description, which authors can use when publishing the library.
- The export keyword is mandatory.
- <parameter_type> is mandatory, contrary to user-defined function parameter definitions in indicators or strategies, which are typeless.
- <script_code> can be any code one would normally use in an indicator, including inputs.

This is an example library:

//@version=6 // @description Provides functions calculating the all-time high/low of values. library("AllTimeHighLow", true) // @function Calculates the all-time high of a series. // @param val Series to use (`high` is used if no argument is supplied). // @returns The all-time high for the series. export hi(float val = high) => var float ath = val ath := math.max(ath, val) // @function Calculates the all-time low of a series. // @param val Series to use (`low` is used if no argument is supplied). // @returns The all-time low for the series. export lo(float val = low) => var float atl = val atl := math.min(atl, val) plot(hi()) plot(lo())

Library functions

Exported functions and methods have slightly different requirements and constraints compared to non-exported functions.

In exported library function signatures (their first line):

- The export is mandatory.
- The function's signature must include type keywords to specify the required type for each parameter.
- Programmers can include either the simple or series qualifier keywords to specify the qualified type that each parameter accepts. See the next section for more information.

Exported library functions have the following constraints:

- They cannot use variables from the library's global scope except for those with the "const" qualifier, meaning they cannot use global variables initialized from script inputs, for example, or globally declared arrays.
- They cannot include calls to any `input.*()` functions.
- They *can* include `request.*()` calls in their local scopes (if the `dynamic_requests` parameter is not set to `false` in the library declaration statement), but the `expression` arguments of these calls cannot depend on any exported function parameters. See this section of the Other timeframes and data page to learn more about dynamic requests.
- They can contain alert() function calls, and a user can create an alert from a script that consumes such a function, but a user *cannot* create an alert directly from that library running on the chart.

Library functions always return "simple" or "series" results. They do not return values with *weaker* qualifiers. Consequently, scripts cannot use their returned values in locations requiring "const" or "input" values. For example, a library function cannot calculate an argument for the `show_last` parameter in a plot() call because the parameter requires an "input int" or "const int" value.

Qualified type control

Pine Script automatically determines the qualified types of the arguments in calls to library functions based on how the functions use those arguments. If a parameter is compatible with the "series" type qualifier, its arguments automatically inherit that qualifier by default. Otherwise, Pine attempts to evaluate the argument with the "simple" qualifier. For example, consider the following exported function:

export myEma(int x) => ta.ema(close, length = x)

The `length` parameter of the ta.ema() function used in our function's scope has the expected type "simple int". The parameter can accept "simple int", "const int" or "input int" values, but **not** "series int" values. Therefore, the Pine Script compiler automatically detects that the `x` parameter's qualified type is "simple int".

This behavior explains why a call such as `myCustomLibrary.myEma(x = 20)` compiles successfully, but a call such as `myCustomLibrary.myEma(x = bar_index)` causes a *compilation error*. A literal value of 20 is of the type “const int”, meaning the script can convert it to a “simple int” argument. In contrast, bar_index has the type “series int”, and “series” arguments **cannot** inherit *weaker* qualifiers such as “simple”.

While library functions cannot return “const” or “input” values, they can be written to produce “simple” results. This makes them useful in more contexts than functions returning “series” results, as some built-in functions do not allow “series” arguments. For example, request.security() requires a “simple string” argument for its `symbol` parameter when a script does not allow dynamic requests. If we wrote a library function to assemble the argument to `symbol` in the following way, the function's result would not work with a non-dynamic `request.*()` call because it is of the “series string” qualified type:

export makeTickerid(string prefix, string ticker) => prefix + ":" + ticker

However, by restricting the parameter qualifiers to “simple”, we can force the function to yield a “simple” result. We can achieve this by prefixing the parameters' type with the simple keyword:

export makeTickerid(simple string prefix, simple string ticker) => prefix + ":" + ticker

Note that for the function to return a “simple” value, no “series” values can be used in its calculation; otherwise the result will be a “series” value.

One can also use the series keyword to prefix the type of a library function parameter. However, because library function parameters are qualified as “series” by default, using the keyword to specify the qualifier is often unnecessary.

> NoteInstances of reference types automatically inherit the “series” qualifier, regardless of how a script uses them. Therefore, libraries **cannot** create exported functions that rely on reference types and return “simple” values, because all calculations involving “series” data always produce “series” results. See the Type system page to learn more.

User-defined types and objects

Libraries can export user-defined types (UDTs), and library functions can return the *references (IDs)* to objects of these types.

To export a UDT, prefix its definition with the export keyword, similar to exporting a function:

//@version=6 library("Point") export type point int x float y bool isHi bool wasBreached = false

Another script can import this library to create objects of the `point` UDT. For example, the following script imports the hypothetical “Point” library published by the `userName` user, then uses the `point.new()` function to create a new `point` instance on each bar:

//@version=6 indicator("") import userName/Point/1 as pt pt.point newPoint = pt.point.new()

Note that:

- This example **does not** compile, because there is no “Point” library published by a `userName` user. This example simply demonstrates how another script might use the hypothetical library.

- This example script uses the alias `pt` in the import statement. It then uses this alias as the *namespace* to access the `point.new()` function and the `point` type from our “Point” library.
- The script uses `pt.point` as the *type keyword* for the `newPoint` variable declaration. Using this keyword is optional, because `newPoint` has a defined reference. However, declaring the type explicitly using this keyword is *required* if we change the variable’s initial reference to na.

A library is **required** to export a UDT if any exported functions or methods accept or return references of that type or the IDs of collections that store references of the type. Likewise, a library must export a UDT if at least one field of *another* exported UDT uses the type.

By contrast, if a library uses objects of a UDT only for *internal* calculations, it *does not* need to export the type. For example, the following library uses the custom `point` type in its only exported function (`drawPivots()`). However, that function does not have a parameter of the `point` type, and it does not return a reference to any object of the type. Therefore, the `point` type does not require the export keyword in its declaration:

//@version=6 library("PivotLabels", true) // The library uses this UDT in the exported `drawPivots()` function, but exporting the type is *not* required, because: // - The exported function's *parameters* do not accept this type. // - The exported function does not *return* a reference of this type. // - The library does not export any other function that uses the type. type point int x float y bool isHi bool wasBreached = false fillPivotsArray(qtyLabels, leftLegs, rightLegs) => // Create an array of the specified qty of pivots to maintain. var pivotsArray = array.new<point>(math.max(qtyLabels, 0)) // Detect pivots. float pivotHi = ta.pivothigh(leftLegs, rightLegs) float pivotLo = ta.pivotlow(leftLegs, rightLegs) // Create a new `point` object when a pivot is found. point foundPoint = switch not na(pivotHi) => point.new(time[rightLegs], pivotHi, true) not na(pivotLo) => point.new(time[rightLegs], pivotLo, false) => na // Add new pivot info to the array and remove the oldest pivot. if not na(foundPoint) array.push(pivotsArray, foundPoint) array.shift(pivotsArray) array<point> result = pivotsArray detectBreaches(pivotsArray) => // Detect breaches. for [i, eachPoint] in pivotsArray if not na(eachPoint) if not eachPoint.wasBreached bool hiWasBreached = eachPoint.isHi and high[1] <= eachPoint.y and high > eachPoint.y bool loWasBreached = not eachPoint.isHi and low[1] >= eachPoint.y and low < eachPoint.y if hiWasBreached or loWasBreached // This pivot was breached; change its `wasBreached` field. point p = array.get(pivotsArray, i) p.wasBreached := true array.set(pivotsArray, i, p) drawLabels(pivotsArray) => for eachPoint in pivotsArray if not na(eachPoint) label.new(eachPoint.x, eachPoint.y, str.tostring(eachPoint.y, format.mintick), xloc.bar_time, color = eachPoint.wasBreached ? color.gray : eachPoint.isHi ? color.teal : color.red, style = eachPoint.isHi ? label.style_label_down: label.style_label_up, textcolor = eachPoint.wasBreached ? color.silver : color.white) // @function Displays a label for each of the last `qtyLabels` pivots. // Colors high pivots in green, low pivots in red, and breached pivots in gray. // @param qtyLabels (simple int) Quantity of last labels to display. // @param leftLegs (simple int) Left pivot legs. // @param rightLegs (simple int) Right pivot legs. export drawPivots(int qtyLabels, int leftLegs, int rightLegs) => // Gather pivots as they occur. pointsArray = fillPivotsArray(qtyLabels, leftLegs, rightLegs) // Mark breached pivots. detectBreaches(pointsArray) // Draw labels once. if barstate.islastconfirmedhistory drawLabels(pointsArray) // Example use of the function. drawPivots(20, 10, 5)

If the TradingView user published the above library, it could be used like this:

//@version=6 indicator("") import TradingView/PivotLabels/1 as dpl dpl.drawPivots(20, 10, 10)

Enum types

Libraries can also export enum types, allowing other scripts to import sets of predefined, named constants that help control the values accepted by variables, conditional expressions, and collections.

For example, this library exports a `State` enum with three fields representing distinct signal states: `long`, `short`, and `neutral`. These fields represent the *possible values* a variable, expression, or collection of the enum type can take on:

//@version=6 library("Signal") //@enum An enumeration of named signal states. //@field long Represents a "Long" signal. //@field short Represents a "Short" signal. //@field neutral Represents a "Neutral" signal. export enum State long = "Long" short = "Short" neutral = "Neutral"

A script that imports this library can use the members (values) of the `State` enum as named states in its logic. Here, we show a simple, hypothetical script that imports the “Signal” library published by the `userName` user and uses the `Signal.State` enum to assign one of three possible values to a `mySignal` variable:

//@version=6 indicator("") import userName/Signal/1 as Signal // Calculate the median and quarter range values. float medianValue = ta.median(close, 100) float rangeValue = ta.range(close, 100) * 0.25 // Calculate upper and lower channel values. float upper = medianValue + rangeValue float lower = medianValue - rangeValue //@variable Returns `Signal.State.long`, `Signal.State.short`, or `Signal.State.neutral` based on the price action. Signal.State mySignal = switch close > upper => Signal.State.long close < lower => Signal.State.short => Signal.State.neutral plot(close, color = mySignal == Signal.State.long ? color.green : mySignal == Signal.State.short ? color.red : na)

Similar to exporting UDTs, a library **must** export an enum when its exported functions or methods accept or return the enum’s members, or when the fields of an exported UDT accept values of that enum type.

Publishing a library

Before you or other Pine Script programmers can reuse any library, it must be published. If you want to share your library with all TradingViewers, publish it publicly. To use it privately, use a private publication. As with indicators or strategies, the active chart when you publish a library will appear in both its widget (the small placeholder denoting libraries in the TradingView scripts stream) and script page (the page users see when they click on the widget).

Private libraries can be used in public Protected or Invite-only scripts.

After adding our example library to the chart and setting up a clean chart showing our library plots the way we want them, we use the Pine Editor's “Publish Script” button. The “Publish Library” window comes up:

Note that:

- We leave the library's title as is (the `title` argument in our library() declaration statement is used as the default). While you can change the publication's title, it is preferable to keep its default value because the `title` argument is used to reference imported libraries in the import statement. It makes life easier for library users when your publication's title matches the actual name of the library.
- A default description is built from the compiler annotations we used in our library. We will publish the library wihout retouching it.
- We chose to publish our library publicly, so it will be visible to all TradingViewers.
- We do not have the possibility of selecting a visibility type other than “Open” because libraries are always open-source.

- The list of categories for libraries is different than for indicators and strategies. We have selected the “Statistics and Metrics” category.
- We have added some custom tags: “all-time”, “high” and “low”.

The intended users of public libraries being other Pine programmers; the better you explain and document your library's functions, the more chances others will use them. Providing examples demonstrating how to use your library's functions in your publication's code will also help.

House Rules

Pine libraries are considered “public domain” code in our House Rules on Script Publishing, which entails that permission is not required from their author if you call their functions or reuse their code in your open-source scripts. However, if you intend to reuse code from a Pine Script library's functions in a public protected or invite-only publication, explicit permission for reuse in that form is required from its author.

Whether using a library's functions or reusing its code, you must credit the author in your publication's description. It is also good form to credit in open-source comments.

Using a library

Using a library from another script (which can be an indicator, a strategy or another library), is done through the import statement:

```
import <username>/<libraryName>/<libraryVersion> [as <alias>]
```

where:

- The <username>/<libraryName>/<libraryVersion> path will uniquely identify the library.
- The <libraryVersion> must be specified explicitly. To ensure the reliability of scripts using libraries, there is no way to automatically use the latest version of a library. Every time a library update is published by its author, the library's version number increases. If you intend to use the latest version of the library, the <libraryVersion> value will require updating in the import statement.
- The `as <alias>` part is optional. When used, it defines the namespace that will refer to the library's functions. For example, if you import a library using the `allTime` alias as we do in the example below, you will refer to that library's functions as `allTime.<function_mame>()`. When no alias is defined, the library's name becomes its namespace.

To use the library we published in the previous section, our next script will require an import statement:

import PineCoders/AllTimeHighLow/1 as allTime

As you type the user name of the library's author, you can use the Editor's `ctrl` + `space` / `cmd` “Auto-complete” command to display a popup providing selections that match the available libraries:

This is an indicator that reuses our library:

//@version=6 indicator("Using AllTimeHighLow library", "", true) import PineCoders/AllTimeHighLow/1 as allTime plot(allTime.hi()) plot(allTime.lo()) plot(allTime.hi(close))

Note that:

- We have chosen to use the “allTime” alias for the library's instance in our script. When typing that alias in the Editor, a popup will appear to help you select the particular function you want to use from the library.
- We use the library's `hi()` and `lo()` functions without an argument, so the default high and low built-in variables will be used for their series, respectively.
- We use a second call to `allTime.hi()`, but this time using close as its argument, to plot the highest close in the chart's history.

Non-standard charts data

Introduction

Pine Script® features several `ticker.*()` functions that generate *ticker identifiers* for requesting data from *non-standard* chart feeds. The available functions that create these ticker IDs are ticker.heikinashi(), ticker.renko(), ticker.linebreak(), ticker.kagi(), and ticker.pointfigure(). Scripts can use these functions' returned values as the `symbol` argument in request.security() calls to access non-standard chart data while running on *any* chart type.

> NoteThe Renko, Line Break, Kagi, and Point & Figure chart types construct bars using price data from *lower timeframes*. Therefore, the bars on these chart types only *approximate* the values that correspond to calculations using tick data.

`ticker.heikinashi()`

Heikin-Ashi means *average bar* in Japanese. The open/high/low/close values of Heikin-Ashi candlesticks are synthetic; they are not actual market prices. They are calculated by averaging combinations of real OHLC values from the current and previous bar. The calculations used make Heikin-Ashi bars less noisy than normal candlesticks. They can be useful to make visual assessments, but are unsuited to backtesting or automated trading, as orders execute on market prices — not Heikin-Ashi prices.

The ticker.heikinashi() function creates a special ticker identifier for requesting Heikin-Ashi data with the request.security() function.

This script requests the close value of Heikin-Ashi bars and plots them on top of the normal candlesticks:

//@version=6 indicator("HA Close", "", true) haTicker = ticker.heikinashi(syminfo.tickerid) haClose = request.security(haTicker, timeframe.period, close) plot(haClose, "HA Close", color.black, 3)

Note that:

- The close values for Heikin-Ashi bars plotted as the black line are very different from those of real candles using market prices. They act more like a moving average.
- The black line appears over the chart bars because we have selected "Visual Order/Bring to Front" from the script's "More" menu.

If you wanted to omit values for extended hours in the last example, an intermediary ticker without extended session information would need to be created first:

//@version=6 indicator("HA Close", "", true) regularSessionTicker = ticker.new(syminfo.prefix, syminfo.ticker, session.regular) haTicker = ticker.heikinashi(regularSessionTicker) haClose = request.security(haTicker, timeframe.period, close, gaps = barmerge.gaps_on) plot(haClose, "HA Close", color.black, 3, plot.style_linebr)

Note that:

- We use the ticker.new() function first, to create a ticker without extended session information.

- We use that ticker instead of syminfo.tickerid in our ticker.heikinashi() call.
- In our request.security() call, we set the `gaps` parameter's value to `barmerge.gaps_on`. This instructs the function not to use previous values to fill slots where data is absent. This makes it possible for it to return na values outside of regular sessions.
- To be able to see this on the chart, we also need to use a special `plot.style_linebr` style, which breaks the plots on na values.

This script plots Heikin-Ashi candles under the chart:

//@version=6 indicator("Heikin-Ashi candles") CANDLE_GREEN = #26A69A CANDLE_RED = #EF5350 haTicker = ticker.heikinashi(syminfo.tickerid) [haO, haH, haL, haC] = request.security(haTicker, timeframe.period, [open, high, low, close]) candleColor = haC >= haO ? CANDLE_GREEN : CANDLE_RED plotcandle(haO, haH, haL, haC, color = candleColor)

Note that:

- We use a tuple with request.security() to fetch four values with the same call.
- We use plotcandle() to plot our candles. See the Bar plotting page for more information.

`ticker.renko()`

Renko bars only plot price movements, without taking time or volume into consideration. They look like bricks stacked in adjacent columns. A new brick is only drawn after the price passes the top or bottom by a predetermined amount. The ticker.renko() function creates a ticker id which can be used with request.security() to fetch Renko values, but there is no Pine Script function to draw Renko bars on the chart:

//@version=6 indicator("", "", true) renkoTicker = ticker.renko(syminfo.tickerid, "ATR", 10) renkoLow = request.security(renkoTicker, timeframe.period, low) plot(renkoLow)

`ticker.linebreak()`

The *Line Break* chart type displays a series of vertical boxes that are based on price changes. The ticker.linebreak() function creates a ticker id which can be used with request.security() to fetch “Line Break” values, but there is no Pine Script function to draw such bars on the chart:

//@version=6 indicator("", "", true) lineBreakTicker = ticker.linebreak(syminfo.tickerid, 3) lineBreakClose = request.security(lineBreakTicker, timeframe.period, close) plot(lineBreakClose)

`ticker.kagi()`

Kagi charts are made of a continuous line that changes directions. The direction changes when the price changes beyond a predetermined amount. The ticker.kagi() function creates a ticker id which can be used with request.security() to fetch “Kagi” values, but there is no Pine Script function to draw such bars on the chart:

//@version=6 indicator("", "", true) kagiBreakTicker = ticker.linebreak(syminfo.tickerid, 3) kagiBreakClose = request.security(kagiBreakTicker, timeframe.period, close) plot(kagiBreakClose)

`ticker.pointfigure()`

Point and Figure (PnF) charts only plot price movements, without taking time into consideration. A column of X's is plotted as the price rises, and O's are plotted when price drops. The ticker.pointfigure() function creates a ticker id which can be used with request.security() to fetch “PnF” values, but there is no Pine Script function to draw such bars on the chart. Every column of X's or O's is represented with four numbers. You may think of them as synthetic OHLC PnF values:

```
//@version=6 indicator("", "", true) pnfTicker = ticker.pointfigure(syminfo.tickerid, "hl", "ATR", 14, 3) [pnfO, pnfC] = request.security(pnfTicker, timeframe.period, [open, close], barmerge.gaps_on) plot(pnfO, "PnF Open", color.green, 4, plot.style_linebr) plot(pnfC, "PnF Close", color.red, 4, plot.style_linebr)
```

Other timeframes and data

Introduction

Pine Script® allows users to request data from sources and contexts other than those their charts use. The functions we present on this page can fetch data from a variety of alternative sources:

- request.security() retrieves data from another symbol, timeframe, or other context.
- request.security_lower_tf() retrieves *intrabar* data, i.e., data from a timeframe lower than the chart timeframe.
- request.currency_rate() requests a *daily rate* to convert a value expressed in one currency to another.
- request.dividends(), request.splits(), and request.earnings() respectively retrieve information about an issuing company's dividends, splits, and earnings.
- request.financial() retrieves financial data from FactSet.
- request.economic() retrieves economic and industry data.
- request.footprint() retrieves *volume footprint* data.
- request.seed() retrieves data from a *user-maintained* GitHub repository.

> NoteThroughout this page, and in other parts of our documentation that discuss `request.*()` functions, we often use the term *"context"* to describe the symbol, timeframe, and any modifications – such as price adjustments, session settings, and non-standard chart types – that apply to a chart or the data retrieved by a script.

These are the signatures of the functions in the `request.*` namespace:

```
request.security(symbol, timeframe, expression, gaps, lookahead, ignore_invalid

request.security_lower_tf(symbol, timeframe, expression, ignore_invalid_symbol,

request.currency_rate(from, to, ignore_invalid_currency) → series float

request.dividends(ticker, field, gaps, lookahead, ignore_invalid_symbol, curren

request.splits(ticker, field, gaps, lookahead, ignore_invalid_symbol) → series

request.earnings(ticker, field, gaps, lookahead, ignore_invalid_symbol, currenc

request.financial(symbol, financial_id, period, gaps, ignore_invalid_symbol, cu

request.economic(country_code, field, gaps, ignore_invalid_symbol) → series flo

request.footprint(ticks_per_row, va_percent, imbalance_percent) → series footpr
```

```
request.seed(source, symbol, expression, ignore_invalid_symbol, calc_bars_count
```

The `request.*()` family of functions has numerous potential applications. Throughout this page, we discuss in detail these functions and some of their typical use cases.

> TipProgrammers can also enable compatible scripts to perform calculations on data from another timeframe, without requiring `request.*()` calls, by supplying an argument to the `timeframe` parameter of the indicator() declaration statement.

Common characteristics

Many functions in the `request.*()` namespace share some common properties and parameters. Before we explore each function in depth, let's familiarize ourselves with these characteristics.

Behavior

All `request.*()` functions have similar internal behavior, even though they do not all share the same required parameters. Every unique `request.*()` call in a script requests a dataset from a defined *context* (i.e., ticker ID and timeframe) and evaluates an *expression* across the retrieved data.

The request.security() and request.security_lower_tf() functions allow programmers to specify the context of a request and the expression directly via the `symbol`, `timeframe`, and `expression` parameters, making them suitable for a wide range of data requests.

For example, the request.security() call in this simple script requests daily "AMEX:SPY" data, and it calculates the slope of a 20-bar linear regression line using the retrieved hl2 prices. The first two arguments specify the context of the request, and the third specifies the expression to evaluate across the requested data:

//@version=6 indicator("Behavior of `request.security()` demo") //@variable The 20-bar linear regression slope of `hl2` prices from the "AMEX:SPY" symbol on the "1D" timeframe. float requestedSlope = request.security("AMEX:SPY", "1D", ta.linreg(hl2, 20, 0) - ta.linreg(hl2, 20, 1)) //@variable Is `color.teal` when the `requestedSlope` is positive, and `color.maroon` otherwise. color plotColor = requestedSlope > 0 ? color.teal : color.maroon // Plot the `requestedSlope` with the `plotColor`. plot(requestedSlope, "Requested slope", plotColor, 1, plot.style_area)

Other functions within the `request.*()` namespace *do not* allow programmers to directly define the full context of a request or the evaluated expression. Instead, these functions determine some of the necessary information *internally* because they perform only specific types of requests.

For instance, request.financial() exclusively retrieves periodic financial data. Its required parameters (`symbol`, `financial_id`, and `period`) all define parts of a specific financial *ticker ID*. The function does not allow specification of the timeframe or expression, as it determines these details internally. The script below demonstrates a simple call to this function that retrieves the annual cost of goods data for the chart symbol's issuing company:

//@version=6 indicator("Behavior of `request.financial()` demo", format = format.volume) //@variable The annual cost of goods sold by the chart symbol's issuing company. float costOfGoods = request.financial(syminfo.tickerid, "COST_OF_GOODS", "FY") // Plot the `costOfGoods`. plot(costOfGoods, "Cost of goods", color.purple, 3, plot.style_stepline_diamond)

Scripts can perform up to 40 unique requests using any combination of `request.*()` function calls, or up to 64 if the user has the Ultimate plan. Unique `request.*()` calls count toward this limit because they are the only calls that fetch *new data*. By contrast, redundant calls to the same `request.*()` function with identical arguments *do not* typically perform new requests. Instead, they *reuse* the data from the first executed call. See the `request.*()` calls section of the Limitations page for more information.

`gaps`

When using a `request.*()` function to retrieve data from another context, the data might not come in on each new bar as it would with the current chart. The `gaps` parameter of a `request.*()` function controls how the function responds to nonexistent values in the requested series.

> NoteThe `timeframe_gaps` parameter of the indicator() declaration statement is similar to the `gaps` parameter for `request.*()` functions. When the declaration statement includes a `timeframe` argument, causing the script to evaluate its code using data from a specific timeframe, the `timeframe_gaps` parameter specifies how the script handles nonexistent values on each chart bar.

Suppose we have a script that requests hourly data for the chart's symbol using request.security() executing on a 1-minute chart. The function call returns new values only on the 1-minute bars that cover the opening or closing times of the symbol's hourly bars. On other chart bars, we can decide whether the function returns na values or the last available values via the `gaps` parameter.

If the `gaps` parameter uses barmerge.gaps_on, the function returns na results on all chart bars where new data is not yet confirmed from the requested context. Otherwise, if the parameter uses barmerge.gaps_off, the function fills the gaps in the requested data with the last confirmed values on historical bars and the most recent developing values on realtime bars.

The script below demonstrates the difference in behavior by plotting the results from two request.security() calls that fetch the close price of the current symbol from the hourly timeframe on a 1-minute chart. The first call uses `gaps = barmerge.gaps_off` and the second uses `gaps = barmerge.gaps_on`:

//@version=6 indicator("`gaps` demo", overlay = true) //@variable The `close` requested from the hourly timeframe without gaps. float dataWithoutGaps = request.security(syminfo.tickerid, "60", close, gaps = barmerge.gaps_off) //@variable The `close` requested from the hourly timeframe with gaps. float dataWithGaps = request.security(syminfo.tickerid, "60", close, gaps = barmerge.gaps_on) // Plot the requested data. plot(dataWithoutGaps, "Data without gaps", color.blue, 3, plot.style_linebr) plot(dataWithGaps, "Data with gaps", color.purple, 15, plot.style_linebr) // Highlight the background for realtime bars. bgcolor(barstate.isrealtime ? color.new(color.aqua, 70) : na, title = "Realtime bar highlight")

Note that:

- barmerge.gaps_off is the default value for the `gaps` parameter in all applicable `request.*()` functions.

- The script plots the requested series as lines with breaks (plot.style_linebr), which do not bridge over na values as the default style (plot.style_line) does.
- When using barmerge.gaps_off, the request.security() function returns the last confirmed close from the hourly timeframe on all historical bars. When running on *realtime bars* (the bars with the color.aqua background in this example), it returns the symbol's current close value, regardless of confirmation. For more information, see the Historical and realtime behavior section of this page.

`ignore_invalid_symbol`

The `ignore_invalid_symbol` parameter of `request.*()` functions determines how a function handles invalid data requests, e.g.:

- Using a `request.*()` function with a nonexistent ticker ID as the `symbol/ticker` parameter.
- Using request.financial() to retrieve information that does not exist for the specified `symbol` or `period`.
- Using request.economic() to request a `field` that does not exist for a `country_code`.

A `request.*()` function call produces a *runtime error* and halts the execution of the script when making an erroneous request if its `ignore_invalid_symbol` parameter is `false`. When this parameter's value is `true`, the function returns na values in such a case instead of raising an error.

This example uses `request.*()` calls within a user-defined function to retrieve data for estimating an instrument's market capitalization (market cap). The user-defined `calcMarketCap()` function calls request.financial() to retrieve the total shares outstanding for a symbol and request.security() to retrieve a tuple containing the symbol's close and syminfo.currency values. We've included `ignore_invalid_symbol = true` in both of these `request.*()` calls to prevent runtime errors for invalid requests.

The script displays a formatted string representing the symbol's estimated market cap value and currency in a table on the chart and uses a plot() call to visualize the `marketCap` history:

//@version=6 indicator("`ignore_invalid_symbol` demo", "Market cap estimate", format = format.volume) //@variable The symbol to request data from. string symbol = input.symbol("TSX:SHOP", "Symbol") //@function Estimates the market capitalization of the specified `tickerID` if the data exists. calcMarketCap(simple string tickerID) => //@variable The quarterly total shares outstanding for the `tickerID`. Returns `na` when the data isn't available. float tso = request.financial(tickerID, "TOTAL_SHARES_OUTSTANDING", "FQ", ignore_invalid_symbol = true) //@variable The `close` price and currency for the `tickerID`. Returns `[na, na]` when the `tickerID` is invalid. [price, currency] = request.security(tickerID, timeframe.period, [close, syminfo.currency], ignore_invalid_symbol = true) // Return a tuple containing the market cap estimate and the quote currency. [tso * price, currency] //@variable A `table` object with a single cell that displays the `marketCap` and `quoteCurrency`. var table infoTable = table.new(position.top_right, 1, 1) // Initialize the table's cell on the first bar. if barstate.isfirst table.cell(infoTable, 0, 0, "", text_color = color.white, text_size = size.huge, bgcolor = color.teal) // Get the market cap estimate and quote currency for the `symbol`. [marketCap, quoteCurrency] = calcMarketCap(symbol) if barstate.islast //@variable The formatted text displayed inside the `infoTable`. string tableText = str.format("Market cap:\n{0} {1}", str.tostring(marketCap, format.volume), quoteCurrency) // Update the `infoTable`. table.cell_set_text(infoTable, 0, 0, tableText) // Plot the `marketCap` value. plot(marketCap, "Market cap", color.new(color.purple, 60), style = plot.style_area)

Note that:

- The `calcMarketCap()` function only returns non-na values on valid instruments with total shares outstanding data, such as the one we selected for this example. It returns na on others that do not have financial data, including forex, crypto, and derivatives.
- Not all issuing companies publish quarterly financial reports. If the issuing company of the `symbol` does not report on a quarterly basis, change the "FQ" value in this script to the company's minimum reporting period. See the request.financial() section for more information.
- We included format.volume in the indicator() and str.tostring() calls to specify that the y-axis of the chart pane represents volume-formatted values and the "string" representation of the `marketCap` value shows as volume-formatted text.
- For efficiency, this script creates a table and initializes its cell on the *first* chart bar, then updates the cell's text on the *last* bar. To learn more about working with tables, see the Tables page.

`currency`

The `currency` parameter of a `request.*()` function enables programmers to specify the currency of the requested data. If this parameter's value differs from the symbol's syminfo.currency value, the function converts the requested values to express them in the specified currency. The `currency` parameter accepts a built-in constant from the `currency.*` namespace, such as currency.JPY, or a string representing a valid currency code (e.g., "JPY"). By default, this parameter accepts a "series" argument that can change across executions. However, if dynamic requests are not enabled, it accepts only a value with the "simple" qualifier or a weaker one.

The conversion rate between the syminfo.currency of the requested data and the specified `currency` depends on the *previous daily value* of the corresponding currency pair from the most popular exchange. If no exchange provides the rate directly, the function derives the rate using a spread symbol.

> NoteNot all `request.*()` function calls return values expressed as a currency amount. Therefore, currency conversion is *not* always necessary. For example, some of the series that the request.financial() function can retrieve – such as the "PIOTROSKI_F_SCORE" and "NUMBER_OF_EMPLOYEES" metrics – use units other than currency. It is up to programmers to determine when currency conversion is appropriate for their data requests.

`lookahead`

The `lookahead` parameter in request.security(), request.dividends(), request.splits(), and request.earnings() specifies the lookahead behavior of the function call. Its default value is barmerge.lookahead_off.

When requesting data from a higher-timeframe (HTF) context, the `lookahead` value determines whether the `request.*()` function can return values from times *beyond* those of the historical bars it executes on. In other words, the `lookahead` paremeter determines whether the requested data may contain *lookahead bias* on historical bars.

When requesting data from a lower-timeframe (LTF) context, the `lookahead` parameter determines whether the function requests values from the first or last *intrabar* (LTF bar) of each chart-timeframe bar.

Programmers should exercise extreme caution when using lookahead in their requests, especially when requesting data from higher timeframes. When using barmerge.lookahead_on as the `lookahead`

value, ensure that it does not compromise the integrity of the script's logic by leaking *future data* into historical chart bars.

The following scenarios are cases where enabling lookahead is acceptable in a `request.*()` call:

- The `expression` argument in a request.security() call includes a *historical offset* (e.g., `close[1]`), which prevents the function from requesting future values that it would **not** have access to on a realtime basis.
- The `timeframe` argument of the call represents the same timeframe as that of the chart on which the script executes, i.e., timeframe.period.
- The function call requests data from an intrabar timeframe, i.e., a timeframe smaller than the timeframe.period. See the Lower-timeframes section for more information.

> NoticeScripts that use request.security() calls with lookahead to leak future data into the past are extremely **misleading**. As such, they are **not allowed** as script publications. Although the results of such a script might look great across history due to its apparent aquisition of prescience, those results are *unrealistic* because the retrieved data was not knowable at the time of each bar. Furthermore, the same behavior is *impossible* to reproduce on realtime bars. Therefore, before publishing a script to share it with others, ensure that its requests **do not** mislead traders by using future data on historical bars.

This example demonstrates how the `lookahead` parameter affects the behavior of higher-timeframe data requests and why enabling lookahead in request.security() without offsetting the `expression` is misleading. The script calls request.security() to get the HTF high price for the current chart's symbol in three different ways and plots the resulting series on the chart for comparison.

The first call uses barmerge.lookahead_off (default), and the others use barmerge.lookahead_on. However, the third request.security() call also *offsets* its `expression` using the history-referencing operator [] to avoid leaking future data into the past.

As we see on the chart, the plot of the series requested using barmerge.lookahead_on without an offset (fuchsia line) shows final HTF high prices *before* they are actually available on historical bars, whereas the other two calls do not:

//@version=6 indicator("`lookahead` demo", overlay = true) //@variable The timeframe to request the data from. string timeframe = input.timeframe("30", "Timeframe") //@variable The requested `high` price from the current symbol on the `timeframe` without lookahead bias. // On realtime bars, it returns the current `high` of the `timeframe`. float lookaheadOff = request.security(syminfo.tickerid, timeframe, high, lookahead = barmerge.lookahead_off) //@variable The requested `high` price from the current symbol on the `timeframe` with lookahead bias. // Returns values that should NOT be accessible yet on historical bars. float lookaheadOn = request.security(syminfo.tickerid, timeframe, high, lookahead = barmerge.lookahead_on) //@variable The requested `high` price from the current symbol on the `timeframe` without lookahead bias or repainting. // Behaves the same on historical and realtime bars. float lookaheadOnOffset = request.security(syminfo.tickerid, timeframe, high[1], lookahead = barmerge.lookahead_on) // Plot the values. plot(lookaheadOff, "High, no lookahead bias", color.new(color.blue, 40), 5) plot(lookaheadOn, "High with lookahead bias", color.fuchsia, 3) plot(lookaheadOnOffset, "High, no lookahead bias or repaint", color.aqua, 3) // Highlight the background on realtime bars. bgcolor(barstate.isrealtime ? color.new(color.orange, 60) : na, title = "Realtime bar highlight")

Note that:

- The series requested using barmerge.lookahead_off has a new historical value at the *end* of each HTF period, and both series requested using barmerge.lookahead_on have new historical data at the *start* of each period.
- On realtime bars, the plot of the series without lookahead (blue) and the series with lookahead and no historical offset (fuchsia) show the *same value* (i.e., the HTF period's unconfirmed high price), as no data exists beyond those points to leak into the past. Both of these plots *repaint* their results after the user reloads the script, because the *elapsed* realtime bars from the previous run become *historical* bars in the new run.
- The series that uses lookahead and a historical offset (aqua) *does not* repaint its results, because it always uses the last *confirmed* value from the higher timeframe. See the Avoiding repainting section of this page for more information.

> NoticeIn Pine Script versions 1 and 2, the `security()` function did not include a `lookahead` parameter. However, the request behaved the same as those with `lookahead = barmerge.lookahead_on` in later versions of Pine, meaning that it systematically accessed future data from a higher timeframe on historical bars. Therefore, *exercise caution* with Pine v1 or v2 scripts that use HTF `security()` calls, unless those calls offset the requested series with the [] operator.

Dynamic requests

By default, unlike all previous Pine Script versions, `request.*()` function calls in Pine Script v6 are *dynamic*.

In contrast to non-dynamic requests, dynamic requests can:

- Access data from different data feeds using a single `request.*()` instance with "series" arguments.
- Execute within the local scopes of conditional structures, loops, and exported functions.
- Execute nested requests.

Aside from the features listed above, there are insignificant differences in the behavior of dynamic and non-dynamic requests. However, for backward compatibility, programmers can deactivate dynamic requests by specifying `dynamic_requests = false` in the indicator(), strategy(), or library() declaration statement.

> Note
>
> In Pine Script v5, it is possible for scripts to call user-defined functions or methods containing `request.*()` calls inside loops or conditional structures *without* enabling dynamic requests. However, those wrapped requests are **not** truly dynamic, and they still require **"simple"** or weaker qualifiers for all arguments that define the requested context.
>
> In Pine Script v6, scripts **cannot** use wrapped `request.*()` calls within the local blocks of these structures without enabling dynamic requests.

"series" arguments

Scripts without dynamic requests enabled cannot use “series” arguments for most `request.*()` function parameters, which means the argument values *cannot change*. The only exception is the `expression` parameter in request.security(), request.security_lower_tf(), and request.seed(), which *always* allows “series” values.

In contrast, when a script allows dynamic requests, all `request.*()` function parameters that define parts of the ticker ID or timeframe of a request accept “series” arguments that *can change* with each script execution. In other words, with dynamic requests, it’s possible for a single `request.*()` instance to fetch data from *different contexts* in different executions. Some other optional parameters, such as `ignore_invalid_symbol`, can also accept “series” arguments, allowing additional flexibility in `request.*()` call behaviors.

The following script declares a `symbolSeries` variable that is assigned four different symbol strings in 20-bar cycles, with its value changing after every five bars. The request.security() call uses this variable as the `symbol` argument. The script plots the `requestedClose` values, which therefore represent a different symbol’s close prices for each five-bar period.

//@version=6 indicator("'series' arguments demo") //@variable A "series" that cycles through four different symbol strings. Its value changes every five bars. string symbolSeries = switch int(bar_index / 5) % 4 1 => "NASDAQ:MSFT" 2 => "NASDAQ:AMD" 3 => "NASDAQ:INTC" => "AMEX:SPY" //@variable The requested `close` value from one of the four `symbolSeries` values on the chart's timeframe. float requestedClose = request.security(symbolSeries, timeframe.period, close) // Plot the `requestedClose`. plot(requestedClose, "Requested close", color.purple, 3) // Draw a label displaying the requested symbol each time the `symbolSeries` changes. if symbolSeries != symbolSeries[1] label.new(bar_index, requestedClose, symbolSeries, textcolor = color.white)

Note that:

- The script draws a label every time the `symbolSeries` changes, to signify which symbol’s data the `requestedClose` currently represents.
- Pine v6 scripts enable dynamic requests by default, allowing this script to use a “series string” `symbol` argument in its request.security() call without error. If the dynamic behavior is disabled by including `dynamic_requests = false` in the indicator() declaration, then the “series” argument causes a compilation error.

An important limitation is that when using dynamic `request.*()` calls with “series” arguments or within local scopes, scripts must request all required datasets while executing on **historical bars**. All `request.*()` calls on *realtime* bars can retrieve data from the datasets that the script previously accessed on historical bars, but they **cannot** request a new context or evaluate a new expression.

To illustrate this limitation, let’s revisit the above script. Notice that it requests close data for all four symbols on the chart’s timeframe during its historical executions. The external datasets for those four contexts are the **only** ones that any `request.*()` call on realtime bars can access.

Below, we changed the `timeframe` argument in the script’s request.security() call to specify that it requests `symbolSeries` data from the chart’s timeframe on historical bars and the “240” (240 minutes = 4H) timeframe on realtime bars. This version raises a runtime error on the first realtime tick, if it is run on any timeframe other than the 4H timeframe, because it **cannot** access the 4H data feeds without requesting them on historical bars first:

//@version=6 indicator("Invalid realtime request demo") //@variable A "series" that cycles through four different symbol strings. Its value changes every five bars. string symbolSeries = switch int(bar_index / 5) % 4 1 => "NASDAQ:MSFT" 2 => "NASDAQ:AMD" 3 => "NASDAQ:INTC" => "AMEX:SPY" // Request the `close` of the `symbolSeries` from the chart's timeframe on historical bars and the "240" (4H) timeframe // on realtime bars. Causes a runtime error on the first realtime tick because the script did not previously access // data from the "240" timeframe on any historical bars. float requestedClose = request.security(symbolSeries, barstate.isrealtime ? "240" : timeframe.period, close) // Plot the `requestedClose`. plot(requestedClose, "Requested close", color.purple, 3) // Draw a label displaying the requested symbol each time the `symbolSeries` changes. if symbolSeries != symbolSeries[1] label.new(bar_index, requestedClose, symbolSeries, textcolor = color.white)

In local scopes

When scripts do not allow dynamic requests, all `request.*()` calls execute once on *every* bar or realtime tick, which prevents their use within most local scopes. The only exception is for `request.*()` calls in the scopes of *non-exported* functions and methods, because the Pine Script compiler extracts such calls into the *global scope* during translation.

Scripts that allow dynamic requests *do not* restrict the execution of `request.*()` calls to the global scope. They can call `request.*()` functions directly within the scopes of conditional structures and loops, meaning that each `request.*()` instance in the code can activate zero, one, or several times on each script execution.

The following example uses a single request.security() instance within a loop to request data from multiple forex data feeds. The script declares an array of `symbols` on the first chart bar, which it iterates through on all bars using a for…in loop. Each loop iteration calls request.security() to retrieve the volume value for one of the symbols and pushes the result into the `requestedData` array. After the loop terminates, the script calculates the average, maximum, and minimum values from the `requestedData` array using built-in methods, then plots the results on the chart:

//@version=6 indicator("In local scopes demo", format = format.volume) //@variable An array of "string" values representing different symbols to request. var array<string> symbols = array.from("EURUSD", "USDJPY", "GBPUSD", "AUDUSD", "USDCAD", "USDCHF", "NZDUSD", "EURJPY", "GBPJPY", "EURGBP") //@variable An array containing the data retrieved for each requested symbol. array<float> requestedData = array.new<float>() // Retrieve `volume` data for each symbol in the `symbols` array and push the results into the `requestedData` array. for symbol in symbols float data = request.security("OANDA:" + symbol, timeframe.period, volume) requestedData.push(data) // Calculate the average, maximum, and minimum tick volume in the `requestedData`. float avgVolume = requestedData.avg() float maxVolume = requestedData.max() float minVolume = requestedData.min() // Plot the `avgVolume`, `maxVolume`, and `minVolume`. plot(avgVolume, "Average volume", color.gray, 3) plot(maxVolume, "Highest volume", color.teal, 3) plot(minVolume, "Lowest volume", color.maroon, 3)

Notice that the `expression` argument in the above example (volume) is *loop-invariant*, i.e., it does not change on any loop iteration. When using `request.*()` calls within a loop, all parameters defining parts of the requested *context* can accept arguments that depend on variables from the loop's header or mutable variables that change within the loop's local scope. However, a `request.*()` call's evaluated expression **cannot** depend on the values of those variables.

Here, we modified the above script to use the *second form* of the for…in loop statement, which creates a tuple containing the index and value of each element in the `symbols` array. The request.security() instance in

this version uses the index (`i`) in its `expression` argument, resulting in a *compilation error*:

//@version=6 indicator("Loop-dependent expression demo", format = format.volume) //@variable An array of "string" values representing different symbols to request. var array<string> symbols = array.from("EURUSD", "USDJPY", "GBPUSD", "AUDUSD", "USDCAD", "USDCHF", "NZDUSD", "EURJPY", "GBPJPY", "EURGBP") //@variable An array containing the data retrieved for each requested symbol. array<float> requestedData = array.new<float>() // Retrieve `volume` data for each symbol in the `symbols` array, weighted using the element index. // Causes a compilation error because the `expression` in `request.security()` cannot depend on loop variables // or mutable variables that change within the loop's scope. for [i, symbol] in symbols float data = request.security("OANDA:" + symbol, timeframe.period, volume * (10 - i)) requestedData.push(data) // Calculate the average, maximum, and minimum tick volume in the `requestedData`. float avgVolume = requestedData.avg() float maxVolume = requestedData.max() float minVolume = requestedData.min() // Plot the `avgVolume`, `maxVolume`, and `minVolume`. plot(avgVolume, "Average volume", color.gray, 3) plot(maxVolume, "Highest volume", color.teal, 3) plot(minVolume, "Lowest volume", color.maroon, 3)

In libraries

Libraries with dynamic requests enabled can *export* functions and methods that utilize `request.*()` calls within their local scopes, provided that the evaluated expressions **do not** depend on any exported function parameters.

For example, this simple library exports an `htfPrices()` function that requests a tuple of confirmed open, high, low, and close prices using a specified `tickerID` and `timeframe`. If we publish this library, another script can *import* the function to request higher-timeframe prices without explicitly calling request.security().

//@version=6 library("DynamicRequests") //@function Requests a tuple containing confirmed HTF OHLC data for a specified `tickerID` and `timeframe`. //@param tickerID The ticker identifier to request data for. //@param timeframe The timeframe of the requested data. //@returns A tuple containing the last confirmed `open`, `high`, `low`, and `close` from the requested context. export htfPrices(string tickerID, string timeframe) => if timeframe.in_seconds() >= timeframe.in_seconds(timeframe) runtime.error("The `timeframe` argument of `getHTFPrices()` must be higher than the chart's timeframe.") request.security(tickerID, timeframe, [open[1], high[1], low[1], close[1]], lookahead = barmerge.lookahead_on)

Note that:

- The tuple that the request.security() call includes as the `expression` argument *does not* depend on the `htfPrices()` parameters.
- The `htfPrices()` function includes a runtime.error() call that raises a custom runtime error when the `timeframe` argument does not represent a higher timeframe than the chart's timeframe. See the higher timeframes section for more information.
- The request.security() call uses barmerge.lookahead_on and offsets each item in the tuple by one bar. This is the only recommended method to avoid repainting.

Nested requests

Scripts can use dynamic requests to execute *nested requests*, i.e., `request.*()` calls that dynamically evaluate other `request.*()` calls that their `expression` arguments depend on.

When a request.security() or request.security_lower_tf() call uses an empty string or syminfo.tickerid for its `symbol` argument, or if it uses an empty string or timeframe.period for the `timeframe` argument, the requested ticker ID or timeframe *depends* on the context where the call executes. This context is normally the ticker ID or timeframe of the chart that the script is running on. However, if such a request.security() or request.security_lower_tf() function call is evaluated by another `request.*()` call, the nested request *inherits* that `request.*()` call's ticker ID or timeframe information.

For example, the script below contains two request.security() calls and uses Pine Logs to display their results. The first call uses empty strings as its `symbol` and `timeframe` arguments, meaning that the requested context depends on where the call executes. It evaluates a concatenated string containing the call's requested ticker ID and timeframe, and the script assigns its result to the `info1` variable.

The second call requests data for a specific `symbol` and `timeframe` using the `info1` variable as its `expression` argument. Since the `info1` variable depends on the first request.security() call, the second call evaluates the first call *within* its own context. Therefore, the first call adopts the second call's ticker ID and timeframe while executing within that context, resulting in a different returned value:

//@version=6 indicator("Nested requests demo") //@variable A concatenated string containing the current `syminfo.tickerid` and `timeframe.period`. string info1 = request.security("", "", syminfo.tickerid + "_" + timeframe.period) //@variable A concatenated string representing the `info1` value calculated within the "NASDAQ:AAPL, 240" context. // This call evaluates the call on line 5 within its context to determine its result because the script // allows dynamic requests. string info2 = request.security("NASDAQ:AAPL", "240", info1) // Log the results from both calls in the Pine Logs pane on the last historical bar. if barstate.islastconfirmedhistory log.info("First request: {0}", info1) log.info("Second request: {0}", info2)

This script allows the execution of the first request.security() call within the context of the second call because Pine v6 scripts enable dynamic `request.*()` calls by default. We can disable this behavior by including `dynamic_requests = false` in the indicator() declaration statement. Without dynamic requests enabled, the script evaluates each call *independently*, passing the first call's calculated value directly into the second call rather than executing the first call within the second context. Consequently, the second call's returned value is the *same* as the first call's value, as we see below:

//@version=6 indicator("Nested requests demo", dynamic_requests = false) //@variable A concatenated string containing the current `syminfo.tickerid` and `timeframe.period`. string info1 = request.security("", "", syminfo.tickerid + "_" + timeframe.period) //@variable The same value as `info1`. This call does not evalutate the call on line 5 because dynamic requests aren't // allowed. Instead, it only uses the value of `info1`, meaning its result does not change. string info2 = request.security("NASDAQ:AAPL", "240", info1) // Log the results from both calls in the Pine Logs pane on the last historical bar. if barstate.islastconfirmedhistory log.info("First request: {0}", info1) log.info("Second request: {0}", info2)

Data feeds

TradingView's data providers supply different data feeds that scripts can access to retrieve information about an instrument, including:

- Intraday historical data (for timeframes < 1D)
- End-of-day (EOD) historical data (for timeframes >= 1D)
- Realtime data (which may be delayed, depending on your account type and extra data services)
- Extended hours data

Not all of these data feed types exist for every instrument. For example, the symbol “BNC:BLX” only has EOD data available.

For some instruments with intraday and EOD historical feeds, volume data may not be the same since some trades (block trades, OTC trades, etc.) may only be available at the *end* of the trading day. Consequently, the EOD feed will include this volume data, but the intraday feed will not. Differences between EOD and intraday volume feeds are almost nonexistent for instruments such as cryptocurrencies, but they are commonplace in stocks.

Slight price discrepancies may also occur between EOD and intraday feeds. For example, the high value on one EOD bar may not match any intraday high values supplied by the data provider for that day.

Another distinction between EOD and intraday data feeds is that EOD feeds do not contain information from *extended hours*.

When retrieving information on realtime bars with `request.*()` functions, it’s important to note that historical and realtime data reported for an instrument often rely on *different* data feeds. A broker/exchange may retroactively modify values reported on realtime bars, which the data will only reflect after refreshing the chart or restarting the script.

Another important consideration is that the chart’s data feeds and feeds requested from providers by the script are managed by *independent*, concurrent processes. Consequently, in some *rare* cases, it’s possible for races to occur where requested results temporarily fall out of synch with the chart on a realtime bar, which a script retroactively adjusts after restarting its executions.

These points may account for variations in the values retrieved by `request.*()` functions when requesting data from other contexts. They may also result in discrepancies between data received on realtime bars and historical bars. There are no steadfast rules about the variations one may encounter in their requested data feeds.

> NoteAs a rule, TradingView *does not* generate data; it relies on its data providers for the information displayed on charts and accessed by scripts.

When using data feeds requested from other contexts, it’s also crucial to consider the *time axis* differences between the chart the script executes on and the requested feeds since `request.*()` functions adapt the returned series to the chart’s time axis. For example, requesting “BTCUSD” data on the “SPY” chart with request.security() will only show new values when the “SPY” chart has new data as well. Since “SPY” is not a 24-hour symbol, the “BTCUSD” data returned will contain gaps that are otherwise not present when viewing its chart directly.

`request.security()`

The request.security() function allows scripts to request data from other contexts than the chart the script executes on, such as:

- Other symbols, including spread symbols
- Other timeframes (see our User Manual’s page on Timeframes to learn about timeframe specifications in Pine Script)

- Custom contexts, including alternative sessions, price adjustments, chart types, etc. using `ticker.*()` functions

This is the function's signature:

```
request.security(symbol, timeframe, expression, gaps, lookahead, ignore_invalid
```

The `symbol` value is the ticker identifier representing the symbol to fetch data from. This parameter accepts values in any of the following formats:

- A "string" representing a symbol (e.g., "IBM" or "EURUSD") or an *"Exchange:Symbol" pair* (e.g., "NYSE:IBM" or "OANDA:EURUSD"). When the value does not contain an exchange prefix, the function selects the exchange automatically. We recommend specifying the exchange prefix when possible for consistent results. Users can also pass an empty string to this parameter, which prompts the function to use the current chart's symbol.
- A "string" representing a spread symbol (e.g., "AMD/INTC"). Note that "Bar Replay" mode does not work with these symbols.
- The syminfo.ticker or syminfo.tickerid built-in variables, which return the symbol or the "Exchange:Symbol" pair that the current chart references. We recommend using syminfo.tickerid to avoid ambiguity unless the exchange information does not matter in the data request. For more information on `syminfo.*` variables, see this section of our Chart information page.
- A custom ticker identifier created using `ticker.*()` functions. Ticker IDs constructed from these functions may contain additional settings for requesting data using non-standard chart calculations, alternative sessions, and other contexts. See the Custom contexts section for more information.

The `timeframe` value specifies the timeframe of the requested data. This parameter accepts "string" values in our timeframe specification format (e.g., a value of "1D" represents the daily timeframe). To request data from the same timeframe as the chart the script executes on, use the timeframe.period variable or an empty string.

The `expression` parameter of the request.security() function determines the data it retrieves from the specified context. This versatile parameter accepts "series" values of int, float, bool, color, string, and chart.point types. It can also accept tuples, collections, user-defined types, and the outputs of function and method calls. For more details on the data one can retrieve, see the Requestable data section below.

> NoticeIf a `request.*()` call uses the value from a source input in its `expression` argument, and that input accesses a plotted series from another indicator, the request evaluates that series using the data for the **chart's symbol**, and **not** the data for the specified symbol. This behavior occurs because `request.*()` functions cannot evaluate the scopes required by an external series. Therefore, some `request.*()` calls that use a `symbol` argument and request the value of a source input can return unintended results.

Timeframes

The request.security() function can request data from any available timeframe, regardless of the chart the script executes on. The timeframe of the data retrieved depends on the `timeframe` argument in the function

call, which may represent a higher timeframe (e.g., using “1D” as the `timeframe` value while running the script on an intraday chart) or the chart’s timeframe (i.e., using timeframe.period or an empty string as the `timeframe` argument).

Scripts can also request *limited* data from lower timeframes with request.security() (e.g., using “1” as the `timeframe` argument while running the script on a 60-minute chart). However, we don’t typically recommend using this function for LTF data requests. The request.security_lower_tf() function is more optimal for such cases.

Higher timeframes

Most use cases of request.security() involve requesting data from a timeframe higher than or the same as the chart timeframe. For example, this script retrieves the hl2 price from a requested `higherTimeframe`. It plots the resulting series on the chart alongside the current chart’s hl2 for comparison:

//@version=6 indicator("Higher timeframe security demo", overlay = true) //@variable The higher timeframe to request data from. string higherTimeframe = input.timeframe("240", "Higher timeframe") //@variable The `hl2` value from the `higherTimeframe`. Combines lookahead with an offset to avoid repainting. float htfPrice = request.security(syminfo.tickerid, higherTimeframe, hl2[1], lookahead = barmerge.lookahead_on) // Plot the `hl2` from the chart timeframe and the `higherTimeframe`. plot(hl2, "Current timeframe HL2", color.teal, 2) plot(htfPrice, "Higher timeframe HL2", color.purple, 3)

Note that:

- We’ve included an offset to the `expression` argument and used barmerge.lookahead_on in request.security() to ensure the series returned behaves the same on historical and realtime bars. See the Avoiding repainting section for more information.

Notice that in the above example, it is possible to select a `higherTimeframe` value that actually represents a *lower timeframe* than the one the chart uses, as the code does not prevent it. When designing a script to work specifically with higher timeframes, we recommend including conditions to prevent it from accessing lower timeframes, especially if you intend to publish it.

Below, we’ve added an if structure to our previous example. If the `higherTimeframe` value represents a timeframe that is smaller than the chart’s timeframe, the script calls runtime.error() within the structure’s local block to raise a custom runtime error, effectively preventing the script from requesting LTF data:

//@version=6 indicator("Higher timeframe security demo", overlay = true) //@variable The higher timeframe to request data from. string higherTimeframe = input.timeframe("240", "Higher timeframe") // Raise a runtime error when the `higherTimeframe` is smaller than the chart's timeframe. if timeframe.in_seconds() > timeframe.in_seconds(higherTimeframe) runtime.error("The requested timeframe is smaller than the chart's timeframe. Select a higher timeframe.") //@variable The `hl2` value from the `higherTimeframe`. Combines lookahead with an offset to avoid repainting. float htfPrice = request.security(syminfo.tickerid, higherTimeframe, hl2[1], lookahead = barmerge.lookahead_on) // Plot the `hl2` from the chart timeframe and the `higherTimeframe`. plot(hl2, "Current timeframe HL2", color.teal, 2) plot(htfPrice, "Higher timeframe HL2", color.purple, 3)

Lower timeframes

Although the request.security() function is intended to operate on timeframes greater than or equal to the chart timeframe, it *can* request data from lower timeframes as well, with limitations. When calling this function to access a lower timeframe, it will evaluate the `expression` from the LTF context. However, it returns the results from only a *single* intrabar (LTF bar) on each chart bar.

The intrabar that the function returns data from on each historical chart bar depends on the `lookahead` value in the function call. When using barmerge.lookahead_on, it will return the *first* available intrabar from the chart period. When using barmerge.lookahead_off, it will return the *last* intrabar from the chart period. On realtime bars, it returns the last available value of the `expression` from the timeframe, regardless of the `lookahead` value, as the realtime intrabar information retrieved by the function is not yet sorted.

This script retrieves close data from the valid timeframe closest to a fourth of the size of the chart timeframe. It makes two calls to request.security() with different `lookahead` values. The first call uses barmerge.lookahead_on to access the first intrabar value in each chart bar. The second uses the default `lookahead` value (barmerge.lookahead_off), which requests the last intrabar value assigned to each chart bar. The script plots the outputs of both calls on the chart to compare the difference:

//@version=6 indicator("Lower timeframe security demo", overlay = true) //@variable The valid timeframe closest to 1/4 the size of the chart timeframe. string lowerTimeframe = timeframe.from_seconds(int(timeframe.in_seconds() / 4)) //@variable The `close` value on the `lowerTimeframe`. Represents the first intrabar value on each chart bar. float firstLTFClose = request.security(syminfo.tickerid, lowerTimeframe, close, lookahead = barmerge.lookahead_on) //@variable The `close` value on the `lowerTimeframe`. Represents the last intrabar value on each chart bar. float lastLTFClose = request.security(syminfo.tickerid, lowerTimeframe, close) // Plot the values. plot(firstLTFClose, "First intrabar close", color.teal, 3) plot(lastLTFClose, "Last intrabar close", color.purple, 3) // Highlight the background on realtime bars. bgcolor(barstate.isrealtime ? color.new(color.orange, 70) : na, title = "Realtime background highlight")

Note that:

- The script determines the value of the `lowerTimeframe` by calculating the number of seconds in the chart timeframe with timeframe.in_seconds(), then dividing by four and converting the result to a valid timeframe string via timeframe.from_seconds().
- The plot of the series without lookahead (purple) aligns with the close value on the chart timeframe, as this is the last intrabar value in the chart bar.
- Both request.security() calls return the *same* value (the current close) on each *realtime* bar, as shown on the bars with the orange background.
- Scripts can retrieve up to 200,000 intrabars from a lower-timeframe context. The number of chart bars with available intrabar data varies with the requested lower timeframe, the `calc_bars_count` value, and the user's plan. For more information, see this section of the Limitations page.

> TipWhile scripts can use request.security() to retrieve limited intrabar data, we recommend using request.security_lower_tf() function for such requests in most cases. Instead of retrieving data for only a single LTF bar on each chart bar, it returns an array containing the data for *all* available LTF bars in the chart bar. See the `request.security_lower_tf()` section below to learn more.

Requestable data

The request.security() function is quite versatile, as it can retrieve values of any fundamental type (int, float, bool, color, or string). It can also request the IDs of data structures and built-in or user-defined types that reference fundamental types. The data this function requests depends on its `expression` parameter, which accepts any of the following arguments:

- Built-in variables and function calls
- Variables declared by the script
- Tuples
- Calls to user-defined functions
- Chart points
- Collections
- User-defined types

Built-in variables and functions

A frequent use case of request.security() is requesting the output of a built-in variable or function/method call from another symbol or timeframe.

For example, suppose we want to calculate the 20-bar SMA of a symbol's ohlc4 prices from the daily timeframe while on an intraday chart. We can accomplish this task with a single line of code:

float ma = request.security(syminfo.tickerid, "1D", ta.sma(ohlc4, 20))

The above line calculates the value of `ta.sma(ohlc4, 20)` on the current symbol's data from the daily timeframe.

It's important to note that newcomers to Pine might sometimes confuse the above line of code as being equivalent to the following:

float ma = ta.sma(request.security(syminfo.tickerid, "1D", ohlc4), 20)

However, this line returns an entirely *different* result. Rather than requesting a 20-bar SMA from the daily timeframe, it requests the ohlc4 price from the daily timeframe and calclates the ta.sma() of the results over 20 **chart bars**.

In essence, when the intention is to request the results of an expression from other contexts, pass the expression *directly* to the `expression` parameter in the request.security() call, as demonstrated in the initial example.

Let's expand on this concept. The script below calculates a multi-timeframe (MTF) ribbon of moving averages, where each moving average in the ribbon calculates over the same number of bars on its respective timeframe. Each request.security() call uses a ta.sma() call as its `expression` argument to return a `length`-bar SMA from the specified timeframe:

//@version=6 indicator("Requesting built-ins demo", "MTF Ribbon", true) //@variable The length of each moving average. int length = input.int(20, "Length", 1) //@variable The number of seconds in the chart timeframe. int chartSeconds = timeframe.in_seconds() // Calculate the higher timeframes closest to 2, 3, and 4 times the size of the chart timeframe. string htf1 = timeframe.from_seconds(chartSeconds * 2) string htf2 = timeframe.from_seconds(chartSeconds * 3) string htf3 = timeframe.from_seconds(chartSeconds * 4) // Calculate the `length`-bar moving averages from each timeframe. float chartAvg = ta.sma(ohlc4, length) float htfAvg1 = request.security(syminfo.tickerid, htf1, ta.sma(ohlc4, length)) float htfAvg2 =

request.security(syminfo.tickerid, htf2, ta.sma(ohlc4, length)) float htfAvg3 = request.security(syminfo.tickerid, htf3, ta.sma(ohlc4, length)) // Plot the results. plot(chartAvg, "Chart timeframe SMA", color.red, 3) plot(htfAvg1, "Double timeframe SMA", color.orange, 3) plot(htfAvg2, "Triple timeframe SMA", color.green, 3) plot(htfAvg3, "Quadruple timeframe SMA", color.blue, 3) // Highlight the background on realtime bars. bgcolor(barstate.isrealtime ? color.new(color.aqua, 70) : na, title = "Realtime highlight")

Note that:

- The script calculates the ribbon's higher timeframes by multiplying the chart's timeframe.in_seconds() value by 2, 3, and 4, then converting each result into a valid timeframe string using timeframe.from_seconds().
- Instead of calling ta.sma() within each request.security() call, one could use the `chartAvg` variable as the `expression` in each call to achieve the same result. See the next section for more information.
- On realtime bars, this script also tracks *unconfirmed* SMA values from each higher timeframe. See the Historical and realtime behavior section to learn more.

Declared variables

The request.security() function's `expression` parameter can accept declared variables that are accessible to the scope from which the function call executes. When using a declared variable as the `expression` argument, the function call *duplicates* all *preceding code* that determines the assigned value or reference. This duplication allows the function to evaluate necessary calculations and logic in the requested context without affecting the original variable.

For instance, this line of code declares a `priceReturn` variable that holds the current bar's arithmetic price return:

float priceReturn = (close - close[1]) / close[1]

We can evaluate the `priceReturn` variable's calculations in another context by using it as the `expression` in a request.security() call. The call below duplicates the variable's calculation and evaluates it across the data from another `symbol`, returning a *separate series* adapted to the chart's time axis:

float requestedReturn = request.security(symbol, timeframe.period, priceReturn)

This example script compares the price returns of the current chart's symbol and a user-specified symbol. It calculates the value of the `priceReturn` variable, then uses that variable as the `expression` in a request.security() call to evaluate the calculation on the input symbol's data. After the request, the script calculates the correlation between the `priceReturn` and `requestedReturn` series using ta.correlation() and plots the result on the chart:

//@version=6 indicator("Requesting calculated variables demo", "Price return correlation") //@variable The symbol to compare to the chart symbol. string symbol = input.symbol("SPY", "Symbol to compare") //@variable The number of bars in the calculation window. int length = input.int(60, "Length", 1) //@variable The close-to-close price return. float priceReturn = (close - close[1]) / close[1] //@variable The close-to-close price return calculated on another `symbol`. float requestedReturn = request.security(symbol, timeframe.period, priceReturn) //@variable The correlation between the `priceReturn` and `requestedReturn` over `length` bars. float correlation = ta.correlation(priceReturn, requestedReturn, length) //@variable The color of the correlation plot. color plotColor = color.from_gradient(correlation, -1, 1,

color.purple, color.orange) // Plot the correlation value. plot(correlation, "Correlation", plotColor, style = plot.style_area)

Note that:

- The request.security() call executes the same calculation used in the `priceReturn` declaration, but the request's calculation operates on the close values from the specified symbol's data.
- The script uses the color.from_gradient() function to calculate the color for the plot of the `correlation` series on each bar. See this section of the Colors page to learn more about color gradients.

When using a variable as the `expression` argument of a `request.*()` call, it's important to note that the function only duplicates code that affects the variable *before* the call. It *cannot* copy any subsequent code following the call. Consequently, if the script reassigns the variable or modifies its referenced data *after* calling request.security(), the code evaluated on the requested data **does not** include those additional operations.

For example, the following script declares a `counter` variable and calls request.security() to evaluate the variable from the same context as the chart. After the call, the script increments the `counter` value by one with the addition assignment operator (+=), then uses plots and Pine Logs to display the `counter` and `requestedCounter` values for comparison.

As shown below, the plots and logs of the two variables display *different* values. The `requestedCounter` variable has a consistent value of 0 because the request.security() call evaluates only the initial variable declaration. The request cannot evaluate the addition assignment operation because the script includes that code *after* the function call:

//@version=6 indicator("Modifying variables after requests demo") //@variable A counter that starts at 0 and increments by 1 on each bar. var int counter = 0 //@variable Holds a consistent value of 0. // `request.security()` cannot evaluate `counter += 1` in its requested context // because that modification occurs *after* the call. int requestedCounter = request.security(syminfo.tickerid, timeframe.period, counter) // Increment the `counter` by 1. This operation is *not* included in the `requestedCounter` calculation. counter += 1 // Plot both variables for comparison. plot(counter, "Original counter", color.purple, 3) plot(requestedCounter, "Requested counter", color.red, 3) // Log the values of both variables in the Pine Logs pane. if barstate.isconfirmed log.info("counter: {0}, requestedCounter: {1}", counter, requestedCounter)

Tuples

Tuples in Pine Script are comma-separated lists of expressions enclosed in square brackets. Programmers often use tuples when creating functions, conditional structures, or loops that return multiple values or references from their local scopes.

The request.security() function can accept a tuple as its `expression` argument, allowing scripts to request multiple series of different types using a single function call. The expressions within requested tuples can be of any type outlined throughout the Requestable data section of this page, excluding other tuples.

> NoteThe combined size of all tuples returned by `request.*()` calls in a script cannot exceed 127 elements. See the Tuple element limit section of the Limitations page for more information.

Tuples are particularly helpful when a script needs to retrieve more than one value from a specific context.

For example, the following script calculates the percent rank of the close series over `length` bars and assigns the result to the `rank` variable. It then calls request.security() to request a tuple containing the values of `rank`, `ta.crossover(rank, 50)`, and `ta.crossunder(rank, 50)` from a specified timeframe. The script plots the `requestedRank` series in a separate pane, then uses the result of a ternary expression based on the `crossOver` and `crossUnder` values within a bgcolor() call to conditionally highlight the pane's background:

//@version=6 indicator("Requesting tuples demo", "Percent rank cross") //@variable The timeframe of the request. string timeframe = input.timeframe("240", "Timeframe") //@variable The number of bars in the calculation. int length = input.int(20, "Length") //@variable The previous bar's percent rank of the `close` price over `length` bars. float rank = ta.percentrank(close, length)[1] // Request the `rank` value from another `timeframe`, and two "bool" values indicating the `rank` from the `timeframe` // crossed over or under 50. [requestedRank, crossOver, crossUnder] = request.security(syminfo.tickerid, timeframe, [rank, ta.crossover(rank, 50), ta.crossunder(rank, 50)], lookahead = barmerge.lookahead_on) // Plot the `requestedRank` and create a horizontal line at 50. plot(requestedRank, "Percent Rank", linewidth = 3) hline(50, "Cross line", linewidth = 2) // Highlight the background of all bars where the `timeframe`'s `crossOver` or `crossUnder` value is `true`. bgcolor(crossOver ? color.new(color.green, 50) : crossUnder ? color.new(color.red, 50) : na)

Note that:

- We've offset the `rank` variable's expression by one bar using the history-referencing operator [] and included barmerge.lookahead_on in the request.security() call to ensure the values on realtime bars do not repaint after becoming historical bars. See the Avoiding repainting section for more information.
- The request.security() call returns a tuple, so we use a *tuple declaration* to declare the `requestedRank`, `crossOver`, and `crossUnder` variables. To learn more about using tuples, see this section of our User Manual's Type system page.

User-defined functions

User-defined functions and methods are custom functions written by users. They allow users to define sequences of operations associated with an identifier that scripts can conveniently call throughout their executions (e.g., `myUDF()`).

The request.security() function can request the results of user-defined functions and methods whose scopes consist of any types outlined throughout this page's Requestable data section.

For example, this script contains a user-defined `weightedBB()` function that calculates Bollinger Bands with the basis average weighted by a specified `weight` series. The function returns a tuple of custom band values. The script calls the `weightedBB()` as the `expression` argument in request.security() to retrieve a tuple of band values calculated on the specified `timeframe` and plots the results on the chart:

//@version=6 indicator("Requesting user-defined functions demo", "Weighted Bollinger Bands", true) //@variable The timeframe of the request. string timeframe = input.timeframe("480", "Timeframe") //@function Calculates Bollinger Bands with a custom weighted basis. //@param source The series of values to process. //@param length The number of bars in the calculation. //@param mult The standard deviation multiplier. //@param weight The series of weights corresponding to each `source` value. //@returns A tuple containing the basis, upper band, and lower band respectively. weightedBB(float source, int length, float mult = 2.0, float weight = 1.0) => //@variable The basis of the bands. float ma =

math.sum(source * weight, length) / math.sum(weight, length) //@variable The standard deviation from the `ma`. float dev = 0.0 // Loop to accumulate squared error. for i = 0 to length - 1 difference = source[i] - ma dev += difference * difference // Divide `dev` by the `length`, take the square root, and multiply by the `mult`. dev := math.sqrt(dev / length) * mult // Return the bands. [ma, ma + dev, ma - dev] // Request weighted bands calculated on the chart symbol's prices over 20 bars from the // last confirmed bar on the `timeframe`. [basis, highBand, lowBand] = request.security(syminfo.tickerid, timeframe, weightedBB(close[1], 20, 2.0, (high - low)[1]), lookahead = barmerge.lookahead_on) // Plot the values. basisPlot = plot(basis, "Basis", color.orange, 2) upperPlot = plot(highBand, "Upper", color.teal, 2) lowerPlot = plot(lowBand, "Lower", color.maroon, 2) fill(upperPlot, lowerPlot, color.new(color.gray, 90), "Background")

Note that:

- We offset the `source` and `weight` arguments in the `weightedBB()` call used as the `expression` in request.security() and used barmerge.lookahead_on to ensure the requested results reflect the last confirmed values from the `timeframe` on realtime bars. See this section to learn more.

Chart points

Chart points are objects that represent coordinates on the chart. Lines, boxes, polylines, and labels use these objects to set their display locations.

The request.security() function can use the ID of a chart.point instance in its `expression` argument, allowing scripts to retrieve chart coordinates from other contexts.

The example below requests a tuple of historical chart points from a higher timeframe and uses them to draw boxes on the chart. The script declares the `topLeft` and `bottomRight` variables that reference chart.point IDs from the last confirmed bar. It then uses request.security() to request a tuple containing the IDs of chart points representing the `topLeft` and `bottomRight` from a `higherTimeframe`.

When a new bar starts on the `higherTimeframe`, the script draws a new box using the `time` and `price` coordinates from the `requestedTopLeft` and `requestedBottomRight` chart points:

//@version=6 indicator("Requesting chart points demo", "HTF Boxes", true, max_boxes_count = 500) //@variable The timeframe to request data from. string higherTimeframe = input.timeframe("1D", "Timeframe") // Raise a runtime error if the `higherTimeframe` is smaller than the chart's timeframe. if timeframe.in_seconds(higherTimeframe) < timeframe.in_seconds(timeframe.period) runtime.error("The selected timeframe is too small. Choose a higher timeframe.") //@variable A `chart.point` containing top-left coordinates from the last confirmed bar. topLeft = chart.point.now(high)[1] //@variable A `chart.point` containing bottom-right coordinates from the last confirmed bar. bottomRight = chart.point.from_time(time_close, low)[1] // Request the last confirmed `topLeft` and `bottomRight` chart points from the `higherTimeframe`. [requestedTopLeft, requestedBottomRight] = request.security(syminfo.tickerid, higherTimeframe, [topLeft, bottomRight], lookahead = barmerge.lookahead_on) // Draw a new box when a new `higherTimeframe` bar starts. // The box uses the `time` fields from the `requestedTopLeft` and `requestedBottomRight` as x-coordinates. if timeframe.change(higherTimeframe) box.new(requestedTopLeft, requestedBottomRight, color.purple, 3, xloc = xloc.bar_time, bgcolor = color.new(color.purple, 90))

Note that:

- Because we designed this example to request data from higher timeframes, we've included a runtime.error() call that the script executes if the `higherTimeframe` value represents a lower timeframe than timeframe.period.

Collections

Pine Script *collections* (arrays, matrices, and maps) are data structures that contain an arbitrary number of elements with specified types. The request.security() function can retrieve the IDs of collections whose elements consist of:

- Fundamental types
- Chart points
- User-defined types that satisfy the criteria listed in the section below

This example below calculates the ratio of a confirmed bar's high-low range to the range between the highest and lowest prices over 10 bars from a from a specified `symbol` and `timeframe`. It uses maps to hold the values used in the calculations.

The script uses a `data` map with "string" keys and "float" values to store the current bar's high, low, ta.highest(), and ta.lowest() results. It passes the map as the `expression` argument in a request.security() call on each bar to retrieve another map containing the values calculated from the specified context, then assigns that map's reference to the `otherData` variable. The script uses the "float" values associated with the "High", "Low", "Highest", and "Lowest" keys of the `otherData` map to calculate the `ratio` series that it plots in the chart pane:

//@version=6 indicator("Requesting collections demo", "Bar range ratio") //@variable The ticker ID to request data from. string symbol = input.symbol("", "Symbol") //@variable The timeframe of the request. string timeframe = input.timeframe("30", "Timeframe") //@variable A map with "string" keys and "float" values. var map<string, float> data = map.new<string, float>() // Put key-value pairs into the `data` map. map.put(data, "High", high) map.put(data, "Low", low) map.put(data, "Highest", ta.highest(10)) map.put(data, "Lowest", ta.lowest(10)) //@variable A new `map` whose data is calculated from the last confirmed bar of the requested context. map<string, float> otherData = request.security(symbol, timeframe, data[1], lookahead = barmerge.lookahead_on) //@variable The ratio of the context's bar range to the max range over 10 bars. Returns `na` if no data is available. float ratio = na if not na(otherData) ratio := (otherData.get("High") - otherData.get("Low")) / (otherData.get("Highest") - otherData.get("Lowest")) //@variable A gradient color for the plot of the `ratio`. color ratioColor = color.from_gradient(ratio, 0, 1, color.purple, color.orange) // Plot the `ratio`. plot(ratio, "Range Ratio", ratioColor, 3, plot.style_area)

Note that:

- The request.security() call in this script can return na if no data is available from the specified context. Since one cannot call methods on a map variable when its value is na, we've added an if structure to only calculate a new `ratio` value when `otherData` references a valid map instance.

User-defined types

User-defined types (UDTs) are *composite types* containing an arbitrary number of *fields*, which can be of any available type, including other user-defined types.

The request.security() function can retrieve the IDs of objects produced by UDTs from other contexts if their fields consist of:

- Fundamental types
- Chart points
- Collections that satisfy the criteria listed in the section above
- Other UDTs whose fields consist of any of these types

The following example requests an object ID using a specified `symbol` and displays its field values on a chart pane.

The script contains a `TickerInfo` UDT with "string" fields for `syminfo.*` values, an array field to store recent "float" price data, and an "int" field to hold the requested ticker's bar_index value. It assigns a new `TickerInfo` ID to an `info` variable on every bar and uses the variable as the `expression` in request.security() to retrieve the ID of an object representing the calculated `info` from the specified `symbol`.

The script displays the `requestedInfo` object's `description`, `tickerType`, `currency`, and `barIndex` values in a label and uses plotcandle() to display the values from its `prices` array:

//@version=6 indicator("Requesting user-defined types demo", "Ticker info") //@variable The symbol to request information from. string symbol = input.symbol("NASDAQ:AAPL", "Symbol") //@type A custom type containing information about a ticker. //@field description The symbol's description. //@field tickerType The type of ticker. //@field currency The symbol's currency. //@field prices An array of the symbol's current prices. //@field barIndex The ticker's `bar_index`. type TickerInfo string description string tickerType string currency array<float> prices int barIndex //@variable A `TickerInfo` object containing current data. info = TickerInfo.new(syminfo.description, syminfo.type, syminfo.currency, array.from(open, high, low, close), bar_index) //@variable The `info` requested from the specified `symbol`. TickerInfo requestedInfo = request.security(symbol, timeframe.period, info) // Assign a new `TickerInfo` instance to `requestedInfo` if one wasn't retrieved. if na(requestedInfo) requestedInfo := TickerInfo.new(prices = array.new<float>(4)) //@variable A label displaying information from the `requestedInfo` object. var infoLabel = label.new(na, na, "", color = color.purple, style = label.style_label_left, textcolor = color.white, size = size.large) //@variable The text to display inside the `infoLabel`. string infoText = na(requestedInfo) ? "" : str.format("{0}\nType: {1}\nCurrency: {2}\nBar Index: {3}", requestedInfo.description, requestedInfo.tickerType, requestedInfo.currency, requestedInfo.barIndex) // Set the `point` and `text` of the `infoLabel`. label.set_point(infoLabel, chart.point.now(array.last(requestedInfo.prices))) label.set_text(infoLabel, infoText) // Plot candles using the values from the `prices` array of the `requestedInfo`. plotcandle(requestedInfo.prices.get(0), requestedInfo.prices.get(1), requestedInfo.prices.get(2), requestedInfo.prices.get(3), "Requested Prices")

Note that:

- The `syminfo.*` variables used in this script all return "simple string" qualified types. However, objects in Pine are *always* qualified as "series". Consequently, all values assigned to the `info` object's fields automatically adopt the "series" qualifier.
- It is possible for the request.security() call to return na due to differences between the data requested from the `symbol` and the main chart. This script assigns a new `TickerInfo` object to the `requestedInfo` in that case to prevent runtime errors.

`request.security_lower_tf()`

The request.security_lower_tf() function is an alternative to request.security() designed for reliably requesting information from lower-timeframe (LTF) contexts.

While request.security() can retrieve data from a *single* intrabar (LTF bar) in each chart bar, request.security_lower_tf() retrieves data from *all* available intrabars in each chart bar, which the script can access and use in additional calculations. Each request.security_lower_tf() call can retrieve up to 200,000 intrabars from a lower timeframe, depending on the user's plan. See this section of our Limitations page for more information.

> TipWorking with the request.security_lower_tf() function involves frequent usage of *arrays*, because the function always returns array results. Therefore, we recommend reading the Arrays page to make the most of this function and understand how to use its returned data.

Below is the function's signature, which is similar to the signature of request.security():

```
request.security_lower_tf(symbol, timeframe, expression, ignore_invalid_symbol,
```

This function requests data only from timeframes that are *lower than* or *equal to* the chart's timeframe (timeframe.period). If the `timeframe` argument of the request.security_lower_tf() call represents a higher timeframe, the function raises a runtime error or returns na results, depending on the `ignore_invalid_timeframe` parameter. The parameter's default value is `false`, meaning the function raises an error and halts the script's executions if the `timeframe` argument is invalid.

Requesting intrabar data

Intrabar data can provide a script with additional information that may not be obvious or accessible from solely analyzing data sampled on the chart's timerframe. The request.security_lower_tf() function can retrieve many data types from an intrabar context.

Before you venture further in this section, we recommend exploring the Requestable data portion of the request.security() section above, which provides foundational information about the types of data one can request. The `expression` parameter in request.security_lower_tf() accepts most of the same arguments discussed in that section, excluding direct references to collections and mutable variables. Although it accepts many of the same types of arguments, this function returns array results, which comes with some differences in interpretation and handling, as explained below.

Intrabar data arrays

Lower timeframes contain more data points than higher timeframes, as new values come in at a *higher frequency*. For example, when comparing a 1-minute chart to an hourly chart, the 1-minute chart will have up to 60 times the number of bars per hour, depending on the available data.

To address the fact that multiple intrabars exist within a chart bar, request.security_lower_tf() always creates arrays to store the requested data. The elements in the arrays represent the `expression` values retrieved from the lower timeframe sorted in ascending order based on each intrabar's timestamp.

The *type identifier* of the constructed arrays corresponds to the data types passed in the request.security_lower_tf() call. For example, using an “int” as the `expression` will produce an `array<int>` instance, a “bool” as the `expression` will produce an `array<bool>` instance, etc.

The following script uses intrabar information to decompose the chart’s close-to-close price changes into positive and negative parts. It calls request.security_lower_tf() to fetch a “float” array containing `ta.change(close)` values from a specified lower timeframe on each chart bar, then accesses all the array’s elements using a for…in loop to accumulate `positiveChange` and `negativeChange` sums. The script adds the accumulated values to calculate the `netChange` value, then plots the results on the chart alongside the `priceChange` value for comparison:

//@version=6 indicator("Intrabar arrays demo", "Intrabar price changes") //@variable The lower timeframe of the requested data. string lowerTimeframe = input.timeframe("1", "Timeframe") //@variable The close-to-close price change. float priceChange = ta.change(close) //@variable An array of `close` values from available intrabars on the `lowerTimeframe`. array<float> intrabarChanges = request.security_lower_tf(syminfo.tickerid, lowerTimeframe, priceChange) //@variable The total positive intrabar `close` movement on the chart bar. float positiveChange = 0.0 //@variable The total negative intrabar `close` movement on the chart bar. float negativeChange = 0.0 // Loop to calculate totals, starting from the chart bar's first available intrabar. for change in intrabarChanges // Add the `change` to `positiveChange` if its sign is 1, and add to `negativeChange` if its sign is -1. switch math.sign(change) 1 => positiveChange += change -1 => negativeChange += change //@variable The sum of `positiveChange` and `negativeChange`. Equals the `priceChange` on bars with available intrabars. float netChange = positiveChange + negativeChange // Plot the `positiveChange`, `negativeChange`, and `netChange`. plot(positiveChange, "Positive intrabar change", color.teal, style = plot.style_area) plot(negativeChange, "Negative intrabar change", color.maroon, style = plot.style_area) plot(netChange, "Net intrabar change", color.yellow, 5) // Plot the `priceChange` to compare. plot(priceChange, "Chart price change", color.orange, 2)

Note that:

- The plots based on intrabar data may not appear on all available chart bars, as request.security_lower_tf() can only access up to the most recent 200,000 intrabars available from the requested context. When executing this function on a chart bar that doesn’t have accessible intrabar data, it will return an *empty array*.
- The number of intrabars per chart bar may vary depending on the data available from the context and the chart the script executes on. For example, a provider’s 1-minute data feed may not include data for every minute within the 60-minute timeframe due to a lack of trading activity over some 1-minute intervals. To check the number of intrabars retrieved for a chart bar, one can use array.size() on the resulting array.
- If the `lowerTimeframe` value is greater than the chart’s timeframe, the script will raise a *runtime error*, as we have not supplied an `ignore_invalid_timeframe` argument in the request.security_lower_tf() call.

Tuples of intrabar data

When passing a tuple or a function call that returns a tuple as the `expression` argument in request.security_lower_tf(), the result is a tuple of arrays with type templates corresponding to the types within the argument. For example, using a `[float, string, color]` tuple as the `expression` will result in `[array<float>, array<string>, array<color>]` data returned by the function. Using a tuple `expression`

allows a script to fetch the IDs of several arrays containing intrabar data with a single request.security_lower_tf() function call.

> NoteThe combined size of all tuples returned by `request.*()` calls in a script cannot exceed 127 elements. See the Tuple element limit section of the Limitations page for more information.

The following example requests OHLC data from a lower timeframe and visualizes the current bar's intrabars on the chart using lines and boxes. The script calls request.security_lower_tf() with the `[open, high, low, close]` tuple as its `expression` to retrieve a tuple of arrays representing OHLC information from a calculated `lowerTimeframe`. It then uses a for loop to set line coordinates with the retrieved data and current bar indices to display the results next to the current chart bar, providing a "magnified view" of the price movement within the latest candle. It also draws a box around the lines to indicate the chart region occupied by intrabar drawings:

//@version=6 indicator("Tuples of intrabar data demo", "Candle magnifier", max_lines_count = 500) //@variable The maximum number of intrabars to display. int maxIntrabars = input.int(20, "Max intrabars", 1, 250) //@variable The width of the drawn candle bodies. int candleWidth = input.int(20, "Candle width", 2) //@variable The largest valid timeframe closest to `maxIntrabars` times smaller than the chart timeframe. string lowerTimeframe = timeframe.from_seconds(math.ceil(timeframe.in_seconds() / maxIntrabars)) //@variable An array of lines to represent intrabar wicks. var array<line> wicks = array.new<line>() //@variable An array of lines to represent intrabar bodies. var array<line> bodies = array.new<line>() //@variable A box that surrounds the displayed intrabars. var box magnifierBox = box.new(na, na, na, na, bgcolor = na) // Fill the `wicks` and `bodies` arrays with blank lines on the first bar. if barstate.isfirst for i = 1 to maxIntrabars array.push(wicks, line.new(na, na, na, na, color = color.gray)) array.push(bodies, line.new(na, na, na, na, width = candleWidth)) //@variable A tuple of "float" arrays containing `open`, `high`, `low`, and `close` prices from the `lowerTimeframe`. [oData, hData, lData, cData] = request.security_lower_tf(syminfo.tickerid, lowerTimeframe, [open, high, low, close]) //@variable The number of intrabars retrieved from the `lowerTimeframe` on the chart bar. int numIntrabars = array.size(oData) if numIntrabars > 0 // Define the start and end bar index values for intrabar display. int startIndex = bar_index + 2 int endIndex = startIndex + numIntrabars // Loop to update lines. for i = 0 to maxIntrabars - 1 line wickLine = array.get(wicks, i) line bodyLine = array.get(bodies, i) if i < numIntrabars //@variable The `bar_index` of the drawing. int candleIndex = startIndex + i // Update the properties of the `wickLine` and `bodyLine`. line.set_xy1(wickLine, startIndex + i, array.get(hData, i)) line.set_xy2(wickLine, startIndex + i, array.get(lData, i)) line.set_xy1(bodyLine, startIndex + i, array.get(oData, i)) line.set_xy2(bodyLine, startIndex + i, array.get(cData, i)) line.set_color(bodyLine, bodyLine.get_y2() > bodyLine.get_y1() ? color.teal : color.maroon) continue // Set the coordinates of the `wickLine` and `bodyLine` to `na` if no intrabar data is available at the index. line.set_xy1(wickLine, na, na) line.set_xy2(wickLine, na, na) line.set_xy1(bodyLine, na, na) line.set_xy2(bodyLine, na, na) // Set the coordinates of the `magnifierBox`. box.set_lefttop(magnifierBox, startIndex - 1, array.max(hData)) box.set_rightbottom(magnifierBox, endIndex, array.min(lData))

Note that:

- The script draws each candle using two lines: one to represent wicks and the other to represent the body. Since the script can display up to 500 lines on the chart, we've limited the `maxIntrabars` input to 250.

- The `lowerTimeframe` value is the result of calculating the math.ceil() of the timeframe.in_seconds() divided by the `maxIntrabars` and converting to a valid timeframe string with timeframe.from_seconds().
- The script sets the top of the box drawing using the array.max() of the requested `hData` array, and it sets the box's bottom using the array.min() of the requested `lData` array. As we see on the chart, these values correspond to the high and low of the chart bar.

Requesting collections

In some cases, a script might need to request collections from an intrabar context. However, in contrast to request.security(), scripts cannot use collection references or calls to functions that return them as the `expression` argument in a request.security_lower_tf() call, because arrays cannot directly store references to other collections.

Despite these limitations, it is possible to request collections from lower timeframes, if needed, with the help of *wrapper* types.

> NoticeThe technique described below is **advanced** and **not** recommended for beginners, because it requires an understanding of how user-defined types with collection fields work. When possible, use *simpler* methods to manage LTF requests. Use the following technique only if others *do not* suffice.

To make collections requestable with request.security_lower_tf(), we must create a UDT with a field to reference a collection ID. This step is necessary since arrays cannot reference other collections directly but *can* reference UDTs with collection fields:

//@type A "wrapper" type for storing an `array<float>` reference. type Wrapper array<float> collection

With our `Wrapper` UDT defined, we can now pass the IDs of objects of the UDT to the `expression` parameter in request.security_lower_tf().

A straightforward approach is to use a call to the type's built-in `*.new()` function as the `expression` argument. For example, this line of code uses a call to `Wrapper.new()` with `array.from(close)` as the `collection` argument directly within the request.security_lower_tf() call:

//@variable An array of `Wrapper` IDs requested from the 1-minute timeframe. array<Wrapper> wrappers = request.security_lower_tf(syminfo.tickerid, "1", Wrapper.new(array.from(close)))

Alternatively, we can create a user-defined function or method that returns a reference to an object of the UDT and call that function within request.security_lower_tf(). For instance, this code calls a custom `newWrapper()` function that returns a `Wrapper` ID as the `expression` argument:

//@function Creates a new `Wrapper` instance to wrap the specified `collection`. newWrapper(array<float> collection) => Wrapper.new(collection) //@variable An array of `Wrapper` IDs requested from the 1-minute timeframe. array<Wrapper> wrappers = request.security_lower_tf(syminfo.tickerid, "1", newWrapper(array.from(close)))

The result with either of the above is an array containing `Wrapper` IDs from all available intrabars in the chart bar, which the script can use to reference `Wrapper` instances from specific intrabars and use their `collection` fields in additional operations.

The script below utilizes this approach to collect the IDs of arrays containing intrabar data from a `lowerTimeframe`, then uses those arrays to display data from a specific lower-timeframe bar. Its custom `Prices` type contains a single `data` field to reference `array<float>` instances that hold price data, and the user-defined `newPrices()` function returns the ID of a `Prices` object.

The script calls request.security_lower_tf() with a `newPrices()` call as its `expression` argument to retrieve the ID of an array containing `Prices` IDs from each intrabar in the chart bar, then uses array.get() to get the ID from a specified available intrabar, if it exists. Lastly, it uses array.get() on the `data` array referenced by that instance and calls plotcandle() to display its values on the chart:

//@version=6 indicator("Requesting LTF collections demo", "Intrabar viewer", true) //@variable The timeframe of the LTF data request. string lowerTimeframe = input.timeframe("1", "Timeframe") //@variable The index of the intrabar to show on each chart bar. 0 is the first available intrabar. int intrabarIndex = input.int(0, "Intrabar to show", 0) //@variable A custom type to store an `array<float>` reference. type Prices array<float> data //@function Returns the ID of a new `Prices` instance containing current `open`, `high`, `low`, and `close` prices. newPrices() => Prices.new(array.from(open, high, low, close)) //@variable An array of `Prices` IDs requested from the `lowerTimeframe`. array<Prices> requestedPrices = request.security_lower_tf(syminfo.tickerid, lowerTimeframe, newPrices()) //@variable The `Prices` ID from the `requestedPrices` array at the `intrabarIndex`, or `na` if not available. Prices intrabarPrices = array.size(requestedPrices) > intrabarIndex ? array.get(requestedPrices, intrabarIndex) : na //@variable The `data` array ID from the `intrabarPrices` object, or the ID of an array of `na` values if `intrabarPrices` is `na`. array<float> intrabarData = na(intrabarPrices) ? array.new<float>(4, na) : intrabarPrices.data // Plot the `intrabarData` values as candles. plotcandle(intrabarData.get(0), intrabarData.get(1), intrabarData.get(2), intrabarData.get(3))

Note that:

- The `intrabarPrices` variable references a `Prices` object only if the size of the `requestedPrices` array is greater than the `intrabarIndex`, because attempting to use array.get() to retrieve an element that doesn't exist causes an out of bounds error.
- The `intrabarData` variable references an array from the `intrabarPrices.data` field only if the `intrabarPrices` variable references a `Prices` object. If `intrabarPrices` holds na because intrabar data is not available for a bar, the `intrabarData` variable references an array of na values.
- The process used in this example is *not* necessary to achieve the intended result. Instead of using UDTs, we can use the tuple `[open, high, low, close]` as the `expression` argument in the request to retrieve a tuple of arrays for further operations. See the Tuples of intrabar data section above for more information.

Custom contexts

Pine Script includes multiple `ticker.*()` functions that allow scripts to construct *custom* ticker IDs that specify additional settings for data requests when used as a `symbol` argument in request.security() and request.security_lower_tf():

- ticker.new() constructs a custom ticker ID from a specified `prefix` and `ticker` with additional `session` and `adjustment` settings.
- ticker.modify() constructs a modified form of a specified `tickerid` with additional `session` and `adjustment` settings.

- ticker.heikinashi(), ticker.renko(), ticker.pointfigure(), ticker.kagi(), and ticker.linebreak() construct a modified form a `symbol` with non-standard chart settings.
- ticker.inherit() constructs a new ticker ID for a `symbol` with additional parameters inherited from the `from_tickerid` specified in the function call, allowing scripts to request the `symbol` data with the same modifiers as the `from_tickerid`, including session, dividend adjustment, currency conversion, non-standard chart type, back-adjustment, settlement-as-close, etc.
- ticker.standard() constructs a standard ticker ID representing the `symbol` *without* additional modifiers.

Let's explore some practical examples of applying `ticker.*()` functions to request data from custom contexts.

Suppose we want to include dividend adjustment in a stock symbol's prices without enabling the "Adjust data for dividends" option in the "Symbol" section of the chart's settings. We can achieve this in a script by constructing a custom ticker ID for the instrument using ticker.new() or ticker.modify() with an `adjustment` value of adjustment.dividends.

This script creates an `adjustedTickerID` using ticker.modify(), uses that ticker ID as the `symbol` in request.security() to retrieve a tuple of adjusted price values, then uses plotcandle() to plot the result as candles on the chart. It also highlights the background of bars where the requested prices differ from the prices without dividend adjustment.

As we see on the "NYSE:XOM" chart below, enabling dividend adjustment results in different historical values before the date of the latest dividend:

//@version=6 indicator("Custom contexts demo 1", "Adjusted prices", true) //@variable A custom ticker ID representing the chart's symbol with the dividend adjustment modifier. string adjustedTickerID = ticker.modify(syminfo.tickerid, adjustment = adjustment.dividends) // Request the adjusted prices for the chart's symbol. [o, h, l, c] = request.security(adjustedTickerID, timeframe.period, [open, high, low, close]) //@variable The color of the candles on the chart. color candleColor = c > o ? color.teal : color.maroon // Plot the adjusted prices. plotcandle(o, h, l, c, "Adjusted Prices", candleColor) // Highlight the background when `c` is different from `close`. bgcolor(c != close ? color.new(color.orange, 80) : na)

Note that:

- If a modifier included in a constructed ticker ID does not apply to the symbol, the script will *ignore* that modifier when requesting data. For instance, this script will display the same values as the main chart on forex symbols such as "EURUSD".

While the example above demonstrates a simple way to modify the chart's symbol, a more frequent use case for `ticker.*()` functions is applying custom modifiers to another symbol while requesting data. If a ticker ID referenced in a script already has the modifiers one would like to apply (e.g., adjustment settings, session type, etc.), they can use ticker.inherit() to quickly and efficiently add those modifiers to another symbol.

In the example below, we've edited the previous script to request data for a `symbolInput` using modifiers inherited from the `adjustedTickerID`. This script calls ticker.inherit() to construct an `inheritedTickerID` and uses that ticker ID in a request.security() call. It also requests data for the `symbolInput` without additional modifiers and plots candles for both ticker IDs in a separate chart pane to compare the difference.

As shown on the chart, the data requested using the `inheritedTickerID` includes dividend adjustment, whereas the data requested using the `symbolInput` directly does not:

```
//@version=6 indicator("Custom contexts demo 2", "Inherited adjustment") //@variable The symbol to request data from. string symbolInput = input.symbol("NYSE:PFE", "Symbol") //@variable A custom ticker ID representing the chart's symbol with the dividend adjustment modifier. string adjustedTickerID = ticker.modify(syminfo.tickerid, adjustment = adjustment.dividends) //@variable A custom ticker ID representing the `symbolInput` with modifiers inherited from the `adjustedTickerID`. string inheritedTickerID = ticker.inherit(adjustedTickerID, symbolInput) // Request prices using the `symbolInput`. [o1, h1, l1, c1] = request.security(symbolInput, timeframe.period, [open, high, low, close]) // Request prices using the `inheritedTickerID`. [o2, h2, l2, c2] = request.security(inheritedTickerID, timeframe.period, [open, high, low, close]) //@variable The color of the candles that use the `inheritedTickerID` prices. color candleColor = c2 > o2 ? color.teal : color.maroon // Plot the `symbol` prices. plotcandle(o1, h1, l1, c1, "Symbol", color.gray, color.gray, bordercolor = color.gray) // Plot the `inheritedTickerID` prices. plotcandle(o2, h2, l2, c2, "Symbol With Modifiers", candleColor) // Highlight the background when `c1` is different from `c2`. bgcolor(c1 != c2 ? color.new(color.orange, 80) : na)
```

Note that:

- Since the `adjustedTickerID` represents a modified form of the syminfo.tickerid, if we modify the chart's context in other ways, such as changing the chart type or enabling extended trading hours in the chart's settings, those modifiers will also apply to the `adjustedTickerID` and `inheritedTickerID`. However, they will *not* apply to the `symbolInput` since it represents a *standard* ticker ID.

Another frequent use case for requesting custom contexts is retrieving data that uses non-standard chart calculations. For example, suppose we want to use Renko price values to calculate trade signals in a strategy() script. If we simply change the chart type to "Renko" to get the prices, the strategy will also simulate its trades based on those synthetic prices, producing misleading results:

```
//@version=6 strategy( "Custom contexts demo 3", "Renko strategy", true, default_qty_type = strategy.percent_of_equity, default_qty_value = 2, initial_capital = 50000, slippage = 2, commission_type = strategy.commission.cash_per_contract, commission_value = 1, margin_long = 100, margin_short = 100 ) //@variable When `true`, the strategy places a long market order. bool longEntry = ta.crossover(close, open) //@variable When `true`, the strategy places a short market order. bool shortEntry = ta.crossunder(close, open) if longEntry strategy.entry("Long Entry", strategy.long) if shortEntry strategy.entry("Short Entry", strategy.short)
```

To ensure our strategy shows results based on *actual* prices, we can create a Renko ticker ID using ticker.renko() while keeping the chart on a *standard type*, allowing the script to request and use Renko prices to calculate its signals without calculating the strategy results on them:

```
//@version=6 strategy( "Custom contexts demo 3", "Renko strategy", true, default_qty_type = strategy.percent_of_equity, default_qty_value = 2, initial_capital = 50000, slippage = 1, commission_type = strategy.commission.cash_per_contract, commission_value = 1, margin_long = 100, margin_short = 100 ) //@variable A Renko ticker ID. string renkoTickerID = ticker.renko(syminfo.tickerid, "ATR", 14) // Request the `open` and `close` prices using the `renkoTickerID`. [renkoOpen, renkoClose] = request.security(renkoTickerID, timeframe.period, [open, close]) //@variable When `true`, the strategy places a long market order. bool longEntry = ta.crossover(renkoClose, renkoOpen) //@variable When `true`, the strategy places a short market order. bool shortEntry = ta.crossunder(renkoClose, renkoOpen) if longEntry strategy.entry("Long Entry", strategy.long) if shortEntry strategy.entry("Short Entry", strategy.short) plot(renkoOpen) plot(renkoClose)
```

Historical and realtime behavior

Functions in the `request.*()` namespace can behave differently on historical and realtime bars. This behavior is closely related to Pine's Execution model.

Consider how a script behaves within the main context. Throughout the chart's history, the script calculates its required values once and *commits* them to that bar so their states are accessible on subsequent executions. On an unconfirmed bar, however, the script recalculates its values on *each update* to the bar's data to align with realtime changes. Before recalculating the values on that bar, it reverts calculated values to their last committed states, otherwise known as *rollback*, and it only commits values to that bar once the bar closes.

Now consider the behavior of data requests from other contexts with request.security(). As when evaluating historical bars in the main context, request.security() only returns new historical values when it confirms a bar in its specified context. When executing on realtime bars, it returns recalculated values on each chart bar, similar to how a script recalculates values in the main context on the open chart bar.

However, the function only *confirms* the requested values when a bar from its context closes. When the script restarts, what were previously *realtime* bars become *historical* bars. Therefore, request.security() only returns the values it confirmed on those bars. In essence, this behavior means that requested data may *repaint* when its values fluctuate on realtime bars without confirmation from the context.

> TipIt is often helpful to distinguish historical bars from realtime bars when working with `request.*()` functions. Scripts can determine whether bars have historical or realtime states by using the barstate.ishistory and barstate.isrealtime variables.

In most circumstances where a script requests data from a broader context, one will typically require confirmed, stable values that *do not* fluctuate on realtime bars. The section below explains how to achieve such a result and avoid repainting data requests.

Avoiding repainting

Higher-timeframe data

When requesting values from a higher timeframe, they are subject to repainting since realtime bars can contain *unconfirmed* information from developing HTF bars, and the script may adjust the times that new values come in on historical bars. To avoid repainting HTF data, one must ensure that the function only returns confirmed values with consistent timing on all bars, regardless of bar state.

The most reliable approach to achieve non-repainting results is to use an `expression` argument that only references past bars (e.g., `close[1]`) while using barmerge.lookahead_on as the `lookahead` value.

Using barmerge.lookahead_on with non-offset HTF data requests is discouraged since it prompts request.security() to "look ahead" to the final values of an HTF bar, retrieving confirmed values *before* they're actually available in the script's history. However, if the values used in the `expression` are offset by at least one bar, the "future" data the function retrieves is no longer from the future. Instead, the data represents confirmed values from established, *available* HTF bars. In other words, applying an offset to the

`expression` effectively prevents the requested data from repainting when the script restarts its executions and eliminates lookahead bias in the historical series.

The following example demonstrates a repainting HTF data request. The script uses request.security() without offset modifications or additional arguments to retrieve the results of a ta.wma() call from a higher timeframe. It also highlights the background to indicate which bars were in a realtime state during its calculations.

As shown on the chart below, the plot of the requested WMA only changes on historical bars when HTF bars close, whereas it fluctuates on all realtime bars since the data includes unconfirmed values from the higher timeframe:

//@version=6 indicator("Avoiding HTF repainting demo", overlay = true) //@variable The multiplier applied to the chart's timeframe. int tfMultiplier = input.int(10, "Timeframe multiplier", 1) //@variable The number of bars in the moving average. int length = input.int(5, "WMA smoothing length") //@variable The valid timeframe string closest to `tfMultiplier` times larger than the chart timeframe. string timeframe = timeframe.from_seconds(timeframe.in_seconds() * tfMultiplier) //@variable The weighted MA of `close` prices over `length` bars on the `timeframe`. // This request repaints because it includes unconfirmed HTF data on realtime bars and it may offset the // times of its historical results. float requestedWMA = request.security(syminfo.tickerid, timeframe, ta.wma(close, length)) // Plot the requested series. plot(requestedWMA, "HTF WMA", color.purple, 3) // Highlight the background on realtime bars. bgcolor(barstate.isrealtime ? color.new(color.orange, 70) : na, title = "Realtime bar highlight")

To avoid repainting in this script, we can add `lookahead = barmerge.lookahead_on` to the request.security() call and offset the call history of ta.wma() by one bar with the history-referencing operator [], ensuring the request always retrieves the last confirmed HTF bar's WMA at the start of each new `timeframe`. Unlike the previous script, this version has consistent behavior on historical and realtime bar states, as we see below:

//@version=6 indicator("Avoiding HTF repainting demo", overlay = true) //@variable The multiplier applied to the chart's timeframe. int tfMultiplier = input.int(10, "Timeframe multiplier", 1) //@variable The number of bars in the moving average. int length = input.int(5, "WMA smoothing length") //@variable The valid timeframe string closest to `tfMultiplier` times larger than the chart timeframe. string timeframe = timeframe.from_seconds(timeframe.in_seconds() * tfMultiplier) //@variable The weighted MA of `close` prices over `length` bars on the `timeframe`. // This request does not repaint, as it always references the last confirmed WMA value on all bars. float requestedWMA = request.security(syminfo.tickerid, timeframe, ta.wma(close, length)[1], lookahead = barmerge.lookahead_on) // Plot the requested value. plot(requestedWMA, "HTF WMA", color.purple, 3) // Highlight the background on realtime bars. bgcolor(barstate.isrealtime ? color.new(color.orange, 70) : na, title = "Realtime bar highlight")

Lower-timeframe data

The request.security() and request.security_lower_tf() functions can retrieve data from lower-timeframe contexts. The request.security() function can only retrieve data from a *single* intrabar in each chart bar, and request.security_lower_tf() retrieves data from *all* available intrabars.

When using these functions to retrieve intrabar data, it's important to note that such requests are **not** immune to repainting behavior. Historical and realtime series often rely on *separate* data feeds. Data providers may retroactively modify realtime data, and it's possible for races to occur in realtime data feeds,

as explained in the Data feeds section of this page. Either case may result in intrabar data retrieved on realtime bars repainting after the script restarts its executions.

Additionally, a particular case that *will* cause repainting LTF requests is using request.security() with barmerge.lookahead_on to retrieve data from the first intrabar in each chart bar. While it will generally work as expected on historical bars, it will track only the most recent intrabar on realtime bars, as request.security() does not retain all intrabar information, and the intrabars the function retrieves on realtime bars are unsorted until restarting the script:

//@version=6 indicator("Avoiding LTF repainting demo", overlay = true) //@variable The lower timeframe of the requested data. string lowerTimeframe = input.timeframe("1", "Timeframe") //@variable The first intrabar `close` requested from the `lowerTimeframe` on each bar. // Only works as intended on historical bars. float requestedClose = request.security(syminfo.tickerid, lowerTimeframe, close, lookahead = barmerge.lookahead_on) // Plot the `requestedClose`. plot(requestedClose, "First intrabar close", linewidth = 3) // Highlight the background on realtime bars. bgcolor(barstate.isrealtime ? color.new(color.orange, 60) : na, title = "Realtime bar Highlight")

One can mitigate this behavior and track the values from the first intrabar, or any available intrabar in the chart bar, by using request.security_lower_tf() since it maintains an array of intrabar values ordered by the times they come in. Here, we call array.first() on a requested array of intrabar data to retrieve the close price from the first available intrabar in each chart bar:

//@version=6 indicator("Avoiding LTF repainting demo", overlay = true) //@variable The lower timeframe of the requested data. string lowerTimeframe = input.timeframe("1", "Timeframe") //@variable An array of intrabar `close` values requested from the `lowerTimeframe` on each bar. array<float> requestedCloses = request.security_lower_tf(syminfo.tickerid, lowerTimeframe, close) //@variable The first intrabar `close` on each bar with available data. float firstClose = requestedCloses.size() > 0 ? requestedCloses.first() : na // Plot the `firstClose`. plot(firstClose, "First intrabar close", linewidth = 3) // Highlight the background on realtime bars. bgcolor(barstate.isrealtime ? color.new(color.orange, 60) : na, title = "Realtime bar Highlight")

Note that:

- While request.security_lower_tf() is more optimized for handling historical and realtime intrabars, it's still possible in some cases for minor repainting to occur due to data differences from the provider, as outlined above.
- This code may not show intrabar data on all available chart bars, depending on how many intrabars each chart bar contains, as `request.*()` functions can retrieve up to 200,000 intrabars from an LTF context. The maximum number of requestable intrabars depends on the user's plan. See this section of the Limitations page for more information.

`request.currency_rate()`

When a script needs to convert values expressed in one currency to another, one can use request.currency_rate(). This function requests a *daily rate* for currency conversion calculations based on currency pair or spread data from the most popular exchanges, providing a simpler alternative to fetching specific pairs or spreads with request.security().

While one can use request.security() to retrieve daily currency rates, its use case is more involved than request.currency_rate(), as one needs to supply a valid *ticker ID* for a currency pair or spread to request the

rate. Additionally, a historical offset and barmerge.lookahead_on are necessary to prevent the results from repainting, as explained in this section.

The request.currency_rate() function, on the other hand, only requires *currency codes*. No ticker ID is needed when requesting rates with this function, and it ensures non-repainting results without requiring additional specification.

The function's signature is as follows:

```
request.currency_rate(from, to, ignore_invalid_currency) → series float
```

The `from` parameter specifies the currency to convert, and the `to` parameter specifies the target currency. Both parameters accept "string" values representing valid currency codes (e.g., "USD") or any built-in `currency.*` variable (e.g., currency.USD).

When the function cannot calculate a valid conversion rate between the specified `from` and `to` currencies, programmers can decide whether it raises a runtime error or returns na via the `ignore_invalid_currency` parameter. The default value is `false`, meaning the function raises a runtime error and halts the script's executions.

The following example demonstrates a simple use case for request.currency_rate(). Suppose we want to convert values expressed in Turkish lira (currency.TRY) to South Korean won (currency.KRW) using a daily conversion rate. If we use request.security() to retrieve the rate, we must supply a valid ticker ID and request the last confirmed close from the previous day.

In this case, no valid symbol exists that would allow us to retrieve a conversion rate directly with request.security(). Therefore, we first need a ticker ID for a spread that converts TRY to an intermediate currency, such as USD, then converts the intermediate currency to KRW. We can then use that ticker ID within request.security() with `close[1]` as the `expression` and barmerge.lookahead_on as the `lookahead` value to request a non-repainting daily rate.

Alternatively, we can achieve the same result more simply by calling request.currency_rate(). This function does all the heavy lifting for us, only requiring `from` and `to` currency arguments to perform its calculation.

As we see below, both approaches return the same daily rate:

//@version=6 indicator("Requesting currency rates demo") //@variable The currency to convert. simple string fromCurrency = currency.TRY //@variable The resulting currency. simple string toCurrency = currency.KRW //@variable The spread symbol to request. Required in `request.security()` because no direct symbol exists. simple string spreadSymbol = str.format("{0}{2} * {2}{1}", fromCurrency, toCurrency, currency.USD) //@variable The non-repainting conversion rate from `request.security()` using the `spreadSymbol`. float securityRequestedRate = request.security(spreadSymbol, "1D", close[1], lookahead = barmerge.lookahead_on) //@variable The non-repainting conversion rate from `request.currency_rate()`. float nonSecurityRequestedRate = request.currency_rate(fromCurrency, toCurrency) // Plot the requested rates. We can multiply TRY values by these rates to convert them to KRW. plot(securityRequestedRate, "`request.security()` value", color.purple, 5) plot(nonSecurityRequestedRate, "`request.currency_rate()` value", color.yellow, 2)

`request.dividends()`, `request.splits()`, and `request.earnings()`

Analyzing a stock's earnings data and corporate actions provides helpful insights into its underlying financial strength. Pine Script provides the ability to retrieve essential information about applicable stocks via request.dividends(), request.splits(), and request.earnings().

These are the functions' signatures:

```
request.dividends(ticker, field, gaps, lookahead, ignore_invalid_symbol, curren

request.splits(ticker, field, gaps, lookahead, ignore_invalid_symbol) → series

request.earnings(ticker, field, gaps, lookahead, ignore_invalid_symbol, currenc
```

Each function has the same parameters in its signature, with the exception of request.splits(), which doesn't have a `currency` parameter.

Note that unlike the `symbol` parameter in other `request.*()` functions, the `ticker` parameter in these functions only accepts an *"Exchange:Symbol" pair*, such as "NASDAQ:AAPL". The built-in syminfo.ticker variable does not work with these functions since it does not contain exchange information. Instead, one must use syminfo.tickerid for such cases.

The `field` parameter determines the data the function will retrieve. Each of these functions accepts different built-in variables as the `field` argument since each requests different information about a stock:

- The request.dividends() function retrieves current dividend information for a stock, i.e., the amount per share the issuing company paid out to investors who purchased shares before the ex-dividend date. Passing the built-in dividends.gross or dividends.net variables to the `field` parameter specifies whether the returned value represents dividends before or after factoring in expenses the company deducts from its payouts.
- The request.splits() function retrieves current split and reverse split information for a stock. A split occurs when a company increases its outstanding shares to promote liquidity. A reverse split occurs when a company consolidates its shares and offers them at a higher price to attract specific investors or maintain their listing on a market that has a minimum per-share price. Companies express their split information as *ratios*. For example, a 5:1 split means the company issued additional shares to its shareholders so that they have five times the number of shares they had before the split, and the raw price of each share becomes one-fifth of the previous price. Passing splits.numerator or splits.denominator to the `field` parameter of request.splits() determines whether it returns the numerator or denominator of the split ratio.
- The request.earnings() function retrieves the earnings per share (EPS) information for a stock `ticker`'s issuing company. The EPS value is the ratio of a company's net income to the number of outstanding stock shares, which investors consider an indicator of the company's profitability. Passing earnings.actual, earnings.estimate, or earnings.standardized as the `field` argument in request.earnings() respectively determines whether the function requests the actual, estimated, or standardized EPS value.

For a detailed explanation of the `gaps`, `lookahead`, and `ignore_invalid_symbol` parameters of these functions, see the Common characteristics section at the top of this page.

It's important to note that the values returned by these functions reflect the data available as it comes in. This behavior differs from financial data originating from a request.financial() call in that the underlying data from such calls becomes available according to a company's fiscal reporting period.

> TipScripts can also retrieve information about upcoming earnings and dividends for an instrument via the `earnings.future_*` and `dividends.future_*` built-in variables.

Here, we've included an example that displays a handy table containing the most recent dividend, split, and EPS data. The script calls the `request.*()` functions discussed in this section to retrieve the data, then converts the values to "strings" with `str.*()` functions and displays the results in the `infoTable` with table.cell():

//@version=6 indicator("Dividends, splits, and earnings demo", overlay = true) //@variable The size of the table's text. string tableSize = input.string(size.large, "Table size", [size.auto, size.tiny, size.small, size.normal, size.large, size.huge]) //@variable The color of the table's text and frame. var color tableColor = chart.fg_color //@variable A `table` displaying the latest dividend, split, and EPS information. var table infoTable = table.new(position.top_right, 3, 4, frame_color = tableColor, frame_width = 1) // Add header cells on the first bar. if barstate.isfirst table.cell(infoTable, 0, 0, "Field", text_color = tableColor, text_size = tableSize) table.cell(infoTable, 1, 0, "Value", text_color = tableColor, text_size = tableSize) table.cell(infoTable, 2, 0, "Date", text_color = tableColor, text_size = tableSize) table.cell(infoTable, 0, 1, "Dividend", text_color = tableColor, text_size = tableSize) table.cell(infoTable, 0, 2, "Split", text_color = tableColor, text_size = tableSize) table.cell(infoTable, 0, 3, "EPS", text_color = tableColor, text_size = tableSize) //@variable The amount of the last reported dividend as of the current bar. float latestDividend = request.dividends(syminfo.tickerid, dividends.gross, barmerge.gaps_on) //@variable The numerator of that last reported split ratio as of the current bar. float latestSplitNum = request.splits(syminfo.tickerid, splits.numerator, barmerge.gaps_on) //@variable The denominator of the last reported split ratio as of the current bar. float latestSplitDen = request.splits(syminfo.tickerid, splits.denominator, barmerge.gaps_on) //@variable The last reported earnings per share as of the current bar. float latestEPS = request.earnings(syminfo.tickerid, earnings.actual, barmerge.gaps_on) // Update the "Value" and "Date" columns when new values come in. if not na(latestDividend) table.cell(infoTable, 1, 1, str.tostring(math.round(latestDividend, 3)), text_color = tableColor, text_size = tableSize) table.cell(infoTable, 2, 1, str.format_time(time, "yyyy-MM-dd"), text_color = tableColor, text_size = tableSize) if not na(latestSplitNum) table.cell(infoTable, 1, 2, str.format("{0}-for-{1}", latestSplitNum, latestSplitDen), text_color = tableColor, text_size = tableSize) table.cell(infoTable, 2, 2, str.format_time(time, "yyyy-MM-dd"), text_color = tableColor, text_size = tableSize) if not na(latestEPS) table.cell(infoTable, 1, 3, str.tostring(latestEPS), text_color = tableColor, text_size = tableSize) table.cell(infoTable, 2, 3, str.format_time(time, "yyyy-MM-dd"), text_color = tableColor, text_size = tableSize)

Note that:

- We've included barmerge.gaps_on in the `request.*()` calls, so they only return values when new data is available. Otherwise, they return na.
- The script assigns a table ID to the `infoTable` variable on the first chart bar. On subsequent bars, it updates necessary cells with new information whenever data is available.
- If no information is available from any of the `request.*()` calls throughout the chart's history (e.g., if the `ticker` has no dividend information), the script does not initialize the corresponding cells since it's unnecessary.

`request.financial()`

Financial metrics provide investors with insights about a company's economic and financial health that are not tangible from solely analyzing its stock prices. TradingView offers a wide variety of financial metrics from FactSet that traders can access via the "Financials" tab in the "Indicators" menu of the chart. Scripts can access available metrics for an instrument directly via the request.financial() function.

This is the function's signature:

```
request.financial(symbol, financial_id, period, gaps, ignore_invalid_symbol, cu
```

As with the first parameter in request.dividends(), request.splits(), and request.earnings(), the `symbol` parameter in request.financial() requires an *"Exchange:Symbol" pair*. To request financial information for the chart's ticker ID, use syminfo.tickerid, as syminfo.ticker will not work.

The `financial_id` parameter accepts a "string" value representing the ID of the requested financial metric. TradingView has numerous financial metrics to choose from. See the Financial IDs section below for an overview of all accessible metrics and their "string" identifiers.

The `period` parameter specifies the fiscal period for which new requested data comes in. It accepts one of the following "string" arguments: **"FQ" (quarterly), "FH" (semiannual), "FY" (annual), or "TTM" (trailing twelve months)**. Not all fiscal periods are available for all metrics or instruments. To confirm which periods are available for specific metrics, see the second column of the tables in the Financial IDs section.

See this page's Common characteristics section for a detailed explanation of this function's `gaps`, `ignore_invalid_symbol`, and `currency` parameters.

It's important to note that the data retrieved from this function comes in at a *fixed frequency*, independent of the precise date on which the data is made available within a fiscal period. For a company's dividends, splits, and earnings per share (EPS) information, one can request data reported on exact dates via request.dividends(), request.splits(), and request.earnings().

This script uses request.financial() to retrieve information about the income and expenses of a stock's issuing company and visualize the profitability of its typical business operations. It requests the "OPER_INCOME", "TOTAL_REVENUE", and "TOTAL_OPER_EXPENSE" financial IDs for the syminfo.tickerid over the latest `fiscalPeriod`, then plots the results on the chart:

//@version=6 indicator("Requesting financial data demo", format = format.volume) //@variable The size of the fiscal reporting period. Some options may not be available, depending on the instrument. string fiscalPeriod = input.string("FQ", "Period", ["FQ", "FH", "FY", "TTM"]) //@variable The operating income after expenses reported for the stock's issuing company. float operatingIncome = request.financial(syminfo.tickerid, "OPER_INCOME", fiscalPeriod) //@variable The total revenue reported for the stock's issuing company. float totalRevenue = request.financial(syminfo.tickerid, "TOTAL_REVENUE", fiscalPeriod) //@variable The total operating expenses reported for the stock's issuing company. float totalExpenses = request.financial(syminfo.tickerid, "TOTAL_OPER_EXPENSE", fiscalPeriod) //@variable Is aqua when the `totalRevenue` exceeds the `totalExpenses`, fuchsia otherwise. color incomeColor = operatingIncome > 0 ? color.new(color.aqua, 50) : color.new(color.fuchsia, 50) // Display the requested data. plot(operatingIncome, "Operating income", incomeColor, 1, plot.style_area)

plot(totalRevenue, "Total revenue", color.green, 3) plot(totalExpenses, "Total operating expenses", color.red, 3)

Note that:

- Not all `fiscalPeriod` options are available for every ticker ID. For example, companies in the US typically publish *quarterly* reports, whereas many European companies publish *semiannual* reports. See this page in our Help Center for more information.

Calculating financial metrics

The request.financial() function can provide scripts with numerous useful financial metrics that don't require additional calculations. However, some commonly used financial estimates require combining an instrument's current market price with requested financial data. Such is the case for:

- Market Capitalization (market price * total shares outstanding)
- Earnings Yield (12-month EPS / market price)
- Price-to-Book Ratio (market price / BVPS)
- Price-to-Earnings Ratio (market price / EPS)
- Price-to-Sales Ratio (market cap / 12-month total revenue)

The following script contains user-defined functions that calculate the above financial metrics for the syminfo.tickerid. We've created these functions so users can easily copy them into their scripts. This example uses them within a str.format() call to construct a `tooltipText`, which it displays in tooltips on the chart using labels. Hovering over any bar's label will expose the tooltip containing the metrics calculated on that bar:

//@version=6 indicator("Calculating financial metrics demo", overlay = true, max_labels_count = 500) //@function Calculates the market capitalization (market cap) for the chart's symbol. marketCap() => //@variable The most recent number of outstanding shares reported for the symbol. float totalSharesOutstanding = request.financial(syminfo.tickerid, "TOTAL_SHARES_OUTSTANDING", "FQ") // Return the market cap value. totalSharesOutstanding * close //@function Calculates the Earnings Yield for the chart's symbol. earningsYield() => //@variable The most recent 12-month earnings per share reported for the symbol. float eps = request.financial(syminfo.tickerid, "EARNINGS_PER_SHARE", "TTM") //Return the Earnings Yield percentage. 100.0 * eps / close //@function Calculates the Price-to-Book (P/B) ratio for the chart's symbol. priceBookRatio() => //@variable The most recent Book Value Per Share (BVPS) reported for the symbol. float bookValuePerShare = request.financial(syminfo.tickerid, "BOOK_VALUE_PER_SHARE", "FQ") // Return the P/B ratio. close / bookValuePerShare //@function Calculates the Price-to-Earnings (P/E) ratio for the chart's symbol. priceEarningsRatio() => //@variable The most recent 12-month earnings per share reported for the symbol. float eps = request.financial(syminfo.tickerid, "EARNINGS_PER_SHARE", "TTM") // Return the P/E ratio. close / eps //@function Calculates the Price-to-Sales (P/S) ratio for the chart's symbol. priceSalesRatio() => //@variable The most recent number of outstanding shares reported for the symbol. float totalSharesOutstanding = request.financial(syminfo.tickerid, "TOTAL_SHARES_OUTSTANDING", "FQ") //@variable The most recent 12-month total revenue reported for the symbol. float totalRevenue = request.financial(syminfo.tickerid, "TOTAL_REVENUE", "TTM") // Return the P/S ratio. totalSharesOutstanding * close / totalRevenue //@variable The text to display in label tooltips. string tooltipText = str.format("Market Cap: {0} {1}\nEarnings Yield: {2}%\nP/B Ratio: {3}\nP/E Ratio: {4}\nP/S Ratio: {5}", str.tostring(marketCap(), format.volume), syminfo.currency, earningsYield(), priceBookRatio(), priceEarningsRatio(), priceSalesRatio()) //@variable

Displays a blank label with a tooltip containing the `tooltipText`. label info = label.new(chart.point.now(high), tooltip = tooltipText)

Note that:

- Since not all companies publish quarterly financial reports, one may need to change the “FQ” in these functions to match the minimum reporting period for a specific company, as the request.financial() calls will return na when “FQ” data isn’t available.

Financial IDs

Below is an overview of all financial metrics one can request via request.financial(), along with the periods in which reports may be available. We’ve divided this information into four tables corresponding to the categories displayed in the “Financials” section of the “Indicators” menu:

- Income statements
- Balance sheet
- Cash flow
- Statistics

Each table has the following three columns:

- The first column contains descriptions of each metric with links to Help Center pages for additional information.
- The second column lists the possible `period` arguments allowed for the metric. Note that all available values may not be compatible with specific ticker IDs, e.g., while “FQ” may be a possible argument, it will not work if the issuing company does not publish quarterly data.
- The third column lists the “string” IDs for the `financial_id` argument in request.financial().

> TipThe tables in these sections are quite lengthy, because there are many `financial_id` arguments available. Use the **“Click to show/hide”** option above each table to toggle its visibility.

Income statements

This table lists the available metrics that provide information about a company’s income, costs, profits and losses.

▼ Click to show/hide

Financial	`period`	`financial_id`
After tax other income/expense	FQ, FH, FY, TTM	AFTER_TAX_OTHER_INCOME
Average basic shares outstanding	FQ, FH, FY	BASIC_SHARES_OUTSTANDING
Basic earnings per share (Basic EPS)	FQ, FH, FY, TTM	EARNINGS_PER_SHARE_BASIC
Cost of goods sold	FQ, FH, FY, TTM	COST_OF_GOODS
Deprecation and amortization	FQ, FH, FY, TTM	DEP_AMORT_EXP_INCOME_S
Diluted earnings per share (Diluted EPS)	FQ, FH, FY, TTM	EARNINGS_PER_SHARE_DILUTED

Financial	period	financial_id
Diluted net income available to common stockholders	FQ, FH, FY, TTM	DILUTED_NET_INCOME
Diluted shares outstanding	FQ, FH, FY	DILUTED_SHARES_OUTSTANDING
Dilution adjustment	FQ, FH, FY, TTM	DILUTION_ADJUSTMENT
Discontinued operations	FQ, FH, FY, TTM	DISCONTINUED_OPERATIONS
EBIT	FQ, FH, FY, TTM	EBIT
EBITDA	FQ, FH, FY, TTM	EBITDA
Equity in earnings	FQ, FH, FY, TTM	EQUITY_IN_EARNINGS
Gross profit	FQ, FH, FY, TTM	GROSS_PROFIT
Interest capitalized	FQ, FH, FY, TTM	INTEREST_CAPITALIZED
Interest expense on debt	FQ, FH, FY, TTM	INTEREST_EXPENSE_ON_DEBT
Interest expense, net of interest capitalized	FQ, FH, FY, TTM	NON_OPER_INTEREST_EXP
Miscellaneous non-operating expense	FQ, FH, FY, TTM	OTHER_INCOME
Net income	FQ, FH, FY, TTM	NET_INCOME
Net income before discontinued operations	FQ, FH, FY, TTM	NET_INCOME_BEF_DISC_OPER
Non-controlling/minority interest	FQ, FH, FY, TTM	MINORITY_INTEREST_EXP
Non-operating income, excl. interest expenses	FQ, FH, FY, TTM	NON_OPER_INCOME
Non-operating income, total	FQ, FH, FY, TTM	TOTAL_NON_OPER_INCOME
Non-operating interest income	FQ, FH, FY, TTM	NON_OPER_INTEREST_INCOME
Operating expenses (excl. COGS)	FQ, FH, FY, TTM	OPERATING_EXPENSES
Operating income	FQ, FH, FY, TTM	OPER_INCOME
Other cost of goods sold	FQ, FH, FY, TTM	COST_OF_GOODS_EXCL_DEP_AMORT
Other operating expenses, total	FQ, FH, FY, TTM	OTHER_OPER_EXPENSE_TOTAL
Preferred dividends	FQ, FH, FY, TTM	PREFERRED_DIVIDENDS
Pretax equity in earnings	FQ, FH, FY, TTM	PRETAX_EQUITY_IN_EARNINGS
Pretax income	FQ, FH, FY, TTM	PRETAX_INCOME
Research & development	FQ, FH, FY, TTM	RESEARCH_AND_DEV
Selling/general/admin expenses, other	FQ, FH, FY, TTM	SELL_GEN_ADMIN_EXP_OTHER
Selling/general/admin expenses, total	FQ, FH, FY, TTM	SELL_GEN_ADMIN_EXP_TOTAL
Taxes	FQ, FH, FY, TTM	INCOME_TAX
Total operating expenses	FQ, FH, FY, TTM	TOTAL_OPER_EXPENSE
Total revenue	FQ, FH, FY, TTM	TOTAL_REVENUE
Unusual income/expense	FQ, FH, FY, TTM	UNUSUAL_EXPENSE_INC

Balance sheet

This table lists the metrics that provide information about a company's capital structure.

▼ Click to show/hide

Financial	`period`	`financial_id`
Accounts payable	FQ, FH, FY	ACCOUNTS_PAYABLE
Accounts receivable - trade, net	FQ, FH, FY	ACCOUNTS_RECEIVABLES_NET
Accrued payroll	FQ, FH, FY	ACCRUED_PAYROLL
Accumulated depreciation, total	FQ, FH, FY	ACCUM_DEPREC_TOTAL
Additional paid-in capital/Capital surplus	FQ, FH, FY	ADDITIONAL_PAID_IN_CAPITAL
Book value per share	FQ, FH, FY	BOOK_VALUE_PER_SHARE
Capital and operating lease obligations	FQ, FH, FY	CAPITAL_OPERATING_LEASE_OBLIGATIONS
Capitalized lease obligations	FQ, FH, FY	CAPITAL_LEASE_OBLIGATIONS
Cash & equivalents	FQ, FH, FY	CASH_N_EQUIVALENTS
Cash and short term investments	FQ, FH, FY	CASH_N_SHORT_TERM_INVEST
Common equity, total	FQ, FH, FY	COMMON_EQUITY_TOTAL
Common stock par/Carrying value	FQ, FH, FY	COMMON_STOCK_PAR
Current portion of LT debt and capital leases	FQ, FH, FY	CURRENT_PORT_DEBT_CAPITAL_LEASES
Deferred income, current	FQ, FH, FY	DEFERRED_INCOME_CURRENT
Deferred income, non-current	FQ, FH, FY	DEFERRED_INCOME_NON_CURRENT
Deferred tax assets	FQ, FH, FY	DEFERRED_TAX_ASSESTS
Deferred tax liabilities	FQ, FH, FY	DEFERRED_TAX_LIABILITIES
Dividends payable	FY	DIVIDENDS_PAYABLE
Goodwill, net	FQ, FH, FY	GOODWILL
Gross property/plant/equipment	FQ, FH, FY	PPE_TOTAL_GROSS
Income tax payable	FQ, FH, FY	INCOME_TAX_PAYABLE
Inventories - finished goods	FQ, FH, FY	INVENTORY_FINISHED_GOODS
Inventories - progress payments & other	FQ, FH, FY	INVENTORY_PROGRESS_PAYMENTS
Inventories - raw materials	FQ, FH, FY	INVENTORY_RAW_MATERIALS
Inventories - work in progress	FQ, FH, FY	INVENTORY_WORK_IN_PROGRESS
Investments in unconsolidated subsidiaries	FQ, FH, FY	INVESTMENTS_IN_UNCONCSOLIDATE
Long term debt	FQ, FH, FY	LONG_TERM_DEBT
Long term debt excl. lease liabilities	FQ, FH, FY	LONG_TERM_DEBT_EXCL_CAPITAL_LEASE
Long term investments	FQ, FH, FY	LONG_TERM_INVESTMENTS
Minority interest	FQ, FH, FY	MINORITY_INTEREST
Net debt	FQ, FH, FY	NET_DEBT
Net intangible assets	FQ, FH, FY	INTANGIBLES_NET
Net property/plant/equipment	FQ, FH, FY	PPE_TOTAL_NET
Note receivable - long term	FQ, FH, FY	LONG_TERM_NOTE_RECEIVABLE
Notes payable	FY	NOTES_PAYABLE_SHORT_TERM_DEBT

Financial	`period`	`financial_id`
Operating lease liabilities	FQ, FH, FY	OPERATING_LEASE_LIABILITIES
Other common equity	FQ, FH, FY	OTHER_COMMON_EQUITY
Other current assets, total	FQ, FH, FY	OTHER_CURRENT_ASSETS_TOTAL
Other current liabilities	FQ, FH, FY	OTHER_CURRENT_LIABILITIES
Other intangibles, net	FQ, FH, FY	OTHER_INTANGIBLES_NET
Other investments	FQ, FH, FY	OTHER_INVESTMENTS
Other long term assets, total	FQ, FH, FY	LONG_TERM_OTHER_ASSETS_TOTAL
Other non-current liabilities, total	FQ, FH, FY	OTHER_LIABILITIES_TOTAL
Other receivables	FQ, FH, FY	OTHER_RECEIVABLES
Other short term debt	FY	OTHER_SHORT_TERM_DEBT
Paid in capital	FQ, FH, FY	PAID_IN_CAPITAL
Preferred stock, carrying value	FQ, FH, FY	PREFERRED_STOCK_CARRYING_VALUE
Prepaid expenses	FQ, FH, FY	PREPAID_EXPENSES
Provision for risks & charge	FQ, FH, FY	PROVISION_F_RISKS
Retained earnings	FQ, FH, FY	RETAINED_EARNINGS
Shareholders' equity	FQ, FH, FY	SHRHLDRS_EQUITY
Short term debt	FQ, FH, FY	SHORT_TERM_DEBT
Short term debt excl. current portion of LT debt	FQ, FH, FY	SHORT_TERM_DEBT_EXCL_CURRENT_PORT
Short term investments	FQ, FH, FY	SHORT_TERM_INVEST
Tangible book value per share	FQ, FH, FY	BOOK_TANGIBLE_PER_SHARE
Total assets	FQ, FH, FY	TOTAL_ASSETS
Total current assets	FQ, FH, FY	TOTAL_CURRENT_ASSETS
Total current liabilities	FQ, FH, FY	TOTAL_CURRENT_LIABILITIES
Total debt	FQ, FH, FY	TOTAL_DEBT
Total equity	FQ, FH, FY	TOTAL_EQUITY
Total inventory	FQ, FH, FY	TOTAL_INVENTORY
Total liabilities	FQ, FH, FY	TOTAL_LIABILITIES
Total liabilities & shareholders' equities	FQ, FH, FY	TOTAL_LIABILITIES_SHRHLDRS_EQUITY
Total non-current assets	FQ, FH, FY	TOTAL_NON_CURRENT_ASSETS
Total non-current liabilities	FQ, FH, FY	TOTAL_NON_CURRENT_LIABILITIES
Total receivables, net	FQ, FH, FY	TOTAL_RECEIVABLES_NET
Treasury stock - common	FQ, FH, FY	TREASURY_STOCK_COMMON

Cash flow

This table lists the available metrics that provide information about how cash flows through a company.

▼ Click to show/hide

Financial	`period`	`financial_id`
Amortization	FQ, FH, FY, TTM	AMORTIZATION
Capital expenditures	FQ, FH, FY, TTM	CAPITAL_EXPENDITURES
Capital expenditures - fixed assets	FQ, FH, FY, TTM	CAPITAL_EXPENDITURES_FIXED_ASSETS
Capital expenditures - other assets	FQ, FH, FY, TTM	CAPITAL_EXPENDITURES_OTHER_ASSETS
Cash from financing activities	FQ, FH, FY, TTM	CASH_F_FINANCING_ACTIVITIES
Cash from investing activities	FQ, FH, FY, TTM	CASH_F_INVESTING_ACTIVITIES
Cash from operating activities	FQ, FH, FY, TTM	CASH_F_OPERATING_ACTIVITIES
Change in accounts payable	FQ, FH, FY, TTM	CHANGE_IN_ACCOUNTS_PAYABLE
Change in accounts receivable	FQ, FH, FY, TTM	CHANGE_IN_ACCOUNTS_RECEIVABLE
Change in accrued expenses	FQ, FH, FY, TTM	CHANGE_IN_ACCRUED_EXPENSES
Change in inventories	FQ, FH, FY, TTM	CHANGE_IN_INVENTORIES
Change in other assets/liabilities	FQ, FH, FY, TTM	CHANGE_IN_OTHER_ASSETS
Change in taxes payable	FQ, FH, FY, TTM	CHANGE_IN_TAXES_PAYABLE
Changes in working capital	FQ, FH, FY, TTM	CHANGES_IN_WORKING_CAPITAL
Common dividends paid	FQ, FH, FY, TTM	COMMON_DIVIDENDS_CASH_FLOW
Deferred taxes (cash flow)	FQ, FH, FY, TTM	CASH_FLOW_DEFERRED_TAXES
Depreciation & amortization (cash flow)	FQ, FH, FY, TTM	CASH_FLOW_DEPRECATION_N_AMORTIZATION
Depreciation/depletion	FQ, FH, FY, TTM	DEPRECIATION_DEPLETION
Financing activities - other sources	FQ, FH, FY, TTM	OTHER_FINANCING_CASH_FLOW_SOURCES
Financing activities - other uses	FQ, FH, FY, TTM	OTHER_FINANCING_CASH_FLOW_USES
Free cash flow	FQ, FH, FY, TTM	FREE_CASH_FLOW
Funds from operations	FQ, FH, FY, TTM	FUNDS_F_OPERATIONS
Investing activities - other sources	FQ, FH, FY, TTM	OTHER_INVESTING_CASH_FLOW_SOURCES
Investing activities - other uses	FQ, FH, FY	OTHER_INVESTING_CASH_FLOW_USES
Issuance of long term debt	FQ, FH, FY, TTM	SUPPLYING_OF_LONG_TERM_DEBT
Issuance/retirement of debt, net	FQ, FH, FY, TTM	ISSUANCE_OF_DEBT_NET
Issuance/retirement of long term debt	FQ, FH, FY, TTM	ISSUANCE_OF_LONG_TERM_DEBT
Issuance/retirement of other debt	FQ, FH, FY, TTM	ISSUANCE_OF_OTHER_DEBT
Issuance/retirement of short term debt	FQ, FH, FY, TTM	ISSUANCE_OF_SHORT_TERM_DEBT
Issuance/retirement of stock, net	FQ, FH, FY, TTM	ISSUANCE_OF_STOCK_NET
Net income (cash flow)	FQ, FH, FY, TTM	NET_INCOME_STARTING_LINE
Non-cash items	FQ, FH, FY, TTM	NON_CASH_ITEMS
Other financing cash flow items, total	FQ, FH, FY, TTM	OTHER_FINANCING_CASH_FLOW_ITEMS_TOTAL
Other investing cash flow items, total	FQ, FH, FY	OTHER_INVESTING_CASH_FLOW_ITEMS_TOTAL
Preferred dividends paid	FQ, FH, FY	PREFERRED_DIVIDENDS_CASH_FLOW

Financial	period	financial_id
Purchase of investments	FQ, FH, FY, TTM	PURCHASE_OF_INVESTMENTS
Purchase/acquisition of business	FQ, FH, FY, TTM	PURCHASE_OF_BUSINESS
Purchase/sale of business, net	FQ, FH, FY	PURCHASE_SALE_BUSINESS
Purchase/sale of investments, net	FQ, FH, FY, TTM	PURCHASE_SALE_INVESTMENTS
Reduction of long term debt	FQ, FH, FY, TTM	REDUCTION_OF_LONG_TERM_DEBT
Repurchase of common & preferred stock	FQ, FH, FY, TTM	PURCHASE_OF_STOCK
Sale of common & preferred stock	FQ, FH, FY, TTM	SALE_OF_STOCK
Sale of fixed assets & businesses	FQ, FH, FY, TTM	SALES_OF_BUSINESS
Sale/maturity of investments	FQ, FH, FY	SALES_OF_INVESTMENTS
Total cash dividends paid	FQ, FH, FY, TTM	TOTAL_CASH_DIVIDENDS_PAID

Statistics

This table contains a variety of statistical metrics, including commonly used financial ratios.

▼ Click to show/hide

Financial	period	financial_id
Accruals	FQ, FH, FY	ACCRUALS_RATIO
Altman Z-score	FQ, FH, FY	ALTMAN_Z_SCORE
Asset turnover	FQ, FH, FY	ASSET_TURNOVER
Beneish M-score	FQ, FH, FY	BENEISH_M_SCORE
Buyback yield %	FQ, FH, FY	BUYBACK_YIELD
COGS to revenue ratio	FQ, FH, FY	COGS_TO_REVENUE
Cash conversion cycle	FQ, FY	CASH_CONVERSION_CYCLE
Cash to debt ratio	FQ, FH, FY	CASH_TO_DEBT
Current ratio	FQ, FH, FY	CURRENT_RATIO
Days inventory	FQ, FY	DAYS_INVENT
Days payable	FQ, FY	DAYS_PAY
Days sales outstanding	FQ, FY	DAY_SALES_OUT
Debt to EBITDA ratio	FQ, FH, FY	DEBT_TO_EBITDA
Debt to assets ratio	FQ, FH, FY	DEBT_TO_ASSET
Debt to equity ratio	FQ, FH, FY	DEBT_TO_EQUITY
Debt to revenue ratio	FQ, FH, FY	DEBT_TO_REVENUE
Dividend payout ratio %	FQ, FH, FY, TTM	DIVIDEND_PAYOUT_RATIO
Dividend yield %	FQ, FH, FY	DIVIDENDS_YIELD
Dividends per share - common stock primary issue	FQ, FH, FY, TTM	DPS_COMMON_STOCK_PRIM_ISSUE

Financial	`period`	`financial_id`
EBITDA margin %	FQ, FH, FY, TTM	EBITDA_MARGIN
EPS basic one year growth	FQ, FH, FY, TTM	EARNINGS_PER_SHARE_BASIC_ONE_YEAR_GROWTH
EPS diluted one year growth	FQ, FH, FY	EARNINGS_PER_SHARE_DILUTED_ONE_YEAR_GROWTH
EPS estimates	FQ, FH, FY	EARNINGS_ESTIMATE
Effective interest rate on debt %	FQ, FH, FY	EFFECTIVE_INTEREST_RATE_ON_DEBT
Enterprise value	FQ, FH, FY	ENTERPRISE_VALUE
Enterprise value to EBIT ratio	FQ, FH, FY	EV_EBIT
Enterprise value to EBITDA ratio	FQ, FH, FY	ENTERPRISE_VALUE_EBITDA
Enterprise value to revenue ratio	FQ, FH, FY	EV_REVENUE
Equity to assets ratio	FQ, FH, FY	EQUITY_TO_ASSET
Float shares outstanding	FY	FLOAT_SHARES_OUTSTANDING
Free cash flow margin %	FQ, FH, FY	FREE_CASH_FLOW_MARGIN
Fulmer H factor	FQ, FY	FULMER_H_FACTOR
Goodwill to assets ratio	FQ, FH, FY	GOODWILL_TO_ASSET
Graham's number	FQ, FY	GRAHAM_NUMBERS
Gross margin %	FQ, FH, FY, TTM	GROSS_MARGIN
Gross profit to assets ratio	FQ, FY	GROSS_PROFIT_TO_ASSET
Interest coverage	FQ, FH, FY	INTERST_COVER
Inventory to revenue ratio	FQ, FH, FY	INVENT_TO_REVENUE
Inventory turnover	FQ, FH, FY	INVENT_TURNOVER
KZ index	FY	KZ_INDEX
Long term debt to total assets ratio	FQ, FH, FY	LONG_TERM_DEBT_TO_ASSETS
Net current asset value per share	FQ, FY	NCAVPS_RATIO
Net income per employee	FY	NET_INCOME_PER_EMPLOYEE
Net margin %	FQ, FH, FY, TTM	NET_MARGIN
Number of employees	FY	NUMBER_OF_EMPLOYEES
Operating earnings yield %	FQ, FH, FY	OPERATING_EARNINGS_YIELD
Operating margin %	FQ, FH, FY	OPERATING_MARGIN
PEG ratio	FQ, FY	PEG_RATIO
Piotroski F-score	FQ, FH, FY	PIOTROSKI_F_SCORE
Price earnings ratio forward	FQ, FY	PRICE_EARNINGS_FORWARD
Price sales ratio forward	FQ, FY	PRICE_SALES_FORWARD
Quality ratio	FQ, FH, FY	QUALITY_RATIO

Financial	`period`	`financial_id`
Quick ratio	FQ, FH, FY	QUICK_RATIO
Research & development to revenue ratio	FQ, FH, FY	RESEARCH_AND_DEVELOP_TO_REVENUE
Return on assets %	FQ, FH, FY	RETURN_ON_ASSETS
Return on common equity %	FQ, FH, FY	RETURN_ON_COMMON_EQUITY
Return on equity %	FQ, FH, FY	RETURN_ON_EQUITY
Return on equity adjusted to book value %	FQ, FH, FY	RETURN_ON_EQUITY_ADJUST_TO_BOOK
Return on invested capital %	FQ, FH, FY	RETURN_ON_INVESTED_CAPITAL
Return on tangible assets %	FQ, FH, FY	RETURN_ON_TANG_ASSETS
Return on tangible equity %	FQ, FH, FY	RETURN_ON_TANG_EQUITY
Revenue estimates	FQ, FH, FY	SALES_ESTIMATES
Revenue one year growth	FQ, FH, FY, TTM	REVENUE_ONE_YEAR_GROWTH
Revenue per employee	FY	REVENUE_PER_EMPLOYEE
Shares buyback ratio %	FQ, FH, FY	SHARE_BUYBACK_RATIO
Sloan ratio %	FQ, FH, FY	SLOAN_RATIO
Springate score	FQ, FY	SPRINGATE_SCORE
Sustainable growth rate	FQ, FY	SUSTAINABLE_GROWTH_RATE
Tangible common equity ratio	FQ, FH, FY	TANGIBLE_COMMON_EQUITY_RATIO
Tobin's Q (approximate)	FQ, FH, FY	TOBIN_Q_RATIO
Total common shares outstanding	FQ, FH, FY	TOTAL_SHARES_OUTSTANDING
Zmijewski score	FQ, FY	ZMIJEWSKI_SCORE

`request.economic()`

The request.economic() function provides scripts with the ability to retrieve economic data for a specified country or region, including information about the state of the economy (GDP, inflation rate, etc.) or of a particular industry (steel production, ICU beds, etc.).

Below is the signature for this function:

```
request.economic(country_code, field, gaps, ignore_invalid_symbol) → series flo
```

The `country_code` parameter accepts a "string" value representing the identifier of the country or region to request economic data for (e.g., "US", "EU", etc.). See the Country/region codes section for a complete list of codes this function supports. Note that the economic metrics available depend on the country or region specified in the function call.

The `field` parameter accepts a “string” specifying the metric that the function requests. The Field codes section covers all accessible metrics and the countries/regions they’re available for.

For a detailed explanation on the last two parameters of this function, see the Common characteristics section at the top of this page.

This simple example requests the growth rate of the Gross Domestic Product (“GDPQQ”) for the United States (“US”) using request.economic(), then plots its value on the chart with a gradient color:

//@version=6 indicator("Requesting economic data demo") //@variable The GDP growth rate for the US economy. float gdpqq = request.economic("US", "GDPQQ") //@variable The all-time maximum growth rate. float maxRate = ta.max(gdpqq) //@variable The all-time minimum growth rate. float minRate = ta.min(gdpqq) //@variable The color of the `gdpqq` plot. color rateColor = switch gdpqq >= 0 => color.from_gradient(gdpqq, 0, maxRate, color.purple, color.blue) => color.from_gradient(gdpqq, minRate, 0, color.red, color.purple) // Plot the results. plot(gdpqq, "US GDP Growth Rate", rateColor, style = plot.style_area)

Note that:

- This example does not include a `gaps` argument in the request.economic() call, so the function uses the default barmerge.gaps_off. In other words, it returns the last retrieved value when new data isn’t yet available.

> TipThe tables in the sections below are rather large, because there are numerous `country_code` and `field` arguments available. Use the **“Click to show/hide”** option above each table to toggle its visibility.

Country/region codes

The table in this section lists all country/region codes available for use with request.economic(). The first column of the table contains the “string” values that represent the country or region code, and the second column contains the corresponding country/region names.

It’s important to note that the value used as the `country_code` argument determines which field codes are accessible to the function.

▼ Click to show/hide

`country_code`	**Country/region name**
AF	Afghanistan
AL	Albania
DZ	Algeria
AD	Andorra
AO	Angola
AG	Antigua and Barbuda
AR	Argentina
AM	Armenia
AW	Aruba

country_code	**Country/region name**
AU	Australia
AT	Austria
AZ	Azerbaijan
BS	Bahamas
BH	Bahrain
BD	Bangladesh
BB	Barbados
BY	Belarus
BE	Belgium
BZ	Belize
BJ	Benin
BM	Bermuda
BT	Bhutan
BO	Bolivia
BA	Bosnia and Herzegovina
BW	Botswana
BR	Brazil
BN	Brunei
BG	Bulgaria
BF	Burkina Faso
BI	Burundi
KH	Cambodia
CM	Cameroon
CA	Canada
CV	Cape Verde
KY	Cayman Islands
CF	Central African Republic
TD	Chad
CL	Chile
CN	China
CO	Colombia
KM	Comoros
CG	Congo
CR	Costa Rica
HR	Croatia
CU	Cuba

country_code	**Country/region name**
CY	Cyprus
CZ	Czech Republic
DK	Denmark
DJ	Djibouti
DM	Dominica
DO	Dominican Republic
TL	East Timor
EC	Ecuador
EG	Egypt
SV	El Salvador
GQ	Equatorial Guinea
ER	Eritrea
EE	Estonia
ET	Ethiopia
EU	Euro area
FO	Faroe Islands
FJ	Fiji
FI	Finland
FR	France
GA	Gabon
GM	Gambia
GE	Georgia
DE	Germany
GH	Ghana
GR	Greece
GL	Greenland
GD	Grenada
GT	Guatemala
GN	Guinea
GW	Guinea Bissau
GY	Guyana
HT	Haiti
HN	Honduras
HK	Hong Kong
HU	Hungary
IS	Iceland

country_code	**Country/region name**
IN	India
ID	Indonesia
IR	Iran
IQ	Iraq
IE	Ireland
IM	Isle of Man
IL	Israel
IT	Italy
CI	Ivory Coast
JM	Jamaica
JP	Japan
JO	Jordan
KZ	Kazakhstan
KE	Kenya
KI	Kiribati
XK	Kosovo
KW	Kuwait
KG	Kyrgyzstan
LA	Laos
LV	Latvia
LB	Lebanon
LS	Lesotho
LR	Liberia
LY	Libya
LI	Liechtenstein
LT	Lithuania
LU	Luxembourg
MO	Macau
MK	Macedonia
MG	Madagascar
MW	Malawi
MY	Malaysia
MV	Maldives
ML	Mali
MT	Malta
MR	Mauritania

country_code	**Country/region name**
MU	Mauritius
MX	Mexico
MD	Moldova
MC	Monaco
MN	Mongolia
ME	Montenegro
MA	Morocco
MZ	Mozambique
MM	Myanmar
NA	Namibia
NP	Nepal
NL	Netherlands
NC	New Caledonia
NZ	New Zealand
NI	Nicaragua
NE	Niger
NG	Nigeria
KP	North Korea
NO	Norway
OM	Oman
PK	Pakistan
PS	Palestine
PA	Panama
PG	Papua New Guinea
PY	Paraguay
PE	Peru
PH	Philippines
PL	Poland
PT	Portugal
PR	Puerto Rico
QA	Qatar
CD	Republic of the Congo
RO	Romania
RU	Russia
RW	Rwanda
WS	Samoa

country_code	**Country/region name**
SM	San Marino
ST	Sao Tome and Principe
SA	Saudi Arabia
SN	Senegal
RS	Serbia
SC	Seychelles
SL	Sierra Leone
SG	Singapore
SK	Slovakia
SI	Slovenia
SB	Solomon Islands
SO	Somalia
ZA	South Africa
KR	South Korea
SS	South Sudan
ES	Spain
LK	Sri Lanka
LC	St Lucia
VC	St Vincent and the Grenadines
SD	Sudan
SR	Suriname
SZ	Swaziland
SE	Sweden
CH	Switzerland
SY	Syria
TW	Taiwan
TJ	Tajikistan
TZ	Tanzania
TH	Thailand
TG	Togo
TO	Tonga
TT	Trinidad and Tobago
TN	Tunisia
TR	Turkey
TM	Turkmenistan
UG	Uganda

country_code	Country/region name
UA	Ukraine
AE	United Arab Emirates
GB	United Kingdom
US	United States
UY	Uruguay
UZ	Uzbekistan
VU	Vanuatu
VE	Venezuela
VN	Vietnam
YE	Yemen
ZM	Zambia
ZW	Zimbabwe

Field codes

The table in this section lists the field codes available for use with request.economic(). The first column contains the “string” values used as the `field` argument, and the second column contains names of each metric and links to our Help Center with additional information, including the countries/regions they’re available for.

▼ Click to show/hide

field	Metric
AA	Asylum Applications
ACR	API Crude Runs
AE	Auto Exports
AHE	Average Hourly Earnings
AHO	API Heating Oil
AWH	Average Weekly Hours
BBS	Banks Balance Sheet
BCLI	Business Climate Indicator
BCOI	Business Confidence Index
BI	Business Inventories
BLR	Bank Lending Rate
BOI	NFIB Business Optimism Index
BOT	Balance Of Trade
BP	Building Permits
BR	Bankruptcies
CA	Current Account

field	Metric
CAG	Current Account To GDP
CAP	Car Production
CAR	Car Registrations
CBBS	Central Bank Balance Sheet
CCC	Claimant Count Change
CCI	Consumer Confidence Index
CCOS	Cushing Crude Oil Stocks
CCP	Core Consumer Prices
CCPI	Core CPI
CCPT	Consumer Confidence Price Trends
CCR	Consumer Credit
CCS	Credit Card Spending
CEP	Cement Production
CF	Capital Flows
CFNAI	Chicago Fed National Activity Index
CI	API Crude Imports
CIND	Coincident Index
CIR	Core Inflation Rate, YoY
CJC	Continuing Jobless Claims
CN	API Cushing Number
COI	Crude Oil Imports
COIR	Crude Oil Imports from Russia
CONSTS	Construction Spending
COP	Crude Oil Production
COR	Crude Oil Rigs
CORD	Construction Orders, YoY
CORPI	Corruption Index
CORR	Corruption Rank
COSC	Crude Oil Stocks Change
COUT	Construction Output, YoY
CP	Copper Production
CPCEPI	Core PCE Price Index
CPI	Consumer Price Index
CPIHU	CPI Housing Utilities
CPIM	CPI Median
CPIT	CPI Transportation

field	Metric
CPITM	CPI Trimmed Mean
CPMI	Chicago PMI
CPPI	Core Producer Price Index
CPR	Corporate Profits
CRLPI	Cereals Price Index
CRR	Cash Reserve Ratio
CS	Consumer Spending
CSC	API Crude Oil Stock Change
CSHPI	Case Shiller Home Price Index
CSHPIMM	Case Shiller Home Price Index, MoM
CSHPIYY	Case Shiller Home Price Index, YoY
CSS	Chain Store Sales
CTR	Corporate Tax Rate
CU	Capacity Utilization
DFMI	Dallas Fed Manufacturing Index
DFP	Distillate Fuel Production
DFS	Distillate Stocks
DFSI	Dallas Fed Services Index
DFSRI	Dallas Fed Services Revenues Index
DG	Deposit Growth
DGO	Durable Goods Orders
DGOED	Durable Goods Orders Excluding Defense
DGOET	Durable Goods Orders Excluding Transportation
DIR	Deposit Interest Rate
DPI	Disposable Personal Income
DRPI	Dairy Price Index
DS	API Distillate Stocks
DT	CBI Distributive Trades
EC	ADP Employment Change
ED	External Debt
EDBR	Ease Of Doing Business Ranking
EHS	Existing Home Sales
ELP	Electricity Production
EMC	Employment Change
EMCI	Employment Cost Index
EMP	Employed Persons

field	Metric
EMR	Employment Rate
EOI	Economic Optimism Index
EP	Export Prices
ESI	ZEW Economic Sentiment Index
EWS	Economy Watchers Survey
EXP	Exports
EXPYY	Exports, YoY
FAI	Fixed Asset Investment
FBI	Foreign Bond Investment
FDI	Foreign Direct Investment
FE	Fiscal Expenditure
FER	Foreign Exchange Reserves
FI	Food Inflation, YoY
FO	Factory Orders
FOET	Factory Orders Excluding Transportation
FPI	Food Price Index
FSI	Foreign Stock Investment
FTE	Full Time Employment
FYGDPG	Full Year GDP Growth
GASP	Gasoline Prices
GBP	Government Budget
GBV	Government Budget Value
GCI	Competitiveness Index
GCR	Competitiveness Rank
GD	Government Debt
GDG	Government Debt To GDP
GDP	Gross Domestic Product
GDPA	GDP From Agriculture
GDPC	GDP From Construction
GDPCP	GDP Constant Prices
GDPD	GDP Deflator
GDPGA	GDP Growth Annualized
GDPMAN	GDP From Manufacturing
GDPMIN	GDP From Mining
GDPPA	GDP From Public Administration
GDPPC	GDP Per Capita

field	Metric
GDPPCP	GDP Per Capita, PPP
GDPQQ	GDP Growth Rate
GDPS	GDP From Services
GDPSA	GDP Sales
GDPT	GDP From Transport
GDPU	GDP From Utilities
GDPYY	GDP, YoY
GDTPI	Global Dairy Trade Price Index
GFCF	Gross Fixed Capital Formation
GNP	Gross National Product
GP	Gold Production
GPA	Government Payrolls
GPRO	Gasoline Production
GR	Government Revenues
GRES	Gold Reserves
GS	API Gasoline Stocks
GSC	Grain Stocks Corn
GSCH	Gasoline Stocks Change
GSG	Government Spending To GDP
GSP	Government Spending
GSS	Grain Stocks Soy
GSW	Grain Stocks Wheat
GTB	Goods Trade Balance
HB	Hospital Beds
HDG	Households Debt To GDP
HDI	Households Debt To Income
HICP	Harmonised Index of Consumer Prices
HIRMM	Harmonised Inflation Rate, MoM
HIRYY	Harmonised Inflation Rate, YoY
HMI	NAHB Housing Market Index
HOR	Home Ownership Rate
HOS	Heating Oil Stocks
HOSP	Hospitals
HPI	House Price Index
HPIMM	House Price Index, MoM
HPIYY	House Price Index, YoY

field	Metric
HS	Home Loans
HSP	Household Spending
HST	Housing Starts
IC	Changes In Inventories
ICUB	ICU Beds
IE	Inflation Expectations
IFOCC	IFO Assessment Of The Business Situation
IFOE	IFO Business Developments Expectations
IJC	Initial Jobless Claims
IMP	Imports
IMPYY	Imports, YoY
INBR	Interbank Rate
INTR	Interest Rate
IPA	IP Addresses
IPMM	Industrial Production, MoM
IPRI	Import Prices
IPYY	Industrial Production, YoY
IRMM	Inflation Rate, MoM
IRYY	Inflation Rate, YoY
IS	Industrial Sentiment
ISP	Internet Speed
JA	Job Advertisements
JAR	Jobs To Applications Ratio
JC	Challenger Job Cuts
JC4W	Jobless Claims, 4-Week Average
JO	Job Offers
JV	Job Vacancies
KFMI	Kansas Fed Manufacturing Index
LB	Loans To Banks
LC	Labor Costs
LEI	Leading Economic Index
LFPR	Labor Force Participation Rate
LG	Loan Growth, YoY
LIVRR	Liquidity Injections Via Reverse Repo
LMIC	LMI Logistics Managers Index Current
LMICI	LMI Inventory Costs

field	Metric
LMIF	LMI Logistics Managers Index Future
LMITP	LMI Transportation Prices
LMIWP	LMI Warehouse Prices
LPS	Loans To Private Sector
LR	Central Bank Lending Rate
LTUR	Long Term Unemployment Rate
LWF	Living Wage Family
LWI	Living Wage Individual
M0	Money Supply M0
M1	Money Supply M1
M2	Money Supply M2
M3	Money Supply M3
MA	Mortgage Approvals
MAPL	Mortgage Applications
MCE	Michigan Consumer Expectations
MCEC	Michigan Current Economic Conditions
MD	Medical Doctors
ME	Military Expenditure
MGDPYY	Monthly GDP, YoY
MIE1Y	Michigan Inflation Expectations
MIE5Y	Michigan 5 Year Inflation Expectations
MIP	Mining Production, YoY
MMI	MBA Mortgage Market Index
MO	Machinery Orders
MP	Manufacturing Payrolls
MPI	Meat Price Index
MPRMM	Manufacturing Production, MoM
MPRYY	Manufacturing Production, YoY
MR	Mortgage Rate
MRI	MBA Mortgage Refinance Index
MS	Manufacturing Sales
MTO	Machine Tool Orders
MW	Minimum Wages
NDCGOEA	Orders For Non-defense Capital Goods Excluding Aircraft
NEGTB	Goods Trade Deficit With Non-EU Countries
NFP	Nonfarm Payrolls

field	Metric
NGI	Natural Gas Imports
NGIR	Natural Gas Imports from Russia
NGSC	Natural Gas Stocks Change
NHPI	Nationwide House Price Index
NHS	New Home Sales
NHSMM	New Home Sales, MoM
NMPMI	Non-Manufacturing PMI
NO	New Orders
NODXMM	Non-Oil Domestic Exports, MoM
NODXYY	Non-Oil Domestic Exports, YoY
NOE	Non-Oil Exports
NPP	Nonfarm Payrolls Private
NURS	Nurses
NYESMI	NY Empire State Manufacturing Index
OE	Oil Exports
OPI	Oils Price Index
PCEPI	PCE Price Index
PDG	Private Debt To GDP
PFMI	Philadelphia Fed Manufacturing Index
PHSIMM	Pending Home Sales Index, MoM
PHSIYY	Pending Home Sales Index, YoY
PI	Personal Income
PIN	Private Investment
PIND	MBA Purchase Index
PITR	Personal Income Tax Rate
POP	Population
PPI	Producer Price Index
PPII	Producer Price Index Input
PPIMM	Producer Price Inflation, MoM
PPIYY	Producer Prices Index, YoY
PRI	API Product Imports
PROD	Productivity
PS	Personal Savings
PSC	Private Sector Credit
PSP	Personal Spending
PTE	Part Time Employment

field	Metric
PUAC	Pandemic Unemployment Assistance Claims
RAM	Retirement Age Men
RAW	Retirement Age Women
RCR	Refinery Crude Runs
REM	Remittances
RFMI	Richmond Fed Manufacturing Index
RFMSI	Richmond Fed Manufacturing Shipments Index
RFSI	Richmond Fed Services Index
RI	Redbook Index
RIEA	Retail Inventories Excluding Autos
RPI	Retail Price Index
RR	Repo Rate
RRR	Reverse Repo Rate
RSEA	Retail Sales Excluding Autos
RSEF	Retail Sales Excluding Fuel
RSMM	Retail Sales, MoM
RSYY	Retail Sales, YoY
RTI	Reuters Tankan Index
SBSI	Small Business Sentiment Index
SFHP	Single Family Home Prices
SP	Steel Production
SPI	Sugar Price Index
SS	Services Sentiment
SSR	Social Security Rate
SSRC	Social Security Rate For Companies
SSRE	Social Security Rate For Employees
STR	Sales Tax Rate
TA	Tourist Arrivals
TAXR	Tax Revenue
TCB	Treasury Cash Balance
TCPI	Tokyo CPI
TI	Terrorism Index
TII	Tertiary Industry Index
TOT	Terms Of Trade
TR	Tourism Revenues
TVS	Total Vehicle Sales

field	Metric
UC	Unemployment Change
UP	Unemployed Persons
UR	Unemployment Rate
WAG	Wages
WES	Weapons Sales
WG	Wage Growth, YoY
WHS	Wages High Skilled
WI	Wholesale Inventories
WLS	Wages Low Skilled
WM	Wages In Manufacturing
WPI	Wholesale Price Index
WS	Wholesale Sales
YUR	Youth Unemployment Rate
ZCC	ZEW Current Conditions

`request.footprint()`

The request.footprint() function enables scripts to retrieve volume footprint data for the bars in the datasets on which they run. For a given bar, a volume footprint categorizes volume values from lower timeframes as "buy" (upward) or "sell" (downward) based on intrabar price action, then collects the categorized volume data into equally sized rows that cover the bar's price range. Programmers can use retrieved footprint data to inspect the distribution of "buy", "sell", and total volume across the rows for a bar's range, identify a bar's Point of Control (POC) and other significant price levels, calculate volume delta information, detect volume imbalances, and more.

> NoteVolume footprints are available exclusively to users who have a Premium or Ultimate plan. Accounts with lower-tier plans **cannot** use scripts that request volume footprint data with the request.footprint() function.

The function's signature is as follows:

```
request.footprint(ticks_per_row, va_percent, imbalance_percent) → series footpr
```

The `ticks_per_row` parameter specifies the size of each row in the calculated volume footprint, in ticks. It requires a positive "simple int" value representing a multiplier for the instrument's minimum tick size. For example, if the argument is 100, the price range of each row equals the value of `100 * syminfo.mintick`. The specified row size affects the total number of rows in each bar's footprint. Increase the value for fewer rows with a larger size, or decrease the value for the opposite.

The `va_percent` parameter accepts a “simple float” value specifying the percentage of the footprint’s total volume to use for calculating the bar’s *Value Area (VA)*, where a value of 100 represents 100% of the total volume. Specifying an argument is optional. The default value is 70, meaning that the footprint’s VA includes 70% of the total volume.

The `imbalance_percent` parameter accepts a “simple float” value specifying the required percentage difference between “buy” and “sell” volume in adjacent footprint rows for detecting *volume imbalances*:

- The footprint considers a row to have a *buy imbalance* if the row’s “buy” volume exceeds the “sell” volume of the row *below* it by the specified percentage.
- The footprint considers a row to have a *sell imbalance* if the row’s “sell” volume exceeds the “buy” volume of the row *above* it by the given percentage.

Including an `imbalance_percent` argument is optional. The default value is 300, meaning that the “buy” or “sell” volume of a footprint row must be three times (300%) larger than the opposing volume of an adjacent row to signify an imbalance.

A call to the request.footprint() function returns either the *reference (ID)* of a footprint object that contains the volume footprint data for the current bar, or na if no footprint is available for that bar.

> NoticeScripts cannot perform more than **one** footprint request with the request.footprint() function. If a script contains multiple calls to this function, it raises a *runtime error*.

Scripts can use any returned footprint ID that is not na in calls to the built-in `footprint.*()` functions to retrieve data from a bar’s volume footprint.

For example, the following script calls request.footprint() on each bar to request the ID of a footprint object that contains the bar’s volume footprint data. If the requested data is available, the script then uses the returned ID in calls to four `footprint.*()` functions – footprint.total_volume(), footprint.buy_volume(), footprint.sell_volume(), and footprint.delta() – to retrieve the footprint’s total volume, total “buy” and “sell” volume, and overall volume delta.

The script plots the “buy” volume, the negative “sell” volume, and the volume delta as columns for visual comparison. It also displays a color-coded label at each bar’s high price to indicate whether the bar’s “buy” volume exceeds its “sell” volume or vice versa. Hovering over a label reveals a tooltip that shows the corresponding bar’s total volume and volume delta:

//@version=6 indicator("Requesting volume footprint data demo", max_labels_count = 500) //@variable The number of ticks to use as the price interval for each footprint row. int numTicksInput = input.int(100, "Ticks per footprint row", minval = 1) //@variable References a `footprint` object for the current bar, or holds `na` if no footprint data is available. footprint reqFootprint = request.footprint(numTicksInput) //@variable Is `true` if the requested `footprint` ID is not `na`, and `false` otherwise. bool hasFootprint = not na(reqFootprint) // Retrieve the bar's total, "buy", and "sell" volume sums and overall volume delta from the `footprint` object. float totalVolume = hasFootprint ? footprint.total_volume(reqFootprint) : na float buyVolume = hasFootprint ? footprint.buy_volume(reqFootprint) : na float sellVolume = hasFootprint ? footprint.sell_volume(reqFootprint) : na float deltaVolume = hasFootprint ? footprint.delta(reqFootprint) : na // Plot the total "buy" and "sell" volume as bidirectional columns, where "buy" volume increases in the // positive direction (upward plot), and "sell" volume increases in the negative direction (downward plot). plot(buyVolume, "Total buy volume",

#6eb21c, style = plot.style_columns) plot(sellVolume * -1, "Total sell volume", #b21c2b, style = plot.style_columns) // Plot bar's volume delta on top of the bidirectional columns to show the difference between "buy" and "sell" volume. plot(deltaVolume, "Volume delta", #e8a93c, style = plot.style_columns) hline(0, "Zero line", #d6d6d8, hline.style_solid) // Draw a label at the bar's high. The label is green if the volume delta is positive or zero, and red otherwise. // The label includes a tooltip that shows the bar's total volume and volume delta. label.new(bar_index, high, color = deltaVolume >= 0 ? #6eb21c : #b21c2b, size = 14, tooltip = str.format("Total volume: \t{0}\nVolume delta: \t{1}", totalVolume, deltaVolume), force_overlay = true)

Note that:

- The `id` parameter of each `footprint.*()` function does not allow na arguments. Therefore, this script uses ternary operations that execute `footprint.*()` calls only if the retrieved footprint ID is not na. If no data is available, the operations return na directly without executing the calls.
- On timeframes higher than or equal to “1D”, a footprint’s total volume might differ significantly from the value of the volume variable. Such differences occur for some instruments because *EOD* data feeds can include data from block trades, OTC trades, and other sources, whereas requested *intraday* data feeds do not. See the Data feeds section to learn more about the types of data feeds and their differences.

While some of the `footprint.*()` functions retrieve values representing overall metrics for a requested volume footprint, as shown above, others retrieve the IDs of volume_row objects that contain data for *individual rows* from the footprint for more detailed analysis. For instance, the footprint.poc() function retrieves the ID of the volume_row object that contains data for a footprint’s *Point of Control* row (i.e., the row with the highest total volume), and the footprint.rows() function constructs an array containing the volume_row IDs for *every* row within a footprint.

Scripts can use non-na IDs of the volume_row type in calls to the built-in `volume_row.*()` functions to retrieve data for a specific footprint row, including the row’s price levels, volume sums, volume delta, and buy or sell imbalances.

The advanced example below retrieves and displays detailed volume footprint information for visible chart bars. On each visible bar, the script requests a footprint ID using request.footprint(). If the ID is not na, the script calls footprint.rows() to create an array containing the volume_row IDs for all rows in the footprint, and uses other `footprint.*()` calls to retrieve the individual IDs for the footprint’s POC and Value Area rows.

Afterward, the script loops through the array using a `for...in` loop. It calls multiple `volume_row.*()` functions within the loop to retrieve price levels, categorized volume values, volume delta, and imbalance states for each row. On each iteration, the script formats the retrieved “buy” and “sell” volume, volume delta, and imbalance information for the current row into a string, and then displays the text in a box drawn at the row’s price range in a separate pane. Each box uses a gradient background color based on the row’s volume delta and its total volume relative to the POC row’s total volume. The text color of each box is the chart’s foreground color if the row is the POC, purple if the row is a VA boundary, and gray otherwise.

The script also plots the retrieved POC levels and the VA boundaries as circles on the main chart pane for visual reference:

//@version=6 indicator("Retrieving footprint row data demo", max_boxes_count = 500) //@variable The size of each footprint row, expressed in ticks. int numTicksInput = input.int(100, "Ticks per footprint row", 1) //@variable The percentage difference in opposing volume between rows for detecting volume imbalances.

```
int imbalanceInput = input.int(300, "Imbalance percentage", 1) //@variable References a `footprint` object for the current bar, or holds `na` if no footprint data is available. footprint reqFootprint = request.footprint(numTicksInput, imbalance_percent = imbalanceInput) // Declare a tuple of variables to hold the values returned by the `if` structure for plotting. // The values are not `na` only if the bar is visible and the `reqFootprint` variable does not hold `na`. [pocHigh, pocLow, vaHigh, vaLow] = if ( time >= chart.left_visible_bar_time and time <= chart.right_visible_bar_time and not na(reqFootprint) ) //@variable References an array containing a `volume_row` ID for each row in the volume footprint. array<volume_row> volumeRowsArray = reqFootprint.rows() // Retrieve `volume_row` IDs for the footprint's Point of Control, Value Area High, and Value Area Low rows. volume_row poc = reqFootprint.poc() volume_row vah = reqFootprint.vah() volume_row val = reqFootprint.val() // Loop through the array. The `row` variable holds a `volume_row` ID, starting with the one for the *lowest* row. for row in volumeRowsArray // Get the upper and lower price levels of the current row. float upPrice = row.up_price(), float dnPrice = row.down_price() // Get the row's "buy" and "sell" volume values and the volume delta. float buyVol = row.buy_volume(), float sellVol = row.sell_volume(), float delta = row.delta() // Get the row's buy and sell imbalance states. bool buyImbalance = row.has_buy_imbalance(), bool sellImbalance = row.has_sell_imbalance() //@variable A string representing the row's buy volume, sell volume, volume delta, and imbalances, respectively. string boxText = str.format( "B: {0} | S: {1} | D: {2} | I: {3}", str.tostring(buyVol, format.volume), str.tostring(sellVol, format.volume), str.tostring(delta, format.volume), buyImbalance and sellImbalance ? "Both" : buyImbalance ? "Buy" : sellImbalance ? "Sell" : "None" ) // Calculate a green (for positive delta) or red (for negative delta) gradient color based on the row's volume // relative to the POC volume. color deltaColor = delta >= 0 ? color.green : color.red color boxColor = color.from_gradient( row.total_volume() / poc.total_volume(), 0, 1, color.new(deltaColor, 100), color.new(deltaColor, 70) ) // Draw a box at the price range of the row, in a separate pane, to display the `boxText` value. box rowBox = box.new( bar_index, upPrice, bar_index + 1, dnPrice, #787b8650, 1, text = boxText, text_color = #787b86, text_halign = text.align_left, bgcolor = boxColor ) // Update the text color and formatting of the box if the current row is a Value Area boundary or the POC. if upPrice == vah.up_price() or upPrice == val.up_price() rowBox.set_text_color(color.purple) rowBox.set_text_formatting(text.format_bold) if upPrice == poc.up_price() rowBox.set_text_color(chart.fg_color) rowBox.set_text_formatting(text.format_bold) // Return the POC and VA levels for use in the global scope. [poc.up_price(), poc.down_price(), vah.up_price(), val.down_price()] // Plot the retrieved POC and VA levels on the main chart pane. plot(pocHigh, "POC top", chart.fg_color, 5, plot.style_circles, force_overlay = true) plot(pocLow, "POC bottom", chart.fg_color, 5, plot.style_circles, force_overlay = true) plot(vaHigh, "VA top", color.purple, 3, plot.style_circles, force_overlay = true) plot(vaLow, "VA bottom", color.purple, 3, plot.style_circles, force_overlay = true)
```

Note that:

- As with the built-in functions for most other reference types, scripts can call `footprint.*()` and `volume_row.*()` built-ins as functions or methods. This script calls the built-ins using *method syntax*.
- The array created by footprint.rows() sorts its elements in *ascending order* by price level, where the first element refers to the volume_row object for the row with the lowest prices, and the last refers to the one for the row with the highest prices.
- The results of volume_row.has_buy_imbalance() and volume_row.has_sell_imbalance() calls depend on the `imbalance_percent` argument of the request.footprint() call that creates the parent footprint object. In this example, the “Imbalance percentage” input controls the argument, and therefore controls the behavior of the script’s `volume_row.has_*_imbalance()` calls.

To learn more about the footprint and volume_row types, and the available functions in their namespaces, refer to the footprint and volume_row section of the Type system page.

For more information about volume footprints and how they work, refer to the Volume footprint charts article in our Help Center.

Requesting footprints on other datasets

Scripts can request volume footprint data, or the results of calculations that use footprint data, from *other datasets* by passing an expression that returns or uses IDs of the footprint or volume_row type as the `expression` argument in a request.security() or request.security_lower_tf() function call. If a call to either of these functions requests the result of an expression that relies on volume footprint data, it calculates footprints on the specified dataset rather than the script's main dataset.

For example, the following script requests volume footprint data for the last *confirmed* period on a specified *higher timeframe*, then uses the data to display boxes across the current period for footprint rows whose total volume equals or exceeds an input percentage of the Point of Control (POC) row's total volume.

The script uses a request.security() call with `request.footprint(ticksInput)[1]` as the `expression` argument and barmerge.lookahead_on as the `lookahead` argument to fetch the last confirmed footprint ID for the selected timeframe. If the result is not na at the start of the current HTF period, the script compares each row's volume to the volume of the POC row in a for…in loop, and draws boxes to project the levels of rows whose volume meets the specified threshold. Each box uses a volume-based gradient color, where blue hues correspond to rows with lower volume, and orange hues correspond to rows with higher volume:

//@version=6 indicator("Requesting footprints on other datasets demo", overlay = true, behind_chart = false, max_boxes_count = 500) //@variable The higher timeframe for which to request footprint data. string htfInput = input.timeframe("1D", "Higher timeframe") //@variable The size of each footprint row, expressed in ticks. int ticksInput = input.int(100, "Ticks per row", minval = 1) //@variable The minimum percentage of POC volume required to qualify a footprint row as a high-volume node. float hvInput = input.float(50.0, "High-volume threshold (% of POC)", 0.0, 100.0) // Raise an error if the specified timeframe is not higher than the chart's timeframe. if barstate.isfirst and timeframe.in_seconds(htfInput) <= timeframe.in_seconds() runtime.error("The requested timeframe must be higher than the chart's timeframe.") //@variable The ID of the *last confirmed* `footprint` object from the higher TF, or `na` when new data is not available. // The `request.security()` call includes `barmerge.gaps_on`, because the script requires the data only on the // *first bar* of each new HTF period. footprint htfFootprint = request.security(syminfo.tickerid, htfInput, request.footprint(ticksInput)[1], barmerge.gaps_on, barmerge.lookahead_on) if not na(htfFootprint) //@variable References the `volume_row` object for the footprint's POC row. volume_row pocRow = htfFootprint.poc() //@variable The POC row's total volume. float pocVol = pocRow.total_volume() //@variable The minimum volume required for a high-volume row. float minVol = pocVol * hvInput / 100 // Loop through all `volume_row` IDs stored in the array created by `htfFootprint.rows()`. for row in htfFootprint.rows() //@variable The current row's total volume. float rowVol = row.total_volume() // Skip the rest of the current iteration if the row's total volume is less than the required volume. if rowVol < minVol continue // Draw a box across the current period at the confirmed row's price levels if its volume meets the threshold, // and color the box using a volume-based gradient. color boxColor = color.from_gradient(rowVol, minVol, pocVol, #2196f3, #ff9800) box.new(time, row.up_price(), time_close(htfInput), row.down_price(), boxColor, xloc = xloc.bar_time, bgcolor = color.new(boxColor, 50), text = str.tostring(rowVol, format.volume), text_color = #000000)

Note that:

- The request.security() call includes barmerge.gaps_on as the `gaps` argument to return a non-na footprint ID only on the *first* chart bar of each new HTF period, because the script does not require new footprint data on *every* bar.
- The script calls the time() and time_close() functions to retrieve the expected opening and closing timestamps for the current HTF period. The box drawings use these timestamps as their left and right coordinates.

Any request.security() or request.security_lower_tf() call whose expression depends on footprint objects counts toward the total number of request.footprint() calls, because such requests *copy* all code required to calculate the result on the retrieved dataset. A script raises a *runtime error* if its outputs depend on data from more than **one** unique request.footprint() call, regardless of whether those requests execute on the script's main dataset or on the datasets fetched by other `request.*()` calls.

Consider the following script, which defines a calculation that requests volume footprint data for the current dataset, computes a moving average of the POC row's median price, then declares a `pocMA` variable to store the average. The script uses the variable as the `expression` argument in a request.security() call to calculate the average POC level on the "1D" timeframe, then assigns the result to the `requestedMA` variable. Lastly, the script attempts to plot both the `pocMA` and `requestedMA` series on the chart, but fails and raises a *runtime error*:

//@version=6 indicator("Too many footprint requests demo") //@variable The size of each footprint row, expressed in ticks. int ticksInput = input.int(100, "Ticks per row", 1) //#region // The code in this region is copied into the `request.security()` call's context, because all of it affects // the `pocMA` variable used as the call's `expression` argument. // Request the ID of a `footprint` object. footprint fp = request.footprint(ticksInput) // Get the `volume_row` ID for the POC row if the requested `footprint` ID is not `na`. volume_row poc = na(fp) ? na : fp.poc() // Calculate a moving average of the POC row's midpoint. float pocMA = ta.sma(na(poc) ? na : math.avg(poc.up_price(), poc.down_price()), 5) //#endregion // Retrieve the `pocMA` value calculated on the "1D" timeframe. // This request copies all code in the region above and evaluates it on the requested dataset. // Therefore, it executes its own separate version of the `request.footprint()` call. float requestedMA = request.security(syminfo.tickerid, "1D", pocMA) // Using *both* the `pocMA` and `requestedMA` variables in the script's outputs causes a *runtime error*, // because the variables' values depend on *two* different footprint requests: one for the chart's timeframe, // and the other for the requested timeframe. plot(requestedMA, "Daily POC MA", color.purple, 3) plot(pocMA, "Chart POC MA", color.blue, 3)

The above script raises an error because its *outputs* (in this case, its plots) depend on both the `pocMA` and `requestedMA` variables, and the values of those variables depend on **two** separate footprint requests – one for the script's main dataset and the other for the dataset retrieved by the request.security() call. *Both* of these requests count toward the total number of request.footprint() calls, even though one of those calls is not defined *explicitly* in the code.

A simple way to resolve this error is to *remove* one of the script's two plot() calls. As explained in the Compiled tokens section of the Limitations page, the Pine Script compiler automatically *discards* code that a script's *outputs* do not depend on, including `request.*()` calls. Therefore, if we remove the plot of the `pocMA` series, for example, the request.footprint() call defined explicitly in the global scope *does not* execute. Instead, only the request.footprint() call copied into the request.security() call's context executes, because that is the only one that the script's outputs now require:

//@version=6 indicator("Removed footprint request demo") //@variable The size of each footprint row, expressed in ticks. int ticksInput = input.int(100, "Ticks per row", 1) //#region // Although this code is still defined in the global scope, it does *not* execute in this scope, because the script's // outputs no longer depend on the data assigned to the `fp`, `poc`, or `pocMA` variables now that we removed the // `plot(pocMA)` call. footprint fp = request.footprint(ticksInput) volume_row poc = na(fp) ? na : fp.poc() float pocMA = ta.sma(na(poc) ? na : math.avg(poc.up_price(), poc.down_price()), 5) //#endregion // This request still copies all code in the region above to calculate a POC average on the "1D" timeframe. // The compiler does not discard this request, including the `request.footprint()` call that executes in its context, // because the script still uses the `requestedMA` variable in its outputs. float requestedMA = request.security(syminfo.tickerid, "1D", pocMA) // If we plot only the `requestedMA` series, and *not* the `pocMA` series, the "Too many `request.footprint()` calls" // error no longer occurs, because the script no longer requires two separate footprint requests to determine its outputs. // The only footprint request that it requires now is the one copied into the `request.security()` call's context. plot(requestedMA, "Daily POC MA", color.purple, 3)

Note that:

- Although the `pocMA` variable declaration and its dependencies no longer execute directly, the Pine Profiler displays performance details next to that code in the Pine Editor. Those details represent the performance of the *copied* calculations that execute within the request.security() call's context in this case, as the profiler cannot display that information elsewhere. See the When requesting other contexts section of the Profiling and optimization page to learn more about this behavior.

`request.seed()`

TradingView aggregates a vast amount of data from its many providers, including price and volume information on tradable instruments, financials, economic data, and more, which users can retrieve in Pine Script using the functions discussed in the sections above, as well as multiple built-in variables.

To further expand the horizons of possible data one can analyze on TradingView, we have Pine Seeds, which allows users to supply custom *user-maintained* EOD data feeds via GitHub for use on TradingView charts and within Pine Script code.

> NoticeThe creation of *new* Pine Seeds repositories is currently **unavailable**. However, the data feeds from *existing* repositories are still accessible to charts and scripts. The Pine Seeds documentation on GitHub provides in-depth information about Pine Seeds functionality and instructions for requesting the return of full Pine Seeds support.

To retrieve data from a Pine Seeds data feed within a script, use the request.seed() function. Below is the function's signature:

```
request.seed(source, symbol, expression, ignore_invalid_symbol, calc_bars_count
```

The `source` parameter specifies the unique name of the user-maintained GitHub repository that contains the data feed.

The `symbol` parameter represents the file name from the “data/” directory of the `source` repository, excluding the “.csv” file extension. See this page for information about the structure of the data stored in repositories.

The `expression` parameter is the series to evaluate using data extracted from the requested context. It is similar to the equivalent in request.security() and request.security_lower_tf(). Data feeds stored in user-maintained repos contain time, open, high, low, close, and volume information, meaning the `expression` argument can use the corresponding built-in variables, including variables derived from them (e.g., bar_index, ohlc4, etc.) to request their values from the context of the custom data.

The script below visualizes sample data from the seed_crypto_santiment demo repository. It uses two calls to request.seed() to retrieve the close values from the repository’s BTC_SENTIMENT_POSITIVE_TOTAL and BTC_SENTIMENT_NEGATIVE_TOTAL data feeds and plots the results on the chart as step lines:

//@version=6 indicator("Pine Seeds demo", format=format.volume) //@variable The total positive sentiment for BTC extracted from the "seed_crypto_santiment" repository. float positiveTotal = request.seed("seed_crypto_santiment", "BTC_SENTIMENT_POSITIVE_TOTAL", close) //@variable The total negative sentiment for BTC extracted from the "seed_crypto_santiment" repository. float negativeTotal = request.seed("seed_crypto_santiment", "BTC_SENTIMENT_NEGATIVE_TOTAL", close) // Plot the data. plot(positiveTotal, "Positive sentiment", color.teal, 2, plot.style_stepline) plot(negativeTotal, "Negative sentiment", color.maroon, 2, plot.style_stepline)

Note that:

- This example requests data from the repository highlighted in the Pine Seeds documentation. It exists solely for example purposes, and its data *does not* update on a regular basis.
- Unlike most other `request.*()` functions, request.seed() does not have a `gaps` parameter. It always returns na values when no new data exists.
- Pine Seeds data is searchable from the chart’s symbol search bar. To load a data feed on the chart, enter the *“Repo:File” pair*, similar to searching for an “Exchange:Symbol” pair.

Repainting

Introduction

We define repainting as: **script behavior causing historical vs realtime calculations or plots to behave differently**.

Repainting behavior is widespread and many factors can cause it. Following our definition, our estimate is that more than 95% of indicators in existence exhibit some form of repainting behavior. Commony used indicators such as MACD and RSI, for example, show confirmed values on historical bars, but will fluctuate on a realtime, unconfirmed chart bar until it closes. Therefore, they behave *differently* in historical and realtime states.

Not all repainting behavior is inherently useless or misleading, nor does such behavior prevent knowledgeable traders from using indicators with such behavior. For example, who would think of discrediting a volume profile indicator solely because it updates its values on realtime bars?

One may encounter any of the following forms of repainting in the scripts they use, depending on what a script's calculations entail:

- **Widespread but often acceptable**: A script may use values that update with realtime price changes on the unconfirmed bar. For example, if one uses the close variable in calculations performed on an open chart bar, its values will reflect the most recent price in the bar. However, the script will only commit a new data point to its historical series once the bar closes. Another common case is using request.security() to fetch higher-timeframe data on realtime bars, as explained in the Historical and realtime behavior section of the Other timeframes and data page. As with the unconfirmed chart bar in the chart's timeframe, request.security() can track unconfirmed values from a higher-timeframe context on realtime bars, which can lead to repainting after the script restarts its execution. There is often nothing wrong with using such scripts, provided you understand how they work. When electing to use such scripts to issue alerts or trade orders, however, it's important to understand the difference between their realtime and historical behavior and decide for yourself whether it provides utility for your needs.
- **Potentially misleading**: Scripts that plot values into the past, calculate results on realtime bars that one cannot replicate on historical bars, or relocate past events are potentially misleading. For example, Ichimoku, most scripts based on pivots, most strategies using `calc_on_every_tick = true`, scripts using request.security() when it behaves differently on realtime bars, many scripts using varip, many scripts using timenow, and some scripts that use `barstate.*` variables can exhibit misleading repainting behavior.
- **Unacceptable**: Scripts that leak future information into the past, strategies that execute on non-standard charts, and scripts using realtime intrabars to generate alerts or orders, are examples that can produce heavily misleading repainting behavior.
- **Unavoidable**: Revisions of the data feed from a provider and variations in the starting bar of the chart's history can cause repainting behavior that may be unavoidable in a script.

The first two types of repainting can be perfectly acceptable if:

1. You are aware of the behavior.
2. You can live with it, or

3. You can circumvent it.

It should now be clear that not **all** repainting behavior is wrong and requires avoiding at all costs. In many situations, some forms of repainting may be exactly what a script needs. What's important is to know when repainting behavior is **not** acceptable for one's needs. To avoid repainting that's not acceptable, it's important to understand how a tool works or how you should design the tools you build. If you publish scripts, ensure you mention any potentially misleading behavior along with the other limitations of your script in the publication's description.

For script users

One can decide to use repainting indicators if they understand the behavior, and whether that behavior meets their analysis requirements. Don't be one of those newcomers who slap “repaint” sentences on published scripts in an attempt to discredit them, as doing so reveals a lack of foundational knowledge on the subject.

Simply asking whether a script repaints is relatively meaningless, given that there are forms of repainting behavior that are perfectly acceptable in a script. Therefore, such a question will not beget a meaningful answer. One should instead ask *specific* questions about a script's potential repainting behavior, such as:

- Does the script calculate/display in the same way on historical and realtime bars?
- Do alerts from the script wait for the end of a realtime bar before triggering?
- Do signal markers shown by the script wait for the end of a realtime bar before showing?
- Does the script plot/draw values into the past?
- Does the strategy use `calc_on_every_tick = true`?
- Do the script's request.security() calls leak future information into the past on historical bars?

What's important is that you understand how the tools you use work, and whether their behavior is compatible with your objectives, repainting or not. As you will learn if you read this page, repainting is a complex matter. It has many faces and many causes. Even if you don't program in Pine Script®, this page will help you understand the array of causes that can lead to repainting, and hopefully enable more meaningful discussions with script authors.

For Pine Script programmers

As discussed above, not all forms of repainting behavior must be avoided at all costs, nor is all potential repainting behavior necessarily avoidable. We hope this page helps you better understand the dynamics at play so that you can design your trading tools with these behaviors in mind. This page's content should help make you aware of common coding mistakes that produce misleading repainting results.

Whatever your design decisions are, if you publish your script, explain the script to traders so they can understand how it behaves.

This page covers three broad categories of repainting causes:

- Historical vs realtime calculations
- Plotting in the past
- Dataset variations

Historical vs realtime calculations

Fluid data values

Historical data does not include records of intermediary price movements on bars; only open, high, low and close values (OHLC).

On realtime bars (bars running when the instrument's market is open), however, the high, low and close values are not fixed; they can change values many times before the realtime bar closes and its HLC values are fixed. They are *fluid*. This leads to a script sometimes working differently on historical data and in real time, where only the open price will not change during the bar.

Any script using values like high, low and close in realtime is subject to producing calculations that may not be repeatable on historical bars — thus repaint.

Let's look at this simple script. It detects crosses of the close value (in the realtime bar, this corresponds to the current price of the instrument) over and under an EMA:

//@version=6 indicator("Repainting", "", true) ma = ta.ema(close, 5) xUp = ta.crossover(close, ma) xDn = ta.crossunder(close, ma) plot(ma, "MA", color.black, 2) bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)

Note that:

- The script uses bgcolor() to color the background green when close crosses over the EMA, and red on crosses under the EMA.
- The screen snapshot shows the script in realtime on a 30sec chart. A cross over the EMA has been detected, thus the background of the realtime bar is green.
- The problem here is that nothing guarantees this condition will hold true until the end of the realtime bar. The arrow points to the timer showing that 21 seconds remain in the realtime bar, and anything could happen until then.
- We are witnessing a repainting script.

To prevent this repainting, we must rewrite our script so that it does not use values that fluctuate during the realtime bar. This will require using values from a bar that has elapsed (typically the preceding bar), or the open price, which does not vary in realtime.

We can achieve this in many ways. This method adds a `and barstate.isconfirmed` condition to our cross detections, which requires the script to be executing on the bar's last iteration, when it closes and prices are confirmed. It is a simple way to avoid repainting:

//@version=6 indicator("Repainting", "", true) ma = ta.ema(close, 5) xUp = ta.crossover(close, ma) and barstate.isconfirmed xDn = ta.crossunder(close, ma) and barstate.isconfirmed plot(ma, "MA", color.black, 2) bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)

This uses the crosses detected on the previous bar:

//@version=6 indicator("Repainting", "", true) ma = ta.ema(close, 5) xUp = ta.crossover(close, ma)[1] xDn = ta.crossunder(close, ma)[1] plot(ma, "MA", color.black, 2) bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)

This uses only confirmed close and EMA values for its calculations:

//@version=6 indicator("Repainting", "", true) ma = ta.ema(close[1], 5) xUp = ta.crossover(close[1], ma) xDn = ta.crossunder(close[1], ma) plot(ma, "MA", color.black, 2) bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)

This detects crosses between the realtime bar's open and the value of the EMA from the previous bars. Notice that the EMA is calculated using close, so it repaints. We must ensure we use a confirmed value to detect crosses, thus `ma[1]` in the cross detection logic:

//@version=6 indicator("Repainting", "", true) ma = ta.ema(close, 5) xUp = ta.crossover(open, ma[1]) xDn = ta.crossunder(open, ma[1]) plot(ma, "MA", color.black, 2) bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)

All these methods have one thing in common: while they prevent repainting, they will also trigger signals later than repainting scripts. This is an inevitable compromise if one wants to avoid repainting. You can't have your cake and eat it too.

Repainting `request.security()` calls

The request.security() function behaves differently on historical and realtime bars. On historical bars, it only returns *confirmed* values from its requested context, wheras it can return *unconfirmed* values on realtime bars. When the script restarts its execution, the bars that had a realtime state become historical bars, and will therefore only contain the values it confirmed on those bars. If the values returned by request.security() fluctuate on realtime bars without confirmation from the context, the script will repaint them when it restarts its execution. See the Historical and realtime behavior section of the Other timeframes and data page for a detailed explanation.

One can ensure higher-timeframe data requests only return confirmed values on all bars, regardless of bar state, by offsetting the `expression` argument by at least one bar with the history-referencing operator [] and using barmerge.lookahead_on for the `lookahead` argument in the request.security() call, as explained here.

The script below demonstrates the difference between repainting and non-repainting HTF data requests. It contains two request.security() calls. The first function call requests close data from the `higherTimeframe` without additional specification, and the second call requests the same series with an offset and barmerge.lookahead_on.

As we see on all realtime bars (the ones with an orange background), the `repaintingClose` series contains values that fluctuate without confirmation from the `higherTimeframe` context, meaning the results will *repaint* after the script restarts its executions. The `nonRepaintingClose`, on the other hand, behaves the same on realtime and historical bars, i.e., it only changes its value when new, confirmed data is available:

//@version=6 indicator("Repainting vs non-repainting `request.security()` demo", overlay = true) //@variable The timeframe to request data from. string higherTimeframe = input.timeframe("30", "Timeframe") if timeframe.in_seconds() > timeframe.in_seconds(higherTimeframe) runtime.error("The 'Timeframe' input is smaller than the chart's timeframe. Choose a higher timeframe.") //@variable The current `close` requested from the `higherTimeframe`. Fluctuates without confirmation on realtime bars. float repaintingClose = request.security(syminfo.tickerid, higherTimeframe, close) //@variable The last confirmed `close` requested from the `higherTimeframe`. // Behaves the same on historical and realtime bars. float nonRepaintingClose = request.security(syminfo.tickerid, higherTimeframe, close[1], lookahead = barmerge.lookahead_on) // Plot the values. plot(repaintingClose, "Repainting close", color.new(color.purple, 50), 8) plot(nonRepaintingClose, "Non-repainting close", color.teal, 3) // Plot a shape when a new

`higherTimeframe` starts. plotshape(timeframe.change(higherTimeframe), "Timeframe change marker", shape.square, location.top, size = size.small) // Color the background on realtime bars. bgcolor(barstate.isrealtime ? color.new(color.orange, 60) : na, title = "Realtime bar highlight")

Note that:

- The script uses the plotshape() function to mark the chart when a new bar starts on the specified higher timeframe.
- This script calls runtime.error() to raise a custom runtime error if the `higherTimeframe` value represents a timeframe that is *lower* than the chart's timeframe.
- On historical bars, the `repaintingClose` series has a new value at the *end* of each timeframe, and the `nonRepaintingClose` has a new value at the *start* of each timeframe.

For the sake of easy reusability, below is a simple a `noRepaintSecurity()` function that one can apply in their scripts to request non-repainting higher-timeframe values:

//@function Requests non-repainting `expression` values from the context of the `symbol` and `timeframe`. noRepaintSecurity(symbol, timeframe, expression) => request.security(symbol, timeframe, expression[1], lookahead = barmerge.lookahead_on)

Note that:

- The `[1]` offset to the series and the use of `lookahead = barmerge.lookahead_on` are interdependent. *Neither* can be removed without compromising the integrity of the function.
- Unlike a plain request.security() call, a call to this wrapper function cannot use a tuple as the `expression` argument. For multi-element requests, programmers can pass the ID of an object of a user-defined type whose *fields* contain the desired elements.

Using `request.security()` at lower timeframes

Some scripts use request.security() to request data from a timeframe **lower** than the chart's timeframe. This can be useful when functions specifically designed to handle intrabars at lower timeframes are sent down the timeframe. When this type of user-defined function requires the detection of the intrabars' first bar, as most do, the technique will only work on historical bars. This is due to the fact that realtime intrabars are not yet sorted. The impact of this is that such scripts cannot reproduce in real time their behavior on historical bars. Any logic generating alerts, for example, will be flawed, and constant refreshing will be required to recalculate elapsed realtime bars as historical bars.

When used at lower timeframes than the chart's without specialized functions able to distinguish between intrabars, request.security() will only return the value of the **last** intrabar in the dilation of the chart's bar, which is usually not useful, and will also not reproduce in real time, so lead to repainting.

For all these reasons, unless you understand the subtleties of using request.security() at lower timeframes than the chart's, it is best to avoid using the function at those timeframes. Higher-quality scripts will have logic to detect such anomalies and prevent the display of results which would be invalid when a lower timeframe is used.

For more reliable lower-timeframe data requests, use request.security_lower_tf(), as explained in this section of the Other timeframes and data page.

Future leak with `request.security()`

When request.security() is used with `lookahead = barmerge.lookahead_on` to fetch prices without offsetting the series by `[1]`, it will return data from the future on historical bars, which is dangerously misleading.

While historical bars will magically display future prices before they should be known, no lookahead is possible in realtime because the future there is unknown, as it should, so no future bars exist.

This is an example:

// FUTURE LEAK! DO NOT USE! //@version=6 indicator("Future leak", "", true) futureHigh = request.security(syminfo.tickerid, "1D", high, lookahead = barmerge.lookahead_on) plot(futureHigh)

Note how the higher timeframe line is showing the timeframe's high value before it occurs. The solution to avoid this effect is to use the function as demonstrated in this previous section.

Using lookahead to produce misleading results is not allowed in script publications, as explained in the lookahead section of the Other timeframes and data page. Script publications that use this misleading technique **will be moderated**.

`varip`

Scripts using the varip declaration mode for variables (see our section on varip for more information) save information across realtime updates, which cannot be reproduced on historical bars where only OHLC information is available. Such scripts may be useful in realtime, including to generate alerts, but their logic cannot be backtested, nor can their plots on historical bars reflect calculations that will be done in realtime.

Bar state built-ins

Scripts using bar states may or may not repaint. As we have seen in the previous section, using barstate.isconfirmed is actually one way to **avoid** repainting that **will** reproduce on historical bars, which are always “confirmed”. Uses of other bar states such as barstate.isnew, however, will lead to repainting. The reason is that on historical bars, barstate.isnew is `true` on the bar's close, yet in realtime, it is `true` on the bar's open. Using the other bar state variables will usually cause some type of behavioral discrepancy between historical and realtime bars.

`timenow`

The timenow built-in returns the current time. Scripts using this variable cannot show consistent historical and realtime behavior, so they necessarily repaint.

Strategies

Strategies using `calc_on_every_tick = true` execute on each realtime update, while strategies run on the close of historical bars. They will most probably not generate the same order executions, and so repaint. Note that when this happens, it also invalidates backtesting results, as they are not representative of the strategy's behavior in realtime.

Plotting in the past

Scripts detecting pivots after 5 bars have elapsed will often go back in the past to plot pivot levels or values on the actual pivot, 5 bars in the past. This will often cause unsuspecting traders looking at plots on

historical bars to infer that when the pivot happens in realtime, the same plots will apppear on the pivot when it occurs, as opposed to when it is detected.

Let's look at a script showing the price of high pivots by placing the price in the past, 5 bars after the pivot was detected:

//@version=6 indicator("Plotting in the past", "", true) pHi = ta.pivothigh(5, 5) if not na(pHi) label.new(bar_index[5], na, str.tostring(pHi, format.mintick) + "\n□", yloc = yloc.abovebar, style = label.style_none, textcolor = color.black, size = size.normal)

Note that:

- This script repaints because an elapsed realtime bar showing no price may get a price placed on it if it is identified as a pivot, 5 bars after the actual pivot occurs.
- The display looks great, but it can be misleading.

The best solution to this problem when developing script for others is to plot **without** an offset by default, but give the option for script users to turn on plotting in the past through inputs, so they are necessarily aware of what the script is doing, e.g.:

//@version=6 indicator("Plotting in the past", "", true) plotInThePast = input(false, "Plot in the past") pHi = ta.pivothigh(5, 5) if not na(pHi) label.new(bar_index[plotInThePast ? 5 : 0], na, str.tostring(pHi, format.mintick) + "\n□", yloc = yloc.abovebar, style = label.style_none, textcolor = color.black, size = size.normal)

Dataset variations

Starting points

Scripts begin executing on the chart's first historical bar, and then execute on each bar sequentially, as is explained in this manual's page on Pine Script's execution model. If the first bar changes, then the script will often not calculate the same way it did when the dataset began at a different point in time.

The following factors have an impact on the quantity of bars you see on your charts, and their *starting point*:

- The type of account you hold
- The historical data available from the data supplier
- The alignment requirements of the dataset, which determine its *starting point*

These are the account-specific bar limits:

- 40000 historical bars for the Ultimate plan.
- 25000 historical bars for the Expert plan.
- 20000 historical bars for the Premium plan.
- 10000 historical bars for Essential and Plus plans.
- 5000 historical bars for other plans.

Starting points are determined using the following rules, which depend on the chart's timeframe:

- **Tick-based timeframes**: return the exact number of bars based on the plan.
- **Second-based timeframes**: aligns to the beginning of a day.

- **1 - 14 minutes**: aligns to the beginning of a week.
- **15 - 29 minutes**: aligns to the beginning of a month.
- **30 - 1439 minutes**: aligns to the beginning of a year.
- **1440 minutes and higher**: aligns to the first available historical data point.

As time goes by, these factors cause your chart's history to start at different points in time. This often has an impact on your scripts calculations, because changes in calculation results in early bars can ripple through all the other bars in the dataset. Using functions like ta.valuewhen(), ta.barssince() or ta.ema(), for example, will yield results that vary with early history.

Revision of historical data

Historical and realtime bars are built using two different data feeds supplied by exchanges/brokers: historical data, and realtime data. When realtime bars elapse, exchanges/brokers sometimes make what are usually small adjustments to bar prices, which are then written to their historical data. When the chart is refreshed or the script is re-executed on those elapsed realtime bars, they will then be built and calculated using the historical data, which will contain those usually small price revisions, if any have been made.

Historical data may also be revised for other reasons, e.g., for stock splits.

Sessions

Introduction

Exchanges define a *session* for every symbol, which represents the times of day and days of the week in which the symbol can be traded. Exchanges might also define sessions other than the default one, which are called *subsessions*. Subsessions can be shorter or longer than the default session. If different sessions are available for a symbol, users can switch between them either from the “Sessions” controls in the bottom-right corner of the chart or from the chart’s “Settings/Symbol/Session” menu.

Programmers can use built-in functions and variables to define custom sessions, determine whether bars belong to specific sessions, retrieve data from named subsessions, and access session-related market states.

Time-based sessions

A script can define a custom session by encoding the start time, end time, and, optionally, days of the week of the session into a *session string*. Scripts often use time-based session strings to check whether a bar belongs to certain time period.

Creating time-based sessions

Time-based session strings have the following syntax:

```
<time_period>:<days>
```

Where:

- `<time_period>` specifies the session’s start and end times in `"HHmm-HHmm"` format, where `"HH"` represents the *hour* in 24-hour format (`"00"` to `"23"`) and `"mm"` represents the *minute* (`"00"` to `"59"`) – for example, `"1700"` for 5PM. A comma can separate multiple time periods to specify combinations of discrete periods for the session, e.g., `"0800-0900,1230-1630"`.
- `<days>` specifies the *days of the week* that the session applies to, using a set of digits from 1 to 7 to represent each day. The digits use `"1"` to represent Sunday, and count up through the week, ending with `"7"` to represent Saturday. `"0"` is not a valid day. If unspecified, the session applies every day.

The following table shows some examples of session strings:

Example	Description
`"0000-0000:1234567"`	The normal format for a 7-day, 24-hour session beginning at midnight.
`"0000-0000"`	Equivalent to the previous example, because the default days are `"1234567"`.
`"0000-0000:23456"`	A 24-hour session beginning at midnight, but only Monday to Friday.
`"2000-1630:1234567"`	An overnight session that begins at 20:00 and ends at 16:30 the next day. It applies on all days of the week.

Example	Description
`"0930-1700:146"`	A session that begins at 9:30 and ends at 17:00 on Sundays (1), Wednesdays (4), and Fridays (6).
`"1700-1700:23456"`	An *overnight session*. The Monday session starts Sunday at 17:00 and ends Monday at 17:00. It applies Monday through Friday.
`"1000-1001:26"`	An unusual session that lasts only one minute on Mondays (2) and Fridays (6).
`"0900-1600,1700-2000"`	A session that begins at 9:00, breaks from 16:00 to 17:00, and continues until 20:00. Applies to every day of the week.

Note that a special format exists to represent a 7-day, 24-hour session beginning at midnight: `"24x7"` – this session string is equivalent to the first two examples in the table above.

Using time-based sessions

The `time()` and `time_close()` functions can accept time-based session strings as their `session` parameter arguments:

- The time() function returns a UNIX timestamp for the *opening time* of the current bar, or na if the bar is not in the specified session.
- The time_close() function returns a UNIX timestamp for the *closing time* of the current bar, or na if the bar is not in the specified session.

By testing for a returned na value, scripts can use the above functions to check whether a particular bar falls within a certain session.

To interpret the time zone of the specified `session`, the time() and time_close() functions use the time zone of the exchange by default, unless a `timezone` argument is specified. This time zone can be different from the chart time zone, depending on the chart's settings. For more information on time zones, see the Time zones section of the Time page.

Additionally, the input.session() function also takes a time-based session string as its `defval` argument, to determine the input's default value. Using this input type, users can define session times (but not days of the week) from a script's "Inputs" tab. See the Session input section for more information.

> NoteThe three functions mentioned above are the *only* ones that accept time-based string arguments. Scripts cannot use `request.*()` functions to get data from tickers created using time-based sessions – such usage requires named sessions.

The following example script checks whether the start and end time of a bar fall within a user-defined session. If the bar's opening time, as returned by time(), is within the session (i.e., the value is not na), the script draws a label above the bar. Similarly, if the closing time returned by time_close() is not na, it draws a label below the bar. The labels display the bar open or close times and compare them to the selected session. Here, we run the script on an hourly chart with a short default morning session of "0900-1130":

//@version=6 indicator("Session bar checker", overlay = true) //@variable The session to check for. string sessionInput = input.session(defval = "0900-1130", title = "Session") // Check whether the bar open and close times are within the session. bool isBarOpenInSession = not na(time("", sessionInput)) bool

isBarCloseInSession = not na(time_close("", sessionInput)) // If the bar open time is in the session, show the bar opening time and the session in a label. if isBarOpenInSession label.new(x = bar_index, y = high, text = "Bar open: " + str.format_time(time, "HH:mm") + "\nis in session " + sessionInput, color = color.green, style = label.style_label_down, textcolor = chart.fg_color, size = size.large) // If the bar close time is in the session, show the bar *closing* time and the session in a label. if isBarCloseInSession label.new(x = bar_index, y = low, text = "Bar close: " + str.format_time(time_close, "HH:mm") + "\nis in session " + sessionInput, color = color.red, style=label.style_label_up, textcolor = chart.fg_color, size = size.large)

Note that:

- The script draws labels for the opening and closing times of *all* bars that start within the session, even though the closing time of the last chart bar is *outside* the session. This is because the time() and time_close() functions create their own bar representations according to their parameters. In the image above, which is of an hourly chart, the session ends at 11:30, so the final calculated bar representation in the session starts at 11:00 and ends at 11:30. Therefore, the last bar's end time is reported as being within the session, even though the chart bar ends at 12:00.

> NoticeTo avoid unexpected results, align the start and end times of time-based sessions with the start and end times of chart bars at the expected timeframe.

Scripts can create *dynamic* sessions, whose values can change during script execution, by calculating a "series string" argument for the `session` parameter of the time() or time_close() functions. The following example script creates a dynamic time-based session string that differs on weekdays and weekends. The script uses the time() function to determine whether the current bar is within this dynamic session, and colors the background green if so:

//@version=6 indicator("Dynamic session by day", overlay = true) // Define the weekday and weekend sessions. //@variable A time-based session string that defines the session for weekdays (Mon-Fri). string weekdaySessionInput = input.session(defval = "0800-1900:23456", title = "Weekday Session") //@variable A time-based session string that defines the session for weekends (Sat-Sun). string weekendSessionInput = input.session(defval = "1000-1200:17", title = "Weekend Session") //@variable A "series string" for the session, which sets the session times depending on what day it is. string dynamicSession = (dayofweek >= dayofweek.monday and dayofweek <= dayofweek.friday) ? weekdaySessionInput : weekendSessionInput // Use the `dynamicSession` string in the `time()` function to check if bar opening time is in the dynamic session. //@variable Is `true` if the bar opens within the `dynamicSession` time, i.e., if `time()` does not return `na`. bool isBarOpenInSession = not na(time(timeframe.period, dynamicSession)) // Color the background if the bar is within the session. bgcolor(isBarOpenInSession ? color.new(color.green, 50) : na)

Scripts can retrieve the opening and closing times for a bar other than the current bar by using the `bars_back` parameter of the time() and time_close() functions. For a positive `bars_back` value, the functions count that number of bars backward relative to the current bar, i.e., they retrieve times from *past* bars. Passing a negative `bars_back` integer retrieves the UNIX timestamp for a bar up to 500 bars in the *future*.

The following script determines if a user-defined session is currently active by checking if the last bar is in the session. If so, the script displays the ending time of the active session in a label positioned on the future bar that marks the end of the session. To find the last valid bar closing time in the session, the script uses a

loop to increment a dynamic `bars_back` argument for time_close(), stopping the loop when the returned closing time is na. If the session is not active, the label displays a message to that effect at the current bar.

On the example chart below, we added a vertical line using the chart's drawing tools to show that the bar time matches the session label:

//@version=6 indicator("End of this session", overlay = true) //@variable The session to check for. It applies every day of the week. string sessionInput = input.session("0900-1700:1234567", "Session") if barstate.islast bool isInSession = not na(time(timeframe.period, sessionInput)) //@variable A UNIX timestamp for the closing time of the final bar in the active session, used to position the label. // If session is not active, then the `time_close` of the current bar anchors the label instead. var int sessionEndTime = time_close // If the current bar is in the session, search forwards for the end of the session. if isInSession for i = 1 to 499 // Check closing time of the next future bar, using dynamic `bars_back` argument. //@variable On each loop iteration, holds the closing time of the next future bar, to test if `na`. int futureBarCloseTime = time_close(timeframe.period, sessionInput, bars_back = -i) if na(futureBarCloseTime) break else // Update `sessionEndTime` to hold the last valid closing time found. sessionEndTime := futureBarCloseTime // Draw a label to show the session's end time. If bar is not in the session, display a message to that effect. var label sessionLabel = label.new(na, na, xloc = xloc.bar_time, yloc = yloc.price, text = na, color = color.new(color.green, 50), style = label.style_label_left, textcolor = chart.fg_color) //@variable The timestamp used to anchor the label either at the session's end or the current bar's closing time. int labelTime = isInSession ? sessionEndTime : time_close //@variable If session is active, text shows the "string" representation of the session's ending time. string labelText = isInSession ? "This session ends:\n" + str.format_time(labelTime, "HH:mm") : "Current bar is not in session" // Update the `x`, `y`, and `text` properties of the label. sessionLabel.set_xy(labelTime, open) sessionLabel.set_text(labelText)

Note that:

- We use barstate.islast to avoid unnecessary historical calculations, because we want to highlight only the current session.
- The script supplies a negative integer `-i` as the argument to the `bars_back` parameter to return the closing times for *future* bars.
- We use the `break` keyword to exit the for loop as soon as we reach the end of the session. To learn more about this keyword, see the Keywords and return expressions section of the Loops page.
- The `sessionInput` end time must be within the trading period of the symbol on the chart in order for the script to display the label.
- For an extended example of using this technique to visually identify sessions, see the How can I make an entire custom session visible? entry in the Times, dates, and sessions FAQ.

Named sessions

Exchanges often define *named subsessions*. These sessions can differ from the default session in one or more ways:

- They can include extended hours data, for example, pre-market and post-market trades.
- They can separate longer electronic trading sessions from shorter regular trading sessions, which is common in some futures markets.
- They can define some other periods of interest.

Traders can use named sessions to focus on trading periods with greater volume, or for other regional or timing purposes. A script can use named sessions to retrieve data from a different session than that of the chart, or to maintain the session that the script uses for its calculations even if the user changes the chart session.

To use data from a named session, first identify the exact name of the session, then create a modified ticker that uses that session, and finally request data from that ticker. The sections below discuss these steps in more detail.

Retrieving named sessions

Unlike custom time-based session strings, which are user-defined, session names are *fixed*. Scripts can retrieve the active session's name automatically. Programmers can also supply predefined session names in the code.

The following example script retrieves the name of the active session from the current chart using syminfo.session and displays it in a table. The example chart below shows the script running on an hourly chart of the US stock "NASDAQ:AAPL". We selected "Extended trading hours" from this chart's "Sessions" menu (shown in the bottom-right corner of the image), so the session string displayed in the table is `"extended"`:

//@version=6 indicator("Display active session name", overlay = true) if barstate.islast //@variable A table that displays the name of the active session on the current chart. table sessionTable = table.new(position = position.top_right, columns = 2, rows = 1, border_width = 1) sessionTable.cell(column = 0, row = 0, text = "Active session:", text_color = color.white, bgcolor = color.green, text_size = size.large) sessionTable.cell(column = 1, row = 0, text_font_family = font.family_monospace, text = syminfo.session, text_color = color.white, bgcolor = color.green, text_size = size.large)

If we select "Regular trading hours" from the chart settings, the script displays the session string `"regular"`.

For most US equities, the string `"regular"` is equivalent to the built-in constant session.regular, and the string `"extended"` is equivalent to the built-in constant session.extended. However, this is **not always** the case. Let's look at the same script applied to the "S&P 500 E-mini futures" chart (ticker "ES1!"), with the "Electronic trading hours" session selected:

In the example above, the table shows that the active session is `"regular"`, even though the chart displays the "Electronic trading hours" session, which is *longer* than the "Regular trading hours" session. If we switch to "Regular trading hours" on this chart, the active session is `"us_regular"`, *not* `"regular"`.

> NoticeFor most futures contracts, the longer, **electronic** session "ETH" is considered the default, and therefore uses the session `"regular"`. There is no "Extended trading hours" session available on the chart, and using session.extended is equivalent to session.regular.

Now let's look at some non-standard named sessions. Applying our previous example script to the "DAX Futures" chart (ticker "FDAX1!"), we can choose between the "Regular trading hours", "Xetra trading hours", and "Frankfurt trading hours" sessions on the chart, and the script displays the active session as `"regular"`, `"xetr_regular"`, and `"fwb_regular"`, respectively:

Creating a session-specific ticker

A script can create a ticker that uses a specific session by using ticker.new() or ticker.modify(). Both functions create a new ticker identifier, which can specify additional session and pricing modifiers for the requested context. The only practical difference between the two functions is that ticker.new() creates a ticker from an exchange `prefix` and `ticker` name (two separate "string" arguments), whereas ticker.modify() modifies a full ticker ID (`"prefix:ticker"` as one "string" argument, or a `tickerid` string with additional modifiers returned from `ticker.*()`).

For more information about the available `ticker.*()` functions, see the Custom contexts section of the Other timeframes and data page.

The example script below creates the following five tickers for the "NASDAQ:AAPL" US equity and displays them in a table for comparison:

1. A new ticker with the default session, using ticker.modify().
2. A new ticker with the default session, using ticker.new().
3. A new ticker with an extended session, using ticker.new().
4. A modified version of the first ticker with an extended session, using ticker.modify().
5. A new ticker with an extended session, using ticker.modify().

//@version=6 indicator("Creating session-specific tickers") //@variable A new ticker ID, created using `ticker.modify()` with no optional parameters (default session). string ticker1 = ticker.modify("NASDAQ:AAPL") //@variable A new ticker ID, created using `ticker.new()` with no optional parameters (default session). string ticker2 = ticker.new("NASDAQ", "AAPL") //@variable A new ticker ID for "NASDAQ:AAPL" with an extended session, created using `ticker.new()`. string ticker3 = ticker.new("NASDAQ", "AAPL", session.extended) //@variable A modified version of `ticker1`, using `ticker.modify()` to modify to an extended session. string ticker4 = ticker.modify(ticker1, session.extended) //@variable A new ticker ID for "NASDAQ:AAPL" with an extended session, created using `ticker.modify()`. string ticker5 = ticker.modify("NASDAQ:AAPL", session.extended) // Display all the tickers. if barstate.islastconfirmedhistory //@variable A `table` that displays the values of the different ticker ID strings created. table tickerTable = table.new(position = position.top_right, columns = 1, rows = 5, border_width = 1) tickerTable.cell(0, 0, "Ticker 1 " + ticker1, text_color = color.white, bgcolor = color.green) tickerTable.cell(0, 1, "Ticker 2 " + ticker2, text_color = color.white, bgcolor = color.green) tickerTable.cell(0, 2, "Ticker 3 " + ticker3, text_color = color.white, bgcolor = color.green) tickerTable.cell(0, 3, "Ticker 4 " + ticker4, text_color = color.white, bgcolor = color.green) tickerTable.cell(0, 4, "Ticker 5 " + ticker5, text_color = color.white, bgcolor = color.green)

Note that:

- The `ticker.modify("NASDAQ:AAPL")` function call always returns only a ticker. Requesting data from this ticker returns values from the *regular session* of the equity, regardless of the session settings of the chart.
- The `ticker.new("NASDAQ", "AAPL")` call returns a ticker with extra information encoded, representing the defaults for extra optional parameters (such as settlement options for futures contracts), only for tickers where optional parameters are available. The returned ticker also encodes the session, if a non-default session is selected on the chart and the interval is intraday.
- All the other function calls *always* return tickers with session information, because the session is specified in the call. The tickers can also contain information representing optional parameters.

- The script shows the tickers for the “NASDAQ:AAPL” symbol, regardless of the symbol that the chart displays, because the ticker information for that symbol is passed to the `ticker.*()` calls. To create tickers representing the chart symbol, use `syminfo.*` variables like syminfo.prefix, syminfo.ticker, and syminfo.tickerid.

The previous example demonstrates that the two ticker creation functions are largely equivalent. For consistency, we use ticker.new() in our examples below.

Requesting data from session-specific tickers

Scripts use session-specific tickers in request.security() calls to retrieve data from that particular session.

This simple example script visualizes the close prices of the current asset from both the regular and extended sessions, using the syminfo.prefix and syminfo.ticker variables to create session-specific tickers for the symbol currently on the chart. It plots the prices from the extended session as a black line, and the prices from the regular session as red circles. First, we run the script on a 30-minute chart of “NASDAQ:AAPL”, with the “Extended trading hours” session selected:

//@version=6 indicator("Visualizing extended session data") //@variable The ticker ID for the extended session of the current chart symbol. string extendedTicker = ticker.new(syminfo.prefix, syminfo.ticker, session.extended) //@variable The ticker ID for the regular session of the current chart symbol. string regularTicker = ticker.new(syminfo.prefix, syminfo.ticker, session.regular) //@variable The `close` price requested from the extended session of the chart's symbol. float extendedClose = request.security(extendedTicker, timeframe.period, close, barmerge.gaps_on) //@variable The `close` price requested from the regular session of the chart's symbol. float regularClose = request.security(regularTicker, timeframe.period, close, barmerge.gaps_on) // Plot the `extendedClose` with a black line, and the `regularClose` with red circles. plot(extendedClose, style = plot.style_linebr, color = color.black, linewidth = 2, title = "Extended Session Data") plot(regularClose, style = plot.style_circles, color = color.red, linewidth = 4, title = "Regular Session Data")

Note that:

- The chart automatically highlights the backgrounds for extended hours based on its “Symbol/Data Modification” settings; this is not controlled by the script.
- The `plot(regularClose)` call does not plot any circles during the pre-market and post-market sessions. This is because the request.security() call for `regularClose` allows data gaps by using barmerge.gaps_on, so it returns na for chart bars outside the regular trading session.
- The extended and regular closing prices have the same values on the 30-minute chart above. Running the same script on an *hourly* chart instead produces *different* values for the extended and regular closing prices, because the regular session starts at 09:30 and not on the hour.

Now let’s run the same script on the “S&P 500 E-mini futures” chart (ticker “ES1!”), with the “Electronic trading hours” session selected:

Notice that *both* plots are exactly the same, covering the entire extended trading session. This is because, as we saw in the Retrieving named sessions section, most US futures symbols use `"regular"` and `"us_regular"` as their session names. We can update our code to add a third plot that uses the `"us_regular"` session:

```
//@version=6 indicator("Visualizing extended session data") //@variable The ticker ID for the extended session of the current chart symbol. string extendedTicker = ticker.new(syminfo.prefix, syminfo.ticker, "extended") //@variable The ticker ID for the regular session of the current chart symbol. string regularTicker = ticker.new(syminfo.prefix, syminfo.ticker, "regular") //@variable The ticker ID for the "us_regular" session of the current chart symbol. string usRegularTicker = ticker.new(syminfo.prefix, syminfo.ticker, "us_regular") //@variable The `close` price requested from the extended session of the chart's symbol. float extendedClose = request.security(extendedTicker, timeframe.period, close, barmerge.gaps_on) //@variable The `close` price requested from the regular session of the chart's symbol. float regularClose = request.security(regularTicker, timeframe.period, close, barmerge.gaps_on) //@variable The `close` price requested from the "us_regular" session of the chart's symbol. float usRegularClose = request.security(usRegularTicker, timeframe.period, close, barmerge.gaps_on) // Plot the `usRegularClose` with a blue line. plot(usRegularClose, style = plot.style_linebr, color = color.blue, linewidth = 6, title = "US Regular Session Data") // Plot the `extendedClose` with a black line, and the `regularClose` with red circles. plot(extendedClose, style = plot.style_linebr, color = color.black, linewidth = 2, title = "Extended Session Data") plot(regularClose, style = plot.style_circles, color = color.red, linewidth = 4, title = "Regular Session Data")
```

Note that:

- The new `usRegularClose` plot, as we expect, displays prices only during the regular trading hours.
- We replaced session.regular with the string `"regular"`, and session.extended with the string `"extended"` in the other two ticker.new() function calls, just to show that these values are equivalent.

Lastly, let's look at an example of using data from non-standard sessions. By applying the first example script from the Retrieving named sessions section to the "DAX Futures chart" (ticker "FDAX1!"), we discovered that the chart sessions "Regular trading hours", "Xetra trading hours", and "Frankfurt trading hours" have the named sessions `"regular"`, `"xetr_regular"`, and `"fwb_regular"`, respectively. The following example plots the chart's close prices with a blue line, and requests the "Frankfurt trading hours" session's close prices to plot with teal circles. Here we run the script on the hourly "FDAX1!" chart with "Regular trading hours" selected on the chart:

```
//@version=6 indicator("Visualizing non-standard session data") //@variable The ticker ID for the "Frankfurt trading hours" session of the current chart symbol. string fwbRegularTicker = ticker.new(syminfo.prefix, syminfo.ticker, "fwb_regular") //@variable The `close` price requested from the "Frankfurt trading hours" session of the chart's symbol. float fwbClose = request.security(fwbRegularTicker, timeframe.period, close, barmerge.gaps_on) // Plot the current chart `close` with a blue line, and the `fwbClose` with teal circles. plot(close, style = plot.style_linebr, color = color.blue, linewidth = 3, title = "Chart Session Data") plot(fwbClose, style = plot.style_circles, color = color.teal, linewidth = 4, title = "Frankfurt Session Data")
```

Note that:

- The "Frankfurt trading hours" session is shorter than the "Regular trading hours" session.
- Running this script on a chart that *does not* define a named session `"fwb_regular"` plots circles for *all* the bars.

> NoteIf a script attempts to retrieve data using a named session that does not exist for that symbol, the default session is used instead.

Session variables reference

Programmers can use several built-in variables for session-related data.

Market states

The following Boolean variables track whether the current bar belongs to the pre-market or post-market session:

Variable	Description
session.ismarket	Is `true` when the bar belongs to *regular* trading hours. On “1D” and above timeframes, this variable is always `true`.
session.ispremarket	Is `true` when the bar belongs to the extended session *preceding* regular trading hours. Extended hours data is only shown on intraday timeframes; on “1D” and above, this variable is always `false`.
session.ispostmarket	Is `true` when the bar belongs to the extended session *following* regular trading hours. Extended hours data is only shown on intraday timeframes; on “1D” and above, this variable is always `false`.

For tickers without pre-market and post-market sessions, such as “BTCUSD”, session.ismarket is always `true` and session.ispremarket and session.ispostmarket are always `false`.

For many futures symbols, Electronic trading hours (ETH) are considered the default session and use the named session `"regular"`, so during those hours session.ismarket is `true` and session.ispremarket and session.ispostmarket are both `false`.

First and last bars

The following Boolean variables track whether the current bar is the first or last in different sessions:

Variable	Description
session.isfirstbar	Is `true` if the current bar is the first bar of the day’s session, and `false` otherwise. If extended session information is used, it is only `true` on the first bar of the pre-market bars. Is `true` once for every session on the chart.
session.isfirstbar_regular	Is `true` on the first regular session bar of the day, `false` otherwise. It is `true` only once per session. The result is the same whether extended session information is used or not. For futures, the “Electronic trading hours” session *is* the regular session. This variable is always `false` when the ticker is configured to use a subsession.
session.islastbar	Is `true` if the current bar is the last bar of the day’s session, and `false` otherwise. If extended session information is used, it is only `true` on the last bar of the post-market bars.
session.islastbar_regular	Is `true` on the last regular session bar of the day, `false` otherwise. The result is the same whether extended session information is used or not.

The session.islastbar and session.islastbar_regular variables might not be `true` for any bar in a session if no price or volume updates occur during the time period of the last bar. This is more likely at lower timeframes for thinly traded symbols. In contrast, session.isfirstbar and session.isfirstbar_regular are always `true` once for any session.

Named session variables

Scripts can use the following “string” variables to work with named sessions. The Retrieving named sessions section of this page discusses the use of these variables.

Variable	Description
syminfo.session	Holds the current symbol’s session information.
session.regular	Represents the regular trading session.
session.extended	Represents the extended trading session.

Strategies

Introduction

Pine Script® strategies are specialized scripts that simulate trades across historical and realtime bars, allowing users to backtest and forward test their trading systems. Strategy scripts have many of the same capabilities as indicator scripts, and they provide the ability to place, modify, and cancel hypothetical orders and analyze performance results.

When a script uses the strategy() function as its declaration statement, it gains access to the `strategy.*` namespace, which features numerous functions and variables for simulating orders and retrieving essential strategy information. Additionally, the script generates a detailed strategy report in a dedicated tab below the chart.

A simple strategy example

The following script is a simple strategy that simulates entering a long or short position when two moving averages (MAs) cross. If the fast MA crosses over the slow MA, the strategy places a market order named “Buy” to enter a long position. If the fast MA crosses under the slow MA, it places a market order named “Sell” to open a short position instead:

//@version=6 strategy("Simple strategy demo", overlay = true) //@variable Defines the lengths of the fast and slow moving averages. int lengthInput = input.int(14, "Base length", minval = 2) // Calculate a fast MA using the `lengthInput` value and a slow MA using twice that value. float fastMA = ta.sma(close, lengthInput) float slowMA = ta.sma(close, lengthInput * 2) // If the fast MA crosses over the slow MA, place an order to close any short position and enter a long position. if ta.crossover(fastMA, slowMA) strategy.entry("Buy", direction = strategy.long) // If the fast MA crosses under the slow MA, place an order to close any long position and enter a short position. if ta.crossunder(fastMA, slowMA) strategy.entry("Sell", direction = strategy.short) // Plot both moving averages for reference. plot(fastMA, "Fast MA", color.aqua) plot(slowMA, "Slow MA", color.orange)

Note that:

- The strategy() function call declares that the script is a strategy named “Simple strategy demo” that displays visuals on the main chart pane.
- The strategy.entry() function is the command that the script uses to create *entry orders* and reverse positions. The “Buy” entry order closes any short position and opens a new long position. The “Sell” entry order closes any long position and opens a new short position.

Applying a strategy to a chart

To test a strategy, add it to the chart. Select a built-in, published, or personal strategy from the “Indicators, metrics, and strategies” menu, or write a custom strategy in the Pine Editor and select the “Add to chart” button from the editor’s options:

The script plots trade markers on the bars in the main chart pane and displays a detailed *strategy report* within a separate tab in the chart’s bottom panel:

See the Strategy report section below to learn how to read and interpret the performance data displayed by this tab.

> Notice
>
> The performance results from a strategy applied to *non-standard charts* (Heikin Ashi, Renko, line break, Kagi, point & figure, and range) **do not** reflect actual market conditions by default. The strategy simulates trades using the chart's **synthetic** prices, which do not represent real-world market prices. Consequently, running a strategy on a non-standard chart typically produces **unrealistic** results.
>
> Therefore, we strongly recommend using **standard** chart types when testing strategies. Alternatively, on Heikin Ashi charts, users can simulate order fills using actual prices by selecting the "Standard bars" option from the "Heikin Ashi mode" input in the script's "Settings/Properties" tab. Programmers can specify that a strategy uses this behavior by default by including `fill_orders_on_standard_ohlc = true` in the strategy() declaration statement.

Strategy report

The *strategy report* visualizes the hypothetical trading performance of a simulated strategy. The report automatically appears within a tab in the chart's bottom pannel if at least one strategy script is active on the chart. The report displays performance results for only *one* strategy at a time. If two or more strategies are active on the chart, users can specify which strategy to analyze by selecting its name from the *context menu* opened by the dropdown arrow in the tab's header:

After the selected script executes across the chart's data, the strategy report populates two main sub-tabs with relevant data from the strategy simulation:

- The "Metrics" tab provdes a detailed summary of the strategy's performance. The sections in the tab include a time-series chart for analyzing trade activity and equity growth, multiple graphs for analyzing essential performance data, and several useful metrics for assessing the strategy's long, short, and overall trading performance.
- The "Trades" tab displays a list of the strategy's simulated trades in chronological order. The items in the list show essential information about each trade, including the trade's number and direction, the entry and exit prices, and multiple trade-wise performance metrics.

Users can switch between these sub-tabs by selecting the icons in the top-left corner of the strategy report.

> TipUsers can also *download* all applicable data that populates these sub-tabs by selecting the "Download data as XLSX" option in the context menu.

The additional items next to the "Metrics" and "Trades" icons at the top of the strategy report provide quick options where users can customize the testing period and enable Deep Backtesting mode, set the strategy's initial capital and account currency, control the level of historical bar detail in the simulation, and adjust the script's execution settings, respectively:

When a script author publishes a strategy, the publication's script page includes a *compact* version of the strategy report that displays hypothetical performance results from running the script with specific properties on the published chart. The publication's report contains similar information to the report displayed below a user's chart. It also contains an additional "Properties" tab, which displays the input values and properties that the author used while preparing the publication.

"Metrics" tab

The "Metrics" tab of the strategy report provides a detailed summary of a strategy's performance over a sequence of simulated trades. It organizes the performance details into five main sections:

- Key stats: Displays key overall performance metrics, as well as a chart that plots the series of cumulative equity changes and other overall performance values.
- Return details: Displays graphs and tables for inspecting the strategy's overall returns and costs, the strategy's profitability and potential relative to a buy-and-hold system, and the strategy's risk-adjusted performance ratios.
- Trades analysis: Displays graphs to represent the distribution of trade outcomes, as well as tables containing key trade-wise performance metrics.
- Equity run-ups and drawdowns: Displays graphs and tables with essential information about the strategy's positive and negative equity fluctuations.
- Capital efficiency: Displays overall metrics relating to the strategy's capital and margin at the top, as well as a chart and tables containing details about the strategy's capital usage, margin usage, overall returns, and margin calls.

Some of the tables in this tab contain "All", "Long", and "Short" columns. The "All" column shows the performance metrics for all simulated trades. The "Long" and "Short" columns show relevant metrics separately for long and short trades.

The following sections explain the types of information that each part of the "Metrics" tab contains and how it displays the data.

> TipMost of the tables in the "Metrics" tab reveal a "Show description" icon next to a metric's name when the user hovers over the metric's row. Clicking that icon opens a Help Center article containing details about the metric's calculations and meaning. Refer to the Strategy report metrics page in our Help Center for a list of these metrics and their corresponding articles.

Key stats

The "Key stats" section of the strategy report's "Metrics" tab provides a quick overview of the strategy's overall performance. The top of the section displays a few essential performance metrics, including the strategy's total profit or loss, the maximum drawdown, the total number of profitable trades compared to the total number of closed trades, and the strategy's profit factor.

The "Performance" chart below these metrics optionally displays up to four plots to visualize equity growth and trade volatility:

- The "Cumulative PnL" baseline plot displays the cumulative change in the strategy's equity across all closed trades, experessed as a currency amount or a percentage of the strategy's initial capital.

- The “Buy and hold” plot displays the cumulative capital or percentage change in equity for a strategy that opens a single long trade and holds it throughout the entire trading period.
- The “Trades excursions” plot displays green and red columns representing the maximum *unrealized* profit and loss for each trade, expressed as a currency amount or a percentage of the trade’s size.
- The “Run-ups and drawdowns” plot displays green and red horizontal bars to highlight periods of run-up and drawdown in the strategy’s equity across the trading range.

Users can specify the chart’s scale type, download or share an image of the chart, and expand or collapse the chart view by selecting the icons above the chart’s top-right edge.

Hovering over the points on the “Performance” chart anywhere above the “Run-ups and drawdowns” plot reveals a tooltip containing details for a specific trade, including the trade’s number, direction, and closing time. The tooltip also displays the values from the “Cumulative PnL”, “Buy and hold”, and “Trades excursions” plots, depending on which plots are active. When a user clicks the point highlighted by the tooltip, the main price chart automatically scrolls to the chart bar on which the corresponding trade closed, then displays that bar’s time in a temporary tooltip:

Note that:

- The main price chart automatically scrolls to a trade’s closing bar only if the strategy uses the *default* testing period. Selecting a different period from the “Testing period” menu at the top of the tab activates *Deep Backtesting* mode, which does *not* support scrolling the main chart from the report’s “Performance” chart.

Users can also hover over the horizontal bars in the “Run-ups and drawdowns” plot to view a separate tooltip containing the total run-up or drawdown for each displayed period:

Return details

The “Return details” section of the “Metrics” tab analyzes the strategy’s cumulative returns and compares them to a buy-and-hold benchmark. The “Overview” sub-tab in this section displays two graphs:

- The “Profit structure” graph on the left displays bars representing the strategy’s total profit, total loss, total commissions, open profit or loss, and the total net profit or loss.
- The “Benchmarking” graph displays the maximum, minimum, and current total return of the strategy compared to the buy-and-hold return values over the same testing range.

The sub-tab also displays the strategy’s open profit or loss, expected payoff per trade, outperformance, and Sharpe ratio above the graphs for quick reference:

The other sub-tabs within this section display detailed information about the strategy’s overall returns:

- The “Returns” tab displays the strategy’s initial capital, open profit or loss, net profit or loss, gross profit and loss, profit factor, commissions paid, and expected payoff per trade. It displays applicable metrics separately for all trades, all long trades, and all short trades.
- The “Benchmarking” tab displays the overall returns of the buy-and-hold benchmark and the strategy’s outperformance.
- The “Risk-adjusted performance” tab displays the strategy’s Sharpe and Sortino ratios, which gauge a strategy’s risk-adjusted return relative to the risk-free rate specified by the strategy() declaration statement.

Trades analysis

The “Trades analysis” section of the “Metrics” tab analyzes the overall performance of the strategy’s individual trades. The “Overview” sub-tab in this section displays two graphs for analyzing the distribution of trade outcomes:

- The “Returns distribution” histogram graph displays the distribution of trade returns. Each column in the histogram shows the number of trades that closed with returns within a specific range. Red columns represent negative returns (losses), and green columns represent positive returns (profits). The graph also displays dashed vertical lines at the overall average profit and loss values.
- The “Trades distribution” donut graph displays the quantity of winning, losing, and break-even trades relative to the strategy’s total number of trades.

The “Overview” sub-tab also displays the strategy’s average profit or loss, the average number of bars per trade, and the larget profit and loss from a single trade above the two graphs:

The “Trades analysis details” sub-tab displays multiple useful statistics for all trades, all long trades, and all short trades. It includes metrics such as the total number of trades, the total winning and losing trades, the percentage of profitable trades, the average return amounts, the maximum profit and loss, and the number of outliers in the returns distribution:

Equity run-ups and drawdowns

The “Equity run-ups and drawdowns” section of the “Metrics” tab analyzes the strategy’s periods of equity growth (*run-up*) and decline (*drawdown*). The “Overview” sub-tab in this section includes two graphs to simplify run-up and drawdown inspection:

- The “Alternating growth and decline” graph displays vertical bars for periods of run-up and drawdown in chronological order. Green bars represent run-up periods, and red bars represent drawdown periods.
- The “Comparison of growth and decline periods” graph displays horizontal bars representing the maximum, average, and current run-up and drawdown percentages for a quick visual comparison.

The sub-tab also displays the average run-up and drawdown durations, the maximum drawdown as a percentage of the initial capital, and the ratio of the total profit or loss to the maximum drawdown above the graphs:

The “Run-ups” and “Drawdowns” sub-tabs display tables containing detailed metrics for the strategy’s run-up and drawdown periods, including:

- The average duration of each run-up/drawdown period.
- The average run-up/drawdown amount per period.
- The maximum run-up/drawdown across the testing range on both an intrabar and close-to-close basis.
- The maximum intrabar run-up/drawdown as a percentage of the strategy’s initial capital.
- The ratio of the strategy’s total profit or loss relative to the maximum drawdown.

Capital efficiency

The “Capital efficiency” section of the “Metrics” tab evaluates how the strategy uses its simulated funds and available margin. The “Overview” sub-tab contains a “Margin usage” chart that displays the amount of margin used on each trade in order, expressed as a currency amount or a percentage of the strategy’s available funds. The sub-tab also shows the strategy’s compound annual growth rate (CAGR), the minimum account size required to avoid a margin call, the overall return on the initial capital, and the total number of *margin call events*:

The other sub-tabs in this section contain detailed information about the strategy’s capital and margin usage for all trades, all long trades, and all short trades:

- The “Capital usage” tab displays the strategy’s CAGR, the overall return based on the initial capital, the minimum account size, the overall return based on the minimum account size, and the net profit or loss as a percentage of the largest loss.
- The “Margin usage” tab displays the average margin per trade, the maximum margin used on a trade, the average profit per unit of margin, the total number of margin calls, and the total trade volume liquidated by all margin calls.

“Trades” tab

The “Trades” tab of the strategy report lists the strategy’s simulated trades in ascending or descending order by time. The list includes two columns that are always active by default: “Trade number” and “Type”. The “Trade number” column lists each trade’s number and direction. Users can change the sorting order of the list’s items by clicking on the “Trade number” column heading. The list is sorted in descending order by default. The “Type” column contains fields for each trade’s entry and exit order. If the strategy is not running in Deep Backtesting mode, hovering the mouse over either field in a listed item reveals a “Show on chart” icon, which users can click to scroll the main chart to the trade’s entry or exit bar.

The list optionally includes any of the following additional columns, depending on the user’s selections:

- “Date and time”: Shows the entry and exit date and time for each trade.
- “Signal”: Shows the name or comment assigned to each trade’s entry and exit orders.
- “Price”: Shows each trade’s entry and exit price in the instrument’s quoted currency.
- “Size”: Shows each trade’s size, expressed as a number of contracts/lots/shares/units and a quantity of the strategy’s account currency.
- “Return”: Shows each trade’s net return as a percentage of the trade’s size.
- “Favorable excursion”: Shows each trade’s maximum unrealized profit as a currency amount and a percentage of the trade’s size.
- “Adverse excursion”: Shows each trade’s maximum unrealized loss as a currency amount and a percentage of the trade’s size.
- “Cumulative PnL”: Shows the strategy’s total profit or loss at the time of each trade, expressed as a currency amount and a percentage of the strategy’s initial capital.
- “Duration (bars)”: Shows the number of bars for which each trade remained open.

Users can specify which columns to display by selecting the “Column setup” icon above the list’s top-right edge. The adjacent “Download” icon enables users to download all available data for the list as a *CSV* file.

Note that:

- The downloaded CSV file for the list of trades contains data for *all* available columns, regardless of the display columns selected in the “Column setup” menu.

- If a strategy uses the default testing range, it preserves individual data for up to the latest *9000 trades*. If the strategy uses a different testing range, enabling Deep Backtesting mode, it maintains individual trade data for *all* trades. This behavior affects the results available from the list of trades, any downloaded CSV files, and the built-in `strategy.*()` functions that retrieve individual trade information. However, it does *not* affect the data displayed in the "Metrics" tab or accessed by other `strategy.*` built-ins. See the trade limit section to learn more.

"Properties" tab

When a script author publishes a strategy, the publication's script page displays a *compact* version of the strategy report to demonstrate the script's performance results for a specific dataset. The published report also includes an extra *"Properties"* tab, which displays details about the dataset, script inputs, and strategy properties that the author used while preparing the publication. The tab organizes this information into four collapsible sections:

- The "Date range" section shows the selected testing range and the overall available backtesting range.
- The "Symbol info" section displays the chart's symbol, timeframe, type, point value, currency, and tick size. It also includes the chart's specified precision setting.
- The "Strategy inputs" section lists the names and values of all the inputs available in the strategy's "Settings/Inputs" tab. This section appears in the tab only if the script includes `input*()` calls or specifies a nonzero `calc_bars_count` argument in the strategy() declaration statement.
- The "Strategy properties" section provides an overview of the properties that the author specified in the script's "Settings/Properties" tab, including the strategy's initial capital, account currency, order size, leverage, pyramiding, commission, slippage, and other settings.

Note that:

- If users download performance data as an XLSX file from the strategy report below their charts, the exported data from that report also includes "Properties" tab data for reference.

Broker emulator

TradingView uses a broker emulator to simulate trades while running a strategy script. Unlike brokers in real-world trading, the broker emulator fills a strategy's orders using only the available *chart data* by default. Consequently, it executes orders on historical bars *after a bar closes*. Similarly, depending on the selected calculation behavior, the earliest point at which it can fill orders on realtime bars is *after a new tick*. For detailed information about this behavior, refer to the Execution model page.

Because the broker emulator uses only price data from the chart by default, it makes default *assumptions* about intrabar price movement when filling orders on historical bars. The emulator analyzes the opening, high, low, and closing prices of chart bars to infer historical intrabar activity using the following logic:

- If the opening price of a bar is closer to the high than it is to the low, the emulator assumes that the market price moved in this order: **open** → **high** → **low** → **close**.
- If the opening price of a bar is closer to the low than it is to the high, the emulator assumes that the market price moved in this order: **open** → **low** → **high** → **close**.
- When filling *price-based orders* (all orders except market orders), the emulator assumes that *no gaps* exist inside each chart bar; it considers *any* price within the bar's range as a valid level for filling

pending orders.
- If the market price crosses a price-based order's level during the gap between one bar's closing time and the next bar's opening time, the emulator assumes that intrabar data *does not exist* within the gap. Rather than filling the order at the specified price in that case, the emulator fills the order at the *opening price* of the bar following the gap.

The following image labels the OHLC values of a few historical bars using numbers 1-4 to demonstrate the broker emulator's default assumptions about intrabar price movement. The "1" label represents the bar's first tick, and the "4" label represents the last tick:

Adjusting historical bar detail

Users with Premium and Ultimate plans can override the broker emulator's chart-based assumptions and increase the level of intrabar detail on historical bars, allowing for more precise order fills in the strategy's backtest. To enable high historical bar detail, select the "High" option from the strategy's Bar detalization settings, which are available in the "Settings/Properties" tab and at the top of the strategy report below the chart. Programmers can also configure a strategy to use high historical detail by default by including `use_bar_magnifier = true` in the strategy() declaration statement.

If a strategy enables high historical detail, the broker emulator retrieves open, high, low, and close prices from the bars on a suitable *lower timeframe*, when possible, to increase the number of ticks available for estimating price action and filling orders on historical bars. This setting also allows the script to perform multiple *additional executions* on each historical bar, depending on the selected Script execution settings. See the Altering calculation behavior section below to learn more.

The following example illustrates how changing the level of historical bar detail can enhance the behavior of limit orders. The script below creates entry and exit limit orders, named "Buy" and "Exit", on the first bar whose opening time equals or exceeds an input timestamp. For visual reference, the script highlights the chart's background in orange when it places the orders, and it draws two horizontal lines at the order prices. The strategy() statement does not include `use_bar_magnifier = true`. Therefore, the broker emulator uses only chart data to determine when it can fill both orders by default:

//@version=6 strategy("Default historical detail demo", overlay = true, behind_chart = false) //@variable The minimum opening timestamp for the bar on which to place the orders. int orderTime = input.time(timestamp("08 April 2024 00:00"), "Order time") // Declare variables to reference the lines for the entry and exit levels. var line buyLine = na var line exitLine = na //@variable Translucent orange for the bar on which the script creates the orders, and `na` on all other bars. color orderColor = na // Logic to place and visualize the orders if time[1] < orderTime and time >= orderTime // Calculate the order price levels. float buyPrice = hl2 - (high - low) float exitPrice = buyPrice + (high - low) * 0.25 // Place the "Buy" limit order at the `buyPrice` level. strategy.entry("Buy", strategy.long, limit = buyPrice) // Place the "Exit" limit order at the `exitPrice` level. strategy.exit("Exit", "Buy", limit = exitPrice) // Set the `orderColor` value to a translucent orange for the chart's background. orderColor := #ff980033 // Initialize horizontal lines to visualize the order prices. buyLine := line.new(bar_index, buyPrice, bar_index + 1, buyPrice, color = #2962ff, width = 2) exitLine := line.new(bar_index, exitPrice, bar_index + 1, exitPrice, color = #d500f9, width = 2) // Extend the horizontal lines to the current bar when the position closes. if ta.change(strategy.closedtrades) > 0 buyLine.set_x2(bar_index) exitLine.set_x2(bar_index) // Highlight the chart's background on the order bar. bgcolor(orderColor)

If we apply this script to a weekly “NASDAQ:MSFT” chart, the broker emulator fills the “Buy” order one bar after the script creates the orders, then fills the “Exit” order several bars later. On the bar where the “Buy” order fills, the open is closer to the high than it is to the low, so the emulator assumes that the price moved from open to high, high to low, then low to close. Consequently, the emulator infers that after the market price crossed below the blue line, triggering the “Buy” order, it did not move back up and touch the fuchsia line to trigger the “Exit” order on the same bar. In other words, the strategy could not enter and exit the position in the *same* week, according to the broker emulator’s assumptions:

If we include `use_bar_magnifier = true` in the strategy() statement, the strategy enables high historical detail by default. When this setting is active on a weekly chart, the broker emulator retrieves open, high, low, and close prices from the *daily timeframe*. The bars on the daily timeframe show that, contrary to the broker emulator’s default assumption, the market price *did* move to the “Exit” order’s price after triggering the “Entry” order in the same week. Therefore, the emulator can fill both orders on the *same weekly bar* in this case. Below, we show the strategy’s result on the weekly chart after enabling high historical detail, alongside the bars on the daily chart with the entry and exit points annotated:

//@version=6 strategy("High historical detail demo", overlay = true, behind_chart = false, use_bar_magnifier = true) //@variable The minimum opening timestamp for the bar on which to place the orders. int orderTime = input.time(timestamp("08 April 2024 00:00"), "Order time") // Declare variables to reference the lines for the entry and exit levels. var line buyLine = na var line exitLine = na //@variable Translucent orange for the bar on which the script creates the orders, and `na` on all other bars. color orderColor = na // Logic to place and visualize the orders if time[1] < orderTime and time >= orderTime // Calculate the order price levels. float buyPrice = hl2 - (high - low) float exitPrice = buyPrice + (high - low) * 0.25 // Place the "Buy" limit order at the `buyPrice` level. strategy.entry("Buy", strategy.long, limit = buyPrice) // Place the "Exit" limit order at the `exitPrice` level. strategy.exit("Exit", "Buy", limit = exitPrice) // Set the `orderColor` value to a translucent orange for the chart's background. orderColor := #ff980033 // Initialize horizontal lines to visualize the order prices. buyLine := line.new(bar_index, buyPrice, bar_index + 1, buyPrice, color = #2962ff, width = 2) exitLine := line.new(bar_index, exitPrice, bar_index + 1, exitPrice, color = #d500f9, width = 2) // Extend the horizontal lines to the current bar when the position closes. if ta.change(strategy.closedtrades) > 0 buyLine.set_x2(bar_index) exitLine.set_x2(bar_index) // Highlight the chart's background on the order bar. bgcolor(orderColor)

Note that:

- We used the Horizontal line, Vertical line, and Callout drawing tools to annotate the daily chart on the right.

> Note
>
> A strategy’s “Bar detalization” setting *does not* affect the level of bar detail on the “1S” or “1T” timeframes. The broker emulator always analyzes *one* tick per bar on a “1T” chart, and *four* ticks per bar on a “1S” chart.
>
> Scripts can request a maximum of *200,000* bars from any lower timeframe. Due to this limitation, the requested datasets for some symbols and timeframes might not include intrabar coverage for early bars on the chart. The broker emulator uses its default assumptions when filling orders on bars that do not have intrabar data on the requested timeframe.

Orders and trades

Pine Script strategies use orders to initiate trades and manage positions, similar to real-world trading. In this context, an *order* is an instruction that a strategy sends to the broker emulator to perform a market action, and a *trade* is the resulting transaction that opens after the emulator fills an entry order. A market position is total of all open trades.

Let's take a closer look at how strategy orders work and how they become trades. Every 20 bars, the following script creates a long market order with strategy.entry() and draws a label. It calls strategy.close_all() on each bar from the global scope to generate a market order to close any open position:

//@version=6 strategy("Order execution demo", overlay = true, behind_chart = false) //@variable Is `true` on every 20th bar, and `false` otherwise. bool longCondition = bar_index % 20 == 0 // Place a long market order and draw a label when the `longCondition` value is `true`. if longCondition strategy.entry("My Long Entry ID", direction = strategy.long) label.new(bar_index, high, text = "Long entry order created", color = color.lime, style = label.style_label_lower_right, textcolor = color.black, size = size.large) // Place a market order to close any open position. strategy.close_all()

Note that:

- Although the script calls strategy.close_all() on every bar, the function creates a new exit order only if the strategy has an *open position*. If there is no open position, the function call has no effect.

The blue arrows on the above chart show where the strategy entered a long position, and the purple arrows mark the bars where the strategy closed the position. Notice that the label drawings appear one bar *before* the entry markers, and the entry markers appear one bar *before* the closing markers. This sequence illustrates order creation and execution in action.

By default, the earliest point at which the broker emulator fills an order is on the next available tick, because creating and filling an order on the *same* tick is *unrealistic*. If a strategy uses the default calculation behavior, it updates its calculations only after a bar closes, meaning the next tick on which an order can fill is at the *open* of the *following bar*. For example, when the above script's `longCondition` value is `true` on bar 20, the script places an entry order that fills on the next available tick, which is at the open of bar 21. When the script updates its calculations at the close of bar 21, it then places an exit order to close the current position on the following tick, at the open of bar 22.

Order types

Pine Script strategies can simulate different order types to suit specific trading system needs. The main order types include market, limit, stop, and stop-limit.

Market orders

A *market order* is the simplest type of order. A market order is an instruction to buy or sell an instrument as soon as possible, irrespective of the price. Therefore, the broker emulator always executes it on the next available tick. Most order placement commands generate market orders by default, but also include parameters for creating other types of orders.

The example script below alternates between placing long and short market orders in cycles of a specified length. When the bar_index value is divisible by twice the input length, the script generates a long market order. If the bar_index value is divisible by the input length but not by twice the length, the script places a short market order instead. The script also draws labels to indicate bars on which it creates the orders:

//@version=6 strategy("Market order demo", overlay = true) //@variable The number of bars between long and short entries. int lengthInput = input.int(10, "Cycle length", 1) //@function Displays a specified string in a label at the current bar's high. debugLabel(string txt, color lblColor) => label.new(bar_index, high, text = txt, color = lblColor, textcolor = color.white, style = label.style_label_lower_right, size = size.large) //@variable Is `true` every `2 * lengthInput` bars, and `false` otherwise. bool longCondition = bar_index % (2 * lengthInput) == 0 //@variable Is `true` every `lengthInput` bars, and `false` otherwise. bool shortCondition = bar_index % lengthInput == 0 // Generate a long market order with a green label when the long condition occurs. if longCondition debugLabel("Long market order created", color.green) strategy.entry("My Long Entry Id", strategy.long) // Otherwise, generate a short market order with a red label when the short condition occurs. else if shortCondition debugLabel("Short market order created", color.red) strategy.entry("My Short Entry Id", strategy.short)

Note that:

- This script uses the default calculation behavior: it executes once on each bar's closing tick. Therefore, as indicated by the labels and trade markers, the broker emulator fills each new order at the open of the following bar.
- The strategy.entry() command can automatically *reverse* an open position in the opposite direction. See the Reversing positions section below for more information.

Limit orders

A *limit order* is an instruction to buy or sell an instrument at a specific price or better (lower than specified for long orders, and higher than specified for short orders), irrespective of the time. To simulate a limit order in a strategy script, pass a *price* value to the `limit` parameter of an applicable order placement command.

When the market price reaches a limit order's value, or crosses it in the favorable direction, the broker emulator fills the order at that value or a better price. When a strategy generates a limit order at a *worse* value than the current market price (higher for long orders and lower for short orders), the emulator fills the order on the next available tick rather than waiting for the market price to reach that value.

For example, when the following script executes on the bar that is 100 bars before the latest bar, it calls the strategy.entry() command with a `limit` argument to generate a long limit order 800 ticks above the bar's close value. The script also draws a label on the bar where it creates the order, and it draws a horizontal line to visualize the order's price:

//@version=6 strategy("Limit order demo", overlay = true) //@function Draws a line and a label with specified text at a given price level. debugLabel(float price, string txt) => label.new(bar_index, price, text = txt, color = color.teal, textcolor = color.white, style = label.style_label_lower_right, size = size.large) line.new(bar_index, price, bar_index + 1, price, color = color.teal, extend = extend.right, style = line.style_dashed) // Place a long limit order and draw a label and line 100 bars before the last bar. if last_bar_index - bar_index == 100 float limitPrice = close - syminfo.mintick * 800 debugLabel(limitPrice, "Long Limit order created") strategy.entry("Long", strategy.long, limit = limitPrice)

Notice that the label and the start of the line in the chart above occur several bars before the “Long” entry marker. The broker emulator cannot fill the limit order while the market price remains *above* the `limitPrice` value, because that value is a *worse* price for the long trade. After the price subsequently drops and reaches the order’s price, the emulator fills the order mid-bar at that price.

If we change the previous example to place a long limit order *above* the bar’s close value rather than *below*, the broker emulator fills the order on the next available tick – similar to a market order – because the closing price is already a more *favorable* value for the long trade. In the script version below, we set the limit order’s price to 800 ticks above the bar’s close to demonstrate this effect:

//@version=6 strategy("Limit order demo", overlay = true) //@function Draws a line and a label with specified text at a given price level. debugLabel(float price, string txt) => label.new(bar_index, price, text = txt, color = color.teal, textcolor = color.white, style = label.style_label_lower_right, size = size.large) line.new(bar_index, price, bar_index + 1, price, color = color.teal, extend = extend.right, style = line.style_dashed) // Place a long limit order and draw a label and line 100 bars before the last bar. if last_bar_index - bar_index == 100 float limitPrice = close + syminfo.mintick * 800 debugLabel(limitPrice, "Long Limit order created") strategy.entry("Long", strategy.long, limit = limitPrice)

Stop and stop-limit orders

A *stop order* is an instruction to activate a new market or limit order when the market price reaches a specific price or a *worse* value (higher than specified for long orders and lower than specified for short orders). To simulate a stop order, pass a price value to the `stop` parameter of an applicable order placement command.

If a strategy generates a stop order at a *better* value than the current market price, it activates the subsequent order without waiting for the market price to reach that value.

The following example script calls the strategy.entry() with a `stop` argument to place a stop order 800 ticks above the current close value while executing on the bar that is 100 bars before the last bar. It also draws a label to indicate the bar on which it created the order, and it draws a horizontal line at the stop price.

The following example calls strategy.entry() to place a stop order 800 ticks above the close 100 bars before the last historical chart bar. It also draws a label on the bar where it creates the order and a line to display the stop price. As we see in the chart below, the strategy enters a long position immediately after the price crosses the stop level:

//@version=6 strategy("Stop order demo", overlay = true) //@function Draws a line and a label with specified text at a given price level. debugLabel(float price, string txt) => label.new(bar_index, high, text = txt, color = color.teal, textcolor = color.white, style = label.style_label_lower_right, size = size.large) line.new(bar_index, high, bar_index, price, style = line.style_dotted, color = color.teal) line.new(bar_index, price, bar_index + 1, price, color = color.teal, extend = extend.right, style = line.style_dashed) // Place a long stop order and draw a label and line 100 bars before the last bar. if last_bar_index - bar_index == 100 float stopPrice = close + syminfo.mintick * 800 debugLabel(stopPrice, "Long Stop order created") strategy.entry("Long", strategy.long, stop = stopPrice)

Note that:

- A basic stop order is essentially the *opposite* of a limit order in terms of its execution based on the market price. If we use a limit order instead of a stop order in this scenario, the order executes immediately on the next bar. See the Limit orders section above for an example.

If a strategy.entry() or strategy.order() call includes a `stop` *and* `limit` argument, it creates a *stop-limit order*. Unlike a basic stop order, which triggers a market order when the current price is at the `stop` level or a worse value, a stop-limit order creates a subsequent *limit order* to fill at the specified `limit` price.

Below, we modified the previous script to simulate and visualize a stop-limit order. This script version includes the bar's low as the `limit` price in the strategy.entry() command. It also creates additional drawings to show where the strategy activates the subsequent limit order and to visualize the limit price.

In example chart below, notice how the market price reaches the limit level on the next bar after the script creates the stop-limit order, but the strategy does not enter a position because the limit order is not yet active. After the market price reaches the stop level, the strategy places the subsequent limit order, and then the broker emulator fills that order after the market price reverses to the limit level:

//@version=6 strategy("Stop-Limit order demo", overlay = true) //@function Draws a line and a label with specified text at a given price level. debugLabel(price, txt, lblColor, lineWidth = 1) => label.new(bar_index, high, text = txt, color = lblColor, textcolor = color.white, style = label.style_label_lower_right, size = size.large) line.new(bar_index, close, bar_index, price, style = line.style_dotted, color = lblColor, width = lineWidth) line.new(bar_index, price, bar_index + 1, price, color = lblColor, extend = extend.right, style = line.style_dashed, width = lineWidth) // Declare persistent variables to store the stop and limit prices. var float stopPrice = na var float limitPrice = na // Place a long stop-limit order and draw a label and lines 100 bars before the last bar. if last_bar_index - bar_index == 100 stopPrice := close + syminfo.mintick * 800 limitPrice := low debugLabel(limitPrice, "", color.gray) debugLabel(stopPrice, "Long Stop-Limit order created", color.teal) strategy.entry("Long", strategy.long, stop = stopPrice, limit = limitPrice) // Draw another line and label when the strategy activates the limit order. if high >= stopPrice debugLabel(limitPrice, "Limit order activated", color.green, 2) stopPrice := na

Order placement and cancellation

The `strategy.*` namespace features the following five functions that simulate the placement of orders, known as *order placement commands*: strategy.entry(), strategy.order(), strategy.exit(), strategy.close(), and strategy.close_all().

Additionally, the namespace includes the following two functions that cancel pending orders, known as *order cancellation commands*: strategy.cancel() and strategy.cancel_all().

The sections below explain these commands, their unique characteristics, and how to use them.

`strategy.entry()`

The strategy.entry() command generates *entry orders*. Its unique features help simplify opening and managing positions. This order placement command generates market orders by default. It can also create limit, stop, and stop-limit orders by using the `limit` and `stop` parameters, as explained in the Order types section above.

Reversing positions

One of the strategy.entry() command's unique features is its ability to *reverse* an open position automatically. By default, when an order from a strategy.entry() call executes while a position is open in the opposite direction, the command automatically *adds* the position's size to the new order's size. The added quantity

allows the order to close the current position and open a new position for the specified number of contracts/lots/shares/units in the new direction.

For instance, if a strategy has an open position of 15 shares in the strategy.long direction, then calls strategy.entry() to place a new market order in the strategy.short direction, the size of the resulting transaction is the specified entry size **plus** 15 shares.

The example below demonstrates this behavior in action. On each 100th bar, the script calls strategy.entry() with the argument `qty = 15` to enter a long position of 15 shares. Then, 50 bars later, it executes another call with a `qty` argument of 5, to enter a short position of five shares. The script also highlights the chart's background in blue when the long condition occurs, and in red when the short condition occurs, to indicate the bars where it places each order:

//@version=6 strategy("Reversing positions demo", overlay = true) //@variable Is `true` on every 100th bar, and `false` otherwise. bool buyCondition = bar_index % 100 == 0 //@variable Is `true` on every 50th bar, and `false` otherwise. bool sellCondition = bar_index % 50 == 0 if buyCondition // Place a "buy" market order to close any short position and enter a long position of 15 shares. strategy.entry("buy", strategy.long, qty = 15) else if sellCondition // Place a "sell" market order to close any long position and enter a short position of 5 shares. strategy.entry("sell", strategy.short, qty = 5) // Highlight the background when the `buyCondition` or `sellCondition` value is `true`. bgcolor(buyCondition ? color.new(color.blue, 90) : sellCondition ? color.new(color.red, 90) : na)

Although the long and short orders open trades with different sizes, each trade marker on the chart above shows *20 shares* as the traded quantity. These markers display the total size of each *transaction*, not the size of each resulting *position*. Each execution of one of the strategy.entry() calls automatically reverses the current position by adding the position's size (e.g., 15 for a long trade) to the new entry size (e.g., 5 for a short entry), resulting in a total transaction size of 20 shares. However, the resulting positions are 15 shares for long entries and 5 shares for short entries.

> NoteThe strategy.risk.allow_entry_in() function *overrides* the allowed direction for the strategy.entry() command. If a script specifies a trade direction with this risk management command, orders from strategy.entry() calls in the opposite direction *close* an open position *without* reversing it.

Pyramiding

Another unique characteristic of the strategy.entry() command is its connection to a strategy's *pyramiding* property. Pyramiding specifies the maximum number of open trades, from the orders created by strategy.entry() calls, that a strategy allows for a single position. After the number of open trades from strategy.entry() calls reaches the pyramiding limit, the strategy does not execute new orders from subsequent calls to the command until at least one of those trades closes.

Programmers can specify the default pyramiding limit for a strategy by including a `pyramiding` argument in the strategy() declaration statement. The default argument is 1, meaning the strategy can open new positions but cannot add to them using orders from strategy.entry() calls. Script users can adjust a strategy's pyramiding limit via the "Pyramiding" input in the "Settings/Properties" tab.

The following example calls the strategy.entry() function to place a market order once on every 25th bar. The order direction changes once every 100 bars. Therefore, each 100-bar cycle executes the command using

the same direction four times. The script highlights the chart's background for each bar on which the entry condition occurs.

As shown below, although the script calls the strategy.entry() command using the same `direction` argument four times per 100-bar cycle, only the *first* call in each cycle results in a new trade if we run the script with the default settings. The others do not contribute to the open position because the script's default `pyramiding` value is 1:

//@version=6 strategy("Pyramiding demo", overlay = true) //@variable Represents the direction of the entry orders. A value of 1 means long, and -1 means short. var int direction = 1 //@variable Is `true` once every 25 bars, and `false` otherwise. bool entryCondition = bar_index % 25 == 0 // Change the `direction` value on every 100th bar. if bar_index % 100 == 0 direction *= -1 // Place a market order based on the current `direction` value when the `entryCondition` value is `true`. if entryCondition strategy.entry("Entry", direction == 1 ? strategy.long : strategy.short) //@variable When the entry condition occurs, is a blue color if the `direction` is 1 and a red color otherwise. color bgColor = entryCondition ? (direction == 1 ? color.new(color.blue, 80) : color.new(color.red, 80)) : na // Highlight the chart's background using the `bgColor` value. bgcolor(bgColor, title = "Order highlight")

If we include `pyramiding = 4` in the script's strategy() declaration statement, the script can use strategy.entry() calls to open up to four trades in the same direction by default. After applying this change to our example script, all the script's entry orders now result in new trades:

//@version=6 strategy("Pyramiding demo", overlay = true, pyramiding = 4) //@variable Represents the direction of the entry orders. A value of 1 means long, and -1 means short. var int direction = 1 //@variable Is `true` once every 25 bars, and `false` otherwise. bool entryCondition = bar_index % 25 == 0 // Change the `direction` value on every 100th bar. if bar_index % 100 == 0 direction *= -1 // Place a market order based on the current `direction` value when the `entryCondition` value is `true`. if entryCondition strategy.entry("Entry", direction == 1 ? strategy.long : strategy.short) //@variable When the entry condition occurs, is a blue color if the `direction` is 1 and a red color otherwise. color bgColor = entryCondition ? (direction == 1 ? color.new(color.blue, 80) : color.new(color.red, 80)) : na // Highlight the chart's background using the `bgColor` value. bgcolor(bgColor, title = "Order highlight")

> NoticeIn some cases, *price-based* orders from the strategy.entry() command can cause a strategy's entry count for a position to exceed the specified pyramiding limit. If multiple calls to this command generate limit, stop, or stop-limit orders on the *same tick*, the broker emulator fills each one that the price action triggers, regardless of the specified limit.

`strategy.order()`

The strategy.order() command generates a *basic order*. Unlike other order placement commands, which can behave differently based on a strategy's properties and open trades, this command *ignores* most properties, such as pyramiding, and simply creates orders with the specified parameters. This command generates market orders by default. It can also create limit, stop, and stop-limit orders by using the `limit` and `stop` parameters. Orders from strategy.order() calls can open new positions and modify or close existing ones. When a strategy executes an order from this command, the resulting market position is the *net sum* of the open position size and the filled order quantity.

The following script uses strategy.order() calls to enter and exit positions. The strategy places a long market order for 15 units once every 100 bars. On every 25th bar that is not a multiple of 100, it places a short market order for five units. The script also highlights the background to signify where the strategy places each “buy” and “sell” order:

//@version=6 strategy("`strategy.order()` demo", overlay = true) //@variable Is `true` on every 100th bar, and `false` otherwise. bool buyCondition = bar_index % 100 == 0 //@variable Is `true` on every 25th bar, and `false` otherwise. bool sellCondition = bar_index % 25 == 0 if buyCondition // Place a "buy" market order to trade 15 units in the long direction. strategy.order("buy", strategy.long, qty = 15) else if sellCondition // Place a "sell" market order to trade 5 units in the short direction. strategy.order("sell", strategy.short, qty = 5) // Highlight the background when the `buyCondition` or `sellCondition` value is `true`. bgcolor(buyCondition ? color.new(color.blue, 90) : sellCondition ? color.new(color.red, 90) : na)

The strategy above never opens a *short position*. Unlike the strategy.entry() command, the strategy.order() command *does not* automatically reverse open positions. After filling a “buy” order, the strategy has an open long position of 15 units. The three subsequent “sell” orders *reduce* the position by *five* units each, and 15 - 5 * 3 = 0. In other words, the strategy opens a long position on every 100th bar and gradually reduces the size to 0 using three successive short orders. If we used the strategy.entry() command instead of strategy.order() in this example, the strategy would alternate between entering long and short trades of 15 and five units, respectively.

`strategy.exit()`

The strategy.exit() command generates *exit orders*. It features several unique behaviors that link to open trades, helping to simplify closing market positions and creating multi-level exits with *take-profit*, *stop-loss*, and *trailing stop* orders.

Unlike other order placement commands, which can generate a *single order* per call, each call to strategy.exit() can produce *more than one* type of exit order, depending on its arguments. Additionally, a single call to this command can generate exit orders for *multiple entries*, depending on the specified `from_entry` argument and the strategy’s open trades.

Take-profit and stop-loss

The most basic use of the strategy.exit() command is the placement of limit orders to trigger exits after earning enough money (take-profit), stop orders to trigger exits after losing too much money (stop-loss), or both (bracket).

Four parameters determine the prices of the command’s take-profit and stop-loss orders:

- The `profit` and `loss` parameters accept *relative* values representing the number of *ticks* by which the market price must move away from the entry price to trigger an exit.
- The `limit` and `stop` parameters accept *absolute* values representing the specific *prices* that trigger an exit when the market price reaches them.

When a strategy.exit() call includes arguments for the relative *and* absolute parameters defining take-profit or stop-loss levels (`profit` and `limit` or `loss` and `stop`), it creates orders only at the levels expected to trigger exits *first*.

For instance, if the `profit` distance is 19 ticks and the `limit` level is 20 ticks past the entry price in the favorable direction, the strategy.exit() command places a take-profit order based on the `profit` argument, because the market price will move 19 ticks past the entry price before it reaches the `limit` value. In contrast, if the `profit` distance is 20 ticks and the `limit` level is 19 ticks past the entry price in the favorable direction, the command places a take-profit order at the `limit` value because the price will reach that value first.

> NoticeThe strategy.exit() command's `limit` and `stop` parameters **do not** behave the same as the `limit` and `stop` parameters of the strategy.entry() and strategy.order() commands. Calling strategy.entry() or strategy.order() with `limit` and `stop` arguments creates a single stop-limit order. In contrast, calling strategy.exit() with both arguments creates **two exit orders**: a take-profit order at the `limit` price and a stop-loss order at the `stop` price.

The following example creates exit bracket (take-profit and stop-loss) orders with the strategy.exit() command. When the `buyCondition` value is `true`, the script calls the strategy.entry() command to place a "buy" market order. Then, it calls the strategy.exit() command with `limit` and `stop` arguments to create a take-profit order at the `limitPrice` value and a stop-loss order at the `stopPrice` value. The script plots the `limitPrice` and `stopPrice` values on the chart to visualize the exit order prices:

//@version=6 strategy("Take-profit and stop-loss demo", overlay = true) //@variable Is `true` on every 100th bar, and `false` otherwise. bool buyCondition = bar_index % 100 == 0 //@variable Stores the current take-profit order price. var float takeProfit = na //@variable Stores the current stop-loss order price. var float stopLoss = na if buyCondition // Update the `takeProfit` and `stopLoss` values. if strategy.opentrades == 0 takeProfit := close * 1.01 stopLoss := close * 0.99 // Place a long market order. strategy.entry("buy", strategy.long) // Place a take-profit order at the `takeProfit` value and a stop-loss order at the `stopLoss` value. strategy.exit("exit", "buy", limit = takeProfit, stop = stopLoss) // Set the `takeProfit` and `stopLoss` values to `na` when the position closes. if ta.change(strategy.closedtrades) > 0 takeProfit := na stopLoss := na // Plot the `takeProfit` and `stopLoss` series to visualize the exit levels. plot(takeProfit, "TP", color.green, style = plot.style_circles) plot(stopLoss, "SL", color.red, style = plot.style_circles)

Note that:

- We did not specify a `qty` or `qty_percent` argument in the strategy.exit() call, meaning it creates orders to exit 100% of the "buy" order's size.
- The strategy.exit() command's exit orders *do not* necessarily execute at the specified prices. Strategies can fill limit orders at *better* prices and stop orders at *worse* prices, depending on the range of values available to the broker emulator.

If a strategy.exit() call includes a `from_entry` argument, the resulting exit orders apply to only the existing entry orders that have a matching ID. If the specified `from_entry` value does not match the entry ID of any trade in the current position, the command *does not* create any exit orders.

Below, we changed the `from_entry` argument of the strategy.exit() call in our previous script to `"buy2"`, instructing the command to create exit orders only for open trades that have the "buy2" entry ID. This version does not place *any* exit orders, because the strategy does not create any entry order with the "buy2" ID:

//@version=6 strategy("Invalid `from_entry` ID demo", overlay = true) //@variable Is `true` on every 100th bar, and `false` otherwise. bool buyCondition = bar_index % 100 == 0 //@variable Stores the current take-profit order price. var float takeProfit = na //@variable Stores the current stop-loss order price. var float stopLoss = na if buyCondition // Update the `takeProfit` and `stopLoss` values. if strategy.opentrades == 0 takeProfit := close * 1.01 stopLoss := close * 0.99 // Place a long market order. strategy.entry("buy", strategy.long) // Attempt to place an exit bracket for "buy2" entries. // This call has no effect because the strategy does not create entry orders with the "buy2" ID. strategy.exit("exit", "buy2", limit = takeProfit, stop = stopLoss) // Set the `takeProfit` and `stopLoss` values to `na` when the position closes. if ta.change(strategy.closedtrades) > 0 takeProfit := na stopLoss := na // Plot the `takeProfit` and `stopLoss` series to visualize the exit levels. plot(takeProfit, "TP", color.green, style = plot.style_circles) plot(stopLoss, "SL", color.red, style = plot.style_circles)

Note that:

- If a strategy.exit() call *does not* include a `from_entry` argument, it creates exit orders for *all* open trades in the current position, regardless of their entry IDs. See the Exits for multiple entries section below to learn more.

Partial and multi-level exits

Strategies can use more than one call to the strategy.exit() command to create successive *partial* exit orders for the same entry ID. This behavior helps simplify the formation of multi-level exit strategies. To exit from a position using multiple strategy.exit() calls, include a `qty` or `qty_percent` argument in each call to specify the portion of the traded quantity to close. If the sum of the exit order sizes exceeds the open position, the strategy automatically *reduces* their sizes to match the total size of the position.

> NoteIf a strategy.exit() call includes *both* `qty` and `qty_percent` arguments, the command uses the `qty` value to size the order and ignores the `qty_percent` value.

The following example demonstrates a simple strategy that creates two partial exit order brackets for an entry ID. When the `buyCondition` value is `true`, the script places a “buy” market order for two shares with a strategy.entry() call, and it creates “exit1” and “exit2” bracket orders using two calls to strategy.exit(). The first call uses a `qty` value of 1, and the second uses a `qty` value of 3:

//@version=6 strategy("Multi-level exit demo", "test", overlay = true, behind_chart = false) //@variable Is `true` on every 100th bar, and `false` otherwise. bool buyCondition = bar_index % 100 == 0 //@variable The take-profit price for "exit1" orders. var float takeProfit1 = na //@variable The take-profit price for "exit2" orders. var float takeProfit2 = na //@variable The stop-loss price for "exit1" orders. var float stopLoss1 = na //@variable The stop-loss price for "exit2" orders. var float stopLoss2 = na if buyCondition // Update the `takeProfit*` and `stopLoss*` values. if strategy.opentrades == 0 takeProfit1 := close * 1.01 takeProfit2 := close * 1.02 stopLoss1 := close * 0.99 stopLoss2 := close * 0.98 // Place a long market order with a `qty` value of 2. strategy.entry("buy", strategy.long, qty = 2) // Place an "exit1" bracket with a `qty` value of 1 at the `takeProfit1` and `stopLoss1` levels. strategy.exit("exit1", "buy", limit = takeProfit1, stop = stopLoss1, qty = 1) // Place an "exit2" bracket with a `qty` value of 3 at the `takeProfit2` and `stopLoss2` levels. // The size of the resulting orders *decreases* to match the open position, // because the "exit1" bracket always fills first. strategy.exit("exit2", "buy", limit = takeProfit2, stop = stopLoss2, qty = 3) // Set the `takeProfit1` and `stopLoss1` values to `na` when the price reaches either value. if high >= takeProfit1 or low <= stopLoss1

takeProfit1 := na stopLoss1 := na // Set the `takeProfit2` and `stopLoss2` values to `na` when the price reaches either value. if high >= takeProfit2 or low <= stopLoss2 takeProfit2 := na stopLoss2 := na // Plot the `takeProfit*` and `stopLoss*` series to visualize the levels. plot(takeProfit1, "TP1", color.green, style = plot.style_circles) plot(takeProfit2, "TP2", color.green, style = plot.style_circles) plot(stopLoss1, "SL1", color.red, style = plot.style_circles) plot(stopLoss2, "SL2", color.red, style = plot.style_circles)

As we can see from the trade markers on the chart above, the strategy first executes the “exit1” take-profit or stop-loss order to reduce the open position by one share, leaving only one remaining share in the position. However, the “exit2” order bracket has a specified size of *three shares*, which exceeds the remaining position. Rather than using this specified quantity, the strategy automatically *reduces* the “exit2” orders to *one* share, allowing it to close the position successfully.

Note that:

- The broker emulator fills only **one** exit order from the “exit1” bracket, **not both**. If a strategy.exit() call generates more than one exit order type for an entry ID, the strategy fills the only the *first* triggered one and automatically *cancels* the others.
- The strategy reduces the “exit2” orders because all orders from the strategy.exit() calls automatically belong to the same strategy.oca.reduce *OCA group* by default. See the OCA groups section below to learn more.

When creating multiple exit orders with *different* strategy.exit() calls, it’s crucial to note that the orders from each call automatically *reserve* a portion of the open position. The orders from one strategy.exit() call *cannot* exit the portion of a position that a previous call already reserved.

For example, the script below generates a “buy” entry order for 20 shares with a strategy.entry() call, then creates “limit” and “stop” exit orders with two separate calls to strategy.exit(), while executing on the bar that is 100 bars before the last bar. The exit commands specify a quantity of 19 shares for the “limit” order and 20 for the “stop” order:

//@version=6 strategy("Reserved exit demo", "test", overlay = true, behind_chart = false) //@variable The price of the "limit" exit order. var float limitPrice = na //@variable The price of the "stop" exit order. var float stopPrice = na //@variable Is `true` 100 bars before the last chart bar, and `false` otherwise. bool longCondition = last_bar_index - bar_index == 100 if longCondition // Update the `limitPrice` and `stopPrice` values. limitPrice := close * 1.01 stopPrice := close * 0.99 // Place a long market order for 20 shares. strategy.entry("buy", strategy.long, 20) // Create a take-profit order for 19 shares at the `limitPrice` value. strategy.exit("limit", limit = limitPrice, qty = 19) // Create a stop-loss order at the `stopPrice` value. Although this call specifies a `qty` value of 20, the // previous `strategy.exit()` call reserved 19 shares. // Consequently, this call creates an exit order for only **one share**. strategy.exit("stop", stop = stopPrice, qty = 20) //@variable Is `true` if the strategy has an open position, and `false` otherwise. bool showPlot = strategy.opentrades > 0 // Plot the `limitPrice` and `stopPrice` values when the `showPlot` value is `true`. plot(showPlot ? limitPrice : na, "Limit (take-profit) price", color.green, 2, plot.style_linebr) plot(showPlot ? stopPrice : na, "Stop (stop-loss) price", color.red, 2, plot.style_linebr)

Users who are unfamiliar with the strategy.exit() command’s unique behaviors might expect the above strategy to close the entire market position if it fills the “stop” order before the “limit” order. However, the trade markers on the chart below show that the “stop” order *reduces* the position by only **one share**. The strategy.exit() call for the “limit” order executes *first* in the code. Consequently, the “limit” exit order reserves *19 shares* of the open position. This reservation leaves only one share available for the “stop” order to close, regardless of when the strategy fills it:

Trailing stops

One of the strategy.exit() command's key features is its ability to create *trailing stops*, i.e., stop-loss orders that trail behind the market price by a specified amount whenever it moves to a more favorable value (higher for long positions and lower for short positions).

This type of exit order has two components: an *activation level* and a *trail offset*. The activation level is the value that the market price must cross to activate the trailing stop calculation. The trail offset is the distance by which the activated stop follows behind the market price as it reaches successively better values.

Three strategy.exit() parameters determine the activation level and the trail offset of a trailing stop order:

- The `trail_price` parameter accepts an *absolute price value* for the trailing stop's activation level.
- The `trail_points` parameter is an alternative way to specify the activation level. Its value represents the *tick distance* from the entry price required to activate the trailing stop.
- The `trail_offset` parameter accepts a value representing the order's trail offset as a specified number of ticks.

To create and activate a trailing stop order, a strategy.exit() call must specify a `trail_offset` argument and either a `trail_price` or `trail_points` argument. If the call contains both `trail_price` and `trail_points` arguments, the command uses the level expected to activate the stop *first*. For instance, if the `trail_points` distance is 50 ticks and the `trail_price` value is 51 ticks past the entry price in the favorable direction, the strategy.exit() command uses the `trail_points` value to set the activation level because the market price will move that distance *before* reaching the `trail_price` level.

The example below demonstrates how a trailing stop order works in detail. The strategy places a "Long" market order with the strategy.entry() command 100 bars before the last chart bar. Then, it calls the strategy.exit() command with `trail_price` and `trail_offset` arguments on the following bar to create a trailing stop. The script uses lines, labels, and a plot to visualize the trailing stop's behavior.

The green line on the chart shows the level that the market price must reach to activate the trailing stop order. After the price reaches that level from below, the script uses a blue plot to display the trailing stop's price. Each time the market price reaches a new high after activating the trailing stop, the stop's price *increases* to maintain a maximum distance of `trailOffsetInput` ticks from the best value. The exit order *does not* change its price level when the price decreases or does not reach a new high. Eventually, the market price crosses below the trailing stop, triggering an exit:

//@version=6 strategy("Trailing stop order demo", overlay = true, behind_chart = false) //@variable The distance from the entry price required to activate the trailing stop. int activationOffsetInput = input.int(1000, "Activation level offset (in ticks)", 0) //@variable The distance the stop follows behind the highest `high` value after activation. int trailOffsetInput = input.int(2000, "Trailing stop offset (in ticks)", 0) //@variable Draws a label and an optional line at the specified price level. debugDrawings(float price, string txt, color drawingColor, bool drawLine = false) => label.new(bar_index, price, text = txt, color = drawingColor, textcolor = color.white, style = label.style_label_lower_right, size = size.large) if drawLine line.new(bar_index, price, bar_index + 1, price, color = drawingColor, extend = extend.right, style = line.style_dashed) //@variable The level required to activate the trailing stop. var float activationLevel = na //@variable The price of the trailing stop. var float trailingStop = na //@variable The value that the trailing stop would have if it was currently active. float theoreticalStopPrice = high - trailOffsetInput * syminfo.mintick // Place a long market order 100 bars before the last historical bar. if last_bar_index - bar_index == 100 strategy.entry("Long", strategy.long) // Create and visualize the exit order on the next bar. if last_bar_index -

bar_index == 99 // Update the `activationLevel` value. activationLevel := open + syminfo.mintick * activationOffsetInput // Create the trailing stop order that activates at the `activationLevel` value and trails behind the high by // `trailOffsetInput` ticks. strategy.exit("Trailing Stop", from_entry = "Long", trail_price = activationLevel, trail_offset = trailOffsetInput) // Create drawings to indicate the activation level. debugDrawings(activationLevel, "Trailing Stop Activation Level", color.green, true) // Visualize the trailing stop's levels while the position is open. if strategy.opentrades == 1 // Create drawings when the high is above the `activationLevel` value for the first time to indicate when the // stop activates. if na(trailingStop) and high >= activationLevel debugDrawings(activationLevel, "Activation level crossed", color.green) trailingStop := theoreticalStopPrice debugDrawings(trailingStop, "Trailing Stop Activated", color.blue) // Otherwise, update the `trailingStop` value when the `theoreticalStopPrice` value reaches a new high. else if theoreticalStopPrice > trailingStop trailingStop := theoreticalStopPrice // Plot the `trailingStop` series to visualize the trailing stop's price movement. plot(trailingStop, "Trailing Stop")

Exits for multiple entries

A single call to the strategy.exit() command can generate exit orders for *more than one* entry in an open position, depending on the call's `from_entry` value.

If an open position consists of two or more entries with the same ID, a single strategy.exit() call that uses the ID as the `from_entry` argument places exit orders for *every* corresponding entry created *before* or *on* the bar where the call occurs.

For example, the following script periodically calls the strategy.entry() command on two consecutive bars to enter and add to a long position. Both calls use `"buy"` as the `id` argument. After creating the second entry order, the script calls the strategy.exit() command once with `"buy"` as its `from_entry` argument to generate *separate* exit orders for each trade with that entry ID. When the market price reaches the `takeProfit` or `stopLoss` value, the broker emulator fills *two* exit orders and closes the position:

//@version=6 strategy("Exits for entries with the same ID demo", overlay = true, behind_chart = false, pyramiding = 2) //@variable The take-profit price for exit commands. var float takeProfit = na //@variable The stop-loss price for exit commands. var float stopLoss = na //@variable Is `true` on two consecutive bars in 100-bar cycles, and `false` otherwise. bool buyCondition = math.min(bar_index % 100, math.max(bar_index - 1, 0) % 100) == 0 if buyCondition // Place a "buy" market order to enter a trade. strategy.entry("buy", strategy.long) // Calculate exits after a trade is open. if strategy.opentrades == 1 // Update the `takeProfit` and `stopLoss` values. takeProfit := close * 1.01 stopLoss := close * 0.99 // Place exit orders to close both "buy" trades. strategy.exit("exit", "buy", limit = takeProfit, stop = stopLoss) // Set the `takeProfit` and `stopLoss` values to `na` after both trades close. if ta.change(strategy.closedtrades) == 2 takeProfit := na stopLoss := na // Plot the `takeProfit` and `stopLoss` values. plot(takeProfit, "TP", color.green, style = plot.style_circles) plot(stopLoss, "SL", color.red, style = plot.style_circles)

A single strategy.exit() call can also generate exit orders for *all* entries in an open position, irrespective of entry ID, if it does *not* include a `from_entry` argument.

Below, we changed the strategy.entry() instance in the above script to create an entry order with a distinct ID on each call, and we removed the `from_entry` argument from the strategy.exit() call. Because this script version does not specify the entries to which the exit orders apply, the strategy.exit() call creates orders for *every* trade in the position:

//@version=6 strategy("Exits for entries with different IDs demo", overlay = true, behind_chart = false, pyramiding = 2) //@variable The take-profit price for exit commands. var float takeProfit = na //@variable The stop-loss price for exit commands. var float stopLoss = na //@variable Is `true` on two consecutive bars in 100-bar cycles, and `false` otherwise. bool buyCondition = math.min(bar_index % 100, math.max(bar_index - 1, 0) % 100) == 0 if buyCondition // Place a long market order with a *unique ID*. strategy.entry("buy" + str.tostring(strategy.opentrades + strategy.closedtrades), strategy.long) // Calculate exits after a trade is open. if strategy.opentrades == 1 // Update the `takeProfit` and `stopLoss` values. takeProfit := close * 1.01 stopLoss := close * 0.99 // Place exit orders for *all* entries in the position, irrespective of ID. strategy.exit("exit", limit = takeProfit, stop = stopLoss) // Set the `takeProfit` and `stopLoss` values to `na` after both trades close. if ta.change(strategy.closedtrades) == 2 takeProfit := na stopLoss := na // Plot the `takeProfit` and `stopLoss` values. plot(takeProfit, "TP", color.green, style = plot.style_circles) plot(stopLoss, "SL", color.red, style = plot.style_circles)

It's crucial to note that a call to strategy.exit() without a `from_entry` argument *persists* and creates exit orders for all open trades in a position, regardless of *when* the entries occur. This behavior can affect strategies that manage positions with multiple entries or exits. If a strategy has an open position and calls strategy.exit() on any bar without specifying a `from_entry` ID, it generates exit orders for each entry created *before* or on that bar, and it *continues* to generate exit orders for subsequent entries *after* that bar until the position closes.

Let's explore this behavior and how it works. The script below calls the strategy.entry() command to create a long entry order on each bar within a user-specified time range. It also calls the strategy.exit() command without a `from_entry` argument on only *one bar* within that range to generate exit orders for *every* entry in the open position. The exit command uses a `loss` value of 0, which means that an exit order fills each time the market price is not above one of the entry prices.

The script prompts the user to select three points before it starts its calculations. The first point specifies when order creation begins, the second determines when the single strategy.exit() call occurs, and the third specifies when order creation stops:

//@version=6 strategy("Exit persist demo", overlay = true, behind_chart = false, pyramiding = 100) //@variable The time when order creation starts. int entryStartTime = input.time(0, "Start time for entries", confirm = true) //@variable The time when the `strategy.exit()` call occurs. int exitCallTime = input.time(0, "Exit call time", confirm = true) //@variable The time when order creation stops. int entryEndTime = input.time(0, "End time for entries", confirm = true) // Raise a runtime error if incorrect timestamps are chosen. if exitCallTime <= entryStartTime or entryEndTime <= exitCallTime or entryEndTime <= entryStartTime runtime.error("The input timestamps must follow this condition: entryStartTime < exitCallTime < entryEndTime.") // Create variables to track entry and exit conditions. bool entriesStart = time == entryStartTime bool callExit = time == exitCallTime bool entriesEnd = time == entryEndTime bool callEntry = time >= entryStartTime and time < entryEndTime // Place a long entry order when `callEntry` is `true`. if callEntry strategy.entry("Entry", strategy.long) // Call `strategy.exit()` when `callExit` is `true`, which occurs only once. // This single call persists and creates exit orders for EVERY entry in the position because it does not // specify a `from_entry` ID. if callExit strategy.exit("Exit", loss = 0) // Draw labels to signify when entries start, when the `strategy.exit()` call occurs, and when order placement stops. switch entriesStart => label.new(bar_index, high, "Start placing entry orders.", color = color.green, textcolor = color.white, style = label.style_label_lower_right, size = size.large) callExit => label.new(bar_index, high, "Call `strategy.exit()` once.", color = color.blue, textcolor = color.white, style = label.style_label_lower_right, size = size.large) entriesEnd => label.new(bar_index, high, "Stop placing orders.", color = color.red, textcolor = color.white, style = label.style_label_lower_left, size = size.large) // Create a line and label to visualize the lowest entry price, i.e., the price required to close the position. var line lowestLine = line.new(entryStartTime + 1000, na,

entryEndTime, na, xloc.bar_time, extend.right, color.orange, width = 2) var lowestLabel = label.new(entryStartTime + 1000, na, "Lowest entry price", color = color.orange, style = label.style_label_upper_right, xloc = xloc.bar_time) // Update the price values of the `lowestLine` and `lowestLabel` after each new entry. if callEntry[1] var float lowestPrice = strategy.opentrades.entry_price(0) float entryPrice = strategy.opentrades.entry_price(strategy.opentrades - 1) if not na(entryPrice) lowestPrice := math.min(lowestPrice, entryPrice) lowestLine.set_y1(lowestPrice) lowestLine.set_y2(lowestPrice) lowestLabel.set_y(lowestPrice) // Highlight the background when `entriesStart`, `callExit`, and `entriesEnd` occurs. bgcolor(entriesStart ? color.new(color.green, 80) : na, title = "Entries start highlight") bgcolor(callExit ? color.new(color.blue, 80) : na, title = "Exit call highlight") bgcolor(entriesEnd ? color.new(color.red, 80) : na, title = "Entries end highlight")

Note that:

- We included `pyramiding = 100` in the strategy() declaration statement, allowing the position to include up to 100 open trades from strategy.entry() orders.
- The script uses labels and a bgcolor() call to signify when order placement starts and stops and when the strategy.exit() call occurs.
- The script draws a line and a label at the lowest entry price to show the value the market price must reach to close the entire position.

We can observe the unique exit behavior of the strategy.exit() command in this example by comparing the code itself with the script's visuals. The script calls the strategy.exit() command *one time*, only on the bar with the blue label. However, that single call places exit orders for *every* entry that occurs **before** or on that bar, then continues placing exit orders for each new entry **after** that bar. This behavior occurs because the strategy.exit() function cannot determine when to stop placing orders if it does not link to entries with a specific ID. In this case, the command only ceases to create new exit orders only after the position fully *closes*.

The above script exhibits different behaviors if we include a `from_entry` argument in the strategy.exit() call. If a call to this command specifies a `from_entry` ID, the exit orders apply to entries that the strategy created *before* or *on* the bar of the call using that ID. The command does not place exit orders for subsequent entries created *after* the bar in that case, even if those entries have the same ID.

In the script version below, we added `from_entry = "Entry"` to our script's strategy.exit() call to specify that it produces exit orders only for entries that have the "Entry" ID. Only 17 exits occur across the same range this time, each corresponding to an entry order created *before* or *on* the bar with the blue label. The call does not generate new orders for any trade that the strategy opens *after* that bar, regardless of their entry ID:

//@version=6 strategy("Exit persist demo", overlay = true, behind_chart = false, pyramiding = 100) //@variable The time when order creation starts. int entryStartTime = input.time(0, "Start time for entries", confirm = true) //@variable The time when the `strategy.exit()` call occurs. int exitCallTime = input.time(0, "Exit call time", confirm = true) //@variable The time when order creation stops. int entryEndTime = input.time(0, "End time for entries", confirm = true) // Raise a runtime error if incorrect timestamps are chosen. if exitCallTime <= entryStartTime or entryEndTime <= exitCallTime or entryEndTime <= entryStartTime runtime.error("The input timestamps must follow this condition: entryStartTime < exitCallTime < entryEndTime.") // Create variables to track entry and exit conditions. bool entriesStart = time == entryStartTime bool callExit = time == exitCallTime bool entriesEnd = time == entryEndTime bool callEntry = time >= entryStartTime and time < entryEndTime // Place a long entry order when `callEntry` is `true`. if callEntry strategy.entry("Entry", strategy.long) // Call `strategy.exit()` when `callExit` is `true`, which occurs

only once. // This single call only places exit orders for all entries with the "Entry" ID created before or on the bar where // `callExit` occurs. It DOES NOT affect any subsequent entries created after that bar. if callExit strategy.exit("Exit", from_entry = "Entry", loss = 0) // Draw labels to signify when entries start, when the `strategy.exit()` call occurs, and when order placement stops. switch entriesStart => label.new(bar_index, high, "Start placing entry orders.", color = color.green, textcolor = color.white, style = label.style_label_lower_right, size = size.large) callExit => label.new(bar_index, high, "Call `strategy.exit()` once.", color = color.blue, textcolor = color.white, style = label.style_label_lower_right, size = size.large) entriesEnd => label.new(bar_index, high, "Stop placing orders.", color = color.red, textcolor = color.white, style = label.style_label_lower_left, size = size.large) // Create a line and label to visualize the lowest entry price, i.e., the price required to close the position. var line lowestLine = line.new(entryStartTime + 1000, na, entryEndTime, na, xloc.bar_time, extend.right, color.orange, width = 2) var lowestLabel = label.new(entryStartTime + 1000, na, "Lowest entry price", color = color.orange, style = label.style_label_upper_right, xloc = xloc.bar_time) // Update the price values of the `lowestLine` and `lowestLabel` after each new entry. if callEntry[1] var float lowestPrice = strategy.opentrades.entry_price(0) float entryPrice = strategy.opentrades.entry_price(strategy.opentrades - 1) if not na(entryPrice) lowestPrice := math.min(lowestPrice, entryPrice) lowestLine.set_y1(lowestPrice) lowestLine.set_y2(lowestPrice) lowestLabel.set_y(lowestPrice) // Highlight the background when `entriesStart`, `callExit`, and `entriesEnd` occurs. bgcolor(entriesStart ? color.new(color.green, 80) : na, title = "Entries start highlight") bgcolor(callExit ? color.new(color.blue, 80) : na, title = "Exit call highlight") bgcolor(entriesEnd ? color.new(color.red, 80) : na, title = "Entries end highlight")

`strategy.close()` and `strategy.close_all()`

The strategy.close() and strategy.close_all() commands generate orders to exit from an open position. Unlike strategy.exit(), which creates *price-based* exit orders (e.g., stop-loss), these commands generate market orders that the broker emulator fills on the next available tick, irrespective of the price.

The example script below demonstrates a simple strategy that enters and exits a position using only market orders. The script executes a strategy.entry() call to place a “buy” market order once every 50 bars, then calls the strategy.close() command to close any open “buy” trade 25 bars later:

//@version=6 strategy("Close demo", "test", overlay = true) //@variable Is `true` on every 50th bar. buyCond = bar_index % 50 == 0 //@variable Is `true` on every 25th bar except for those that are divisible by 50. sellCond = bar_index % 25 == 0 and not buyCond if buyCond strategy.entry("buy", strategy.long) if sellCond strategy.close("buy") bgcolor(buyCond ? color.new(color.blue, 90) : na) bgcolor(sellCond ? color.new(color.red, 90) : na)

Notice that the strategy.close() call in this script uses “buy” as its required `id` argument. Unlike strategy.exit(), this command’s `id` parameter specifies the *entry ID* of an open trade. It **does not** represent the ID of the resulting exit order. If a market position consists of multiple open trades with the same entry ID, a single strategy.close() call with that ID as its `id` argument generates a single market order to close all of those trades in one transaction.

The following script creates a “buy” order using a strategy.entry() call once every 25 bars, and it calls the strategy.close() command with `"buy"` as its `id` argument to close all open trades with that entry ID once every 100 bars. The market order from the strategy.close() call closes the entire position in this case because every open trade in the position has the same “buy” entry ID:

//@version=6 strategy("Multiple close demo", "test", overlay = true, pyramiding = 3) //@variable Is `true` on every 100th bar. sellCond = bar_index % 100 == 0 //@variable Is `true` on every 25th bar except for those that are divisible by 100. buyCond = bar_index % 25 == 0 and not sellCond if buyCond strategy.entry("buy", strategy.long) if sellCond strategy.close("buy") bgcolor(buyCond ? color.new(color.blue, 90) : na) bgcolor(sellCond ? color.new(color.red, 90) : na)

Note that:

- We included `pyramiding = 3` in the strategy() declaration statement, allowing the script to generate up to three entries per position using strategy.entry() calls.

The strategy.close_all() command generates a market order to close any open position. Unlike strategy.close(), this command *does not* link to any specific entry ID. Using the strategy.close_all() command is helpful when a strategy must exit as soon as possible from a position consisting of multiple open trades with different entry IDs.

The script below places “A”, “B”, and “C” entry orders sequentially based on the number of open trades as tracked by the strategy.opentrades variable. Then, it calls the strategy.close_all() command to create a single order that closes the entire position on the following bar:

//@version=6 strategy("Close multiple ID demo", "test", overlay = true, pyramiding = 3) switch strategy.opentrades 0 => strategy.entry("A", strategy.long) 1 => strategy.entry("B", strategy.long) 2 => strategy.entry("C", strategy.long) 3 => strategy.close_all()

`strategy.cancel()` and `strategy.cancel_all()`

The strategy.cancel() and strategy.cancel_all() commands allow strategies to cancel *unfilled* orders before the broker emulator processes them. These order cancellation commands are most helpful when working with *price-based orders*, including all orders from strategy.exit() calls and the orders from strategy.entry() or strategy.order() calls that include `limit` or `stop` arguments.

The strategy.cancel() command has a required `id` parameter, which specifies the ID of the entry or exit orders to cancel. The strategy.cancel_all() command does not have such a parameter because it cancels *all* unfilled orders, regardless of ID.

The following strategy calls the strategy.entry() command to place a “buy” limit order 500 ticks below the closing price while executing on the bar that is 100 bars before the latest bar. Then, on the next bar, it calls the strategy.cancel() command to cancel the order before the broker emulator executes it.

The following strategy places a “buy” limit order 500 ticks below the closing price 100 bars before the last chart bar with strategy.entry(), and it cancels the order on the next bar with strategy.cancel(). The script highlights the chart’s background to signify when it places and cancels the “buy” order, and it draws a horizontal line at the order’s price. As shown below, our example chart shows no entry marker when the market price crosses the horizontal line, because the strategy already cancels the order (when the chart’s background is orange) before the price reaches that level:

//@version=6 strategy("Cancel demo", "test", overlay = true) //@variable Draws a horizontal line at the `limit` price of the "buy" order. var line limitLine = na //@variable Is `color.green` when the strategy places the "buy" order, `color.orange` when it cancels the order. color bgColor = na if last_bar_index - bar_index == 100 float limitPrice = close - syminfo.mintick * 500 strategy.entry("buy", strategy.long, limit = limitPrice) limitLine := line.new(bar_index, limitPrice, bar_index + 1, limitPrice, extend = extend.right) bgColor :=

color.new(color.green, 50) if last_bar_index - bar_index == 99 strategy.cancel("buy") bgColor := color.new(color.orange, 50) bgcolor(bgColor)

The strategy.cancel() command cancels *all* unfilled orders that have the specified entry ID. It does nothing if the specified `id` represents the ID of an order that does not exist. If the strategy has more than one unfilled order with the same entry ID, the command cancels *all* of them at once.

Below, we modified the previous script to place a “buy” limit order on three consecutive bars, starting 100 bars before the last chart bar. After placing all three orders, the strategy cancels them using a single strategy.cancel() call with “buy” as the `id` argument, resulting in no new trades when the market price reaches any of the order prices indicated by the horizontal lines:

//@version=6 strategy("Multiple cancel demo", "test", overlay = true, pyramiding = 3) //@variable Draws a horizontal line at the `limit` price of the "buy" order. var line limitLine = na //@variable Is `color.green` when the strategy places the "buy" order, `color.orange` when it cancels the order. color bgColor = na if last_bar_index - bar_index <= 100 and last_bar_index - bar_index >= 98 float limitPrice = close - syminfo.mintick * 500 strategy.entry("buy", strategy.long, limit = limitPrice) limitLine := line.new(bar_index, limitPrice, bar_index + 1, limitPrice, extend = extend.right) bgColor := color.new(color.green, 50) if last_bar_index - bar_index == 97 strategy.cancel("buy") bgColor := color.new(color.orange, 50) bgcolor(bgColor)

Note that:

- We included `pyramiding = 3` in the strategy() declaration statement, allowing three successive entries from strategy.entry() calls per position. The script would also achieve the same result without this setting if it called the strategy.order() command instead, because a strategy’s pyramiding setting *does not* affect orders from that command.

The strategy.cancel() and strategy.cancel_all() commands can cancel orders of any type, including market orders. However, it is important to note that either command can cancel a market order only if its call occurs on the *same* script execution as the order placement command. The cancellation command has *no effect* if the call happens after that point, because the broker emulator always fills market orders by the *next available tick*.

The example below places a “buy” market order with a strategy.entry() call 100 bars before the latest bar. Then, it attempts to cancel that order with a strategy.cancel_all() call on the next bar. The cancellation command *does not* affect the “buy” order in this case. The broker emulator fills the market order on the next bar’s opening tick, *before* the script executes the strategy.cancel_all() call:

//@version=6 strategy("Cancel market demo", overlay = true) //@variable Is `color.green` when the strategy places the "buy" order, `color.orange` when it tries to cancel the order. color bgColor = na if last_bar_index - bar_index == 100 strategy.entry("buy", strategy.long) bgColor := color.new(color.green, 50) if last_bar_index - bar_index == 99 strategy.cancel_all() bgColor := color.new(color.orange, 50) bgcolor(bgColor)

Position sizing

Pine Script strategies feature two ways to control the sizes of the orders that open and modify positions:

- Set a default *fixed* quantity type and value for the orders. Programmers can specify defaults for these properties by including `default_qty_type` and `default_qty_value` arguments in the strategy()

declaration statement. Script users can adjust these defaults via the “Default order size” inputs in the “Settings/Properties” tab.
- Include a *non-na* `qty` argument in the strategy.entry() or strategy.order() call. If a call to either of these commands specifies a non-na `qty` value, that call *ignores* the strategy’s default quantity type and value and places an order for `qty` contracts/shares/lots/units instead.

The following example uses strategy.entry() calls with different `qty` arguments for long and short trades. When the current bar’s low equals the `lowest` value, the script places a “Buy” order to enter a long position of `longAmount` units. Otherwise, when the high equals the `highest` value, it places a “Sell” order to enter a short position of `shortAmount` units:

//@version=6 strategy("Orders with specified quantities demo", overlay = true, default_qty_type = strategy.cash, default_qty_value = 5000) int length = input.int(20, "Length", 1) float longAmount = input.float(4.0, "Long Amount", 0.0) float shortAmount = input.float(2.0, "Short Amount", 0.0) float highest = ta.highest(length) float lowest = ta.lowest(length) switch low == lowest => strategy.entry("Buy", strategy.long, longAmount) high == highest => strategy.entry("Sell", strategy.short, shortAmount)

Notice that although we’ve included `default_qty_type` and `default_qty_value` arguments in the strategy() declaration statement, the strategy *does not* use this default setting to size its orders. Instead, the specified `qty` value in the entry commands takes precedence. To use the strategy’s default order size, we must *remove* the `qty` arguments from the strategy.entry() calls or set their values to na.

Below, we edited the previous script by including ternary expressions for the `qty` arguments in both strategy.entry() calls. These expressions replace input values of 0 with na. Now, if the specified `longAmount` or `shortAmount` value is 0, the corresponding entry orders use the strategy’s default order size instead, as we see below:

//@version=6 strategy("Orders using default quantities demo", overlay = true, default_qty_type = strategy.cash, default_qty_value = 5000) int length = input.int(20, "Length", 1) float longAmount = input.float(0.0, "Long Amount", 0.0) float shortAmount = input.float(0.0, "Short Amount", 0.0) float highest = ta.highest(length) float lowest = ta.lowest(length) switch low == lowest => strategy.entry("Buy", strategy.long, longAmount == 0.0 ? na : longAmount) high == highest => strategy.entry("Sell", strategy.short, shortAmount == 0.0 ? na : shortAmount)

Closing a market position

By default, strategies close a market position using the *First In, First Out (FIFO)* method. If a strategy uses this behavior, any exit order closes or reduces the position starting with the *first* open trade, even if the exit command specifies the entry ID of a *different* open trade. To override this default behavior, include `close_entries_rule = "ANY"` in the strategy() declaration statement.

The following example script places “Buy1” and “Buy2” entry orders sequentially, starting 100 bars before the latest chart bar. When the position size is 0, it calls the strategy.entry() command to place the “Buy1” order for five units. After the strategy’s position size matches the size of that order, the second strategy.entry() call places the “Buy2” order for ten units. The strategy then creates “bracket” exit orders for both entries using a single strategy.exit() call without a `from_entry` argument. For visual reference, the script also plots the strategy.position_size value in a separate pane:

//@version=6 strategy("Exit demo", pyramiding = 2) float positionSize = strategy.position_size if positionSize == 0 and last_bar_index - bar_index <= 100 strategy.entry("Buy1", strategy.long, 5) else if positionSize == 5

strategy.entry("Buy2", strategy.long, 10) else if positionSize == 15 strategy.exit("bracket", loss = 10, profit = 10) plot(positionSize, "Position size", color.lime, 4, plot.style_histogram)

Note that:

- We included `pyramiding = 2` in the strategy() declaration statement, allowing two successive entries from strategy.entry() calls per position.

Each time the market price triggers an exit order, the above script exits from the open position, starting with the *oldest* open trade. This FIFO behavior applies even if we explicitly specify an exit from "Buy2" before "Buy1" in the code.

The script version below calls the strategy.close() command with "Buy2" as the `id` argument, and it includes "Buy1" as the `from_entry` argument in the strategy.exit() call. The market order from the strategy.close() call executes on the next available tick. Therefore, the broker emulator fills it *before* the take-profit and stop-loss orders from the strategy.exit() call:

//@version=6 strategy("Exit demo", pyramiding = 2) float positionSize = strategy.position_size if positionSize == 0 and last_bar_index - bar_index <= 100 strategy.entry("Buy1", strategy.long, 5) else if positionSize == 5 strategy.entry("Buy2", strategy.long, 10) else if positionSize == 15 strategy.close("Buy2") strategy.exit("bracket", "Buy1", loss = 10, profit = 10) plot(positionSize, "Position size", color.lime, 4, plot.style_histogram)

The market order from the script's strategy.close() call is for 10 units because it links to the open trade with the "Buy2" entry ID. A user might expect this strategy to close that trade completely when the order executes. However, the "Trades" tab in the strategy report shows that *five* units of the order close the "Buy1" trade *first* because it is the oldest, and the remaining five units close *half* of the "Buy2" trade. Then, the "bracket" orders from the strategy.exit() call close the rest of the position later:

Note that:

- If we included `close_entries_rule = "ANY"` in the strategy() declaration statement, the market order from the strategy.close() call would close the open trade with the "Buy2" entry ID *first*, and then the "bracket" orders from the strategy.exit() call would close the trade with the "Buy1" entry ID.

OCA groups

One-Cancels-All (OCA) groups allow a strategy to fully or partially *cancel* specific orders when the broker emulator executes another order from the same group. To assign an order to an OCA group, include an `oca_name` argument in the call to the order placement command. The strategy.entry() and strategy.order() commands also allow programmers to specify an *OCA type*, which defines whether a strategy cancels, reduces, or does not modify the order after executing other orders.

> NoteAll order placement commands that issue orders for the same OCA group must specify the same group name **and** OCA type. If two commands have the same `oca_name` value but *different* `oca_type` values, the strategy considers them to be from **two distinct groups**. In other words, an OCA group **cannot** mix the strategy.oca.cancel, strategy.oca.reduce, and strategy.oca.none OCA types.

`strategy.oca.cancel`

If an order placement command uses strategy.oca.cancel as its `oca_type` argument, the strategy completely *cancels* the resulting order if another order from the same OCA group executes first.

To understand how this OCA type impacts a strategy's orders, consider the following script, which places orders when the `ma1` value crosses the `ma2` value. If the strategy.position_size value is 0 when the cross occurs, the strategy places two stop orders using strategy.order() calls. The first is a long order at the bar's high, and the second is a short order at the bar's low. If the strategy already has an open position during the cross, it calls the strategy.close_all() command to close the position with a market order:

//@version=6 strategy("OCA Cancel Demo", overlay=true) float ma1 = ta.sma(close, 5) float ma2 = ta.sma(close, 9) if ta.cross(ma1, ma2) if strategy.position_size == 0 strategy.order("Long", strategy.long, stop = high) strategy.order("Short", strategy.short, stop = low) else strategy.close_all() plot(ma1, "Fast MA", color.aqua) plot(ma2, "Slow MA", color.orange)

Depending on the price action, the strategy might fill *both* stop orders before creating the closing market order. In that case, the strategy exits the position without executing the strategy.close_all() call because both orders have the same size. We see this behavior in the chart below, where the strategy alternated between filling the "Long" and "Short" orders a few times without executing an order from the strategy.close_all() command:

To eliminate cases in which the strategy fills the "Long" and "Short" orders before evaluating the strategy.close_all() call, we can instruct it to *cancel* one of the orders after the broker emulator fills the other. Below, we included "Entry" as the `oca_name` argument and strategy.oca.cancel as the `oca_type` argument in both strategy.order() calls. Now, after the either the "Long" or "Short" order fills, the strategy cancels the other order and waits for the market order from the strategy.close_all() command to close the position:

//@version=6 strategy("OCA Cancel Demo", overlay=true) float ma1 = ta.sma(close, 5) float ma2 = ta.sma(close, 9) if ta.cross(ma1, ma2) if strategy.position_size == 0 strategy.order("Long", strategy.long, stop = high, oca_name = "Entry", oca_type = strategy.oca.cancel) strategy.order("Short", strategy.short, stop = low, oca_name = "Entry", oca_type = strategy.oca.cancel) else strategy.close_all() plot(ma1, "Fast MA", color.aqua) plot(ma2, "Slow MA", color.orange)

`strategy.oca.reduce`

If an order placement command uses strategy.oca.reduce as its OCA type, the strategy *does not* cancel the resulting order entirely if another order with the same OCA name executes first. Instead, it *reduces* the order's size by the filled number of contracts/shares/lots/units. This behavior is particularly useful for custom exit strategies.

The following example demonstrates a *long-only* strategy that generates a single stop-loss order and two take-profit orders for each new entry. When a faster moving average crosses over a slower one, the script calls the strategy.entry() command with the argument `qty = 6` to create an entry order. Then, it uses three strategy.order() calls to create a stop order at the `stop` value and two limit orders at the `limit1` and `limit2` values. The strategy.order() call for the "Stop" order uses the argument `qty = 6`, and the two calls for the "Limit 1" and "Limit 2" orders both use `qty = 3`:

//@version=6 strategy("Multiple limits without reduction demo", overlay = true, behind_chart = false) var float stop = na var float limit1 = na var float limit2 = na bool longCondition = ta.crossover(ta.sma(close, 5),

ta.sma(close, 9)) if longCondition and strategy.position_size == 0 stop := close * 0.99 limit1 := close * 1.01 limit2 := close * 1.02 strategy.entry("Long", strategy.long, 6) strategy.order("Stop", strategy.short, stop = stop, qty = 6) strategy.order("Limit 1", strategy.short, limit = limit1, qty = 3) strategy.order("Limit 2", strategy.short, limit = limit2, qty = 3) bool showPlot = strategy.position_size != 0 plot(showPlot ? stop : na, "Stop", color.red, style = plot.style_linebr) plot(showPlot ? limit1 : na, "Limit 1", color.green, style = plot.style_linebr) plot(showPlot ? limit2 : na, "Limit 2", color.green, style = plot.style_linebr)

After adding this strategy to the chart, we see that it does not work as initially intended. The problem with this script is that the orders from the strategy.order() command **do not** belong to an OCA group by default (unlike strategy.exit(), whose orders automatically belong to a strategy.oca.reduce OCA group). Because the strategy does not assign the strategy.order() calls to any OCA group, it cannot reduce any unfilled stop or limit orders after a pending order executes. Consequently, if the broker emulator fills the stop order and at least one of the limit orders, the traded quantity **exceeds** the open long position, resulting in an open *short* position:

For our long-only strategy to work as we intended, we must instruct it to *reduce* the sizes of the unfilled stop/limit orders after one of them executes to prevent selling a larger quantity than the open long position.

Below, we specified “Bracket” as the `oca_name` argument and strategy.oca.reduce as the `oca_type` argument in all the script’s strategy.order() calls. These changes tell the strategy to reduce the sizes of the orders in the “Bracket” group each time the broker emulator fills one of them. This version of the strategy never simulates a short position, because the total size of its filled stop and limit orders never *exceeds* the long position’s size:

//@version=6 strategy("Multiple limits with reduction demo", overlay = true, behind_chart = false) var float stop = na var float limit1 = na var float limit2 = na bool longCondition = ta.crossover(ta.sma(close, 5), ta.sma(close, 9)) if longCondition and strategy.position_size == 0 stop := close * 0.99 limit1 := close * 1.01 limit2 := close * 1.02 strategy.entry("Long", strategy.long, 6) strategy.order("Stop", strategy.short, stop = stop, qty = 6, oca_name = "Bracket", oca_type = strategy.oca.reduce) strategy.order("Limit 1", strategy.short, limit = limit1, qty = 3, oca_name = "Bracket", oca_type = strategy.oca.reduce) strategy.order("Limit 2", strategy.short, limit = limit2, qty = 6, oca_name = "Bracket", oca_type = strategy.oca.reduce) bool showPlot = strategy.position_size != 0 plot(showPlot ? stop : na, "Stop", color.red, style = plot.style_linebr) plot(showPlot ? limit1 : na, "Limit 1", color.green, style = plot.style_linebr) plot(showPlot ? limit2 : na, "Limit 2", color.green, style = plot.style_linebr)

Note that:

- We also changed the `qty` value of the “Limit 2” order to 6 instead of 3 because the strategy reduces that order’s amount by three units after it executes the “Limit 1” order. Keeping the `qty` value of 3 would cause the second limit order’s size to decrease to 0 after the strategy fills the first limit order, resulting in no effect on the position.

`strategy.oca.none`

If an order placement command uses strategy.oca.none as its `oca_type` value, all orders from that command execute *independently* of any OCA group. This value is the default `oca_type` argument for the strategy.order() and strategy.entry() commands.

Currency

Pine Script strategies can simulate trades using different account currencies in their calculations. To set the default account currency for a strategy, include one of the `currency*` variables as the `currency` argument in the strategy() declaration statement. The default argument is currency.NONE, which specifies that the strategy uses the quoted currency on the chart by default (the currency indicated by the syminfo.currency variable). Users can change the strategy's account currency via the "Initial capital" inputs in the "Settings/Properties" tab and at the top of the strategy report.

If a strategy script uses an account currency that differs from the chart's currency, it uses the *previous daily value* of a corresponding currency pair from the most popular exchange to determine the conversion rate for all necessary calculations. If no available exchange provides the conversion rate directly, the strategy uses a spread to calculate the rate. The strategy multiplies all monetary values, such as simulated profits and losses, by the retrieved rate to express them in the account currency. Likewise, it inverts the rate to convert values in the account currency to prices in the quoted currency. To retrieve the rate that the strategy uses for currency conversion, call the request.currency_rate() function with syminfo.currency as the `from` argument and strategy.account_currency as the `to` argument.

> TipProgrammers can also directly convert values expressed in a strategy's account currency to the chart's currency, and vice versa, by using the strategy.convert_to_symbol() and strategy.convert_to_account() functions.

The following example demonstrates how currency conversion affects a strategy's monetary values, and how conversion rate calculations in a strategy match the calculations used by the `request.*()` functions.

On each of the latest 500 bars, the strategy below places an entry market order with the strategy.entry() command, then places a take-profit and stop-loss order *one tick* away from the entry price using the strategy.exit() command. Each entry order uses the size `1.0 / syminfo.mintick`. Therefore, the resulting profit or loss from each trade is syminfo.pointvalue units of chart currency per tick. We included currency.EUR as the `currency` argument in the strategy() declaration statement, causing the strategy to convert necessary values to *Euros* by default.

The script calculates the absolute one-bar change in the ratio of the strategy's net profit to the symbol's point value to derive the value of *one unit* of the chart's currency in Euros. It plots the result alongside the value returned by a request.currency_rate() call that uses syminfo.currency and strategy.account_currency as the `from` and `to` arguments. As shown on the chart below, both plots show the same values, confirming that the strategy and the data request use the same daily conversion rate in their calculations:

//@version=6 strategy("Currency Test", currency = currency.EUR) if last_bar_index - bar_index < 500 // Place an entry order with a size that results in a P/L of `syminfo.pointvalue` units of chart currency per tick. strategy.entry("LE", strategy.long, math.round_to_mintick(1.0 / syminfo.mintick)) // Place exit orders one tick above and below the "LE" entry price, // meaning each trade closes with one point of profit or loss in the chart's currency. strategy.exit("LX", "LE", profit = 1, loss = 1) // Plot the absolute change in `strategy.netprofit / syminfo.pointvalue`, which represents 1 chart unit of profit/loss. plot(math.abs(ta.change(strategy.netprofit / syminfo.pointvalue)), "1 chart unit of profit/loss in EUR", color = color.fuchsia, linewidth = 4) // Plot the requested currency rate. plot(request.currency_rate(syminfo.currency, strategy.account_currency), "Requested conversion rate", color.lime)

Note that:

- If a strategy executes on a chart with a timeframe higher than “1D”, it uses the data from *one day before* each *historical* bar’s closing time in currency conversions. For example, on a weekly chart, the strategy performs currency conversions using the confirmed rate from the previous Thursday’s close. However, it still uses the latest confirmed daily rate on *realtime* bars.

Altering calculation behavior

Similar to an indicator, a strategy script executes across all historical chart bars in order, then continues to execute across realtime bars that become available after updates from the data feed. However, while indicators always executes *once per bar* on historical bars and *once per tick* on realtime bars, a strategy script can execute *differently* on historical bars, realtime bars, or both, depending on the selected strategy properties.

> TipUnderstanding how these settings impact a strategy requires some knowledge of Pine’s Execution model. Therefore, we recommend reviewing the basics of the model before exploring the information below.

Unless otherwise specified, a strategy script executes strictly *once per closed bar*, regardless of whether a bar is part of the historical dataset or a new bar from the realtime data feed. If the current bar is open, the strategy *waits* until the bar closes before updating calculations or placing new orders. Additionally, each new order in the strategy’s simulation has a *one-tick delay* by default. Therefore, when using these default behaviors, a strategy places orders on a bar only when it reaches the bar’s close, and the earliest point at which the broker emulator can fill those orders is at the *open* of the following bar.

Users can configure a strategy to perform *more than one* execution per historical or realtime bar, allowing for more granular calculations and order fills, by selecting the checkboxes in the strategy’s Script execution settings. Users can also change the strategy’s order delay on closed bars to zero ticks by selecting the “None” option from the “Order execution delay” input in the “Settings/Properties” tab. Programmers can set the default script execution and order delay behaviors by including arguments for the `calc_on_every_tick`, `calc_on_order_fills`, `calc_on_every_history_tick`, and `process_orders_on_close` parameters in the strategy() declaration statement.

The sections below explain how each of these parameters affects a strategy’s default calculation behaviors.

> NoticeModifying a strategy’s “Script execution” settings can cause repainting in the strategy’s calculations and results, because they define different behaviors for historical and realtime bars, and both types of bars often have different levels of intrabar detail. Such differences can affect the behavior of order placement commands, Pine Logs, alerts, and any variables or fields that use the `varip` declaration mode. Therefore, when modifying these settings, we recommend inspecting a strategy’s logic carefully to ensure that it behaves as intended.

`calc_on_every_tick`

The `calc_on_every_tick` parameter of the strategy() declaration statement specifies the default behavior of the strategy’s executions on *realtime bars*. If the value is `true`, the script executes to update its calculations

on *every new tick* in an open realtime bar, similar to an indicator. The default value is `false`. Script users can override the specified default by selecting the *"On realtime bar tick"* checkbox from the Script execution settings in the "Settings/Properties" tab and at the top of the strategy report.

Configuring a strategy to execute on each realtime tick is sometimes useful when *forward testing* or updating *visuals* on live data, because it allows the strategy to perform more granular calculations using the latest price and volume updates for an open bar. However, depending on the strategy's logic, using this setting can also cause the strategy to behave *differently* across historical and realtime bars, leading to repainting in the strategy's orders and performance results. After a script reloads, all *elapsed* realtime bars from the previous run become *historical* bars in the new run, and the new historical bars **do not** preserve intrabar data from previous realtime ticks. Consequently, if a strategy places orders on the ticks within an open bar, it might not be able to reproduce the same orders after reloading.

The following example demonstrates how executing a strategy's logic on each new tick can significantly change the behavior of orders on realtime bars. If the current value of the close variable is the highest value in the series over a specified length, the script below calls the strategy.entry() command to place a long entry order. Otherwise, if the close value is the lowest over the same length, the script uses a separate strategy.entry() call to place a short entry order. We included `calc_on_every_tick = true` in the script's strategy() declaration statement to allow a new execution on each realtime tick by default:

//@version=6 strategy("Executions on realtime ticks demo", overlay = true, behind_chart = false, pyramiding = 20, calc_on_every_tick = true) //@variable The number of bars in the highest and lowest calculations. int lengthInput = input.int(15, "Length", minval = 1) // Calculate the highest and lowest `close` values over the input length. float highest = ta.highest(close, lengthInput) float lowest = ta.lowest(close, lengthInput) // Place a long market order if the `close` value equals the `highest` value. if close == highest strategy.entry("Buy", strategy.long) // Otherwise, place a short market order if the `close` value equals the `lowest` value. if close == lowest strategy.entry("Sell", strategy.short) // Highlight the background of realtime bars. bgcolor(barstate.isrealtime ? color.new(color.orange, 80) : na, title = "Realtime bar highlight") // Plot the `highest` and `lowest` series. plot(highest, "Highest close", color = color.lime) plot(lowest, "Lowest close", color = color.red)

Note that:

- The script uses a pyramiding value of 20, which allows to 20 entries per position using calls to the strategy.entry() command.
- The script indicates realtime bars by highlighting the chart's background in orange when the barstate.isrealtime value is `true`.

Because this strategy allows executions on every realtime tick, it updates its calculations and can place new orders after *each new update* from the data feed. Below, we applied the script to a chart and let it run on several realtime bars. On the chart's historical bars, the script places up to *one* market order per bar, and the broker emulator fills each order at the *open* of the following bar. By contrast, on realtime bars (the bars with an orange background), the script places *multiple* orders per bar – one for every *tick* on which the latest available close value equals the highest or lowest value over the specified length. Additionally, the broker emulator fills most of the orders on each highlighted bar *before* the next bar opens, because every update to a realtime bar is a *valid tick* for filling orders:

After we refresh our chart and run the script on the same bars again, the *elapsed* realtime bars from the previous script run become *historical* bars in the new run, and the script's behavior *changes* on those bars. Rather than placing multiple orders per bar on the former realtime bars, the script places only *one* order on

each closed bar whose *final* price equals the highest or lowest value, and the broker emulator fills that order at the *open* of the following bar:

Note that:

- This strategy also behaves differently if we enable executions after *each order fill* or on *each historical tick*, because both settings allow *additional* executions and fill prices for orders on *historical bars*. See the `calc_on_order_fills` and `calc_on_every_history_tick` sections below to learn more.

> TipWhen using the “On realtime bar tick” setting, users can help align script behaviors on historical and realtime bars by enabling the “On history bar tick” setting. When active, the script executes on each available tick across *historical* bars to *approximate* how the strategy would behave when executing on those bars as they formed. Allowing executions on historical ticks does not *eliminate* repainting. However, it can help reduce significant differences between the script’s behaviors on historical and realtime bars for more realistic results. See the `calc_on_every_history_tick` section for more information.

`calc_on_order_fills`

The `calc_on_order_fills` parameter of the strategy() declaration statement affects the default behavior of the strategy’s executions on both historical bars *and* realtime bars. If the value is `true`, the script performs an additional execution on *each tick* where the broker emulator *fills* an order. The default value is `false`. Script users can override the specified default behavior by selecting the *“On order fill”* checkbox from the Script execution settings in the “Settings/Properties” tab and at the top of the strategy report.

When using this setting, a strategy updates its calculations and can place new orders *immediately* after an order fills, without waiting for a bar’s closing tick. Those extra calculations update the `strategy.*` built-ins with more granular information that is otherwise not available until the closing tick, such as the current size or the average price of a new or modified position. This behavior is sometimes useful when backtesting or forward testing strategies that enter and exit trades mid-bar.

The example script below uses the strategy.entry() command to place a “Buy” entry order on any bar where the value of the strategy.position_size variable is 0. It also calls the strategy.exit() command on each bar to place “Exit” bracket orders to close any active “Buy” trade. The script calculates the stop-loss and take-profit prices for the exit orders based on the value of the strategy.position_avg_price variable.

The strategy() declaration statement includes the argument `calc_on_order_fills = true`. Therefore, in addition to updating calculations on each bar’s close, the strategy performs new calculations on each tick where the broker emulator fills a “Buy” or “Exit” order by default. Each time an “Exit” order fills, the strategy’s position size reverts to 0, triggering a new “Buy” order. The emulator then fills the “Buy” order on the next available tick, and the strategy.position_avg_price value automatically updates on that tick to store the average price of the new position. The strategy then uses the updated price to set the prices of new “Exit” orders. This cycle of intrabar entries and exits repeats across the bars in the dataset:

//@version=6 strategy("Executions after order fills demo", overlay = true, calc_on_order_fills = true) float stopSize = input.float(5.0, "SL %", minval = 0.0) / 100.0 float profitSize = input.float(5.0, "TP %", minval = 0.0) / 100.0 // The condition that triggers the entry order is `true` after each tick where an "Exit" // order fills. if strategy.position_size == 0.0 strategy.entry("Buy", strategy.long) // After each order fill, the values of these variables update immediately rather // than only at the bar's closing tick. float stopLoss =

strategy.position_avg_price * (1.0 - stopSize) float takeProfit = strategy.position_avg_price * (1.0 + profitSize) strategy.exit("Exit", stop = stopLoss, limit = takeProfit)

Note that:

- This strategy can repaint after running on realtime bars. As the strategy runs on historical bars, the broker emulator uses the chart's OHLC prices or the prices from a lower timeframe to determine the ticks available for filling orders, depending on the selected level of historical bar detail. By contrast, *each new update* to a realtime bar is a valid tick on which an order can fill. Consequently, when *elapsed* realtime bars become *historical* bars after the script reloads, the strategy may place or fill orders on those bars at *different* prices or times.
- If we deactivate executions after order fills, the strategy would *not* place new orders *before* a bar closes. Instead, it would wait for a bar's closing tick before placing any "Buy" order. The broker emulator would then fill the order at the open of the following bar, and the strategy.position_avg_price variable would return a usable value for calculating the "Exit" order prices on that bar.

It's crucial to note that enabling some strategies to execute after order fills can cause lookahead bias on historical bars. As a strategy executes across a dataset's history while using this setting, built-in variables that store price and volume data for the current bar – including high, low, close, and volume – consistently hold the bar's **final values**. Consequently, if the strategy uses these built-ins to control order logic on the ticks *within* a historical bar, it may produce **misleading** backtest results, as the logic relies on data that would **not** be available in live trading until the market reaches the bar's closing tick. Furthermore, the strategy's apparent future awareness on historical bars is *impossible* to reproduce on *realtime bars*. The bottom of the strategy report typically displays a *warning banner* when the "On order fill" execution setting is active to inform users about this behavior.

The following example demonstrates a simple strategy that produces lookahead bias when using the "On order fill" execution setting. The script calls the strategy.entry() command to place a long market order, then calls the strategy.exit() command to place a take-profit order at the current bar's high value. By default, the script places the entry order at the current bar's closing tick, and the broker emulator fills that order at the open of the following bar. The order fill on that bar triggers an additional execution, causing the strategy.exit() call to set the exit level to that bar's high price. As shown below, on most historical bars where the script enters a new long trade, it then exits the trade at the bar's *exact high*. This behavior is **misleading**, because knowing the exact high of a bar on the opening tick – let alone numerous consecutive times – is *impossible* to achieve in real-world trading:

//@version=6 strategy("Lookahead bias in orders demo", calc_on_order_fills = true, default_qty_type = strategy.percent_of_equity, default_qty_value = 2) //@variable A persistent variable that updates to hold the current bar's opening time. // The script uses this variable to limit the executions of the order placement / commands. varip int openTime = 0 //@variable Holds a string to append to each order's ID to indicate the trade number. string num = str.tostring(strategy.closedtrades) if openTime != time strategy.entry("Buy" + num, strategy.long) // Create a take-profit order to exit the open trade at the current `high` value. // When an entry order fills on a historical bar's opening tick, the call below // typically places the take-profit order at that bar's **final** high price. // This behavior is extremely **misleading**, and not reproducible on realtime bars, // because it is impossible to know the final high of a bar before the bar closes. strategy.exit("Exit" + num, limit = high) openTime := time

Note that:

- This script declares the `openTime` variable using the varip keyword. If a variable declaration uses this keyword, the variable persists across all executions without resetting to a previous state. The script uses this variable in the if structure to limit the placement of new orders to the current bar's first tick – where the variable's value does not yet match the current bar's opening time. To learn more about the behavior of this keyword, refer to the `varip` section of the Variable declarations page.
- This script behaves very differently on *realtime bars*. Rather than exiting trades at a bar's exact high, the strategy exits each trade at the bar's *developing* high as of the next tick or the current tick, because the final high price on a realtime bar is *unknown* until after the bar closes.

> TipTo enable script executions on historical intrabar ticks *without* causing this form of lookahead bias, use the *"On history bar tick"* execution setting rather than "On order fill". See the `calc_on_every_history_tick` section below to learn more.

`calc_on_every_history_tick`

The `calc_on_every_history_tick` parameter of the strategy() declaration statement specifies the default behavior of the strategy's executions on *historical bars*. This parameter requires a *named argument*, which includes the parameter's name and the assigned value (e.g., `calc_on_every_history_tick = true`). If the value is `true`, the script executes to update its calculations on *every available tick* in each historical bar. The default value is `false`. Script users can override the specified default by selecting the *"On history bar tick"* checkbox from the Script execution settings in the "Settings/Properties" tab and at the top of the strategy report.

> NoteThe "On history bar tick" feature is available only to users who have a Premium or Ultimate plan. Additionally, the feature requires a *standard* chart type. If the setting is enabled by default, the strategy raises a runtime error if it is not compatible with the user's plan or the current chart.

A strategy that uses the "On history bar tick" setting treats historical bars similarly to *realtime bars*. As the script executes on the available ticks of each historical bar, multiple built-in variables that hold price and volume data for the bar, including high, low, close, and volume, *update on every tick* to approximate the progression of values that would have been visible to the strategy when the bar was still open. This behavior helps reduce the risk of lookahead bias while executing on historical ticks, as the strategy cannot access the *final* price or volume data for a historical bar before reaching the bar's closing tick.

> NoteThis setting does not affect the behavior of variables that track bar states. For example, the value of the barstate.isconfirmed variable is always `true` on historical bars. It never changes to `false` during the executions across a historical bar's intrabar ticks while the "On history bar tick" setting is active.

The visibility of values within each historical bar depends on the selected level of historical bar detail. If the strategy uses high historical detail, it determines the high, low, close, and volume values available on each tick by requesting data from a *lower timeframe*. Otherwise, if the strategy uses the default level of detail, the values available on each tick depend on the broker emulator's assumptions about the price action within the historical bar. The following tables describe the values stored by the high, low, close, and volume variables

on each tick in a historical bar when using the default level of detail, based on the assumed order of price action within the bar:

Open → High → Low → Close

Tick	high	low	close	volume
1	Open	Open	Open	Total volume * 0.25
2	High	Open	High	Total volume * 0.5
3	High	Low	Low	Total volume * 0.75
4	High	Low	Close	Total volume

Open → Low → High → Close

Tick	high	low	close	volume
1	Open	Open	Open	Total volume * 0.25
2	Open	Low	Low	Total volume * 0.5
3	High	Low	High	Total volume * 0.75
4	High	Low	Close	Total volume

Note that:

- Variables that store values calculated from the chart's OHLCV data also update on each tick based on these assumptions. For example, if the *second* tick of the current historical bar is at the *high*, the value of the ohlc4 variable equals `(open + open + high + high) / 4`. However, the value still equals `(open + high + low + close) / 4` on the bar's final tick.

The following example demonstrates a strategy that enters trades on the ticks within historical bars when using the "On history bar tick" execution setting. The script declares two persistent variables named `openTime` and `firstPrice` to track time and price information from the first tick on which it executes in each historical bar. On each tick where the `openTime` value does not match the bar's opening time, the script reassigns that variable to store the current timestamp, then reassigns the `firstPrice` variable to hold the close value (i.e., the current price) for that tick.

The script uses the `firstPrice` value and the 10-bar highest and lowest prices to control order placement commands. On each tick where the current price is less than the `firstPrice` value and not equal to the current highest or lowest price, the script calls the strategy.entry command to place a long market order, then calls the strategy.exit() command to place take-profit and stop-loss orders at the highest and lowest values:

//@version=6 strategy("Executions on all historical ticks demo", overlay = true, behind_chart = false, default_qty_type = strategy.percent_of_equity, default_qty_value = 2, calc_on_every_history_tick = true) //@variable A persistent variable that tracks the current bar's opening time. varip int openTime = 0 //@variable A persistent variable that tracks the `close` value on each bar's *first* tick (i.e., the opening price). varip float firstPrice = na // The local scope of this structure executes only on the first tick of each bar, because the `time` variable's // value changes only once per bar. if time != openTime // Update the `openTime` variable, preventing subsequent executions of the scope on the same bar. openTime := time // Update the `firstPrice` variable to store the current `close` value. // When using the "On history bar tick"

setting, the value is the price as of the historical bar's *first tick*. // When not using the setting, the value consistently equals the historical bar's *final* closing price. firstPrice := close // Calculate the highest high and lowest low over the latest 10 bars. float highest = ta.highest(high, 10) float lowest = ta.lowest(low, 10) // When using the "On history bar tick" setting, this variable's value is `true` for each tick historical tick // where the *current* price is less than the price on the bar's first tick and not equal to the current highest high // or lowest low. // Without this setting enabled, the condition is *never* `true` on historical bars, because `close` and `firstPrice` // both refer to the *same value*. bool entryCondition = close < firstPrice and close != highest and close != lowest if entryCondition // Place a long entry order on the current tick. strategy.entry("Entry", strategy.long) // Place take-profit and stop-loss orders at the current tick's `highest` and `lowest` values, respectively. strategy.exit("Exit", "Entry", limit = highest, stop = lowest) // Plot the `highest` and `lowest` series for visual reference. plot(highest, "10-bar high", color.green) plot(lowest, "10-bar low", color.red)

Note that:

- This script simulates historical trades only while using the “On history bar tick” setting. Without the setting enabled, the value of the close variable on historical bars always represents the *final* closing price rather than the intrabar price on each tick. Therefore, when the script reassigns the `firstPrice` variable on any historical bar, that variable also stores the bar’s final price, and the `close < firstPrice` condition consistently evaluates to `false`, resulting in *no trades*.
- By default, this script does not simulate trades on *realtime bars*, because it executes only *once* at each bar’s close on that part of the dataset. To enable trades on realtime bars, select the “On realtime bar tick” checkbox in the “Script execution” settings or add `calc_on_every_tick = true` to the strategy() declaration statement.
- By default, the broker emulator fills the strategy’s orders based on the *chart prices* of each historical bar, because the script uses the default level of historical bar detail. Users can simulate the trades using more granular prices by selecting the “High” option in the script’s Bar detalization settings or including `use_bar_magnifier = true` in the declaration statement.

`process_orders_on_close`

The `process_orders_on_close` parameter of the strategy() declaration statement specifies the default behavior of orders created on a bar’s *closing tick*. If the value is `true`, the broker emulator can fill orders created on a bar’s close immediately, on the *same tick*, rather than waiting for the next available tick. If the value is `false` (the default), the earliest point at which the emulator can fill orders created on a bar’s closing tick is on the *next* tick, at the *open* of the following bar. Script users can override the specified default behavior via the “Order execution delay” dropdown menu in the “Settings/Properties” tab.

Strategies apply a one-tick delay to order fills by default because creating and filling an order on the same tick is typically *unrealistic* in real-world trading. However, simulating order fills at a bar’s close can be useful in some scenarios, such as when backtesting manual strategies in which traders exit a position immediately before the market closes, or when using order fill alerts to potentially trigger real-world orders before the start of the next trading session.

> Note
>
> Sending alerts to a third-party service for filling orders when using this setting might not work as a trader intends, especially in non-continuous markets, because the alerts still occur *after* the session closes.

> Depending on the external service and the type of market, real-world orders based on such alerts might not fill until after the market opens again.
>
> This setting *does not* affect the execution of orders that a strategy places *before* a bar's closing tick. If a strategy places orders on the ticks within a bar, the broker emulator fills those orders on the next available tick.

The following example demonstrates how removing the execution delay in orders at a bar's close changes the timing of trades. The initial script below uses the strategy.entry() command to place alternating long and short market orders across the dataset. The script places an order only at the close of each bar, then highlights the chart's background in blue or red to indicate the order direction. The strategy() declaration statement does not include a `process_orders_on_close` argument. Therefore, the broker emulator applies a one-tick delay to all orders placed on a bar's close by default. As shown by the trade markers on the chart's bars, after the strategy creates an order, the broker emulator fills the order at the open of the following bar:

//@version=6 strategy("Default delay demo", overlay = true) //@variable A translucent blue or red on each bar where a `strategy.entry()` call executes. color entryColor = na if bar_index % 10 == 0 // Place a long entry order and assign a blue color to `entryColor` if a short position or no position is open. if strategy.position_size <= 0 strategy.entry("Buy", strategy.long) entryColor := color.new(color.blue, 80) // Otherwise, place a short entry order and assign a red color to `entryColor`. else strategy.entry("Sell", strategy.short) entryColor := color.new(color.red, 80) // Highlight the chart's background to indicate when the entry orders occur. bgcolor(entryColor, title = "Entry order highlight")

If we add `process_orders_on_close = true` to the strategy() declaration statement, broker emulator fills each market order on the same closing tick where the script creates it by default. After applying this change to the script above, the trade markers on the chart now align with the displayed background colors, indicating that each order fills immediately rather than at the next bar's opening tick:

//@version=6 strategy("No delay for orders on close demo", overlay = true, process_orders_on_close = true) //@variable A translucent blue or red on each bar where a `strategy.entry()` call executes. color entryColor = na if bar_index % 10 == 0 // Place a long entry order and assign a blue color to `entryColor` if a short position or no position is open. if strategy.position_size <= 0 strategy.entry("Buy", strategy.long) entryColor := color.new(color.blue, 80) // Otherwise, place a short entry order and assign a red color to `entryColor`. else strategy.entry("Sell", strategy.short) entryColor := color.new(color.red, 80) // Highlight the chart's background to indicate when the entry orders occur. bgcolor(entryColor, title = "Entry order highlight")

> TipThe strategy.close() and strategy.close_all() commands feature an `immediately` parameter, which enables programmers to remove the delay for closing market orders *without* affecting the behavior of other order placement commands. If the value is `true`, the broker emulator fills the resulting market order on the same tick where the strategy calls the command. If `false` (the default), the behavior of the order depends on the strategy's "Order execution delay" setting.

Simulating trading costs

Strategy performance reports are more relevant and meaningful when they include potential real-world trading costs. Without modeling the potential costs associated with their trades, traders may overestimate a strategy's historical profitability, potentially leading to suboptimal decisions in live trading. Pine Script strategies include inputs and parameters for simulating trading costs in performance results.

Commission

Commission is the fee that a broker/exchange charges when executing orders. The commission can be a flat fee per order, a fee per contract/share/lot/unit, or a percentage of the total transaction value. Programmers can specify the default commission settings for their strategies by including `commission_type` and `commission_value` arguments in the strategy() declaration statement. If not specified, the strategy applies no commission to filled orders. Users can override the specified defaults by adjusting the "Commision" inputs in the strategy's "Settings/Properties" tab.

The following script is a simple strategy that enters a long position using 2% of its available equity when the current close value is the highest value over a user-specified length, and closes the trade when the close value is the lowest across that same number of bars. Because the declaration statement does not specify any `commission_*` arguments, the strategy simulates transactions without any commission by default:

//@version=6 // This declaration statement's `default_qty_*` arguments specify that `strategy.entry()` calls create orders // with 2% of the available equity by default. // It does not specify `commission_*` arguments, so this strategy applies no commission by default. strategy("Commission demo", overlay = true, default_qty_value = 2, default_qty_type = strategy.percent_of_equity) //@variable The number of bars for the `ta.highest()` and `ta.lowest()` calculations. int lengthInput = input.int(10, "Length", minval = 1) // Determine the highest and lowest `close` values over `lengthInput` bars. float highest = ta.highest(close, lengthInput) float lowest = ta.lowest(close, lengthInput) switch close // Place a long market order if the current `close` value is equal to the `highest` value. highest => strategy.entry("Long", strategy.long) // Place an order to close the position if the `close` value is equal to the `lowest` value. lowest => strategy.close("Long") // Plot the calculated `highest` and `lowest` series across the bars for reference. plot(highest, "Highest close", color.new(color.lime, 50), 2) plot(lowest, "Lowest close", color.new(color.red, 50), 2)

Note that:

- The `default_qty_type` and `default_qty_value` arguments in the strategy() statement set the strategy's default order type and size. This default order size applies only to orders from strategy.entry() and strategy.order() calls that do *not* include a `qty` argument. See the Position sizing section above for more information.

For our daily "NASDAQ:AAPL" chart above, the results in the strategy report show that the strategy had a positive equity growth of 18.67% over the testing range. However, these backtesting results do not account for any fees that the broker/exchange may charge.

Let's see what happens to these results when we add commission to every trade in the strategy simulation. In the example below, we modified the previous script to include the arguments `commission_type = strategy.commission.percent` and `commission_value = 1` in the strategy() declaration statement. With this change, the broker emulator now applies a commission of 1% of the transaction size to each filled order by default.

As shown below, after applying 1% commission to the strategy's orders on the same dataset, the strategy report shows a significantly reduced net profit, as well as increased volatility in the strategy's cumulative returns and an elevated maximum drawdown. These results highlight the impact that commission can have on a strategy's simulated performance:

//@version=6 // This declaration statement's `default_qty_*` arguments specify that `strategy.entry()` calls create orders // with 2% of the available equity by default. // The `commission_*` arguments specify that the strategy simulates a commission of 1% of each transaction's value. strategy("Commission demo", overlay = true, default_qty_value = 2, default_qty_type = strategy.percent_of_equity, commission_type = strategy.commission.percent, commission_value = 1) //@variable The number of bars for the `ta.highest()` and `ta.lowest()` calculations. lengthInput = input.int(10, "Length", minval = 1) // Determine the highest and lowest `close` values over `lengthInput` bars. float highest = ta.highest(close, lengthInput) float lowest = ta.lowest(close, lengthInput) switch close // Place a long market order if the current `close` value is equal to the `highest` value. highest => strategy.entry("Long", strategy.long) // Place an order to close the position if the `close` value is equal to the `lowest` value. lowest => strategy.close("Long") // Plot the calculated `highest` and `lowest` series across the bars for reference. plot(highest, "Highest close", color.new(color.lime, 50), 2) plot(lowest, "Lowest close", color.new(color.red, 50), 2)

Slippage and unfilled limits

In real-world trading, orders may execute at prices that differ from what the trader intended, due to volatility, liquidity, order size, and other market factors. Such differences can profoundly impact a strategy's performance. The disparity between the expected fill price of an order and the actual fill price is known as *slippage*. Slippage is dynamic and unpredictable, making it impossible to simulate precisely. However, applying a small, fixed amount of slippage to every order in a backtest or forward test can help overall performance results align more closely with reality. Programmers can specify a strategy's default slippage amount, as a fixed number of *ticks*, by including a `slippage` argument in the strategy() declaration statement. The default argument is 0. Script users can override the specified default by adjusting the "Slippage" input in the strategy's "Settings/Properties" tab.

The following example demonstrates how simulating slippage impacts the fill prices of market orders in a strategy test. The script below places a "Buy" market order of 2% equity when the market price is above a rising EMA, then closes the position with another market order when the price dips below the EMA while it's falling. The strategy() declaration statement includes the argument `slippage = 20`. Therefore, by default, each order fills 20 ticks beyond the intended price in the *unfavorable* direction (higher for long orders, and lower for short orders). The script plots the each order's expected fill price along with the simulated fill price after slippage to visually compare the difference:

//@version=6 strategy("Slippage Demo", overlay = true, slippage = 20, default_qty_value = 2, default_qty_type = strategy.percent_of_equity) int length = input.int(5, "Length") //@variable Exponential Moving Average with an input `length`. float ma = ta.ema(close, length) //@variable Is `true` when `ma` has increased and `close` is above it, `false` otherwise. bool longCondition = close > ma and ma > ma[1] //@variable Is `true` when `ma` has decreased and `close` is below it, `false` otherwise. bool shortCondition = close < ma and ma < ma[1] // Enter a long market position on `longCondition` and close the position on `shortCondition`. if longCondition strategy.entry("Buy", strategy.long) if shortCondition strategy.close("Buy") //@variable The `bar_index` of the position's entry order fill. int entryIndex = strategy.opentrades.entry_bar_index(0) //@variable The `bar_index` of the position's close order fill. int exitIndex = strategy.closedtrades.exit_bar_index(strategy.closedtrades - 1) //@variable The fill price simulated by the strategy. float fillPrice = switch bar_index entryIndex => strategy.opentrades.entry_price(0)

exitIndex => strategy.closedtrades.exit_price(strategy.closedtrades - 1) //@variable The expected fill price of the open market position. float expectedPrice = not na(fillPrice) ? open : na color expectedColor = na color filledColor = na if bar_index == entryIndex expectedColor := color.green filledColor := color.blue else if bar_index == exitIndex expectedColor := color.red filledColor := color.fuchsia plot(ma, color = color.new(color.orange, 50)) plotchar(not na(fillPrice) ? open : na, "Expected fill price", "—", location.absolute, expectedColor) plotchar(fillPrice, "Fill price after slippage", "—", location.absolute, filledColor)

Note that:

- Because the strategy applies constant slippage to *all* order fills, some orders can fill *outside* the candle range in the simulation. Therefore, we recommend exercising caution with this setting, as adding excessive simulated slippage can produce unrealistically *worse* testing results.

Some traders might assume that they can avoid the adverse effects of slippage by using limit orders. While market orders execute as soon as possible, irrespective of the price, limit orders execute at the specified price or a better value. They cannot execute at a *worse* price. However, in real-world trading, some limit orders *might not fill* when the market price reaches the specified value, due to insufficient liquidity or price action. Programmers can simulate the possibility of *unfilled* limit orders in their scripts by using the `backtest_fill_limits_assumption` parameter in the strategy() declaration statement. The parameter specifies the number of ticks that the price must move *beyond* each limit order's level to trigger an order fill at that level. The default argument is 0, meaning that the broker emulator fills any limit order immediately if the price reaches the order's level.

> NoteThe "Limit order execution" input in a strategy's "Settings/Properties" tab overrides the default assumption for filling limit orders. This input includes the options to fill each limit order when either the market price reaches the specified level or it moves *one tick* beyond that level. If the `backtest_fill_limits_assumption` argument is greater than 1, users can restore the default assumption specified in the declaration statement after using this input by selecting "Reset settings" from the "Defaults" dropdown menu at the bottom of the "Properties" tab.

The following example script places a limit order for 2% of the strategy's equity at a bar's hlcc4 price if the current high value is the highest price over a specified number of bars and no pending entry orders are active. The strategy closes the market position and cancels all pending orders if the current low value is the lowest price over the same length. Each time that the strategy triggers an order, it draws a horizontal line at the `limitPrice` value. It then updates the line on each bar until it closes the position or cancels the order:

//@version=6 strategy("Verify price for limits example", overlay = true, default_qty_type = strategy.percent_of_equity, default_qty_value = 2) int length = input.int(25, title = "Length") //@variable Draws a line at the limit price of the most recent entry order. var line limitLine = na // Highest high and lowest low highest = ta.highest(length) lowest = ta.lowest(length) // Place an entry order and draw a new line when the the `high` equals the `highest` value and `limitLine` is `na`. if high == highest and na(limitLine) float limitPrice = hlcc4 strategy.entry("Long", strategy.long, limit = limitPrice) limitLine := line.new(bar_index, limitPrice, bar_index + 1, limitPrice) // Close the open market position, cancel orders, and set `limitLine` to `na` when the `low` equals the `lowest` value. if low == lowest strategy.cancel_all() limitLine := na strategy.close_all() // Update the `x2` value of `limitLine` if it isn't `na`. if not na(limitLine)

limitLine.set_x2(bar_index + 1) plot(highest, "Highest High", color = color.new(color.green, 50)) plot(lowest, "Lowest Low", color = color.new(color.red, 50))

By default, the broker emulator assumes that all limit orders are guaranteed to fill when the market price reaches their values. However, that may not be the case in a real-world market. Let's add price verification to our script's limit orders to account for potentially unfilled ones. In the example below, we added `backtest_fill_limits_assumption = 3` to the strategy() statement. With this change, the broker emulator assumes that there is sufficient liquidity to fill a limit order only if the price moves *three ticks* beyond the order's level. As shown below, using this fill assumption prevents the execution of some orders and changes the *times* of others:

> NoticeUsing a nonzero `backtest_fill_limits_assumption` argument can affect the *times* at which limit orders execute, as shown above. However, regardless of the fill time, the broker emulator still executes verified limit orders at their specified *prices*. This behavior is a necessary compromise to preserve the intended fill prices of limit orders without causing *lookahead bias*, but it can also cause the orders to execute at times that may *not* be possible in real-word trading, especially if the argument is a large value. Therefore, we recommend that users exercise caution and understand this price-time limitation when applying price verification to limit orders.

Risk management

Designing a strategy that performs well, especially across a broad range of markets, is a challenging task. Most strategies are designed for specific market patterns or conditions and can produce uncontrolled losses when applied to other datasets. Therefore, a strategy's risk management behavior can be critical to its performance. Programmers can specify risk management criteria in their strategy scripts by using the `strategy.risk.*()` commands.

Strategies can incorporate any number of risk management criteria in any combination. All risk management commands execute *on every tick and order execution event*, regardless of any changes to the strategy's calculation behavior. There is no way to deactivate any of these commands on specific script executions. Irrespective of a risk management command's location, the command *always* applies to the strategy unless the programmer removes the call from the code. Below, we list the available risk management commands and the behaviors they define:

strategy.risk.allow_entry_in()

This command overrides the market direction allowed for all calls to the strategy.entry() command. If a user specifies the trade direction with the strategy.risk.allow_entry_in() function (e.g., strategy.direction.long), the strategy enters trades only in that direction. If the script calls an entry command in the opposite direction of an open market position, the strategy generates a market order to *close* the position without entering a new trade in that direction.

strategy.risk.max_cons_loss_days()

This command cancels all pending orders, closes any open market position, and stops all additional trade actions after the strategy simulates a defined number of trading days with consecutive losses.

strategy.risk.max_drawdown()

This command cancels all pending orders, closes any open market position, and stops all additional trade actions after the strategy's drawdown reaches the amount specified in the function call.

strategy.risk.max_intraday_filled_orders()

This command specifies the maximum number of filled orders per trading day (or per chart bar if the chart's timeframe is higher than "1D"). If the strategy fills more orders than the specified limit, the command cancels all pending orders, closes any open market position, and halts trading activity until the end of the current session.

strategy.risk.max_intraday_loss()

This command controls the maximum allowed loss per trading day (or per chart bar if the chart's timeframe is higher than "1D"). If the strategy's losses reach the specified threshold, the command cancels all pending orders, closes the open market position, and stops all trading activity until the end of the current session.

strategy.risk.max_position_size()

This command specifies the maximum possible position size when calling the strategy.entry() command. If the size of an entry order results in a market position that exceeds the specified threshold, the strategy reduces the order quantity so that the resulting position does not exceed the limit.

Margin and leverage

Margin is the minimum percentage of a market position that a trader must hold in their account as collateral to open and maintain that position. With a margin of 100%, the trader must cover the entire position using their account's available funds. With a margin of 25%, the trader must cover only *one-fourth* of each position using their account's funds to maintain a *loan* for the other three-fourths from the broker. Most brokers define a trader's margin requirements based on a specified *leverage* amount, where leverage is the *inverse* of margin. For example, a leverage ratio of 4:1 is equivalent to 25% margin. A trader can open a position for up to *four times* their available funds when using this amount of leverage, because they must maintain only a fourth of each position's size as collateral. In other words, the trader has four times the purchasing power that they would otherwise have when trading using only their account's funds.

The `margin_long` and `margin_short` parameters of the strategy() declaration statement define the default required *margin percentages* for long and short trades, respectively. The strategy *converts* the specified percentages to leverage ratios and uses those ratios as the default values for the "Long leverage" and "Short leverage" inputs in the "Settings/Properties" tab. The default argument for both parameters is 100, which is equivalent to a leverage ratio of 1:1.

> NoticeA margin requirement of less than 0.2% (i.e., leverage greater than 500:1) is typically *unrealistic* in a real-world market. Using unrealistic levels of margin in a strategy can cause very *misleading* backtest results. Furthermore, using a margin of 0% is *extremely* misleading because it is equivalent to *infinite* leverage, which is impossible to achieve in any live trading environment. Therefore, when setting a strategy's margin via the `margin_*` parameters, or adjusting leverage using the "Leverage *" inputs, we recommend specifying realistic values that align with current market conditions.

Trading with less than 100% margin can significantly increase a trader's potential profits and their potential **losses**. In real-world trading, if the loss from a position causes the trader's available margin to fall below the required margin, the broker issues a *margin call*, which is a demand for the trader to immediately deposit additional funds to cover the loss. If the trader fails to meet the demand, or if the losses reach beyond the broker's limits, the broker forcibly *liquidates* all or part of the position to prevent further losses that the trader cannot cover.

To simulate this process in strategies, the broker emulator generates *margin call events* if a strategy's available funds fall below the required margin percentage. Each time that a margin call event occurs, the emulator immediately liquidates *four times* the number of contracts/shares/lots/units required to cover the loss to help prevent continuous margin calls across subsequent bars. The emulator uses the following algorithm to determine the liquidated quantity for each event:

1. Calculate the amount of capital spent on the position: `Money Spent = Quantity * Entry Price`
2. Calculate the Market Value of Security (MVS): `MVS = Position Size * Current Price`
3. Calculate the Open Profit as the difference between `MVS` and `Money Spent`. If the position is short, multiply this value by -1.
4. Calculate the strategy's equity value: `Equity = Initial Capital + Net Profit + Open Profit`
5. Calculate the margin ratio: `Margin Ratio = Margin Percent / 100`
6. Calculate the margin value, which is the cash required to cover the hypothetical account's portion of the position: `Margin = MVS * Margin Ratio`
7. Calculate the strategy's available funds: `Available Funds = Equity - Margin`
8. Calculate the total amount of money lost: `Loss = Available Funds / Margin Ratio`
9. Calculate the number of contracts/shares/lots/units the account must liquidate to cover the loss, truncated to the same decimal precision as the minimum position size for the current instrument: `Cover Amount = TRUNCATE(Loss / Current Price).`
10. Multiply the quantity required to cover the loss by four to determine the margin call size: `Margin Call Size = Cover Amount * 4`

> NoteA short trade typically entails *borrowing* shares from a broker to sell them at the current market price, then buying the shares back at a different price and returning them to the lender. The trade thus produces a profit if the trader purchases the shares at a *lower* price, or a loss if they purchase the shares at a *higher* price. Consequently, unlike long trades, short trades carry the risk of *uncapped losses*, because there is no definite limit on how far an instrument's price can rise. Therefore, short trades in a strategy that uses *any* nonzero margin are subject to forced liquidation from margin call events, even if the `margin_short` argument is 100 or the "Short leverage" input value is 1.

To examine the above calculations in detail, the following example applies the Supertrend Strategy built-in script to a daily "NASDAQ:TSLA" chart. In the strategy's properties, we set the order size to 300% of equity, and we set the strategy's long leverage to 4. On the chart below, the strategy opens a long trade, then the broker emulator triggers margin call event a few bars later:

In the above image, the script enters the long position at the bar's opening price on 16 Sep 2010. The strategy purchases 682,438 shares (Position Size) at 4.43 USD (Entry Price) using 25% margin. Then, on 23 Sep 2010, when the price drops to 3.9 USD (Current Price), the emulator triggers the margin call event and forcibly liquidates *111,052 shares*. The calculations below explain how the broker emulator determines the liquidation quantity for this event:

```
Money spent: 682438 * 4.43 = 3023200.34
MVS: 682438 * 3.9 = 2661508.2
Open Profit: -361692.14
Equity: 1000000 + 0 - 361692.14 = 638307.86
Margin Ratio: 25 / 100 = 0.25
Margin: 2661508.2 * 0.25 = 665377.05
Available Funds: 638307.86 - 665377.05 = -27069.19
Money Lost: -27069.19 / 0.25 = -108276.76
Cover Amount: TRUNCATE(-108276.76 / 3.9) = TRUNCATE(-27763.27) = -27763
Margin Call Size: -27763 * 4 = - 111052
```

> TipProgrammers can use the strategy.margin_liquidation_price variable to retrieve the current *price level* at which the broker emulator will trigger a margin call event and forcibly liquidate part or all of an open position if the market price reaches it. For more information about how margin and leverage work in strategies, as well as details on how to calculate the liquidation price, refer to the How to simulate trading with leverage in Pine Script article in our Help Center.

Using strategy information in scripts

Numerous built-ins within the `strategy.*` namespace and its *sub-namespaces* provide convenient solutions for programmers to use a strategy's trade and performance information, including data shown in the strategy report, directly within their code's logic and calculations.

Several `strategy.*` variables store essential information about a strategy, including its starting capital, equity, profits and losses, run-up and drawdown, and open position:

- strategy.account_currency
- strategy.initial_capital
- strategy.equity
- strategy.netprofit and strategy.netprofit_percent
- strategy.grossprofit and strategy.grossprofit_percent
- strategy.grossloss and strategy.grossloss_percent
- strategy.openprofit and strategy.openprofit_percent
- strategy.max_runup and strategy.max_runup_percent
- strategy.max_drawdown and strategy.max_drawdown_percent
- strategy.position_size
- strategy.position_avg_price
- strategy.position_entry_name

Additionally, the namespace features multiple variables that store general trade information, such as the number of open and closed trades, the number of winning and losing trades, average trade profits, and maximum trade sizes:

- strategy.opentrades
- strategy.closedtrades

- strategy.wintrades
- strategy.losstrades
- strategy.eventrades
- strategy.avg_trade and strategy.avg_trade_percent
- strategy.avg_winning_trade and strategy.avg_winning_trade_percent
- strategy.avg_losing_trade and strategy.avg_losing_trade_percent
- strategy.max_contracts_held_all
- strategy.max_contracts_held_long
- strategy.max_contracts_held_short

Programmers can use these variables to display relevant strategy information on their charts, create customized trading logic based on strategy data, calculate custom performance metrics, and more.

The following example demonstrates a few simple use cases for these `strategy.*` variables. The script uses them in its order placement and display calculations. When the calculated `rank` value crosses above 10 and the strategy.opentrades value is 0, the script calls the strategy.entry() command to place a “Buy” market order. On the following bar, where that order fills, the script calls strategy.exit() to create a stop-loss order at a user-specified percentage below the strategy.position_avg_price value. If the `rank` crosses above 80 during the open trade, the script calls strategy.close() to exit the position on the next bar.

The script creates a table to display formatted strings representing information from several of the above `strategy.*` variables on the main chart pane. The text in the table shows the strategy’s net profit and net profit percentage, the account currency, the number of winning trades and the win percentage, the ratio of the average winning trade to the average losing trade, and the profit factor (the ratio of the gross profit to the gross loss). The script also plots the strategy.equity series in a separate pane and highlights the pane’s background based on the strategy.openprofit value:

//@version=6 strategy("Using strategy information demo", default_qty_type = strategy.percent_of_equity, default_qty_value = 5, margin_long = 100, margin_short = 100) //@variable The number of bars in the `rank` calculation. int lengthInput = input.int(50, "Length", 1) //@variable The stop-loss percentage. float slPercentInput = input.float(4.0, "SL %", 0.0, 100.0) / 100.0 //@variable The percent rank of `close` prices over `lengthInput` bars. float rank = ta.percentrank(close, lengthInput) // Entry and exit signals. bool entrySignal = ta.crossover(rank, 10) and strategy.opentrades == 0 bool exitSignal = ta.crossover(rank, 80) and strategy.opentrades == 1 // Place orders based on the `entrySignal` and `exitSignal` occurrences. switch entrySignal => strategy.entry("Buy", strategy.long) entrySignal[1] => strategy.exit("SL", "Buy", stop = strategy.position_avg_price * (1.0 - slPercentInput)) exitSignal => strategy.close("Buy") if barstate.islastconfirmedhistory or barstate.isrealtime //@variable A table displaying strategy information on the main chart pane. var table dashboard = table.new(position.top_right, 2, 10, border_color = chart.fg_color, border_width = 1, force_overlay = true) //@variable The strategy's currency. string currency = strategy.account_currency // Display the net profit as a currency amount and percentage. dashboard.cell(0, 1, "Net P/L") dashboard.cell(1, 1, str.format("{0, number, 0.00} {1} ({2}%)", strategy.netprofit, currency, strategy.netprofit_percent), text_color = chart.fg_color, bgcolor = strategy.netprofit > 0 ? color.lime : color.red) // Display the number of winning trades as an absolute value and percentage of all completed trades. dashboard.cell(0, 2, "Winning trades") dashboard.cell(1, 2, str.format("{0} ({1, number, #.##%})", strategy.wintrades, strategy.wintrades / strategy.closedtrades), text_color = chart.fg_color, bgcolor = strategy.wintrades > strategy.losstrades ? color.lime : color.red) // Display the ratio of average trade profit to average trade loss. dashboard.cell(0, 3, "Avg. win / Avg. loss") dashboard.cell(1, 3, str.format("{0, number, #.###}", strategy.avg_winning_trade / strategy.avg_losing_trade), text_color = chart.fg_color, bgcolor = strategy.avg_winning_trade > strategy.avg_losing_trade ? color.lime : color.red) // Display the profit factor,

i.e., the ratio of gross profit to gross loss. dashboard.cell(0, 4, "Profit factor") dashboard.cell(1, 4, str.format("{0, number, #.###}", strategy.grossprofit / strategy.grossloss), text_color = chart.fg_color, bgcolor = strategy.grossprofit > strategy.grossloss ? color.lime : color.red) // Plot the current equity in a separate pane and highlight the pane's background while there is an open position. plot(strategy.equity, "Total equity", strategy.equity > strategy.initial_capital ? color.teal : color.maroon, 3) bgcolor(strategy.openprofit > 0 ? color.new(color.teal, 80) : strategy.openprofit < 0 ? color.new(color.maroon, 80) : na, title = "Open position highlight")

Note that:

- This script creates a stop-loss order one bar after the entry order because it uses strategy.position_avg_price to determine the price level. This variable has a non-na value only when the strategy has an *open position*.
- The script draws the table only on the last historical bar and all realtime bars because the *historical* states of tables are **never visible**. See the Reducing drawing updates section of the Profiling and optimization page for more information.
- The table.new() call includes `force_overlay = true` to display the table on the main chart pane.

Individual trade information

The `strategy.*` namespace features two sub-namespaces that provide access to *individual trade* information: `strategy.opentrades.*` and `strategy.closedtrades.*`. The `strategy.opentrades.*` built-ins return data for *incomplete* (open) trades, and the `strategy.closedtrades.*` built-ins return data for *completed* (closed) trades. With these built-ins, programmers can use granular trade data in their scripts, allowing for more detailed strategy analysis and advanced calculations.

Both sub-namespaces contain several similar functions that return information about a trade's orders, simulated costs, and profit/loss, including:

- strategy.opentrades.entry_id() / strategy.closedtrades.entry_id()
- strategy.opentrades.entry_price() / strategy.closedtrades.entry_price()
- strategy.opentrades.entry_bar_index() / strategy.closedtrades.entry_bar_index()
- strategy.opentrades.entry_time() / strategy.closedtrades.entry_time()
- strategy.opentrades.entry_comment() / strategy.closedtrades.entry_comment()
- strategy.opentrades.size() / strategy.closedtrades.size()
- strategy.opentrades.profit() / strategy.closedtrades.profit()
- strategy.opentrades.profit_percent() / strategy.closedtrades.profit_percent()
- strategy.opentrades.commission() / strategy.closedtrades.commission()
- strategy.opentrades.max_runup() / strategy.closedtrades.max_runup()
- strategy.opentrades.max_runup_percent() / strategy.closedtrades.max_runup_percent()
- strategy.opentrades.max_drawdown() / strategy.closedtrades.max_drawdown()
- strategy.opentrades.max_drawdown_percent() / strategy.closedtrades.max_drawdown_percent()
- strategy.closedtrades.exit_id()
- strategy.closedtrades.exit_price()
- strategy.closedtrades.exit_time()
- strategy.closedtrades.exit_bar_index()
- strategy.closedtrades.exit_comment()

Note that:

- Most built-ins within these namespaces are *functions*. However, the `strategy.opentrades.*` namespace also features a unique *variable*: strategy.opentrades.capital_held. Its value represents the amount of capital *reserved* by *all* open trades.
- Only the `strategy.closedtrades.*` namespace has `.exit_*()` functions that return information about *exit orders*.

All `strategy.opentrades.*()` and `strategy.closedtrades.*()` functions have a `trade_num` parameter, which accepts an "int" value representing the index of the open or closed trade. The index of the first open/closed trade is 0, and the last trade's index is *one less* than the value of the strategy.opentrades/strategy.closedtrades variable.

The following example places up to five long entry orders per position, each with a unique ID, and it calculates metrics for specific closed trades.

The strategy places a new entry order when the close crosses above the `median` value without reaching the `highest` value, but only if the number of open trades is less than five. It exits each position using stop-loss orders from strategy.exit() or a market order from strategy.close_all(). Each successive entry order's ID depends on the number of open trades. The first entry ID in each position is `"Buy0"`, and the last possible entry ID is `"Buy4"`.

The script calls `strategy.closedtrades.*()` functions within a for loop to access closed trade entry IDs, profits, entry bar indices, and exit bar indices. It uses this information to calculate the total number of closed trades with the specified entry ID, the number of winning trades, the average number of bars per trade, and the total profit from all the trades. The script then organizes this information in a formatted string and displays the result using a single-cell table:

//@version=6 strategy("Individual trade information demo", pyramiding = 5, default_qty_type = strategy.percent_of_equity, default_qty_value = 1, margin_long = 100, margin_short = 100) //@variable The number of bars in the `highest` and `lowest` calculation. int lengthInput = input.int(50, "Length", 1) string idInput = input.string("Buy0", "Entry ID to analyze", ["Buy0", "Buy1", "Buy2", "Buy3", "Buy4"]) // Calculate the highest, lowest, and median `close` values over `lengthInput` bars. float highest = ta.highest(close, lengthInput) float lowest = ta.lowest(close, lengthInput) float median = 0.5 * (highest + lowest) // Define entry and stop-loss orders when the `close` crosses above the `median` without touching the `highest` value. if ta.crossover(close, median) and close != highest and strategy.opentrades < 5 strategy.entry("Buy" + str.tostring(strategy.opentrades), strategy.long) if strategy.opentrades == 0 strategy.exit("SL", stop = lowest) // Close the entire position when the `close` reaches the `lowest` value. if close == lowest strategy.close_all() // The total number of closed trades with the `idInput` entry, the number of wins, the average number of bars, // and the total profit. int trades = 0 int wins = 0 float avgBars = 0 float totalPL = 0.0 if barstate.islastconfirmedhistory or barstate.isrealtime //@variable A single-cell table displaying information about closed trades with the `idInput` entry ID. var table infoTable = table.new(position.middle_center, 1, 1, color.purple) // Iterate over closed trade indices. for tradeNum = 0 to strategy.closedtrades - 1 // Skip the rest of the current iteration if the `tradeNum` closed trade didn't open with an `idInput` entry. if strategy.closedtrades.entry_id(tradeNum) != idInput continue // Accumulate `trades`, `wins`, `avgBars`, and `totalPL` values. float profit = strategy.closedtrades.profit(tradeNum) trades += 1 wins += profit > 0 ? 1 : 0 avgBars += strategy.closedtrades.exit_bar_index(tradeNum) - strategy.closedtrades.entry_bar_index(tradeNum) + 1 totalPL += profit avgBars /= trades //@variable A formatted string containing the calculated closed trade information. string displayText = str.format("ID: {0}\n\nTotal trades: {1}\nWin trades: {2}\nAvg. bars: {3}\nTotal P/L: {4} {5}", idInput, trades, wins, avgBars, totalPL, strategy.account_currency) // Populate the table's cell with `displayText`. infoTable.cell(0, 0,

displayText, text_color = color.white, text_halign = text.align_left, text_size = size.large) // Plot the highest, median, and lowest values on the main chart pane. plot(highest, "Highest close", force_overlay = true) plot(median, "Median close", force_overlay = true) plot(lowest, "Lowest close", force_overlay = true)

Note that:

- This strategy can open up to five long trades per position by default because we included `pyramiding = 5` in the strategy() declaration statement. See the pyramiding section for more information.
- The strategy.exit() call in this script persists and generates exit orders for every entry in the open position because it does not include a `from_entry` argument. See the Exits for multiple entries section to learn more about this behavior.

Strategy alerts

Pine Script indicators (not strategies) have two different mechanisms to set up custom alert conditions: the alertcondition() function, which defines a separate alert trigger per function call, and the alert() function, which defines a single alert trigger based on all calls in the code, but provides greater flexibility in the number of calls, alert messages, etc.

Pine Script strategies cannot create alert triggers using the alertcondition() function, but they can create triggers with the alert() function. Additionally, each order placement command comes with its own built-in alert functionality that does not require any additional code to implement. As such, any strategy that uses an order placement command can issue alerts upon order execution. The precise mechanics of such built-in strategy alerts are described in the Order Fill events section of the Alerts page.

If a strategy uses both the alert() function and functions that create orders in the same script, the “Create Alert” dialog box provides a choice between the conditions to use as a trigger: alert() events, order fill events, or both.

For many trading strategies, the delay between a triggered alert and a live trade can be a critical performance factor. By default, strategy scripts execute alert() function calls only on the close of realtime bars, as if they used the alert.freq_once_per_bar_close frequency, regardless of the `freq` argument in the call. Users can change the allowed alert frequency by enabling the “On realtime bar tick” or “On order fill” Script execution setting. See the Altering calculation behavior section above to learn more about these settings.

Order fill alert triggers do not suffer the same limitations as the triggers from alert() calls. Alerts from order fill events execute *immediately*, regardless of the script’s execution settings. Therefore, they are often more suitable for sending alerts to third parties for automation. Users can specify the default message for order fill alerts by using the `//@strategy_alert_message` compiler annotation. The text included in the annotation populates the “Message” field in the “Create alert” dialog box.

The following script shows a simple example of a default order fill alert message. Above the strategy() declaration statement, the script includes the `//@strategy_alert_message` annotation with placeholders for the trade action, current position size, ticker name, and fill price values in the message text:

//@version=6 //@strategy_alert_message {{strategy.order.action}} {{strategy.position_size}} {{ticker}} @ {{strategy.order.price}} strategy("Alert Message demo", overlay = true) float fastMa = ta.sma(close, 5) float slowMa = ta.sma(close, 10) if ta.crossover(fastMa, slowMa) strategy.entry("buy", strategy.long) if

ta.crossunder(fastMa, slowMa) strategy.entry("sell", strategy.short) plot(fastMa, "Fast MA", color.aqua) plot(slowMa, "Slow MA", color.orange)

This script populates the “Create alert” dialog box with its default message when the user selects its name from the “Condition” section:

Each time the alert occurs, the strategy *replaces* the placeholders in the alert message with their corresponding values. For example:

Notes on testing strategies

Testing and tuning strategies in historical and live market conditions can provide insight into a strategy’s characteristics, potential weaknesses, and *possibly* its future potential. However, traders should always be cautious of the biases and limitations of simulated strategy results, especially when using the results to support live trading decisions. This section outlines some caveats associated with strategy validation and tuning, and lists possible solutions to mitigate their effects.

> NoticeAlthough testing strategies on existing data might provide traders useful information about a strategy’s qualities, it’s crucial to understand that neither the past nor the present guarantees the *future*. Financial markets can change rapidly and unpredictably, leading to uncontrollable, unforeseen losses. Additionally, simulated results may not fully account multiple real-world factors that can impact trading performance. Therefore, we recommend that traders thoroughly understand the limitations of strategy simulations, and consider them only “parts of the whole” in their validation processes rather than basing decisions solely on the results.

Backtesting and forward testing

Backtesting is a technique to evaluate the past performance of a trading strategy or model by simulating and analyzing its results on historical market data. This technique assumes that a strategy’s results on past data can provide insight into its strengths and weaknesses. When backtesting, many traders adjust the parameters of a strategy in an attempt to optimize its results. Analysis and optimization of historical results can help traders to gain a deeper understanding of a strategy’s characteristics. However, traders should always understand the risks and limitations when basing their live trading decisions on optimized backtest results.

It is prudent to also use realtime analysis as a tool for evaluating a trading system on a forward-looking basis. *Forward testing* aims to gauge the performance of a strategy in live market conditions, where factors such as trading costs, slippage, and liquidity can meaningfully affect overall trading performance. While forward testing has the distinct advantage of not being affected by some types of biases (e.g., lookahead bias or “future data leakage”), it does carry the disadvantage of being limited in the quantity of data to test. Therefore, although it can provide helpful insights into a strategy’s performance in current market conditions, forward testing is not typically used on its own.

Lookahead bias

One typical issue in backtesting strategies that request alternate timeframe data, use repainting variables such as timenow, or alter calculation behavior for intrabar order fills, is the leakage of *future* data into the *past* during evaluation. We refer to this behavior as *lookahead bias*. Not only is lookahead bias a common

cause of unrealistic strategy results, because the future is always unknown, but it is also one of the common causes of strategy repainting.

Traders can often confirm whether a strategy has lookahead bias by forward testing it on realtime data, where no known data exists beyond the latest bar. Because there is no *future* data to leak into the past on realtime bars, the strategy will behave *differently* on historical and realtime bars if it is affected by lookahead bias.

To eliminate lookahead bias in a strategy:

- Do not use repainting variables that leak future values into the past in the strategy's order placement or cancellation logic.
- Do not include barmerge.lookahead_on in `request.*()` calls without offsetting the data series with the history-referencing operator, especially when requesting data from a higher timeframe. See the `lookahead` section of the Other timeframes and data page for more information.
- When using the "On order fill" Script execution setting, avoid using the *current* values of built-in variables that hold price or volume data for each bar. During the script's historical executions, the current value of a variable such as close always refers to the *final value* as of the bar's *closing tick*, even while the script executes on the bar's *first* tick. See the `calc_on_order_fills` section above to learn more about this behavior.

Selection bias

Selection bias occurs when a trader analyzes performance results for a set of specific instruments, timeframes, or testing ranges while ignoring others. This bias can distort the trader's perspective of the strategy's robustness, thus impacting trading decisions and performance optimizations. Traders can reduce the effects of selection bias by evaluating their strategies on multiple, ideally *diverse* datasets, and avoiding ignoring poor performance results or "cherry-picking" testing ranges.

Overfitting

A common problem when optimizing a strategy based on backtest results is *overfitting* ("curve fitting"), which refers to tailoring the strategy for improved performance on specific datasets. A strategy that suffers from overfitting often fails to perform well on new, *unseen* data. One widely-used approach to help reduce overfitting and promote better generalization is to split an instrument's data into two or more parts to test the strategy outside the sample used for optimization. This process is often known as "in-sample" (IS) and "out-of-sample" (OOS) backtesting.

When using this approach, traders optimize strategy parameters on the IS data. Then, they test the optimized configuration on the OOS data without additional fine-tuning. Although this technique and other, more robust approaches might provide a glimpse into how a strategy might fare after optimization, traders should still exercise caution. No trading strategy can *guarantee* future performance, regardless of the data used for optimization and testing, because the future is inherently *unknown*.

Trade limit

By default, strategies preserve information for up to the latest *9000 trades*. If a strategy simulates *more* than 9000 trades, it *trims* the information for the oldest closed trades and maintains data for only the most recent trades. Information for trimmed trades or their orders are **not** visible in the strategy report interface or any downloaded XLSX or CSV files. Likewise, the `strategy.closedtrades.*()` functions return na for all trimmed

trades. However, if the strategy uses Deep Backtesting mode, it maintains information for *every* closed trade and does not trim any orders from the report.

Programmers can retrieve the index of the oldest *untrimmed* trade, which corresponds to the oldest trade listed in the strategy report's "Trades" tab, by using the strategy.closedtrades.first_index variable. Scripts can use the index in `strategy.closedtrades.*()` calls to retrieve information for the oldest available closed trade. If the strategy simulates fewer than 9000 trades or runs in Deep Backtesting mode, the variable's value is 0, representing the actual first trade in the simulation.

> NoteDeep Backtesting mode only affects the testing range of results displayed in the *strategy report*. When using this mode, the strategy's trade markers, plots, alerts, and Pine Logs are calculated using only the available chart data, regardless of the specified testing range.

To learn more about retrieving trade data using the `strategy.closedtrades.*()` built-ins, refer to the Individual trade information section above.

Strings

Introduction

Pine Script® strings are immutable values containing sequences of up to 40,960 encoded characters, such as letters, digits, symbols, spaces, control characters, or other Unicode characters and code points. Strings allow scripts to represent a wide range of data as character patterns and human-readable text.

Scripts written in Pine use strings for several purposes, such as defining titles, expressing symbol and timeframe information, setting the contexts of data requests, creating alert and debug messages, and displaying text on the chart. The specialized functions in the `str` namespace provide convenient ways to construct strings, create modified copies of other strings, and inspect or extract substrings.

This page explains how Pine strings work, and how to construct, inspect, and modify strings using the available `str.*()` functions.

> NoteWe use the format *"U+XXXX"* throughout our documentation when referring to characters and code points in the Unicode Standard.

Literal strings

Literal strings are constant "string" values whose character sequences are defined literally in the source code. In Pine Script, programmers can define literal strings using single-line or multiline syntax.

Single-line strings

A single-line string is a literal character sequence enclosed by a single pair of ASCII quotation marks (") or apostrophes ('). All parts of the string's definition, including the enclosing characters, occupy a *single* line of code.

For example, the following code snippet declares two variables that store equivalent single-line strings containing the text `Hello world!`:

//@variable A literal string containing `Hello world!`. Uses the `"` character as the enclosing delimiter. string hello1 = "Hello world!" //@variable A literal string containing `Hello world!`. Uses the `'` character as the enclosing delimiter. string hello2 = 'Hello world!'

The " or ' enclosing delimiters in a single-line string definition are *not* parts of the specified character sequence. They only mark the start and end of the string in the code. These characters *do not* appear in outputs that display text from strings, such as Pine Logs or various drawings.

This example calls the log.info() function on the first bar to display the contents of the literal value `"Hello world!"` in the Pine Logs pane. The message in the pane displays the `Hello world!` text only, without the " characters:

//@version=6 indicator("Single-line strings demo") // The script's displayed title does not include the quotation marks. if barstate.isfirst // Log `Hello world!` on the first bar. The logged text does not include `"`

characters. log.info("Hello world!")

Note that:

- The script also uses a single-line literal string to specify the `title` argument of the indicator() declaration statement.
- Only the `"` and `'` *ASCII* characters are valid enclosing characters for literal strings. Other Unicode characters, such as U+FF02 (Fullwidth Quotation Mark), are not allowed in enclosing delimiters.
- The timestamp in square brackets (`[` and `]`) at the start of the logged message is an *automatic prefix* showing the log's time in the chart's time zone. For more information, refer to the Pine Logs section of the Debugging page.

A single-line string definition can also use the enclosing `"` or `'` character within its character sequence, but only if it includes a single backslash (`\`) before each instance of the character to mark it as a *literal* character instead of a string boundary.

For example, the following script version logs the text `"Hello world!"` in the Pine Logs pane, with quotation marks included, because the string definition uses `\"` to insert literal quotation marks:

//@version=6 indicator("Quotes in single-line strings demo") if barstate.isfirst // Log `"Hello world!"` on the first bar. // The `\"` sequences in the defined string insert literal `"` characters. log.info("\"Hello world!\"")

See the Escape sequences section below to learn more about the behavior of the `\` character in literal strings.

> Notice
>
> In Pine v6, programmers can use *line wrapping* to define single-line literal strings across multiple code lines, where each wrapped line has an indentation of one or more spaces. However, the resulting character sequence adds only **one** space to the start of the wrapped lines, and it does *not* automatically add line terminators.
>
> This line-wrapping behavior is *deprecated*; future versions of Pine Script will not support it. Therefore, rather than defining a single-line string across multiple code lines, we recommend either of the following:
>
> - Splitting the string into smaller strings, then concatenating those strings using line-wrapped + operations. See the Line wrapping section of our Style guide for an example.
> - Defining a multiline string instead. See the next section to learn more.

Multiline strings

A multiline string is a literal character sequence enclosed by *three* pairs of ASCII quotation marks (e.g. `"""..."""`) or apostrophes (e.g., `'''...'''`). This syntax offers convenience for defining literal strings that represent *multiline text*. All parts of a multiline string definition between the enclosing `"""` or `'''` delimiters can occupy *separate* code lines and use *any* amount of indentation. The definition automatically adds the *newline* control character (U+000A) before each new line to insert a line terminator into the resulting string's sequence.

> NoteAlthough a multiline string can occupy multiple visible code lines in the Pine Editor, it is still considered part of a *single-line* expression, similar to other types of line-wrapped code.

For example, the following script defines a multiline string representing the text `Hello` and `world!` on separate text lines, then displays the result in a label on the last historical bar. The string automatically includes *three* newline characters – one before `Hello`, one before `world!`, and one before the end – because those parts of the sequence occupy *separate lines* in the string's definition:

//@version=6 indicator("Multiline strings demo", overlay = true) //@variable A string representing the text `Hello` and `world!` on separate text lines. // This multiline string is equivalent to the single-line string `"\nHello\nworld!\n"` string helloStr = """ Hello world! """ // Create a label to display the string's text on the last historical bar. if barstate.islastconfirmedhistory label.new(bar_index + 10, close, helloStr, style = label.style_label_center, size = 30, textalign = text.align_left)

Note that:

- The enclosing characters of a multiline string cannot contain any other characters between them. For instance, changing the enclosing """ delimiter to " " " in this example causes a *compilation error*.
- Single-line strings can also represent multiline text if their definitions explicitly insert newline characters with the `\n` escape sequence. For example, the single-line string `"\nHello\nworld!\n"` is equivalent to the multiline string defined in this code.

Multiline string definitions treat *all* code between the enclosing """ or ''' delimiters, including any space characters used for indentation, as *literal text*. If the separate lines in a multiline string definition contain leading spaces, the lines of text represented by the string include those spaces as well.

For example, the script below defines a multiline literal string with different numbers of leading spaces on each line, then displays the result in a label on the last historical bar. As shown below, the displayed lines of text preserve all the leading spaces included in the string definition:

//@version=6 indicator("Indentation in multiline strings demo") //@variable A string representing multiline text indented by spaces. // This multiline string is equivalent to the following single-line string: // `"\n0 leading spaces\n 1 leading space\n 4 leading spaces\n 8 leading spaces\n"` string indentedText = """ 0 leading spaces 1 leading space 4 leading spaces 8 leading spaces """ // Create a label to display the string's text on the last historical bar. if barstate.islastconfirmedhistory label.new(bar_index, 0, indentedText, style = label.style_label_center, size = 30, textalign = text.align_left)

This same behavior also applies when defining multiline strings within indented code blocks or line-wrapped expressions. All lines in a multiline string definition after the first include *every* leading space, starting from *column 0* in the Pine Editor. They do *not* include only the spaces starting from the indentation level of the enclosing code.

For example, the following script defines a multiline string *inside* an if structure and displays the string's text in a label. Each new line in the string definition starts with four leading spaces to align with the indentation of the structure's local block. The string definition treats these leading spaces as *literal characters*, not as syntax to indicate the local block. Therefore, the text in the label includes four-space indentation for all lines after the first:

//@version=6 indicator("Multiline strings in local blocks demo") if barstate.islastconfirmedhistory //@variable A multiline string defined in a local block. // The string does not exclude the spaces used for block indentation. // Consequently, each subseqent text line in the string starts with four spaces. string localMultiStr = """This line is not indented. This line has four leading spaces. This line has four additional leading spaces.""" // Draw a label to display the string's defined text. label.new(bar_index, 0, localMultiStr, style = label.style_label_center, size = 30, textalign = text.align_left)

Multiline string definitions can include the same characters as those that define the enclosing delimiters (""" or ''') directly in their character sequences. Unlike a single-line string definition, which must prefix the " or ' character with a backslash (\) to include it literally if it matches the enclosing characters, a multiline string definition does *not* require a backslash to include the " or ' character if:

- The character does not occur three consecutive times in the string's sequence without other characters separating the occurrences.
- There is at least one other character or line break between the last " or ' character and the end of the string.

For example, the following code block demonstrates equivalent literal strings that represent quoted text. The first string uses single-line syntax, and the second uses multiline syntax. The single-line syntax requires a backslash before each " character because it also uses that character for its enclosing delimiters. By contrast, the multiline syntax does not require a backslash before each " character in this example, because no part of the defined character sequence literally matches the enclosing """ delimiters:

//@variable A literal string, with quotes, defined using single-line syntax. // This example requires `\"` to include `"` in the string, // because that character matches the enclosing delimiters. string quoted1 = " \"Some quoted text\" " //@variable An equivalent literal string defined using multiline syntax. // Although the definition uses `"` characters in the enclosing delimiters, using `\"` is *optional*, // because a single `"` in the string does not match the enclosing `"""` sequences. string quoted2 = """ "Some quoted text" """

Escape sequences

The backslash character (\), also known as the Reverse Solidus in Unicode (U+005C), is an *escape character* in Pine strings. This character forms an *escape sequence* when it precedes another character, signaling that the following character has a potentially *different* meaning than usual.

Characters with a special meaning in "string" value definitions, such as quotation marks and backslashes, become *literal* characters when prefixed by a backslash (e.g., \\ includes a single \ in the character sequence).

This simple script declares a variable with an assigned literal "string" value enclosed in apostrophes (') and displays the value's contents in the Pine Logs pane. It uses the \ character to escape an extra apostrophe and another backslash, making them literal characters in the displayed text:

//@version=6 indicator("Escaping special characters demo") //@variable A string containing escaped `\` and `'` characters. string displayText = 'The backslash (\\) can change another character\'s meaning in Pine strings.' if barstate.isfirst log.info(displayText)

Note that:

- This example must prefix the `'` character with a backslash in the string's sequence because it also uses that character to mark its start and end boundaries. Without the backslash, it causes a *compilation error*. The script does not need to escape the apostrophe if we change the literal string's enclosing characters to quotation marks (").

The ASCII characters `n` and `t` usually have a literal meaning in Pine strings. However, when prefixed by the backslash character, they form escape sequences representing *control characters*. The `\n` sequence represents the newline character (U+000A), a line terminator for multiline text. The `\t` sequence represents the horizontal tab character (U+0009), which is helpful for indentation.

The script below creates a "string" value with multiline text on a single line of code, which it displays in a label on the last historical bar. The defined value contains several `\n` and `\t` escape sequences to include line terminators and tab spaces in the displayed text:

//@version=6 indicator("Control characters demo", overlay = true) if barstate.islastconfirmedhistory //@variable A string containing `\n` and `\t` escape sequences. Renders as multiline text with indentation. string displayText = "\This\n\tis\n\t\tmultiline\n\t\t\ttext\n\t\t\t\twith\n\t\t\t\t\ttab spaces." // Draw a label showing the `displayText` at the bar's `high`. label.new(bar_index, high, displayText, style = label.style_label_left, size = 24, textalign = text.align_left)

Note that:

- The "string" value also includes `\` before the `T` character. However, that character still appears *literally* in the displayed text. If a backslash applied to a character does not form a supported escape sequence, the character's meaning *does not change*.

Concatenation

The + and += operators signify *concatenation* when the operands are strings. A concatenation operation appends the second operand's character sequence to the first operand's sequence to form a new, *combined* "string" value.

For example, this script declares a `concatString` variable that holds the result of a concatenation operation. After declaring the variable, it uses the += operator to concatenate additional strings and reassign the variable's value. Then, the script calls log.info() to show the result in the Pine Logs pane:

//@version=6 indicator("Concatenation demo", overlay = true) if barstate.isfirst // Declare two variables that hold concatenated literal strings. string value1 = "\n\nThis 'string' is " + "the result " string value2 = "of multiple " + "concatenation operations." //@variable A string produced by concatenating `value1`, `value2`, and four literal strings. string displayText = value1 + value2 + "\n" // Use two concatenation assignments to create new strings from `displayText` and literal strings. displayText += "\nEach operation creates a new 'string' in memory that" + " combines the character sequences of the operands" displayText += " without modifying the original values." // Log the `displayText` in the Pine Logs pane. log.info(displayText)

Note that:

- Strings are immutable and cannot change. Therefore, every concatenation operation creates a *new* "string" value in memory. The operation does **not** modify either "string" operand directly.
- Another, more advanced way to combine strings is to collect them inside an array and use the array.join() function. For more information, see the Joining section of the Arrays page.

- In many cases, programmers can efficiently create *formatted strings* with str.format() instead of combining individual strings with concatenation or joining. See the Formatting strings section to learn more.

String conversion and formatting

Programmers can use strings to represent data of virtually any type as human-readable character sequences. Converting data to strings allows scripts to perform many helpful tasks, including:

- Displaying dynamic prices and calculations as text inside labels, tables, or boxes.
- Creating alert messages containing realtime market and indicator information.
- Logging debug messages containing calculated script information in the Pine Logs pane.
- Performing custom calculations and logic, such as constructing symbol or timeframe strings for data requests.

Converting values to strings

The simplest way to convert data to strings is to call the str.tostring() function. The function can represent values of several types as strings, based on predefined or custom formats. It has the following two signatures:

```
str.tostring(value) → string
str.tostring(value, format) → string
```

The function's `value` parameter accepts any "int", "float", "bool", or "string" value; the reference of an array or matrix containing values of these types; or a member of an enum type.

For example, this line of code creates a string representing the "float" value `123.456`, with default formatting. The result is usable in "string" operations and any script outputs that display *dynamic* text, such as labels and tables:

//@variable Holds the "string" value `"123.456"`. string numString = str.tostring(123.456)

The str.tostring() function's `format` parameter determines the *numeric format* for converted "int" and "float" values, arrays, and matrices. It can use one of the following `format.*` constants: format.mintick, format.percent, or format.volume. Alternatively, programmers can use strings containing `#` (number sign), `0` (zero), `.` (period), `,` (comma), and `%` (percent sign) tokens for customized *formatting patterns* with specific decimal precision. The default numeric format is `"#.########"`, which rounds fractional digits to eight decimal places without trailing zeros.

The script below uses the str.tostring() function to convert numeric values, a "bool" value, arrays, and a matrix into strings and displays the results in a table on the last bar. The str.tostring() calls that convert numeric values and collections contain different `format` arguments to demonstrate how various formatting patterns affect the results:

//@version=6 indicator("String conversion demo") //@variable A 2-row, 15-column table showing "string" representations of numbers, "bool" values, and collections. var table displayTable = table.new(position.middle_center, 2, 15, frame_color = chart.fg_color, frame_width = 1) //@function Initializes a row to show two specified strings in the display table. makeRow(int row, string str0, string str1)

=> color bgColor = row == 0 ? chart.fg_color : chart.bg_color color textColor = row == 0 ? chart.bg_color : chart.fg_color displayTable.cell(0, row, str0, text_color = textColor, text_halign = text.align_left, bgcolor = bgColor) displayTable.cell(1, row, str1, text_color = textColor, text_halign = text.align_left, bgcolor = bgColor) // Initialize the header row for the `displayTable` on the first bar. if barstate.isfirst makeRow(0, "Variable", "'string' value") // Compute several "string" conversions and populate the `displayTable` with the results on the last bar. if barstate.islast //@variable Represents `ta.vwap` with the default numeric format (`#.########`). string numberRepr1 = str.tostring(ta.vwap) //@variable Represents `ta.vwap` rounded to the minimum tick without trailing zeros. string numberRepr2 = str.tostring(ta.vwap, format.mintick) //@variable Represents `ta.vwap` rounded to three fractional digits without trailing zeros. string numberRepr3 = str.tostring(ta.vwap, "#.###") //@variable Represents `ta.vwap` rounded to five fractional digits with trailing zeros. string numberRepr4 = str.tostring(ta.vwap, "0.00000") //@variable Represents the `ta.vwap` as an integer. string numberRepr5 = str.tostring(ta.vwap, "#") //@variable Represents `ta.vwap` as an integer with leading zeros. string numberRepr6 = str.tostring(ta.vwap, "000000") //@variable Represents `ta.vwap` with commas for each third whole digit from the decimal point. // Values less than 1000000 include leading zeros. The fractional part includes up to two digits. string numberRepr7 = str.tostring(ta.vwap, "0000,000.##") //@variable Represents `100 * ta.tr / close` rounded to two fractional digits with `%` at the end. string numberRepr8 = str.tostring(100 * ta.tr / close, format.percent) //@variable Represents `ta.tr / close` as a percentage rounded to four fractional digits. // With `%` at the end of the `format`, the function multiplies the represented number by 100. string numberRepr9 = str.tostring(ta.tr / close, "#.####%") //@variable Represents `volume` with fixed precision and letter characters for large figures. // `K` means "thousand", `M` means "million", `B` means "billion", and `T` means "trillion". string numberRepr10 = str.tostring(volume, format.volume) //@variable Represents a "bool" value. Is `"true"` when `ta.tr` exceeds the bar's range, `"false"` otherwise. string boolRepr = str.tostring(ta.tr > high - low) // Create an array and matrix of price values to convert into strings. array<float> pricesArray = array.from(open, close, low, high) matrix<float> pricesMatrix = matrix.new<float>(), pricesMatrix.add_row(0, pricesArray), pricesMatrix.reshape(2, 2) //@variable Represents the `pricesArray` values with up to two decimal places and no trailing zeros for each element. // Contains `[` and `]` characters to mark the start and end of the array's contents. string numberArrayRepr = str.tostring(pricesArray, "#.##") //@variable Represents the `pricesMatrix` values with four decimal places and trailing zeros for each element. // Contains `[` and `]` characters to mark the start and end of each row. string numberMatrixRepr = str.tostring(pricesMatrix, "#.0000") //@variable Represents a "string" array containing the symbol's type and currency. string stringArrayRepr = str.tostring(array.from(syminfo.type, syminfo.currency)) // Populate the `displayTable` rows with each of the above variable names and their assigned "string" values. makeRow(1, "numberRepr1", numberRepr1) makeRow(2, "numberRepr2", numberRepr2) makeRow(3, "numberRepr3", numberRepr3) makeRow(4, "numberRepr4", numberRepr4) makeRow(5, "numberRepr5", numberRepr5) makeRow(6, "numberRepr6", numberRepr6) makeRow(7, "numberRepr7", numberRepr7) makeRow(8, "numberRepr8", numberRepr8) makeRow(9, "numberRepr9", numberRepr9) makeRow(10, "numberRepr10", numberRepr10) makeRow(11, "boolRepr", boolRepr) makeRow(12, "numberArrayRepr", numberArrayRepr) makeRow(13, "numberMatrixRepr", numberMatrixRepr) makeRow(14, "stringArrayRepr", stringArrayRepr)

Note that:

- The `#` and `0` tokens control the digits in the represented numbers in a similar way, but with different behaviors for leading and trailing *zeros*, as shown above. The `0` token *always* includes a digit at the specified decimal place, even for a leading or trailing zero, whereas the `#` token allows a leading or trailing digit only if it is *nonzero*.
- The `format` argument requires a `#` or `0` token for each *fractional* digit in a converted number. These tokens are optional for extra *whole* digits, because str.tostring() includes the necessary digits

automatically.

- A single `,` token adds *repeated* comma separation to whole digits. In the str.tostring() call with the format `"0000,000.##"`, the token specifies that the result includes a dividing comma for every set of *three digits* to the left of the decimal point.
- When the `%` token is at the *end* of the formatting string, the representation multiplies numbers by 100 to express them as percentages, as shown by the example that uses `"#.####%"`.

> TipScripts can convert a numeric “string” value back to a “float” value with str.tonumber(). When converting strings to numbers with this function, the value’s character sequence can include only ASCII digits, a sign symbol at the beginning (`+` or `–`), and a single period for the decimal point (`.`). If the specified “string” value does not represent a number with this format, the function returns na. For example, the function can convert `"1234.50"` to a “float” value, but it cannot convert strings such as `"$1,234.50"`.

Formatting strings

The str.format() function can combine multiple “int”, “float”, “bool”, “string”, or array arguments into one output string in a specified format. Using this function is a simpler alternative to creating multiple separate strings and combining them with repeated concatenation operations. Below is the function’s signature:

```
str.format(formatString, arg0, arg1, ...) → string
```

> NoteThe second overloads of all `log.*()` functions have the *same* parameter signature and formatting behaviors as str.format(). However, they generate *logs* with formatted text instead of returning usable strings. See the Pine Logs section of the Debugging page to learn more about these functions.

The `formatString` parameter accepts a “string” value that defines the *format* of the returned string, where the *placeholders* in curly brackets (`{}`) refer to the function call’s *additional arguments*. The placeholder `"{0}"` represents the first additional argument `arg0`, `"{1}"` represents `arg1`, and so on. The function *replaces* each placeholder in the `formatString` with a string representation of the corresponding argument. For instance, the call `str.format("The timeframe multiplier is {0}", timeframe.multiplier)` on a 1D chart returns `"The timeframe multiplier is 1"`.

The following example constructs a formatted string containing various bar information, then displays the result in a label at the bar’s high. The str.format() call’s `formatString` argument includes placeholders for 10 values, where each placeholder’s *number* corresponds to one of the additional “string”, “bool”, “int”, or “float” arguments:

//@version=6 indicator("Formatting strings demo", overlay = true) //@variable Counts script executions in a bar. Does not roll back because it is declared with `varip`. varip int exCount = 0 exCount += 1 //@variable Is `true` when the `close` exceeds the current `open` and the previous `close`; `false` otherwise. bool rising = close > math.max(nz(close[1]), open) //@variable Is `"realtime"` for all new bars after the script's historical executions; `"historical"` otherwise. string barState = barstate.isrealtime ? "realtime" : "historical" //@variable A formatting string containing placeholders for 10 values. string formatString = "Bar: {1} ({9}){0} {0}Executions: {2}{0}O: {3}{0}H: {4}{0}L: {5}{0}C: {6}{0}V: {7}{0}Rising: {8}" //@variable A multiline string

containing formatted information for the current bar. string formattedString = str.format(formatString, "\n\t", bar_index, exCount - nz(exCount[1]), open, high, low, close, volume, rising, barState) // Draw a label displaying the `formattedString` at the bar's `high`. label.new(bar_index, high, formattedString, size = 18, textalign = text.align_left)

Note that:

- The `formatString` argument can use placeholders in *any order* and can repeat specific placeholders *more than once*. The format in this example uses `{0}` multiple times to insert the first argument (`"\n\t"`) to create multiline text with indentation.
- If a placeholder refers to a nonexistent argument, the formatted result treats that placeholder as a *literal* character sequence. For instance, a placeholder such as `{20}` in the `formatString` argument above includes those characters literally in the formatted result, because the str.format() call *does not* contain 21 additional arguments.
- *Non-quoted* left curly brackets (`{`) must have corresponding right curly brackets (`}`) inside formatting strings. If a `formatString` contains unbalanced curly brackets, it causes a *runtime error*.

It's important to note that the apostrophe (`'`) acts as a *quote character* inside formatting strings. When a formatting string contains a character sequence between two apostrophes, the formatted result includes that sequence directly, without treating the characters as placeholders or formatting tokens. This behavior applies even if the formatting string prefixes apostrophes with the backslash character (`\`). The enclosing apostrophes for a non-empty quoted sequence are **not** part of the formatted string. To include literal apostrophes in a str.format() call's result, pass a "string" value containing the character as an *extra argument*, then use that argument's *placeholder* in the specified `formatString`. Alternatively, use pairs of apostrophes with no characters between them directly in the `formatString` (e.g., `''` adds a single `'` character in the result).

The example below demonstrates how using apostrophes directly in formatting strings differs from inserting them via placeholders. The script uses the `'` character directly in the str.format() call's `formatString` to define a *quoted sequence*, and it uses the `{1}` placeholder to insert the character from an extra argument without creating a quoted sequence. The script displays the resulting `formattedString` value in a single-cell table on the first bar:

//@version=6 indicator("Quotes in formatting strings demo") if barstate.isfirst //@variable A concatenated formatting string that contains apostrophes and placeholders. string formatString = "Apostrophes in formatting strings signify quoted character sequences, which do not have special meaning." + "\n\nQuoting a placeholder includes its characters literally: '{0}'" + "\n\nInserting the apostrophe character from arguments does *not* create quoted sequences: {1}{0}{1}" //@variable A formatted string showing how using `'` directly differs from insertion via arguments. string formattedString = str.format(formatString, ticker.standard(), "'") //@variable A single-cell table to display the `formattedString`. table display = table.new(position.middle_center, 1, 1, color.purple) // Initialize the cell with white, left-aligned text. display.cell(0, 0, formattedString, text_color = color.white, text_halign = text.align_left, text_size = 20)

When a str.format() call contains "int" or "float" arguments, the placeholders for those arguments in the `formatString` can include the `number` *modifier* followed by a *formatting pattern* for customized numeric formats (e.g., `"{0,number,#.000}"`).

The possible numeric formatting patterns are similar to those for the `format` parameter of str.tostring(). They can contain `#`, `0`, and `.` tokens to specify decimal precision; use the `,` token for comma separation; and

include `%` at the end for percentage conversion. Alternatively, a placeholder can use one of the following *keywords* that specify *predefined* formatting patterns: `integer`, `currency`, or `percent`.

The script below demonstrates how different numeric formats in a `formatString` placeholder affect the formatted representation of a "float" value. On the last bar, the script generates a pseudorandom value between 0 and 10000 with math.random(), uses several str.format() calls to format the value in different ways, and displays the results in a table:

```
//@version=6 indicator("Numeric formatting demo") //@variable A two-row, eight-column table showing a pseudorandom number formatted with different format patterns. var table display = table.new(position.middle_center, 2, 8, frame_color = chart.fg_color, frame_width = 1) //@variable Initializes a row to show two specified strings in the `display` table. makeRow(int row, string str0, string str1) => color bgColor = row == 0 ? chart.fg_color : chart.bg_color color textColor = row == 0 ? chart.bg_color : chart.fg_color display.cell(0, row, str0, text_color = textColor, text_halign = text.align_left, text_size = 30, bgcolor = bgColor) display.cell(1, row, str1, text_color = textColor, text_halign = text.align_left, text_size = 30, bgcolor = bgColor) // Initialize the header row for the `display` table on the first bar. if barstate.isfirst makeRow(0, "Format pattern", "Result") if barstate.islast //@variable A pseudorandom value between 0 and 10000. float value = math.random(0, 10000) //@variable Represents the `value` using the `str.format()` function's default numeric format (`#,###.###`). // This format uses comma-separated sets of three whole digits, and allows up to three fractional digits. // This default differs from the default for `str.tostring()` (`#.########`). string default = str.format("{0}", value) //@variable Represents the `value` using the `integer` preset (`#,###`). // This format rounds the `value` to the nearest whole and adds comma separation for three-digit sets. string integerPreset = str.format("{0,number,integer}", value) //@variable Represents the `value` as an integer without comma-separated digits (`#`). string integerNoComma = str.format("{0,number,#}", value) //@variable Represents the `value` using the `currency` preset (`'$'#,###.00`). // This format prefixes the result with the `$` symbol, adds comma separation for sets of three whole // digits, and includes two fractional digits. string currencyPreset = str.format("{0,number,currency}", value) //@variable Represents the `value` in dollars with comma-separated whole digits and four fractional digits (`'$',###.0000`). string currencyCustom = str.format("{0,number,'$',###.0000}", value) //@variable Represents the `value` using the `percent` preset (`#,###%`). // This format multiplies the `value` by 100, rounds the result to the nearest whole number, adds comma // separation for three-digit sets, and includes the `%` symbol at the end. string percentPreset = str.format("{0,number,percent}", value) //@variable Represents the `value` as a percentage with comma-separated whole digits and up to three fractional digits. string percentCustom = str.format("{0,number,#,###.###%}", value) // Initialize rows showing each numeric format and the resulting representation of the `value`. makeRow(1, "Default (#,###.###)", default) makeRow(2, "integer (#,###)", integerPreset) makeRow(3, "integer without commas (#)", integerNoComma) makeRow(4, "currency ('$'#,###.00)", currencyPreset) makeRow(5, "currency with 4 fractional digits ('$',###.0000)", currencyCustom) makeRow(6, "percent (#,###%)", percentPreset) makeRow(7, "percent with up to 3 fractional digits (#,###.###%)", percentCustom)
```

Note that:

- In contrast to str.tostring(), the str.format() function does not directly support the preset formats defined by the `format.*` constants (format.mintick, format.percent, and format.volume). To use those formats on numeric values in a formatted string, convert the values with str.tostring() first, then use the resulting strings in the str.format() call.

The str.format() function's `formatString` also supports placeholders with the `date` or `time` modifier, which can format an "int" UNIX timestamp into a *UTC* date or time. For example, this line of code creates a string

representing the current bar's opening timestamp as a date and time in the ISO 8601 standard format:

string formattedTime = str.format("{0,date,yyyy-MM-dd}T{0,time,HH:mm:ssZ}", time)

However, str.format() **cannot** express dates and times in other time zones. It uses *UTC+0* exclusively. The specialized str.format_time() function is more optimal for constructing date-time strings, because it can express dates and times in *any time zone*. See the Formatting dates and times section of the Time page to learn more about this function and the available formatting tokens.

Custom representations

All built-in functions that create "string" values to represent data support a limited subset of built-in types. They *do not* support "color" values or objects of most *reference types* (e.g., labels). Programmers can, however, use custom logic and formatting to create "string" representations of data that the str.tostring() or str.format() functions cannot express as strings directly.

For example, this script demonstrates two ways to represent a "color" value as a string based on its red, green, blue, and transparency components. The first method formats the color components directly. The second calculates and formats each component's *hexadecimal* form. The script displays the results of both custom formats in a label on the last historical bar.

After creating the label object, the script also uses log.info() to create formatted text containing the label's `x`, `y`, and `text` *properties* and display the result in the Pine Logs pane:

//@version=6 indicator("Custom representations demo", overlay = true) //@variable The "color" value to color the label and convert to a string. color colorInput = input.color(#00897b) //@function Constructs a formatted string containing the color's R, G, B, and T components. rgbtString(color value) => str.format("color (R: {0}, G: {1}, B: {2}, T: {3})", color.r(value), color.g(value), color.b(value), color.t(value)) //@variable Constructs a string containing the color's hexadecimal RGBA representation. hexString(color value) => //@variable An array of hexadecimal characters formed by splitting a string. var array<string> chars = str.split("0123456789abcdef", "") // Get the R, G, and B channel values from the `value`. int r = int(color.r(value)) int g = int(color.g(value)) int b = int(color.b(value)) //@variable The A (alpha) channel value (opposite of transparency), scaled to the range [0, 255]. int a = int((100 - color.t(value)) * 255 / 100) //@variable A formatted string combining hex codes for the R, G, B, and A channels. string result = str.format("#{0}{1}{2}{3}{4}{5}{6}{7}", chars.get(int(r / 16)), chars.get(r % 16), chars.get(int(g / 16)), chars.get(g % 16), chars.get(int(b / 16)), chars.get(b % 16), chars.get(int(a / 16)), chars.get(a % 16)) if barstate.islastconfirmedhistory //@variable A formatted string containing the results of `rgbtString()` and `hexString()`. string labelText = str.format("{0}\n{1}", rgbtString(colorInput), hexString(colorInput)) //@variable A label displaying the `labelText`. The script displays this object's properties in the Pine Logs pane. label displayLabel = label.new(bar_index, high, labelText, color = colorInput, textcolor = color.white, size = 36) // Log a custom representation of the `displayLabel` in the Pine Logs pane. log.info("\nlabel object\nx: {0,number,#}\ny: {1,number,#.#####}\ntext: {2}", displayLabel.get_x(), displayLabel.get_y(), displayLabel.get_text())

Note that:

- Not all special types have retrievable properties. For instance, scripts cannot retrieve information from polylines or tables. To create strings for these types, track the data used in their creation with separate variables, then format the values of those variables into strings.

- For an example of creating strings from the field values of user-defined types, see the Debugging objects of UDTs section of the Debugging page.

Modifying strings

Several `str.*()` functions provide simplified ways to modify the character sequence from a "string" value, including str.replace(), str.replace_all(), str.upper(), str.lower(), str.trim(), and str.repeat().

Programmers can use these functions to create copies of strings with replaced character sequences, modified letter cases, trimmed whitespaces, or repeated character patterns.

Replacing substrings

The str.replace() function searches a specified `source` string for the nth *non-overlapping* occurrence of a given substring, then returns a copy of the original string containing a specified replacement at that substring's position.

The str.replace_all() function searches the `source` for *every* non-overlapping occurrence of the substring and replaces each one in its returned value.

Below are the functions' signatures:

```
str.replace(source, target, replacement, occurrence) → string
str.replace_all(source, target, replacement) → string
```

Where:

- `source` is the "string" value containing the substrings to replace with a specified `replacement`.
- `target` is the substring replaced by the `replacement` in the returned copy. If the `source` value does not contain the substring, the function returns a copy of the value without modification.
- `replacement` is the substring inserted in place of the required `target` occurrences in the result.
- The `occurrence` parameter for str.replace() specifies which non-overlapping occurrence of the `target` is swapped for the `replacement` in the result. The default value is 0, meaning the function replaces the *first* occurrence of the `target`. If the specified occurrence does not exist in the `source` value, the function returns a copy of the value without modification.

The following script demonstrates the effects of str.replace() and str.replace_all() calls on a string containing the sequence `Hello world!`. Additionally, it calls these functions to define the `formatString` value for a str.format() call, which formats all the replacement results into a single "string" value. The script displays the formatted text inside a label anchored to the latest bar's opening time:

//@version=6 indicator("Replacing substrings demo") if barstate.isfirst //@variable A string containing the sequence `Hello world!`. string originalString = "Hello world!" //@variable A copy of the `originalString` with `!` replaced by `!!!`. string changePunctuation = str.replace(originalString, "!", "!!!") //@variable A copy of the `originalString` with the second `o` replaced by `0`. string changeLetter = str.replace(originalString, "o", "0", 1) //@variable A copy of the `originalString` with all `l` characters replaced by 1. string changeLetters = str.replace_all(originalString, "l", "1") //@variable A copy of the `originalString` with all zero-width boundaries replaced by `\n-`. string insertNewlines = str.replace_all(originalString, "", "\n") //@variable A copy of the `originalString` without changes, as `H` does not occur two times. string unchanged =

str.replace(originalString, "H ", "_", 1) //@variable A formatting string with the following initial structure: `{0}\n{1}\n{2}\n{3}\n{4}\n{5}`. // The script creates the string by replacing zero-width boundaries in the sequence `012345` with // `}\n{` using `str.replace_all()`, then removing the extra `}\n` and `\n{` from the start and end of the // call's result with two additional `str.replace()` calls. string formatString = str.replace(str.replace(str.replace_all("012345", "", "}\n{"), "}\n", ""), "\n{", "", 5) // Create a copy of the `formatString` with the first `}` replaced by `}\n----------`, then reassign // the variable to use that copy. formatString := str.replace(formatString, "}", "}\n----------") //@variable A formatted string containing the original and modified "Hello world!" strings. string displayText = str.format(formatString, originalString, changePunctuation, changeLetter, changeLetters, insertNewlines, unchanged) // Draw a label anchored to the latest bar's opening time to show the `displayText`. label.new(math.max(last_bar_time, chart.right_visible_bar_time), 0, displayText, xloc.bar_time, style = label.style_label_center, size = 20)

Note that:

- Each `str.replace*()` call creates an independent, modified copy of the specified `source` value. Because each modification of the `originalString` is assigned to a separate variable, each value *does not* contain changes from previous `str.replace*()` calls.
- The `str.replace*()` functions can replace zero-width *boundaries* when the `target` is an empty string, as shown by the `formatString` declaration. The str.replace_all() call inserts `}\n{` around every character in the literal string `"012345"`.

> TipWhen using str.replace(), knowing the precise number of substrings within the `source` value helps ensure correct results. A simple way to count substrings is to remove them with str.replace_all(), then measure the difference in length relative to the length of the removed substring. See the Counting characters and substrings section to learn more.

Changing case

The str.upper() and str.lower() functions create a copy of a `source` string with all ASCII letter characters converted to *uppercase* or *lowercase* variants, providing a convenient alternative to replacing specific characters with several `str.replace*()` calls. The str.upper() function replaces all lowercase characters with uppercase characters, and str.lower() does the opposite. These are the functions' signatures:

```
str.upper(source) → string
str.lower(source) → string
```

This simple example demonstrates how these functions affect strings with *standard* letter characters. The script declares an `originalString` variable to hold a literal string, uses str.upper() on that variable to create a copied string with all letters converted to uppercase, then calls str.lower() to make a copy with only lowercase characters. It logs all three strings in the Pine Logs pane on the first bar:

//@version=6 indicator("Changing case demo", overlay = true) if barstate.isfirst //@variable A literal string containing the sequence `Hello World!` string originalString = "Hello World!" //@variable A copy of the `originalString` with all lowercase ASCII characters changed to uppercase characters. string uppercaseString = str.upper(originalString) //@variable A copy of the `uppercaseString` with all uppercase

ASCII characters changed to lowercase characters. string lowercaseString = str.lower(uppercaseString) // Log a formatted message containing all three values. log.info("\n\nOriginal: {0}\nUppercase: {1}\nLowercase: {2}", originalString, uppercaseString, lowercaseString)

Note that these functions can only change the cases of *ASCII* letter characters. They cannot convert other Unicode letters outside the ASCII range. For example, this script attempts to create uppercase and lowercase versions of a “string” value containing “Mathematical Sans-Serif” Unicode characters using str.upper() and str.lower(). As shown below, both function calls return *identical* copies of the value:

//@version=6 indicator("Non-ASCII case demo", overlay = true) if barstate.isfirst //@variable A literal string that uses Unicode characters in the "Mathematical Sans-Serif" family. string originalString = "This string is unaffected by the case-swapping functions!" // Call `str.upper()` and `str.lower()` to change the case of letters. // Although the characters in the `originalString` have the "Letter" property, they are not part of the standard // ASCII set. Consequently, these calls return unmodified strings. string uppercaseString = str.upper(originalString) string lowercaseString = str.lower(uppercaseString) // Log a formatted message containing all three values. log.info("\n\nOriginal: {0}\nUppercase: {1}\nLowercase: {2}", originalString, uppercaseString, lowercaseString)

Trimming whitespaces

The str.trim() function copies a `source` string and removes leading and trailing whitespace characters, including the standard space ( ), newline (`\n`), and tab space (`\t`). Below is the function’s signature:

```
str.trim(source) → string
```

This simple example demonstrates the str.trim() function’s behavior. The script creates a literal string containing different types of whitespaces at the start and end of the character sequence. Then, it uses str.trim() to create a new “string” value with those characters removed. The script formats both values into a single string, then displays the result in a label on the last historical bar:

//@version=6 indicator("Trimming whitespaces demo") if barstate.islastconfirmedhistory //@variable A literal string containing space, newline, and tab characters. string originalString = "\n\n\t\tABC DEF\t \n\n " //@variable A copy of the `originalString` that contains only `ABC DEF` without the other whitespaces. string trimmedString = str.trim(originalString) //@variable A formatted string containing the `originalString` and `trimmedString` values. string displayText = str.format("Original: \"{0}\"\n---\n\nTrimmed: \"{1}\"", originalString, trimmedString) // Draw a label to show the `displayText`. label.new(bar_index, 0, displayText, style = label.style_label_center, size = 30, textalign = text.align_left)

Note that:

- The str.trim() function removes only the ASCII whitespaces to the left of the *first* non-whitespace character and the right of the *last* non-whitespace character. It does **not** remove whitespaces between other characters.
- The formatting string in the str.format() call uses `\"` to include quotation marks around the `originalString` and `trimmedString` values in the displayed text. See the Escape sequences section above for more information.

The str.trim() function is particularly helpful when supplying calculated or input strings to built-in functions that process “string” arguments, because some function parameters require values *without* leading or trailing

whitespaces.

The following example creates an array of timeframe strings by splitting the value of a text area input based on its comma characters. Within a loop, the script uses each element from the array in a time() call to retrieve an opening time, then concatenates a formatted date and time with the `displayText`.

Although each item listed in the default string represents a valid timeframe, the time() call causes a runtime error. The script splits the value only by its commas, resulting in a *leading space* in each `timeframes` element after the first, and the time() function *does not* allow whitespaces in its `timeframe` argument.

If the user enables the input to trim the input string (which is off by default), the script uses str.trim() to remove surrounding whitespaces from the time() call's argument and prevent the formatting issue and the runtime error.

//@version=6 indicator("Invalid arguments with whitespaces demo", overlay = true) //@variable A string containing a comma-separated list of timeframes. string timeframesInput = input.text_area("1, 5, 15, 30, 60, 240, 720, 1D", "Timeframes") //@variable A boolean variable that defines whether to trim the input string. bool doTrimInput = input.bool(false, "Trim the \"Timeframes\" input string") //@variable An array of timeframe substrings. var array<string> timeframes = str.split(timeframesInput, ",") //@variable A concatenated string containing formatted dates and times for each value in the `timeframes` array. string displayText = "Opening times:" for timeframe in timeframes //@variable A copy of the `timeframe` string with surrounding whitespaces removed. string trimmedTimeframe = str.trim(timeframe) //@variable The UNIX timestamp of the bar's opening time on the timeframe. // This call causes a runtime error if the input string is not trimmed, because the argument contains // a leading whitespace (e.g., " 5"), which is an *unsupported format* for timeframe strings. int timestamp = time(doTrimInput ? trimmedTimeframe : timeframe) // Add a new text line containing the `timestamp` formatted as a date and time in the exchange time zone. displayText += "\n" + timeframe + str.format_time(timestamp, " yyyy-MM-dd, HH:mm:ss") // Draw a label at the bar's `high` with a tooltip showing the `displayText`. label.new(bar_index, high, style = label.style_label_down, tooltip = displayText)

Repeating sequences

The str.repeat() function creates a "string" value that *repeats* a `source` string's character sequence a specified number of times, providing a convenient way to construct strings with repetitive character patterns. Below is the function's signature:

```
str.repeat(source, repeat, separator) → string
```

Where:

- `source` is the "string" value containing the character sequence to repeat in the result.
- `repeat` is an "int" value specifying the number of times the function repeats the `source` sequence. If the value is 0, the function returns an empty string.
- `separator` is an optional "string" value containing a character sequence to *insert* between each repeated instance of the `source` sequence. The default value is an empty string, meaning the function repeats the `source` sequence *without* inserting additional characters.

The following script formats two numbers – the ohlc4 price and its Simple Moving Average – with a variable number of fractional digits. The minimum and maximum number of fractional digits are set by user inputs.

The script uses a str.repeat() call to repeat `0` characters to create a pattern for the *required* digits, and another call that repeats `#` characters to create a pattern for the *optional* digits, which are displayed only if they are nonzero. The script then concatenates these patterns into one pattern and uses that in a str.format() call to format the two numbers.

The script calls log.info() to log the constructed `formatString` on the first bar, and it displays the formatted results for each bar using labels:

//@version=6 indicator("Repeating sequences demo", overlay = true) //@variable The minimum number of fractional digits in the numeric strings. int minPrecisionInput = input.int(2, "Min precision", 0, 16) //@variable The maximum number of fractional digits in the numeric strings. int maxPrecisionInput = input.int(8, "Max precision", 0, 16) // Raise an error if the `maxPrecisionInput` is less than the `minPrecisionInput`. if maxPrecisionInput < minPrecisionInput runtime.error("The 'Max precision' cannot be less than the 'Min precision.") //@variable A string containing the `0` character repeated `minPrecisionInput` times. var string requiredDigits = str.repeat("0", minPrecisionInput) //@variable A string containing the `#` character repeated `maxPrecisionInput - minPrecisionInput` times. var string extraDigits = str.repeat("#", maxPrecisionInput - minPrecisionInput) //@variable A string representing a formatting pattern for numeric strings. // With default inputs, the value is `"0.00######"`, meaning two required digits and six nonzero extra digits. var string formatPattern = "0." + requiredDigits + extraDigits //@variable A formatting string that contains two placeholders for the `formatPattern` for numeric values. var string formatString = str.format("OHLC4: '{'0,number,{0}'}', MA: '{'1,number,{0}'}'", formatPattern) // Log the `formatString` value on the first bar. if barstate.isfirst log.info(formatString) // Draw a label to show the bar's formatted result. label.new(bar_index, high, str.format(formatString, ohlc4, ta.sma(ohlc4, 20)), size = 18)

Note that:

- The apostrophe (`'`) in the str.format() call serves as a *quote character*, **not** a literal character. The `formatString` uses the apostrophe to quote curly brackets (`{` and `}`), treating them as literal characters instead of direct placeholder markers.

The example below demonstrates a more creative use of str.repeat(). This script generates an ASCII art representation of the Pine Script logo using alternating sequences of repeated `.` (period) and `@` (at) characters. The user-defined `makeLine()` function calls str.repeat() seven times to create the repeated sequences, then formats their results into a single “string” value with a str.format() call. On the first bar, the script formats the results of several `makeLine()` calls into a multiline string and displays the result in a single-cell table in the chart’s top-right corner:

//@version=6 indicator("ASCII art from repeated sequences demo", overlay = true) //@function Creates a string containing alternating sequences of repeated `.` and `@` characters with `\n` at the end. // Each `dot*` argument defines a sequence of repeated `.` characters, and each `at*` argument defines a // sequence of repeated `@` characters. The function formats the repeated sequences from `str.repeat()` calls // into a single string, in the order of the parameters. // For example, `makeLine(6, 5, 4, 3, 2)` returns `".....@@@@@....@@@..\n"`. makeLine(int dot1 = 0, int at1 = 0, int dot2 = 0, int at2 = 0, int dot3 = 0, int at3 = 0, int dot4 = 0) => string result = str.format("{0}{1}{2}{3}{4}{5}{6}\n", str.repeat(".", dot1), str.repeat("@", at1), str.repeat(".", dot2), str.repeat("@", at2), str.repeat(".", dot3), str.repeat("@", at3), str.repeat(".", dot4)) result if barstate.isfirst //@variable A string representing the Pine logo using several lines of different repeated `.` and `@` sequences. string asciiArt = str.format("{0}{1}{2}{3}{4}{5}{6}{7}{8}{9}{10}{11}{12}{13}{14}{15}{16}{17}" + "{18}{19}{20}{21}{22}{23}{24}{25}{26}{27}{28}{29}{30}{31}{32}{33}", makeLine(80), makeLine(80), makeLine(38, 6, 36), makeLine(37, 8, 35), makeLine(35, 12, 33), makeLine(34, 15, 31),

makeLine(33, 7, 2, 7, 31), makeLine(31, 8, 4, 8, 29), makeLine(29, 8, 8, 8, 27), makeLine(28, 8, 10, 9, 25), makeLine(27, 7, 14, 8, 24), makeLine(25, 8, 17, 8, 22), makeLine(24, 8, 19, 8, 21), makeLine(23, 7, 22, 9, 19), makeLine(21, 17, 16, 9, 17), makeLine(20, 20, 16, 8, 16), makeLine(18, 25, 14, 9, 14), makeLine(18, 14, 3, 10, 13, 8, 14), makeLine(18, 8, 11, 10, 9, 10, 14), makeLine(39, 10, 4, 12, 15), makeLine(29, 6, 7, 20, 18), makeLine(24, 14, 6, 15, 21), makeLine(20, 21, 5, 10, 24), makeLine(18, 25, 5, 5, 5, 6, 16), makeLine(17, 28, 10, 10, 15), makeLine(16, 32, 4, 14, 14), makeLine(14, 54, 12), makeLine(13, 56, 11), makeLine(12, 58, 10), makeLine(10, 62, 8), makeLine(9, 64, 7), makeLine(9, 64, 7), makeLine(80), makeLine(80)) //@variable A single-cell table to display the `asciiArt` value. table t = table.new(position.top_right, 1, 1, frame_color = chart.fg_color, frame_width = 1) // Initialize a left-aligned cell with monospace font for proper alignment. t.cell(0, 0, asciiArt, text_color = chart.fg_color, text_halign = text.align_left, text_size = 6, text_font_family = font.family_monospace)

Note that:

- The table.cell() call uses text.align_left as the `text_halign` argument and font.family_monospace as the `text_font_family` argument to align the text lines to the *left* with relatively uniform character width.
- The formatted string from each `makeLine()` call uses the `\n` escape sequence at the end to add a line terminator.

String inspection and extraction

Several built-in `str.*()` functions allow scripts to measure a “string” value, check for substrings and retrieve their positions, split a string into several substrings, and extract substrings based on positions or match patterns. These functions include str.length(), str.contains(), str.startswith(), str.endswith(), str.split(), str.pos(), str.substring(), and str.match().

The sections below explain these functions and some helpful techniques to use them effectively.

Counting characters and substrings

The str.length() function measures the length of a specified “string” value, returning an “int” value representing the number of characters in the argument’s character sequence. It has the following signature:

```
str.length(string) → int
```

This function detects *every* character within a “string” value’s sequence, even those that are hard to see, such as leading or repeated spaces, line terminators, and invisible characters like U+200B (Zero Width Space).

For example, this simple script declares two variables with assigned literal strings and measures their length. The script creates the first “string” value using *Em Space* characters (U+2003), and creates the second using *En Space* characters (U+2002) instead. It measures the length of both strings with str.length(), creates modified strings with the "__" parts replaced by the length values, then concatenates the results for display in a single-cell table.

Although the two strings look identical in the output, their lengths differ because one En Space is equivalent to *half* the width of one Em Space, meaning the second string must include *two En Spaces* between each word to match the width of each Em Space in the first string:

```
//@version=6 indicator("Counting characters demo") if barstate.isfirst //@variable A single-cell table to display text. var table display = table.new(position.middle_center, 1, 1, color.teal) //@variable A literal string that uses one Em Space (U+2003) between each word. string testString1 = "This 'string' contains\n__ characters." //@variable A literal string that uses two En Spaces (U+2002) between each word. // An En Space is half the width of an Em Space, so this string contains three extra characters // to match the widths of each Em space in `testString1` and produce an identical output. string testString2 = "This  'string' contains\n__  characters." // Count the number of characters in `testString1` and `testString2`. int length1 = str.length(testString1) int length2 = str.length(testString2) // Replace the "__" in `testString1` and `testString2` with string representations of `length1` and `length2`. testString1 := str.replace(testString1, "__", str.tostring(length1)) testString2 := str.replace(testString2, "__", str.tostring(length2)) // Concatenate both strings and two newline characters, then display the result in the table's cell. string displayString = testString1 + "\n\n" + testString2 display.cell(0, 0, displayString, text_color = color.white, text_size = 50, text_halign = text.align_left)
```

Note that:

- A simple way to verify the added characters is to *split* the `testString2` value into an array of substrings with str.split() and inspect the array's elements. See the Splitting strings section to learn more about this function.

The str.length() function is also useful for counting the number of *substrings* of any size contained within a string's sequence, which is helpful information when replacing substrings or performing custom routines that depend on recurring characters.

The following example defines a `countSubstrings()` function, which uses str.replace_all() and str.length() to count the number of times a `target` substring occurs within a specified `source` value. The function creates a modified copy of the `source` with all instances of the `target` removed, then calls str.length() to measure the length of each separate string. It calculates the number of `target` occurrences by dividing the length difference in the original and reduced strings by the length of the substring.

The script uses str.repeat() to generate a "string" value that repeats the sequence `aba` a pseudorandom number of times with `baab` inserted between each instance, then counts all occurrences of the substring `ab` in the result with a `countSubstrings()` call. It then displays a formatted message containing the repeated sequence and the total number of `ab` substrings in the Pine Logs pane:

```
//@version=6 indicator("Counting substrings demo", overlay = true) //@function Counts the number of times a `target` substring occurs within the specified `source`. countSubstrings(string source, string target) => //@variable A copy of the `source` string with all instances of the `target` removed. string reduced = str.replace_all(source, target, "") // Count the characters in the `source`, `target`, and `reduced` strings. int sourceLength = str.length(source) int targetLength = str.length(target) int reducedLength = str.length(reduced) // Calculate the difference between `sourceLength` and `reducedLength` relative to the `targetLength`. // This value represents the number of `target` substrings inside the original `source`. (sourceLength - reducedLength) / targetLength //@variable A string containing `aba` repeated a pseudorandom number of times with `baab` inserted between each instance. string randString = str.repeat("aba", int(math.random(1, 8)), "baab") //@variable The number of times the `ab` sequence occurs in the `randString`. int count = countSubstrings(randString, "ab") // Log a formatted message containing the `randString` and `count` values in the Pine Logs pane. log.info("randString: {0}, count: {1,number,#}", randString, count)
```

Note that:

- This script uses the *second overload* of log.info(), which shares the same signature as str.format() but logs a formatted message instead of returning a value. See the Pine Logs section of the Debugging page to learn more about the `log*()` functions.

Checking for substrings

The str.contains() function searches a `source` string for a specified substring, returning a "bool" value representing whether it found the substring. Two similar functions, str.startswith() and str.endswith(), check whether the `source` *starts* and *ends* with a specified substring.

These functions have the following signatures:

```
str.contains(source, str) → bool
str.startswith(source, str) → bool
str.endswith(source, str) → bool
```

Where:

- `source` is the "string" value that the function searches to find the substring.
- `str` is a "string" value containing the substring to find in the `source`. The str.contains() function returns `true` if the `source` contains at least one instance of the substring. The str.startswith() function returns `true` only if the `source` starts with the substring, even if the substring exists elsewhere in the character sequence. Likewise, str.endswith() returns `true` only if the `source` ends with the substring.

These functions are convenient when a script needs to check whether a substring exists but does not require the substring in additional calculations. Programmers often use these functions in conditional logic to control script behaviors based on a "string" value's contents.

The following script creates a spread symbol string from two symbol inputs and requests price information from that spread symbol using a request.security() call. Before executing the request, the script calls str.startswith() to check whether the `spreadInput` value starts with a leading space and forward slash (`/`), indicating that the first input is *empty*. If the call returns `true`, the script replaces the missing symbol in the "string" value with the chart's symbol to prevent errors.

The script then plots the retrieved data as candles in a separate pane. The colors of the candles change if the chart is in Bar Replay mode. The script tests for Bar Replay mode by searching for the `replay` substring in the chart's ticker identifier (syminfo.tickerid) using a str.contains() call:

//@version=6 indicator("Checking for substrings demo") //@variable A spread symbol created from two symbol inputs. The first input is an empty string by default. var string spreadInput = input.symbol("", "Symbol 1") + " / " + input.symbol("BATS:SPY", "Symbol 2") //@variable Is `true` if the `spreadInput` starts with ` /`, meaning the first input is empty. var bool missingNumerator = str.startswith(spreadInput, " /") //@variable A copy of the spreadInput` that inserts the chart's standard ticker ID when `missingNumerator` is `true`. var string spread = missingNumerator ? str.replace(spreadInput, " /", ticker.standard() + " /") : spreadInput // Request a tuple of OHLC data from the `spread` context. [o, h, l, c] = request.security(" " + spread, "", [open, high, low, close]) //@variable Is `true` if the chart's ticker ID contains `replay`, meaning Bar Replay mode is active. var bool isReplay = str.contains(syminfo.tickerid, "replay") // Define variables to hold colors for up and down candles. Their values depend on whether `isReplay` is `true`. var color upColor = isReplay ? color.blue : color.teal var color dnColor = isReplay ? color.maroon : color.red // Plot the candles in a separate pane.

plotcandle(o, h, l, c, "Spread candles", c > o ? upColor : dnColor) // Log the original `spreadInput`, the final `spread` value, and the `isReplay` value` in the Pine Logs pane. if barstate.isfirst log.info("\n\nOriginal input: {0}\nFinal spread: {1}\nReplay chart: {2}", spreadInput, spread, isReplay)

Note that:

- We used `" /"` as the substring value in the str.startswith() call because an empty string does *not* detect the empty input value. When the substring specified in a str.contains(), str.startswith(), or str.endswith() call is empty, the function always returns `true` because the argument can match *any position* in a string's sequence.

Splitting strings

The str.split() function splits a single "string" value into one or more substrings based on a `separator` substring in the value's character sequence, then collects the results in an array. Below is the function's signature:

```
str.split(string, separator) → array<string>
```

Where:

- The specified `string` is the value to divide into substrings.
- The `separator` is a "string" value containing the characters that divide each substring. The resulting array does *not* include the separator in its elements. If the value is empty, the function splits the string into single-character substrings.

The str.split() function returns an array of strings, unlike the other `str.*()` functions. Scripts can use `array.*()` functions on these arrays, or iterate through them directly with for…in loops. Programmers often use str.split() to process "string" inputs and parameters that represent *lists* of arguments for dynamic requests and other calculations.

The following script requests data from several contexts based on a text area input containing a comma-separated list of symbols. First, the script splits the input value based on its commas with str.split() to construct an array of symbol strings. Then, it uses a for…in loop to iterate over the array's contents, request data for each symbol, and populate a table with the results. Additionally, the table's first row contains a "string" representation of the array of symbols:

//@version=6 indicator("Splitting strings demo") //@variable A string containing a list of symbols separated by commas and optional spaces. string symbolListInput = input.text_area("AAPL, NVDA, MSFT, AMZN, SPY") //@variable An array of symbols created by splitting the `symbolListInput` by its commas. var array<string> symbolsArray = str.split(symbolListInput, ",") if barstate.islast //@variable A two-column table with a row for each `symbolsArray` item. var table display = table.new(position.middle_center, 2, symbolsArray.size() + 1) // Initialize a merged cell to show the `symbolsArray`. display.cell(0, 0, "Symbols:\n" + str.tostring(symbolsArray), bgcolor = color.blue, text_color = color.white, text_size = 30) display.merge_cells(0, 0, 1, 0) // Loop through the `symbolsArray`. for [i, symbol] in symbolsArray //@variable The `close` value requested for the `symbol` on the chart's timeframe. float requestedValue = request.security(symbol, timeframe.period, close) // Initialize a cell for the `symbol` and a "string" representing the `requestedValue`. display.cell(0, i + 1, symbol, text_color = chart.fg_color, text_size = 20) display.cell(1, i + 1, str.tostring(requestedValue), text_color = chart.fg_color, text_size = 20)

Note that:

- The symbol strings in the array contain *extra whitespaces*, which are not visible in the table. However, in contrast to some other function parameters, the `symbol` parameter of request.security() ignores leading and trailing whitespaces in its argument.
- The script uses str.tostring() and concatenation to create the strings used for the table's text.
- This script can fetch data from other contexts within a loop using "series string" `symbol` values because scripts allow dynamic requests by default. See the Dynamic requests section of the Other timeframes and data page for more information.

Locating and retrieving substrings

The str.pos() function searches a `source` string for the *first* occurrence of a specified substring and returns an "int" value representing the *position* of its initial character boundary. The function's signature is as follows:

```
str.pos(source, str) → int
```

Where:

- `source` is the "string" value to search for the first occurrence of the `str` substring.
- `str` is a "string" value representing the substring to locate in the `source`. If the argument is na or an empty "string" value, the function returns 0 (the first possible position).

The str.substring() function retrieves a substring from a `source` value at specified character positions. This function has the following signatures:

```
str.substring(source, begin_pos) → string
str.substring(source, begin_pos, end_pos) → string
```

Where:

- `source` is the "string" value containing the substring.
- `begin_pos` is an "int" value representing the position of the substring's first character in the `source`, where the numbering starts from zero. If the value is na, the function sets the initial position to 0. The script raises an error if the specified position is invalid.
- The `end_pos` is an "int" value representing the position *after* the substring's last character in the `source`. This position is *exclusive*, meaning the returned value does **not** contain this position's character. If the value is not specified or represents a position outside the string's length, the substring includes *all* characters from the `begin_pos` onward. If the value is less than the `begin_pos`, it causes a runtime error.

For example, the `begin_pos` value of the substring `"Trading"` in the string `"TradingView"` is 0, because the substring starts at the source string's *first* character position. The `end_pos` value is 7, because the substring's last character (`g`) is at position 6, and `end_pos` represents the position *after* that character. To retrieve only the first character of a string as a substring, use a call such as `str.substring("TradingView", 0, 1)`.

Programmers often use these functions together by retrieving positional values with str.pos() and then using those values to extract substrings with str.substring() for additional calculations. This technique is an efficient

alternative to matching patterns for substrings at specific positions that have unique characters.

The following simple script uses these functions to extract the “area” and “location” parts of the syminfo.timezone variable’s *IANA identifier*. The script calls str.pos() to get the position of the `/` character in the time zone string, which it assigns to the `dividerPos` variable. Then, it uses that variable in two str.substring() calls. The first call retrieves the substring from position 0 to `dividerPos`, and the second retrieves the substring from the position at `dividerPos + 1` to the end of the string.

The script displays the IANA identifier, the retrieved substrings, and the formatted date and time of the latest execution in a single-cell table on the last bar:

//@version=6 indicator("Locating and retrieving substrings demo", overlay = true) if barstate.islast //@variable A single-cell table to show the `displayText`. var table display = table.new(position.bottom_right, 1, 1, frame_color = chart.fg_color, frame_width = 1) //@variable The position of the `/` character in the exchange time zone's IANA identifier. var int dividerPos = str.pos(syminfo.timezone, "/") //@variable The `syminfo.timezone` substring from position 0 to `dividerPos`. // This substring epresents the "area" part of the time zone identifier. // The character at `dividerPos` is not included. var string areaString = str.substring(syminfo.timezone, 0, dividerPos) //@variable The `syminfo.timezone` substring from `dividerPos + 1` to the end of the string, without low lines. // This substring represents the "location" part of the time zone identifier. var string locationString = str.replace_all(str.substring(syminfo.timezone, dividerPos + 1), "_", " ") //@variable A string representing the latest execution's date and time in the chart's time zone. string formattedTime = str.format_time(timenow, "HH:mm:ss 'on' MMM d, yyyy") //@variable A formatted string containing `syminfo.timezone`, `areaString`, `locationString`, and `formattedTime`. string displayText = str.format("IANA identifier: {0}\n\nArea: {1}\nLocation: {2}\n\nTime of latest tick: {3}", syminfo.timezone, areaString, locationString, formattedTime) // Initialize the table cell with the `displayText`. display.cell(0, 0, displayText, text_size = 24, text_color = chart.fg_color)

Note that:

- The first str.substring() call does *not* include the character at the position specified by `dividerPos`. Its result contains only the characters from position 0 to `dividerPos - 1`.

It’s important to emphasize that the str.pos() function only finds the *first* occurrence of a specified substring. However, in some cases, programmers might require the positions of the substring’s *other* occurrences. One way to achieve this result is by repeatedly *reducing* a “string” value with str.substring() and locating the substring in the new value with str.pos().

The advanced example script below contains a `getPositions()` function that returns an array containing every `substring` position within a specified `source`. The function first uses str.pos() to get the position of the first `substring` and creates an array containing that value with array.from(). If the initial position is not na, the function removes all characters up to the substring’s end position with str.substring(). Then, it executes a for…in loop that repeatedly locates the substring, pushes the calculated position into the array, and reduces the character sequence. The loop stops only after the array contains the position of every substring in the `source` value.

On the first bar, the script uses the function to analyze substring occurrences in four arbitrarily selected strings, then logs formatted messages containing the results in the Pine Logs pane:

//@version=6 indicator("Locating multiple substrings demo", overlay = true) getPositions(string source, string substring) => //@variable The position of the first occurrence of the `substring`. int firstPos = str.pos(source,

substring) //@variable An array containing the starting position of each `substring` occurrence in the `source`. array<int> positions = array.from(firstPos) // Search for extra positions if `firstPos` is not `na`. if not na(firstPos) //@variable The length of the `substring`. int substringLength = str.length(substring) //@variable A substring of `source` from `firstPos + substringLength` onward. string reduced = str.substring(source, firstPos + substringLength) // Loop through the `positions` array. for pos in positions //@variable The end boundary position of the first `substring` occurrence in the `reduced` string. int newPos = str.pos(reduced, substring) + substringLength // Add a new element to the `positions` array and reduce the `reduced` string if `newPos` is not `na`. if not na(newPos) // Push `pos + newPos` into the `positions` array, allowing another iteration. // The `newPos` is added to the latest `pos` to get the actual position in the original `source`. positions.push(pos + newPos) // Assign the substring from `newPos` onward to the `reduced` variable. reduced := str.substring(reduced, newPos) positions if barstate.isfirst // Define four arbitrary strings. string testStr1 = "NASDAQ:AAPL" string testStr2 = "1234321234321" string testStr3 = str.repeat(str.repeat("abc", 3, ", "), 2, " a") string testStr4 = str.format_time(time) // Get arrays containing the positions of various substrings in the four test strings. array<int> positions1 = getPositions(testStr1, "A") array<int> positions2 = getPositions(testStr2, "12") array<int> positions3 = getPositions(testStr3, ", ") array<int> positions4 = getPositions(testStr4, "-") //@variable The formatting string for all `log.info()` calls. The `{0}` placeholder is for apostrophes. string formatString = "Positions of {0}{1}{0} in {0}{2}{0}: {3}" // Log formatted results in the Pine Logs pane. log.info(formatString, "'", "A", testStr1, str.tostring(positions1)) log.info(formatString, "'", "12", testStr2, str.tostring(positions2)) log.info(formatString, "'", ", ", testStr3, str.tostring(positions3)) log.info(formatString, "'", "-", testStr4, str.tostring(positions4))

Note that:

- Although the `positions` array starts with *one* element, the for…in loop performs *more than one* iteration because Pine loops can have dynamic boundaries. After each execution of the array.push() call, the `positions` array's size increases, allowing a *new iteration*. Refer to the Loops page for more information.
- Each reduced version of the string starts at the position *after* the last character of the detected substring. The script identifies the end position by adding the substring's str.length() value to its starting position.

Matching patterns

Pine scripts can dynamically match and retrieve substrings using the str.match() function. In contrast to the other `str.*()` functions, which only match sequences of literal characters, the str.match() function uses regular expressions (regex) to match variable *character patterns*. The function's signature is as follows:

```
str.match(source, regex) → string
```

Where:

- `source` is the "string" value containing the sequence to match using the regular expression.
- `regex` is a "string" value representing the regular expression that specifies the *pattern* to match in the `source`. The function returns the *first substring* that follows the match pattern. If the regex does not match any substring in the `source`, the function returns an empty string.

> TipUsing str.match() requires a basic understanding of regular expressions and how they work. The tables in the next section provide a detailed overview of the syntax supported by Pine's regex engine to help you make the most of this function.

Because the str.match() function matches *patterns* in a string's character sequence rather than strictly literal characters, a single call to this function can perform a wide range of text-matching tasks that would otherwise require multiple calls to other `str.*()` functions or custom operations.

For example, this script requests data from a FINRA Short Sale Volume series for a specified symbol. It uses separate str.startswith() calls to check whether the symbol string has one of the supported exchange prefixes. It locates and removes the exchange prefix with str.pos() and str.substring(), constructs a FINRA ticker ID with str.format() and logs its value, then executes the request.security() call only if one of the str.startswith() calls returns `true`. The script plots the retrieved data on the chart as columns:

//@version=6 indicator("Detecting substrings with other functions demo") //@variable The symbol for which to request Short Sale Volume data. string symbolInput = input.symbol("NASDAQ:AAPL", "Symbol") //@variable `true` if the `symbolInput` starts with `BATS:`, `NASDAQ:`, `NYSE:`, or `AMEX:`. Otherwise, `false`. var bool supportedSymbol = str.startswith(symbolInput, "BATS:") or str.startswith(symbolInput, "NASDAQ:") or str.startswith(symbolInput, "NYSE:") or str.startswith(symbolInput, "AMEX:") //@variable The requested FINRA data if `supportedSymbol` is `true`, `na` otherwise. float requestedData = if supportedSymbol //@variable The `symbolInput` value without the exchange prefix. var string noPrefix = str.substring(symbolInput, str.pos(symbolInput, ":") + 1) //@variable A formatted string representing the ticker ID of a FINRA Short Sale Volume dataset. var string finraTickerID = str.format("FINRA:{0}_SHORT_VOLUME", noPrefix) // Log the `finraTickerID` in the Pine Logs pane on the first bar. if barstate.isfirst log.info(finraTickerID) // Retrieve the data. request.security(finraTickerID, timeframe.isintraday ? "1D" : "", close) // Plot the `requestedData` plot(requestedData, "Short Sale Volume", color.teal, 1, plot.style_columns)

In the script version below, we replaced the multiple str.startswith() calls with an expression containing a str.match() call. The call matches one of the supported exchange prefixes at the start of the string using the following regular expression:

```
^(?:BATS|NASDAQ|NYSE|AMEX):
```

We also replaced the str.pos() and str.substring() calls with a str.match() call. The call calculates the `noPrefix` value with a regex that matches all characters after the input value's colon (`:`):

```
(?<=:).+
```

These changes achieve the same results as the previous script, but with more concise function calls:

//@version=6 indicator("Matching substrings with regex demo") //@variable The symbol for which to request Short Sale Volume data. string symbolInput = input.symbol("NASDAQ:AAPL", "Symbol") //@variable `true` if the `symbolInput` starts with `BATS:`, `NASDAQ:`, `NYSE:`, or `AMEX:`. Otherwise, `false`. // - `^` at the beginning of the regex matches the start of the string. // - `(?:...)` defines a non-capturing group. // - `|` is an OR operator that allows the group to match one of the listed options. var bool supportedSymbol =

str.match(symbolInput, "^(?:BATS|NASDAQ|NYSE|AMEX):") != "" //@variable The requested FINRA data if `supportedSymbol` is `true`, `na` otherwise. float requestedData = if supportedSymbol //@variable The `symbolInput` value without the exchange prefix. // - `(?<=:)` in the regex is a lookbehind assertion that checks if `:` precedes the match. // - `.` matches any character except for line terminators in this context. // - `+` is a quantifier that specifies one or more consecutive `.` matches. var string noPrefix = str.match(symbolInput, "(?<=:).+") //@variable A formatted string representing the ticker ID of a FINRA Short Sale Volume dataset. var string finraTickerID = str.format("FINRA:{0}_SHORT_VOLUME", noPrefix) // Log the `finraTickerID` in the Pine Logs pane on the first bar. if barstate.isfirst log.info(finraTickerID) // Retrieve the data. request.security(finraTickerID, timeframe.isintraday ? "1D" : "", close) // Plot the `requestedData` plot(requestedData, "Short Sale Volume", color.teal, 1, plot.style_columns)

Note that:

- The caret (^) at the beginning of the regex string matches the *beginning* of the `symbolInput`.
- The `(?:...)` syntax in the regex string creates a *non-capturing group*.
- The pipe character (`|`) in the regex acts as an *OR operator*. The group matches only *one* of the character sequences separated by the character (`BATS`, `NASDAQ`, `NYSE`, or `AMEX`).
- The `(?<=...)` syntax defines a *lookbehind assertion*, which checks if the specified pattern precedes the match.
- The `.` (period) character has a special meaning in regex strings. It matches *any* character, excluding line terminators by default.
- The `+` (plus sign) character is a *quantifier*. It specifies that the previous token (`.`) must match *one or more* times.

The flexibility of regular expressions also allows str.match() to perform advanced matching tasks that are impractical or infeasible with other `str.*()` functions.

For instance, suppose we want to create a script that executes dynamic requests for a list of symbols specified in a text area input, and we require a specific input format consisting of only valid ticker patterns, comma separators with optional space characters, and no empty items. This validation is difficult to achieve with the other `str.*()` functions because they rely on *literal* character sequences. However, with str.match(), we can define a single regular expression that matches the input only if it meets our required formatting criteria.

The script below demonstrates a single str.match() function call that validates the format of an input list of symbols. The user-defined `processList()` function combines strings to form the following regex for matching the `list` value:

```
^ *(?:(?:\w+:)?\w+(?:\.\w+){0,2}!? *, *)*(?:\w+:)?\w+(?:\.\w+){0,2}!? *$
```

If the str.match() call returns a *non-empty* string, meaning the constructed pattern matches the `list` argument, the `processList()` function uses str.replace_all() to remove all space characters, then calls str.split() to split the string based on its commas to create an array of symbol substrings. Otherwise, it raises a runtime error with the runtime.error() function.

The script loops through the returned array of substrings to request data for each specified symbol and populate a table with the results:

```
//@version=6 indicator("Processing inputs with regex demo") //@variable A string containing a list of symbols separated by commas and optional spaces. string listInput = input.text_area("AMEX:SPY, AMEX:GLD, NASDAQ:TLT, CME:BTC1!, NYMEX:CL1!, TVC:US10Y", "Symbol list") //@function Checks a string value for a pattern of symbols separated by commas and optional space characters. // If the specified `list` does not match this format, the function raises a runtime error. // Otherwise, it returns an array of substrings representing each listed symbol. processList(string list) => //@variable A pattern for an optional group of one or more word characters (`\w` class) followed by a colon (`:`). // This pattern matches a symbol's exchange prefix, if listed // (e.g., "NASDAQ:" for "NASDAQ:AAPL" or "" for "AAPL"). // - `(?:...)` defines a non-capturing group. // - `\\w` is an ASCII digit, letter, or low line. // - The `+` means the regex checks for one or more consecutive matches of the previous token (`\w`). // - The `?` at the end makes the group optional. var string exchange = "(?:\\w+:)?" //@variable A pattern for 1+ word characters, with 0-2 extra word sequences divided by `.`, then an optional `!`. // This pattern matches symbols without the exchange prefix // (e.g., "AAPL" for "NASDAQ:AAPL", "BTC.D" for "CRYPTOCAP:BTC.D", "ES1!" for "CME_MINI:ES1!"). // - `\\w+` means one or more word characters. // - `\\.` makes the `.` character literal. When not escaped, it matches any character. // - `{0,2}` means the regex matches the group's pattern (`\.\w+`) zero to two times. // - The `?` makes the `!` character optional. var string symbol = "\\w+(?:\\.\\w+){0,2}!?" //@variable A pattern that matches a symbol or comma-separated list of symbols with optional spaces. // (e.g., "AAPL, OANDA:EURUSD, BATS:SPY, BINANCE:BTCUSDT.P") // - `^` at the beginning matches the start of the text line. // - `*` after the spaces and group construction `(?:...)` mean the regex matches them zero or more times. // - `$` at the end matches the end of the text line. // The formatted result combines `exchange` and `symbol` to form this regex pattern: // `^ *(?:(?:\w+:)?\w+(?:\.\w+){0,2}!? *, *)* (?:\w+:)?\w+(?:\.\w+){0,2}!? *$` var string matchPattern = str.format("^ *(?:{0}{1} *, *)*{0}{1} *$", exchange, symbol) //@variable A copy of the `list` if the `matchPattern` produces a match. Otherwise, an empty string. string match = str.match(list, matchPattern) // If the `match` is empty, meaning the `list` does not have the required format, raise a runtime error. if match == "" runtime.error("Invalid list. The value must represent a comma-separated list of symbols with optional spaces.") // Log an `info` message showing the pattern and the match. log.info("\n\nThe pattern:\n\n{0}\n\nmatches:\n\n{1}", matchPattern, match) //@variable A copy of the `match` without space characters. string noSpaces = str.replace_all(match, " ", "") //@variable An array of substrings formed by splitting the `noSpaces` value by its commas. array<string> result = str.split(noSpaces, ",") //@variable An array of symbols from the processed `listInput`. var array<string> symbols = processList(listInput) if barstate.islast //@variable A two-column table with a row for each `symbols` item. var table display = table.new(position.middle_center, 2, symbols.size()) // Loop through the `symbols` array. for [i, symbol] in symbols //@variable The `close` value requested for the `symbol` on the chart's timeframe. float requestedValue = request.security(symbol, timeframe.period, close) // Initialize a cell for the `symbol` and a string representing the `requestedValue`. display.cell(0, i, symbol, text_color = chart.fg_color, text_size = 24) display.cell(1, i, str.tostring(requestedValue), text_color = chart.fg_color, text_size = 24)
```

Note that:

- When creating regex strings, it is often helpful to display them in a script's text outputs to ensure they are formatted as intended. In this script, we included a log.info() call to show the resulting regular expression and its match in the Pine Logs pane.
- Because the backslash (`\`) is an escape character in Pine strings, the value used as the `regex` argument in a str.match() call requires **two** consecutive backslashes for each single backslash in the regular expression.
- The `\\w` parts of the regex string specify the `\w` pattern, a *predefined character class* that matches *word characters* (letters, digits, or low lines).

- The `*` (asterisk) and `{0,2}` parts of the regular expression are *quantifiers*, similar to `+`. The asterisk requires the previous token to match *zero or more* times. The `{0,2}` quantifier requires the match to occur exactly *zero to two* times.
- The `$` (dollar sign) character in this regular expression matches the *end* of the input, excluding any final line terminators (`\n`).
- Because the `.` (period) character has a special meaning in regex strings, we must prefix it with two backslashes (`\\`) in the string to match a literal period.

Regex syntax reference

Every programming language's regex engine has unique characteristics and syntax. Some regex syntax is universal across engines, while other patterns and modifiers are engine-specific.

The tables below provide a categorized overview of the syntax patterns supported by Pine's regex engine along with descriptions, remarks, and examples to explain how they work.

Escapes and character references

▼ Click to show/hide

Token/syntax	Description and remarks
`\`	Changes the meaning of the next character. Because Pine strings natively use `\` as an escape character, regex strings containing it must include an *additional* `\` to use the token in the pattern. For example, `"\\"` represents a single `\` (escape token) in the regex, and `"\\\\"` represents `\\` (literal backslash). Some other characters always or conditionally represent regex syntax, including `.`, `^`, `$`, `*`, `+`, `?`, `(`, `)`, `[`, `]`, `{`, `}`, `|`, and `–`. To match a special character literally, include `"\\"` immediately before it in the string (e.g., `"\\+"` matches the literal + character). Note that some sequences of otherwise literal characters can also have syntactical meaning. See below for examples.
`\Q...\E`	Matches everything between `\Q` and `\E` *literally*, ignoring the syntactical meaning of special characters and sequences. For example, the regex string `"\\Q[^abc]\\E"` matches the literal sequence of `[`, `^`, `a`, `b`, `c`, and `]` characters instead of creating a *character class*.
`a`	Matches the literal character `a` (U+0061). By default, the regex engine is *case-sensitive*. If the string includes the `(?i)` modifier *before* the token, the match becomes *case-insensitive*. For example, the regex string `"(?i)a"` matches the `a` or `A` character.
`\t`	Matches the *tab space* character (U+0009).
`\n`	Matches the *newline* character (U+000A).

Token/syntax	Description and remarks
`\x61`	A *two-digit* Unicode reference that matches the hexadecimal point U+0061 (the a character). This shorthand syntax works only for codes with leading zeros and up to **two** nonzero end digits. It cannot reference other Unicode points. For example, the regex string `"\\x2014"` matches U+0020 (the *space* character) followed by U+0031 (the 1 character) and U+0034 (the 4 character). It **does not** match U+2014 (the – character).
`\u2014`	A *four-digit* Unicode reference that matches the hexadecimal point U+2014 (–, Em Dash). This syntax works only for codes with leading zeros and up to **four** nonzero end digits. It cannot reference larger Unicode points. For example, the regex string `"\\u1F5E0"` matches U+1F5E (unassigned) followed by U+0030 (the `0` character), resulting in no match. It **does not** match U+1F5E0 (the Stock Chart character).
`\x{...}`	The *full-range* Unicode reference syntax. The hexadecimal digits enclosed in the brackets can refer to *any* Unicode point. Leading zeros in the digits *do not* affect the matched Unicode point. For example, the regex strings `"\\x{61}"`, `"\\x{061}"`, `"\\x{0061}"`, and `"\\x{000061}"` all match U+0061 (the a character).

Character class and logical constructions

▼ Click to show/hide

Token/syntax	Description and remarks
`[abc]`	A character class that matches only *one* of the characters listed (`a`, `b`, or `c`). It does **not** match the *entire* `abc` sequence. Each listed character, range, or nested class between two `[]` brackets represents a specific possible match. Note that several special characters have a *literal* meaning inside classes (e.g., the regex string `"[.+$]"` matches `.`, +, or `$` literally). However, regex strings must still escape the following characters to treat them literally because they maintain a special meaning in most cases: `\`, `[`, `]`, `^`, –.
`[a-z]`	A class that matches a single character in the *range* from `a` (U+0061) to `z` (U+007A). It is equivalent to `[\x{61}-\x{7A}]`. Note that the left side of the – character must have a *smaller* Unicode value than the right. For example, the regex string `"[f-a]"` is *invalid* because `f` has the Unicode value U+0066, which is *larger* than the value of `a`. If the dash (–) is at the start or end of the enclosed text, the regex treats it as a *literal* character instead of a character range marker (e.g., `"[-abc]"` matches –, `a`, `b`, or `c` literally).
`[a-zA-Z]`	A class containing a *list* of character *ranges*. It matches any character from `a` (U+0061) to `z` (U+007A) or `A` (U+0041) to `Z` (U+005A) only. It is equivalent to `[\x{61}-\x{7A}\x{41}-\x{5A}]`. The syntax `[a-z[A-Z]]` also produces the same match.
`[^...]`	The syntax for a class that matches any character *except* for the ones specified. For example, the regex string `"[^abc\\n ]"` matches any character except for `a`, `b`, `c`, `\n` (newline), or (space).

Token/syntax	Description and remarks
	Note that only a caret (^) at the *start* of the enclosed text signifies *negation*. If the character comes after that point, the regex considers it a possible *literal* match (e.g., `"[ab^c]"` matches the `a`, `b`, `^`, or `c` character literally).
`[...&& [...]]`	The syntax for a nested class structure that matches any character within the *intersection* of two character classes. Example 1: The regex string `"[abc&&[cde]]"` matches `c` exclusively because it is the only character common to both lists. Example 2: The regex string `"[a-z&&[^def]]"` matches any character from lowercase `a` to `z` except for `d`, `e`, or `f`.
`expr1|expr2`	An OR operation that matches either the `expr1` or `expr2` substring. It does *not* include both in the match.

Predefined classes

▼ Click to show/hide

Token/syntax	Description and remarks
`.`	Matches any character on the line. By default, it *excludes* line terminators (e.g., `\n`). To include line terminators in the match, add the `(?s)` modifier *before* the token in the regex string (e.g., `"(?s)."`).
`\d`	Matches a decimal digit character. By default, it is equivalent to `[0-9]`. However, if the regex string includes the `"(?U)"` modifier before the token, it can match *other* Unicode characters with the "Digit" property. For example, the string `"(?U)\\d"` can match characters such as U+FF11 (Fullwidth Digit One). In contrast, the only "Digit One" character matched by the `[0-9]` class, even with the `(?U)` modifier, is U+0031 (the 1 character).
`\D`	Matches a *non-digit* character. By default, it is equivalent to `[^0-9]`, which does *not* negate other Unicode digits. To exclude other Unicode digits from the match, include the `(?U)` modifier before the token in the regex string (e.g., `"(?U)\\D"`).
`\w`	Matches a *word character* (letter, digit, or low line). By default, it is equivalent to `[a-zA-Z0-9_]`, which excludes other Unicode characters. To include other Unicode letters, digits, or low lines in the match, add the `(?U)` modifier before the token in the regex string. For example, `"(?U)\\w"` can match characters such as U+FE4F (Wavy Low Line), whereas the only low line character the `[a-zA-Z0-9_]` class matches is U+005F (_).
`\W`	Matches a *non-word* character. By default, it is equivalent to `[^a-zA-Z0-9_]`, which does not negate other Unicode characters. To exclude other Unicode word characters from the match, include the `(?U)` modifier before the token (e.g., `"(?U)\\W"`).
`\h`	Matches a *horizontal whitespace* character, such as the tab space (`\t`), standard space, and other characters such as U+2003 (Em Space). The token matches other Unicode characters, even if the regex string includes the `(?-U)` modifier.

Token/syntax	Description and remarks
`\H`	Matches a character that is *not* a horizontal whitespace. It also excludes other Unicode spaces, even if the regex string includes the `(?-U)` modifier
`\s`	Matches a *whitespace* or other *control character*. In contrast to `\h`, this token covers a broader range of characters, including *vertical* spaces such as `\n`.
`\S`	Matches a *non-whitespace* character. In contrast to `\H`, this token excludes a broader character range of characters from the match.

Unicode property classes

▼ Click to show/hide

Token/syntax	Description and remarks
`\p{...}`	The syntax to match a Unicode point that has a specific property, such as script type, block, general category, etc. See the following rows to learn the required syntax for different common Unicode property references. To match any character that does *not* have a specific Unicode property, use the *uppercase* `P` in the syntax (`\P{...}`)
`\p{IsScriptName}` or `\p{Script=ScriptName}`	Unicode script reference syntax. Matches any code point belonging to the `ScriptName` Unicode script. The specified name must not contain spaces. For example, the regex strings `"\\p{IsLatin}"` and `"\\p{Script=Latin}"` both match any Unicode point that is part of the Latin script.
`\p{InBlockName}` or `\p{Block=BlockName}`	Unicode block reference syntax. Matches any code point belonging to the `BlockName` Unicode block. The specified name must not contain spaces. For example, the regex string `"\\p{InBasicLatin}"` matches any Unicode point that is part of the Basic Latin block, and `"\\p{Block=Latin-1Supplement}"` matches any point belonging to the Latin-1 Supplement block.
`\p{category}` or `\p{gc=category}`	Unicode general category reference syntax. Matches any Unicode point with the assigned `category` *abbreviation*. For example, the regex string `"\\p{L}"` or `"\\p{gc=L}"` matches any Unicode point in the *Letter (L)* category, and `"\\p{N}"` matches any point in the *Number (N)* category. Note that, unlike some regex engines, Pine's regex engine does not support the *long form* of a category name, (e.g., `"Letter"` instead of `"L"`).
`\p{ClassName}`	The syntax for referencing the Unicode mapping of a POSIX character class, in Java notation. For example, the regex string `"\\p{XDigit}"` matches a *hexadecimal* digit. By default, it is equivalent to `"[A-Fa-f0-9]"`. Note that the default behavior for POSIX classes matches only *ASCII* characters. To allow other Unicode matches for a POSIX class, use the `(?U)` modifier. For instance, `"(?U)\\p{XDigit}"` can match non-ASCII characters that represent hexadecimal digits, such as U+1D7D9 (the 𝟙 character).

Group constructions

▼ Click to show/hide

Token/syntax	Description and remarks
`(...)`	A *capturing group* that matches the enclosed sequence and stores the matched substring for later reference. Each capturing group construction has an assigned *group number* starting from 1. The regex can reference a capturing group's match with the `\#` syntax, where # represents the group number. For example, the regex string `"(a|b)cde\\1"` matches `a` or `b`, followed by `cde`, and then another occurrence of the `(a|b)` group's initial match. If the group matched `a`, the `\1` reference also matches `a`. If it matched `b`, the reference also matches `b`. If the regex does not need to use a group's match later, using a *non-capturing* group is the more efficient choice, e.g., `(?:...)`.
`(?<name>...)`	A *named* capturing group that matches the enclosed sequence and stores the matched substring with an assigned *identifier*. The regex can use the `\k<name>` syntax to reference the group's match, where `name` is the assigned *identifier*. For example, the string `"(?<myGroup>a|b)cde\\k<myGroup>"` matches `a` or `b`, followed by `cde`, and then another instance of the substring (`a` or `b`) matched by the capturing group. As with a standard capturing group, a named capturing group contributes to the group count and has a *group number*, meaning the regex can also reference a named group with the `\#` syntax, for example, `"(?<myGroup>a|b)cde\\1"`.
`(?:...)`	A *non-capturing* group that matches the enclosed sequence *without* storing the matched substring. Unlike a capturing group, the regex string *cannot* reference a previous non-capturing group's match. For example, the regex string `"(?:a|b)\\1"` matches `a` or `b`, then references an *unassigned* group match, resulting in *no match*. In contrast to all other group constructions, standard non-capturing groups can contain *pattern modifiers* that apply exclusively to their scopes. For example, `"(?i)(?-i:a|b)c"` matches `a` or `b` followed by lowercase `c` or uppercase `C`. The `(?i)` part of the regex activates case-insensitive matching globally, but the `-i` token *deactivates* the behavior for the group's scope only. Note that non-capturing groups typically have a *lower* computational cost than capturing groups.
`(?>...)`	An *independent* non-capturing group (*atomic group*). Unlike a standard non-capturing group, an atomic group consumes as many characters as possible *without* allowing other parts of the pattern to use them. For example, the regex string `"(?s)(?>.+).+"` fails to produce a match because the atomic group `(?>.+)` consumes *every* available character, leaving *nothing* for the following `.+` portion to match. In contrast, the regex string `"(?s)(?:.+).+"` matches the entire source string because the standard non-capturing group `(?:.+)` *releases* characters from its match as needed, allowing `.+` to match *at least one* character.

Quantifiers

▼ Click to show/hide

Token/syntax	Description and remarks
?	Appending ? to a character, group, or class specifies that the matched substring must contain the pattern *once* or *not at all*.

Token/syntax	Description and remarks
	For example, the regex string `"a?bc?"` matches `abc`, `ab`, `bc`, or `b` because `a` and `c` are *optional*. By default, regex quantifiers are *greedy*, meaning they match as many characters as possible, releasing some as necessary. Adding `?` to another quantifier makes it *lazy*, meaning it matches the *fewest* characters possible, expanding its match only when required. For example, with a source string of `"a12b34b"`, the regex string `"a.*b"` matches the entire sequence, whereas `"a.*?b"` matches the *smallest* valid substring with the pattern, which is `a12b`.
`*`	Appending `*` to a character, group, or class specifies that the matched substring must contain the pattern *zero* or *more* times consecutively. For example, the regex string `"a*b"` matches zero or more consecutive `a` characters followed by a single `b` character.
`+`	Appending `+` to a character, group, or class specifies that the matched substring must contain the pattern *one* or *more* times consecutively. For example, the regex string `"\\w+abc"` matches one or more consecutive word characters followed by `abc`. Adding `+` to another quantifier makes it *possessive*. Unlike a greedy quantifier (default), which *releases* characters from the match as necessary, a possessive quantifier consumes as many characters as possible *without* releasing them for use in other parts of the pattern. For instance, the regex string `"\\w++abc"` fails to produce a match because `\w++` consumes *all* word characters in the pattern, including `a`, `b`, and `c`, leaving none for the `abc` portion to match.
`{n}`	Appending `{n}` to a character, group, or class specifies that the matched substring must contain the pattern exactly `n` times consecutively, where `n >= 0`. For example, the regex string `"[abc]{2}"` matches two consecutive characters from the `[abc]` class, meaning the possible substrings are `aa`, `ab`, `ac`, `ba`, `bb`, `bc`, `ca`, `cb`, or `cc`.
`{n,}`	Appending `{n,}` to a character, group, or class specifies that the matched substring must contain the pattern *at least* `n` times consecutively, where `n >= 0`. For example, the regex string `"a{1,}b{2,}"` matches one or more consecutive `a` characters followed by two or more consecutive `b` characters.
`{n, m}`	Appending `{n, m}` to a character, group, or class specifies that the matched substring must contain the pattern at least `n` times but no more than `m` times, where `n >= 0`, `m >= 0`, and `m >= n`. For example, the regex string `"\\w{1,5}b{2,4}"` matches one to five consecutive word characters followed by two to four repeated `b` characters.

Boundary assertions

▼ Click to show/hide

Token/syntax	Description and remarks
`\A`	Matches the *starting point* of the source string without consuming characters. It enables the regex to isolate the initial pattern in a string without allowing matches in other locations. For example, the regex string `"\\A\\w+"` matches a sequence of one or more word characters only if the sequence is at the start of the source string.

Token/syntax	Description and remarks
`^`	When this character is outside a character class construction (i.e., `[^...]`), it matches the *starting point* of a *line* in the source string without consuming characters. By default, the character performs the same match as `\A`. However, if the regex string uses the `(?m)` modifier, it can also match a point immediately after a *newline* character (`\n`). For example, the regex string `"(?m)^[xyz]"` matches `x`, `y`, or `z` if the character is at the start of the source string or immediately after the `\n` character.
`\Z`	Matches the *ending point* of the source string, or the point immediately before the final character if it is `\n`, without consuming characters. It enables the regex to isolate the final pattern in a string without allowing matches in other locations. For example, the regex string `"\\w+\\Z"` matches a sequence of one or more word characters only if the sequence is at the end of the source string or immediately before the final line terminator.
`\z`	Matches the *absolute* ending point of the source string without consuming characters. Unlike `\Z` (uppercase), this token does not match the point *before* any final line terminator. For example, the regex string `"(?s)\\w+.*\\z"` matches a sequence of one or more word characters, followed by zero or more extra characters, only if the sequence is at the absolute end of the source string.
`$`	Matches the *ending point* of a *line* in the source string without consuming characters. By default, it performs the same match as `\Z` (uppercase). However, if the regex string uses the `(?m)` modifier, it can match any point immediately before a newline (`\n`) character. For example, the regex string `"(?m)[123]$"` matches `1`, `2`, or `3` only if the character is at the end of the source string or immediately before the `\n` character.
`\b`	Matches a *word boundary*, which is the point immediately before or after a sequence of word characters (members of the `\w` class). For example, the regex string `"\\babc"` matches `abc` only if it is at the starting point of a word character sequence.
`\B`	Matches a *non-word* boundary, which is any point between characters that is *not* the start or end of a word character sequence. For example, the regex string `"\\Babc"` matches `abc` only if it is not at the start of a word character sequence.

Lookahead and lookbehind assertions

▼ Click to show/hide

Token/syntax	Description and remarks
`(?=...)`	A *positive lookahead* assertion that checks whether the specified sequence immediately *follows* the current match location, without consuming characters. For example, the regex string `"a(?=b)"` matches the `a` character only if `b` occurs immediately after that point, and it does not include `b` in the matched substring.
`(?!...)`	A *negative lookahead* assertion that checks whether the specified sequence *does not* immediately follow the current match location, without consuming characters.

Token/syntax	Description and remarks
	For example, the regex string `"a(?!b)"` matches the a character only if the b character does not immediately follow it.
`(?<=...)`	A *positive lookbehind* assertion that checks whether the specified sequence immediately *precedes* the current match location, without consuming characters. For example, the regex string `"(?<=a)b"` matches the b character only if the a character occurs immediately before that point, and it does not include a in the matched substring.
`(?<!...)`	A *negative lookbehind* assertion that checks whether the specified sequence *does not* immediately precede the current match location, without consuming characters. For example, the regex string `"(?<!a)b"` matches the b character only if the a character does not immediately precede it.

Pattern modifiers

▼ Click to show/hide

Token/syntax	Description and remarks
`(?...)`	This syntax applies a global list of inline *pattern modifiers* (flags) to the regex string. Pattern modifiers change the matching behaviors of the regex engine. All parts of the regex string that come after this syntax update their behaviors based on the specified modifiers, and those behaviors persist from that point until explicitly overridden. For example, `"(?mi)"` activates *multiline* and *case-insensitive* modes for the rest of the regex string. To deactivate modifiers, include the – character before the list of modifier tokens. For instance, `"(?-mi)"` deactivates multiline and case-insensitive modes for the rest of the regex string. Standard *non-capturing groups* can also utilize modifiers *locally*, allowing different behaviors exclusively within group constructions. For example, `"(?U:\\d)123"` activates *Unicode-aware* matching only for the specific group. The modifier does not apply globally, meaning the remaining `123` part of the regex string can only match *ASCII* characters. See the rows below for details about the most common, useful pattern modifiers for Pine regex strings.
`i`	The `i` character represents *case-insensitive* mode when used as a global modifier (`(?i)`) or group modifier (`(?i:...)`). For example, the regex string `"a(?i)b(?-i)c"` matches lowercase a, uppercase B or lowercase b, and then lowercase c. Note that case-insensitive mode only applies to ASCII characters unless Unicode-aware mode is active.
`m`	The m character represents multiline mode when used as a global modifier (`(?m)`) or group modifier (`(?m:...)`). By default, the `^` and `$` boundary assertions match the start and end of the source string, excluding final line terminators. With multiline mode enabled, they match the start and end boundaries of any *separate line* in the string. For example, the regex string `"^abc"` matches `abc` only if the source string starts with that sequence,

Token/syntax	Description and remarks
	whereas "(?m)^abc" matches abc if it is at the start of the string or immediately follows a *newline character* (\n).
s	The lowercase s character represents *single-line mode* (*dotall mode*) when used as a global modifier ((?s)) or group modifier ((?s:...)). By default, the . character matches any character except for line terminators such as \n. With single-line mode enabled, the regex treats the source string as *one line*, allowing the character to match line terminators. For example, using the regex string ".+" on the source string "ab\nc" matches ab only, whereas "(?m).+" matches the *entire* source string.
U	The uppercase U character represents *Unicode-aware* mode when used as a global modifier ((?U)) or group modifier ((?U:...)). By default, most of the regex engine's predefined character classes and mapped POSIX classes do not match *non-ASCII* characters. With Unicode-aware mode enabled, the regex allows these classes, and various ASCII character tokens, to match related Unicode characters. For example, the regex string "\\d(?U)\\d+" matches a single ASCII digit followed by one or more Unicode digit characters.
x	The lowercase x character represents *verbose mode* (*comments mode*) when used as a global modifier ((?x)) or group modifier ((?x:...)). In this mode, the regex string ignores *whitespace* characters and treats sequences starting with # as *comments*. For example, the regex string "(?x)[a-f] 1 2\n3 # this is a comment!" produces the same match as "[a-f]123". It does **not** match the space or newline characters, or anything starting from the # character. Regex strings with this modifier can include multiple comments on *separate lines* (e.g., "a #match 'a' \nb #followed by 'b'" matches ab). To match whitespaces or the # character in this mode, *escape* them using backslashes or the \Q...\E syntax. For instance, "(?x)\\#\\ \\# #comment" and "(?x)\\Q# #\\E #comment" both literally match the sequence # #.

Time

Introduction

In Pine Script®, the following key aspects apply when working with date and time values:

- **UNIX timestamp**: The native format for time values in Pine, representing the absolute number of *milliseconds* elapsed since midnight UTC on 1970-01-01. Several built-ins return UNIX timestamps directly, which users can format into readable dates and times. See the UNIX timestamps section below for more information.
- **Exchange time zone**: The time zone of the instrument's exchange. All calendar-based variables hold values expressed in the exchange time zone, and all built-in function overloads that have a `timezone` parameter use this time zone by default.
- **Chart time zone**: The time zone the chart and Pine Logs message prefixes use to express time values. Users can set the chart time zone using the "Timezone" input in the "Symbol" tab of the chart's settings. This setting only changes the *display* of dates and times on the chart and the times that prefix logged messages. It does **not** affect the behavior of Pine scripts because they cannot access a chart's time zone information.
- `timezone` **parameter**: A "string" parameter of time-related functions that specifies the time zone used in their calculations. For calendar-based functions, such as dayofweek(), the `timezone` parameter determines the time zone of the returned value. For functions that return UNIX timestamps, such as time(), the specified `timezone` defines the time zone of other applicable parameters, e.g., `session`. See the Time zone strings section to learn more.

UNIX timestamps

UNIX time is a standardized date and time representation that measures the number of *non-leap seconds* elapsed since January 1, 1970 at 00:00:00 UTC (the *UNIX Epoch*), typically expressed in seconds or smaller time units. A UNIX time value in Pine Script is an "int" *timestamp* representing the number of *milliseconds* from the UNIX Epoch to a specific point in time.

Because a UNIX timestamp represents the number of consistent time units elapsed from a fixed historical point (epoch), its value is **time zone-agnostic**. A UNIX timestamp in Pine always corresponds to the same distinct point in time, accurate to the millisecond, regardless of a user's location.

For example, the UNIX timestamp `1723472500000` always represents the time 1,723,472,500,000 milliseconds (1,723,472,500 seconds) after the UNIX Epoch. This timestamp's meaning does **not** change relative to any time zone.

To *format* an "int" UNIX timestamp into a readable date/time "string" expressed in a specific time zone, use the str.format_time() function. The function does not *modify* UNIX timestamps. It simply *represents* timestamps in a desired human-readable format.

For instance, the function can represent the UNIX timestamp `1723472500000` as a "string" in several ways, depending on its `format` and `timezone` arguments, without changing the *absolute* point in time that it refers to. The simple script below calculates three valid representations of this timestamp and displays them in the Pine Logs pane:

//@version=6 indicator("UNIX timestamps demo") //@variable A UNIX time value representing the specific point 1,723,472,500,000 ms after the UNIX Epoch. int unixTimestamp = 1723472500000 // These are a few different ways to express the `unixTimestamp` in a relative, human-readable format. // Despite their format and time zone differences, all the calculated strings represent the SAME distinct point: string isoExchange = str.format_time(unixTimestamp) string utcDateTime = str.format_time(unixTimestamp, "MM/dd/yyyy HH:mm:ss.S", "UTC+0") string utc4TimeDate = str.format_time(unixTimestamp, "hh:mm:ss a, MMMM dd, yyyy z", "UTC+4") // Log the `unixTimestamp` and the custom "string" representations on the first bar. if barstate.isfirst log.info("\nUNIX time (ms): {0, number, #}\nISO 8601 representation (Exchange time zone): {1}" + "\nCustom date and time representation (UTC+0 time zone): {2}" + "\nCustom time and date representation (UTC+4 time zone): {3}", unixTimestamp, isoExchange, utcDateTime, utc4TimeDate)

Note that:

- The value enclosed within square brackets in the logged message is an *automatic* prefix representing the historical time of the log.info() call in ISO 8601 format, expressed in the chart time zone.
- The script concatenates three literal strings to create one long `formatString` argument for the log.info() call.

See the Formatting dates and times section to learn more about representing UNIX timestamps with formatted strings.

Time zones

A time zone is a geographic region with an assigned *local time*. The specific time within a time zone is consistent throughout the region. Time zone boundaries typically relate to a location's longitude. However, in practice, they tend to align with administrative boundaries rather than strictly following longitudinal lines.

The local time within a time zone depends on its defined *offset* from Coordinated Universal Time (UTC), which can range from UTC-12:00 (12 hours *behind* UTC) to UTC+14:00 (14 hours *ahead* of UTC). Some regions maintain a consistent offset from UTC, and others have an offset that changes over time due to daylight saving time (DST) and other factors.

Two primary time zones apply to data feeds and TradingView charts: the *exchange time zone* and the *chart time zone*.

The exchange time zone represents the time zone of the current symbol's *exchange*, which Pine scripts can access with the syminfo.timezone variable. Calendar-based variables, such as month, dayofweek, and hour, always hold values expressed in the exchange time zone, and all time function overloads that have a `timezone` parameter use this time zone by default.

The chart time zone is a *visual preference* that defines how the chart and the time prefixes of Pine Logs represent time values. To set the chart time zone, use the "Timezone" input in the "Symbol" tab of the chart's settings or click on the current time shown below the chart. The specified time zone does **not** affect time calculations in Pine scripts because they cannot access this chart information. Although scripts cannot access a chart's time zone, programmers can provide inputs that users can adjust to match the time zone.

For example, the script below uses str.format_time() to represent the last historical bar's opening and closing UNIX timestamps (time and time_close values) as date-time strings expressed in the function's default time zone, the exchange time zone, UTC-0, and a user-specified time zone. It uses a table to display all four representations in the bottom-right corner of the chart for comparison:

```
//@version=6 indicator("Time zone comparison demo", overlay = true) //@variable The time zone of the time values in the last table row. // The "string" can contain either UTC offset notation or an IANA time zone identifier. string timezoneInput = input.string("UTC+4:00", "Time zone") //@variable A `table` showing strings representing bar times in three preset time zones and a custom time zone. var table displayTable = table.new( position.bottom_right, columns = 3, rows = 5, border_color = chart.fg_color, border_width = 2 ) //@function Initializes three `displayTable` cells on the `row` that show the `title`, `text1`, and `text2` strings. tableRow(int row, string title, string text1, string text2, color titleColor = #9b27b066, color infoColor = na) => displayTable.cell(0, row, title, bgcolor = titleColor, text_color = chart.fg_color) displayTable.cell(1, row, text1, bgcolor = infoColor, text_color = chart.fg_color) displayTable.cell(2, row, text2, bgcolor = infoColor, text_color = chart.fg_color) if barstate.islastconfirmedhistory // Draw an empty label to signify the bar that the displayed time strings represent. label.new(bar_index, high, color = #9b27b066, size = size.huge) //@variable The formatting string for all `str.format_time()` calls. Sets the format of the date-time strings. var string formatString = "yyyy-MM-dd HH:mm:ss" // Initialize a header row at the top of the `displayTable`. tableRow(0, "", "OPEN time", "CLOSE time", na, #9b27b066) // Initialize a row showing the bar's times in the default time zone (no specified `timezone` arguments). tableRow(1, "Default", str.format_time(time, formatString), str.format_time(time_close, formatString)) // Initialize a row showing the bar's times in the exchange time zone (`syminfo.timezone`). tableRow(2, "Exchange: " + syminfo.timezone, str.format_time(time, formatString, syminfo.timezone), str.format_time(time_close, formatString, syminfo.timezone) ) // Initialize a row showing the bar's times in the UTC-0 time zone (using "UTC" as the `timezone` arguments). tableRow(3, "UTC-0", str.format_time(time, formatString, "UTC"), str.format_time(time_close, formatString, "UTC")) // Initialize a row showing the bar's times in the custom time zone (`timezoneInput`). tableRow( 4, "Custom: " + timezoneInput, str.format_time(time, formatString, timezoneInput), str.format_time(time_close, formatString, timezoneInput) )
```

Note that:

- The label on the chart signifies which bar's times the displayed strings represent.
- The "Default" and "Exchange" rows in the table show identical results because syminfo.timezone is the str.format_time() function's default `timezone` argument.
- The exchange time zone on our example chart appears as `"America/New_York"`, the IANA identifier for the NASDAQ exchange's time zone. It represents UTC-4 *or* UTC-5, depending on the time of year. See the next section to learn more about time zone strings.

Time zone strings

All built-in functions with a `timezone` parameter accept a "string" argument specifying the time zone they use in their calculations. These functions can accept time zone strings in either of the following formats:

- **UTC** (or *GMT*) offset notation, e.g., `"UTC-5"`, `"UTC+05:30"`, `"GMT+0100"`
- **IANA database** notation, e.g., `"America/New_York"`, `"Asia/Calcutta"`, `"Europe/Paris"`

The IANA time zone database reference page lists possible time zone identifiers and their respective UTC offsets. The listed identifiers are valid as `timezone` arguments.

Note that various time zone strings expressed in UTC or IANA notation can represent the *same* offset from Coordinated Universal Time. For instance, these strings all represent a local time three hours ahead of UTC:

- `"UTC+3"`
- `"GMT+03:00"`
- `"Asia/Kuwait"`

- `"Europe/Moscow"`
- `"Africa/Nairobi"`

For the str.format_time() function and the functions that calculate calendar-based values from a UNIX timestamp, including month(), dayofweek(), and hour(), the "string" passed to the `timezone` parameter changes the returned value's calculation to express the result in the specified time zone. See the Formatting dates and times and Calendar-based functions sections for more information.

The example below shows how time zone strings affect the returned values of calendar-based functions. This script uses three hour() function calls to calculate "int" values representing the opening hour of each bar in the exchange time zone, UTC-0, and a user-specified UTC offset. It plots all three calculated hours in a separate pane for comparison:

//@version=6 indicator("Time zone strings in calendar functions demo") //@variable An "int" representing the user-specified hourly offset from UTC. int utcOffsetInput = input.int(defval = 4, title ="Timezone offset UTC (+/-)", minval = -12, maxval = 14) //@variable A valid time zone string based on the `utcOffsetInput`, in UTC offset notation (e.g., "UTC-4"). string customOffset = "UTC" + (utcOffsetInput > 0 ? "+" : "") + str.tostring(utcOffsetInput) //@variable The bar's opening hour in the exchange time zone (default). Equivalent to the `hour` variable. int exchangeHour = hour(time) //@variable The bar's opening hour in the "UTC-0" time zone. int utcHour = hour(time, "UTC-0") //@variable The bar's opening hour in the `customOffset` time zone. int customOffsetHour = hour(time, customOffset) // Plot the `exchangeHour`, `utcHour`, and `customOffsetHour` for comparison. plot(exchangeHour, "Exchange hour", #E100FF5B, 8) plot(utcHour, "UTC-0 hour", color.blue, 3) plot(customOffsetHour, "Custom offset hour", color.orange, 3)

Note that:

- The `exchangeHour` value is four *or* five hours behind the `utcHour` because the NASDAQ exchange is in the "America/New_York" time zone. This time zone has a UTC offset that *changes* during the year due to daylight saving time (DST). The script's default `customOffsetHour` is consistently four hours ahead of the `utcHour` because its time zone is UTC+4.
- The call to the hour() function without a specified `timezone` argument returns the same value that the hour *variable* holds because both represent the bar's opening hour in the exchange time zone (syminfo.timezone).

For functions that return UNIX timestamps directly, such as time() and timestamp(), the `timezone` parameter defines the time zone of the function's calendar-based *parameters*, including `session`, `year`, `month`, `day`, `hour`, `minute`, and `second`. The parameter does *not* determine the time zone of the returned value, as UNIX timestamps are *time zone-agnostic*. See the Testing for sessions and `timestamp()` sections to learn more.

The following script calls the timestamp() function to calculate the UNIX timestamp of a specific date and time, and it draws a label at the timestamp's corresponding bar location. The user-selected `timezone` argument (`timezoneInput`) determines the time zone of the call's calendar-based arguments. Consequently, the calculated timestamp varies with the `timezoneInput` value because identical local times in various time zones correspond to *different* amounts of time elapsed since the UNIX Epoch:

//@version=6 indicator("Time zone strings in UNIX timestamp functions demo", overlay = true) //@variable The `timezone` argument of the `timestamp()` call, which sets the time zone of all date and time parameters. string timezoneInput = input.string("Etc/UTC", "Time zone") //@variable The UNIX timestamp corresponding to a specific calendar date and time. // The specified `year`, `month`, `day`, `hour`, `minute`, and `second` represent calendar values in the // `timezoneInput` time zone. // Different `timezone` arguments produce

different UNIX timestamps because an identical date in another // time zone does NOT represent the same absolute point in time. int unixTimestamp = timestamp(timezone = timezoneInput, year = 2024, month = 10, day = 31, hour = 0, minute = 0, second = 0) //@variable The `close` value when the bar's opening time crosses the `unixTimestamp`. float labelPrice = ta.valuewhen(ta.cross(time, unixTimestamp), close, 0) // On the last historical bar, draw a label showing the `unixTimestamp` value at the corresponding bar location. if barstate.islastconfirmedhistory label.new(unixTimestamp, nz(labelPrice, close), "UNIX timestamp: " + str.tostring(unixTimestamp), xloc.bar_time, yloc.price, chart.fg_color, label.style_label_down, chart.bg_color, size.large)

Note that:

- `"Etc/UTC"` is the *IANA identifier* for the UTC+0 time zone.
- The label.new() call uses xloc.bar_time as its `xloc` argument, which is required to anchor the drawing to an absolute time value. Without this argument, the function treats the `unixTimestamp` as a relative bar index, leading to an incorrect location.
- The label's `y` value is the close of the bar where the time value crosses the `unixTimestamp` value. If the timestamp represents a future time, the label displays the last historical bar's price.

Although time zone strings can use either UTC or IANA notation, we recommend using *IANA notation* for `timezone` arguments in most cases, especially if a script's time calculations must align with the observed time offset in a specific country or subdivision. When a time function call uses an IANA time zone identifier for its `timezone` argument, its calculations adjust automatically for historical and future changes to the specified region's observed time, such as daylight saving time (DST) and updates to time zone boundaries, instead of using a fixed offset from UTC.

The following script demonstrates how UTC and IANA time zone strings can affect time calculations differently. It uses two calls to the hour() function to calculate the hour from the current bar's opening timestamp using `"UTC-4"` and `"America/New_York"` as `timezone` arguments. The script plots the results of both calls for comparison and colors the main pane's background when the returned values do not match. Although these two hour() calls may seem similar because UTC-4 is an observed UTC offset in New York, they *do not* always return the same results, as shown below:

//@version=6 indicator("UTC vs IANA time zone strings demo") //@variable The hour of the current `time` in the "UTC-4" time zone. // This variable's value represents the hour in New York only during DST. It is one hour ahead otherwise. int hourUTC = hour(time, "UTC-4") //@variable The hour of the current `time` in the "America/New_York" time zone. // This form adjusts to UTC offset changes automatically, so the value always represents the hour in New York. int hourIANA = hour(time, "America/New_York") //@variable Is translucent blue when `hourUTC` does not equal `hourIANA`, `na` otherwise. color bgColor = hourUTC != hourIANA ? color.rgb(33, 149, 243, 80) : na // Plot the values of `hourUTC` and `hourIANA` for comparison. plot(hourUTC, "UTC-4", color.blue, linewidth = 6) plot(hourIANA, "America/New_York", color.orange, linewidth = 3) // Highlight the main chart pane with the `bgColor`. bgcolor(bgColor, title = "Unequal result highlight", force_overlay = true)

The plots in the chart above diverge periodically because New York observes daylight saving time, meaning its UTC offset *changes* at specific points in a year. During DST, New York's local time follows UTC-4. Otherwise, it follows UTC-5. Because the script's first hour() call uses `"UTC-4"` as its `timezone` argument, it returns the correct hour in New York *only* during DST. In contrast, the call that uses the `"America/New_York"` time zone string adjusts its UTC offset automatically to return the correct hour in New York at *any* time of the year.

Time variables

Pine Script has several built-in variables that provide scripts access to different forms of time information:

- The time and time_close variables hold UNIX timestamps representing the current bar's opening and closing times, respectively.
- The time_tradingday variable holds a UNIX timestamp representing the starting time of the last UTC calendar day in a session.
- The timenow variable holds a UNIX timestamp representing the current time when the script executes.
- The year, month, weekofyear, dayofmonth, dayofweek, hour, minute, and second variables reference calendar values based on the current bar's opening time, expressed in the exchange time zone.
- The last_bar_time variable holds a UNIX timestamp representing the last available bar's opening time.
- The chart.left_visible_bar_time and chart.right_visible_bar_time variables hold UNIX timestamps representing the opening times of the leftmost and rightmost visible chart bars.
- The syminfo.timezone variable holds a "string" value representing the time zone of the current symbol's exchange in IANA database notation. All time-related function overloads with a `timezone` parameter use this variable as the default argument.

`time` and `time_close` variables

The time variable holds the UNIX timestamp of the current bar's *opening time*, and the time_close variable holds the UNIX timestamp of the bar's *closing time*.

These timestamps are unique, time zone-agnostic "int" values, which programmers can use to anchor drawing objects to specific bar times, calculate and inspect bar time differences, construct readable date/time strings with the str.format_time() function, and more.

The script below displays bar opening and closing times in different ways. On each bar, it formats the time and time_close timestamps into strings containing the hour, minute, and second in the exchange time zone, and it draws labels displaying the formatted strings at the open and close prices. Additionally, the script displays strings containing the unformatted UNIX timestamps of the last chart bar within a table in the bottom-right corner:

//@version=6 indicator("`time` and `time_close` demo", overlay = true, max_labels_count = 500) //@variable A "string" representing the hour, minute, and second of the bar's opening time in the exchange time zone. string openTimeString = str.format_time(time, "HH:mm:ss") //@variable A "string" representing the hour, minute, and second of the bar's closing time in the exchange time zone. string closeTimeString = str.format_time(time_close, "HH:mm:ss") //@variable Is `label.style_label_down` when the `open` is higher than `close`, `label.style_label_up` otherwise. string openLabelStyle = open > close ? label.style_label_down : label.style_label_up //@variable Is `label.style_label_down` when the `close` is higher than `open`, `label.style_label_up` otherwise. string closeLabelStyle = close > open ? label.style_label_down : label.style_label_up // Draw labels anchored to the bar's `time` to display the `openTimeString` and `closeTimeString`. label.new(time, open, openTimeString, xloc.bar_time, yloc.price, color.orange, openLabelStyle, color.white) label.new(time, close, closeTimeString, xloc.bar_time, yloc.price, color.blue, closeLabelStyle, color.white) if barstate.islast //@variable A `table` displaying the last bar's *unformatted* UNIX timestamps. var table t = table.new(position.bottom_right, 2, 2, bgcolor = #ffe70d) // Populate the `t`

table with "string" representations of the the "int" `time` and `time_close` values. t.cell(0, 0, "`time`") t.cell(1, 0, str.tostring(time)) t.cell(0, 1, "`time_close`") t.cell(1, 1, str.tostring(time_close))

Note that:

- This script's label.new() calls include xloc.bar_time as the `xloc` argument and time as the `x` argument to anchor the drawings to bar opening times.
- The formatted strings express time in the exchange time zone because we did not specify `timezone` arguments in the str.format_time() calls. NYSE, our chart symbol's exchange, is in the "America/New_York" time zone (UTC-4/-5).
- Although our example chart uses an *hourly* timeframe, the table and the labels at the end of the chart show that the last bar closes only *30 minutes* (1,800,000 milliseconds) after opening. This behavior occurs because the chart aligns bars with session opening and closing times. A session's final bar closes when the session ends, and a new bar opens when a new session starts. A typical session on our 60-minute chart with regular trading hours (RTH) spans from 09:30 to 16:00 (6.5 hours). The chart divides this interval into as many 60-minute bars as possible, starting from the session's opening time, which leaves only 30 minutes for the final bar to cover.

It's crucial to note that unlike the time variable, which has consistent behavior across chart types, time_close behaves differently on *time-based* and *non-time-based* charts.

Time-based charts have bars that typically open and close at regular, *predictable* times within a session. Thanks to this predictability, time_close can accurately represent the *expected* closing time of an open bar on a time-based chart, as shown on the last bar in the example above.

In contrast, the bars on tick charts and *price-based* charts (all non-standard charts excluding Heikin Ashi) cover *irregular* time intervals. Tick charts construct bars based on successive ticks in the data feed, and price-based charts construct bars based on significant price movements. The time it takes for new ticks or price changes to occur is *unpredictable*. As such, the time_close value is na on the *open realtime bars* of these charts.

The following script uses the time and time_close variables with str.tostring() and str.format_time() to create strings containing bar opening and closing UNIX timestamps and formatted date-time representations, which it displays in labels at each bar's high and low prices.

When applied to a Renko chart, which forms new bars based on *price movements*, the labels show correct results on all historical bars. However, the last bar has a time_close value of na because the future closing time is unpredictable. Consequently, the bar's closing time label shows a timestamp of `"NaN"` and an *incorrect* date and time:

//@version=6 indicator("`time_close` on non-time-based chart demo", overlay = true) //@variable A formatted "string" containing the date and time that `time_close` represents in the exchange time zone. string formattedCloseTime = str.format_time(time_close, format = "'Date and time:' yyyy-MM-dd HH:mm:ss") //@variable A formatted "string" containing the date and time that `time` represents in the exchange time zone. string formattedOpenTime = str.format_time(time, format = "'Date and time:' yyyy-MM-dd HH:mm:ss") //@variable A "string" containing the `time_close` UNIX timestamp and the `formattedCloseTime`. string closeTimeText = str.format("Close timestamp: {0,number,#}\n{1}", time_close, formattedCloseTime) //@variable A "string" containing the `time` UNIX timestamp and the `formattedOpenTime`. string openTimeText = str.format("Open timestamp: {0,number,#}\n{1}", time, formattedOpenTime) // Define label colors for historical and realtime bars. color closeLabelColor = barstate.islast ? color.purple : color.aqua color

openLabelColor = barstate.islast ? color.green : color.orange // Draw a label at the `high` to display the `closeTimeText` and a label at the `low` to display the `openTimeText`, // both anchored to the bar's `time`. label.new(time, high, closeTimeText, xloc.bar_time, color = closeLabelColor, textcolor = color.white, size = size.large, textalign = text.align_left) label.new(time, low, openTimeText, xloc.bar_time, color = openLabelColor, style = label.style_label_up, size = size.large, textcolor = color.white, textalign = text.align_left) // Highlight the background yellow on the latest bar. bgcolor(barstate.islast ? #f3de22cb : na, title = "Latest bar highlight")

Note that:

- The script draws up to 50 labels because we did not specify a `max_labels_count` argument in the indicator() declaration statement.
- The str.format_time() function replaces na values with 0 in its calculations, which is why it returns an incorrect date-time "string" on the last bar. A timestamp of 0 corresponds to the *UNIX Epoch* (00:00:00 UTC on January 1, 1970). However, the str.format_time() call does not specify a `timezone` argument, so it expresses the epoch's date and time in the *exchange time zone*, which was five hours behind UTC at that point in time.
- The time_close() function, which returns the closing timestamp of a bar on a specified timeframe within a given session, also returns na on the open realtime bars of tick-based and price-based charts.

Scripts can retrieve a realtime bar's closing time on tick charts and price-based charts once the bar is *confirmed*. The closing timestamp of an *elapsed realtime bar* is committed to the realtime data feed as soon as the bar closes, so its time_close value is no longer na.

To demonstrate this, we can modify the script above to use `time_close[1]` to output the previous bar's closing time on each bar. The image below shows two highlighted realtime bars. When we executed the script on the chart, the first bar was initially an unconfirmed realtime bar. Its label shows the previous *historical* bar's closing time. After some time, this realtime bar closed, and the second highlighted bar opened. The label on the new realtime bar shows the *elapsed realtime* bar's closing time:

//@version=6 indicator("`time_close[1]` on non-time-based chart demo", overlay = true) //@variable A formatted "string" containing the date and time that `time_close[1]` represents in the exchange time zone. string formattedCloseTime = str.format_time(time_close[1], format = "'Date and time:' yyyy-MM-dd HH:mm:ss") //@variable A "string" containing the `time_close[1]` UNIX timestamp and the `formattedCloseTime`. string closeTimeText = str.format("Close timestamp of previous bar: {0,number,#}\n{1}", time_close[1], formattedCloseTime) // Define label colors for historical and realtime bars. color closeLabelColor = barstate.islast ? color.purple : color.aqua // Draw a label at the `high` to display the `closeTimeText` anchored to the bar's `time`. label.new(time, high, closeTimeText, xloc.bar_time, color = closeLabelColor, textcolor = color.white, size = size.large, textalign = text.align_left) // Highlight the background yellow on the realtime bar. bgcolor(barstate.islast ? #f3de22cb : na, title = "Realtime bar highlight")

Note that:

- A confirmed realtime bar is **not** the same as a historical bar. Pine's execution model uses separate data feeds for realtime and historical data. The closing time of a confirmed realtime bar is committed to the *realtime* feed, until the script re-executes on the chart. Only then will this bar's closing time load *historically* along with all the other closed bars.

- The barstate.islast value is `true` for all realtime bars in the dataset. Therefore, the elapsed realtime bar and the latest realtime bar both display a purple label and highlighted background. See the Bar states page to learn more about the different `barstate.*` variables in Pine Script.
- The time_close() function can similarly retrieve the previous bar's closing time on price-based charts using a `bars_back = 1` argument.

`time_tradingday`

The time_tradingday variable holds a UNIX timestamp representing the starting time (00:00 UTC) of the last trading day in the current bar's final session. It is helpful primarily for date and time calculations on *time-based* charts for symbols with overnight sessions that start and end on *different* calendar days.

On "1D" and lower timeframes, the time_tradingday timestamp corresponds to the beginning of the day when the current session *ends*, even for bars that open and close on the previous day. For example, the "Monday" session for "EURUSD" starts on Sunday at 17:00 and ends on Monday at 17:00 in the exchange time zone. The time_tradingday values of *all* intraday bars within the session represent Monday at 00:00 UTC.

On timeframes higher than "1D", which can cover *multiple* sessions, time_tradingday holds the timestamp representing the beginning of the last calendar day of the bar's *final* trading session. For example, on a "EURUSD, 1W" chart, the timestamp represents the start of the last trading day in the week, which is typically Friday at 00:00 UTC.

The script below demonstrates how the time_tradingday and time variables differ on Forex symbols. On each bar, it draws labels to display strings containing the variables' UNIX timestamps and formatted dates and times. It also uses the dayofmonth() function to calculate the UTC calendar day from both timestamps, highlighting the background when the calculated days do not match.

When applied to the "FXCM:EURUSD" chart with the "3h" ("180") timeframe, the script highlights the background of the *first bar* in each session, as each session opens on the *previous* calendar day. The dayofmonth() call that uses time calculates the opening day on the session's first bar, whereas the call that uses time_tradingday calculates the day when the session *ends*:

//@version=6 indicator("`time_tradingday` demo", overlay = true) //@variable A concatenated "string" containing the `time_tradingday` timestamp and a formatted representation in UTC. string tradingDayText = "`time_tradingday`: " + str.tostring(time_tradingday) + "\n" + "Date and time: " + str.format_time(time_tradingday, "dd MMM yyyy, HH:mm (z)", "UTC+0") //@variable A concatenated "string" containing the `time` timestamp and a formatted representation in UTC. string barOpenText = "`time`: " + str.tostring(time) + "\n" + "Date and time: " + str.format_time(time, "dd MMM yyyy, HH:mm (z)", "UTC+0") //@variable Is `true` on every even bar, `false` otherwise. This condition determines the appearance of the labels. bool isEven = bar_index % 2 == 0 // The `yloc` and `style` properties of the labels. They alternate on every other bar for visibility. labelYloc = isEven ? yloc.abovebar : yloc.belowbar labelStyle = isEven ? label.style_label_down : label.style_label_up // Draw alternating labels anchored to the bar's `time` to display the `tradingDayText` and `barOpenText`. if isEven label.new(time, 0, tradingDayText + "\n\n\n", xloc.bar_time, labelYloc, color.teal, labelStyle, color.white) label.new(time, 0, barOpenText, xloc.bar_time, labelYloc, color.maroon, labelStyle, color.white) else label.new(time, 0, "\n\n\n" + barOpenText, xloc.bar_time, labelYloc, color.maroon, labelStyle, color.white) label.new(time, 0, tradingDayText, xloc.bar_time, labelYloc, color.teal, labelStyle, color.white) //@variable The day of the month, in UTC, that the `time_tradingday` timestamp corresponds to. int tradingDayOfMonth = dayofmonth(time_tradingday,

"UTC+0") //@variable The day of the month, in UTC, that the `time` timestamp corresponds to. int openingDayOfMonth = dayofmonth(time, "UTC+0") // Highlight the background when the `tradingDayOfMonth` does not equal the `openingDayOfMonth`. bgcolor(tradingDayOfMonth != openingDayOfMonth ? color.rgb(174, 89, 243, 85) : na, title = "Different day highlight")

Note that:

- The str.format_time() and dayofmonth() calls use `"UTC+0"` as the `timezone` argument, meaning the results represent calendar time values with no offset from UTC. In the screenshot, the first bar opens at 21:00 UTC, 17:00 in the exchange time zone (“America/New_York”).
- The formatted strings show `"GMT"` as the acronym of the time zone, which is equivalent to `"UTC+0"` in this context.
- The time_tradingday value is the same for *all* three-hour bars within each session, even for the initial bar that opens on the previous UTC calendar day. The assigned timestamp changes only when a new session starts.

`timenow`

The timenow variable holds a UNIX timestamp representing the script’s *current time*. Unlike the values of other variables that hold UNIX timestamps, the values in the timenow series correspond to times when the script *executes*, not the times of specific bars or trading days.

A Pine script executes only *once* per historical bar, and all historical executions occur when the script first *loads* on the chart. As such, the timenow value is relatively consistent on historical bars, with only occasional millisecond changes across the series. In contrast, on realtime bars, a script executes once for *each new update* in the data feed, which can happen several times per bar. With each new execution, the timenow value updates on the latest bar to represent the current time.

> NoteBecause timenow updates only after script executions, its value does **not** always correspond to the *continuous* time displayed below the chart. When no new updates are available in the realtime data feed, a script on the chart remains *idle*, in which case the variable’s timestamp does not change.

This variable is most useful on realtime bars, where programmers can apply it to track the times of the latest script executions, count the time elapsed within open bars, control drawings based on bar updates, and more.

The script below inspects the value of timenow on the latest chart bars and uses it to analyze realtime bar updates. When the script first reaches the last chart bar, it declares three variables with the varip keyword to hold the latest timenow value, the total time elapsed between the bar’s updates, and the total number of updates. It uses these values to calculate the average number of milliseconds between updates, which it displays in a label along with the current execution’s timestamp, a formatted time and date in the exchange time zone, and the current number of bar updates:

//@version=6 indicator("`timenow` demo", overlay = true, max_labels_count = 500) if barstate.islast //@variable Holds the UNIX timestamp of the latest script execution for timing updates in the data feed. varip int lastUpdateTimestamp = timenow //@variable The total number of milliseconds elapsed across the bar's data updates. varip int totalUpdateTime = 0 //@variable The number of updates that have occurred on the current bar. varip int numUpdates = 0 // Add the time elapsed from the `lastUpdateTimestamp` to

`totalUpdateTime`, increase the `numUpdates` counter, and // update the `lastUpdateTimestamp` when the `timenow` value changes. if timenow != lastUpdateTimestamp totalUpdateTime += timenow - lastUpdateTimestamp numUpdates += 1 lastUpdateTimestamp := timenow //@variable The average number of milliseconds elapsed between the bar's updates. float avgUpdateTime = nz(totalUpdateTime / numUpdates) //@variable Contains the `timenow` value, a custom representation, and the `numUpdates` and `avgUpdateTime` values. string displayText = "`timenow`: " + str.tostring(timenow) + "\nTime and date (exchange): " + str.format_time(timenow, "HH:mm:ss MM/dd/yy") + "\nNumber of updates: " + str.tostring(numUpdates) + "\nAvg. time between updates: " + str.tostring(avgUpdateTime, "#.###") + " ms" //@variable The color of the label. Is blue when the bar is open, and gray after it closes. color labelColor = barstate.isconfirmed ? color.gray : color.blue //@variable The label's y-coordinate. Alternates between `high` and `low` on every other bar. float labelPrice = bar_index % 2 == 0 ? high : low //@variable The label's style. Alternates between "lower-right" and "upper-right" styles. labelStyle = bar_index % 2 == 0 ? label.style_label_lower_right : label.style_label_upper_right // Draw a `labelColor` label anchored to the bar's `time` to show the `displayText`. label.new(time, labelPrice, displayText, xloc.bar_time, color = labelColor, style = labelStyle, textcolor = color.white, size = size.large) // Reset the `totalUpdateTime` and `numUpdates` counters when the bar is confirmed (closes). if barstate.isconfirmed totalUpdateTime := 0 numUpdates := 0

Note that:

- When a bar is open, the drawn label is blue to signify that additional updates can occur. After the bar closes, the final label's color is gray.
- Although we've set the chart time zone to match the exchange time zone, the formatted time in the open bar's label and the time shown below the chart *do not* always align. The script records a new timestamp only when a *new execution* occurs, whereas the time below the chart updates *continuously*.
- The varip keyword specifies that a variable does not revert to the last committed value in its series when new updates occur. This behavior allows the script to use variables to track changes in timenow on an open bar.
- Updates to timenow on open realtime bars do not affect the recorded timestamps on confirmed bars as the script executes. However, the historical series changes (*repaints*) after reloading the chart because timenow references the script's *current time*, not the times of specific bars.

Calendar-based variables

The year, month, weekofyear, dayofmonth, dayofweek, hour, minute, and second variables hold *calendar-based* "int" values calculated from the current bar's *opening time*, expressed in the exchange time zone. These variables reference the same values that calendar-based functions return when they use the default `timezone` argument and time as the `time` argument. For instance, the year variable holds the same value that a `year(time)` call returns.

Programmers can use these calendar-based variables for several purposes, such as:

- Identifying a bar's opening date and time.
- Passing the variables to the timestamp() function to calculate UNIX timestamps.
- Testing when date/time values or ranges occur in a data feed.

One of the most common use cases for these variables is checking for date or time ranges to control when a script displays visuals or executes calculations. This simple example inspects the year variable to determine

when to plot a visible value. If the year is 2022 or higher, the script plots the bar's close. Otherwise, it plots na:

//@version=6 indicator("`year` demo", overlay = true) // Plot the `close` price on bars that open in the `year` 2022 onward. Otherwise, plot `na` to display nothing. plot(year >= 2022 ? close : na, "`close` price (year 2022 and later)", linewidth = 3)

When using these variables in conditions that isolate specific dates or times rather than ranges, it's crucial to consider that certain conditions might not detect some occurrences of the values due to a chart's timeframe, the opening times of chart bars, or the symbol's active session.

For instance, suppose we want to detect when the first calendar day of each month occurs on the chart. Intuitively, one might consider simply checking when the dayofmonth value equals 1. However, this condition only identifies when a bar *opens* on a month's first day. The bars on some charts can open and close in *different* months. Additionally, a chart bar might not contain the first day of a month if the market is *closed* on that day. Therefore, we must create extra conditions that work in these scenarios to identify the first day in *any* month on the chart.

The script below uses the dayofmonth and month variables, and the month() function, to create three conditions that detect the first day of the month in different ways. The first condition detects if the bar opens on the first day, the second checks if the bar opens in one month and closes in another, and the third checks if the chart skips the date entirely. The script draws labels showing bar opening dates and highlights the background with different colors to visualize when each condition occurs:

//@version=6 indicator("Detecting the first day of the month demo", overlay = true, max_labels_count = 500) //@variable Is `true` only if the current bar opens on the first day of the month in exchange time, `false` otherwise. bool opensOnFirst = dayofmonth == 1 //@variable Is `true` if the bar opens in one month and closes in another, meaning its time span includes the first day. bool containsFirst = month != month(time_close) //@variable Is `true` only if the bar opens in a new month and the current or previous bar does not cover the first day. bool skipsFirst = month != month[1] and not (opensOnFirst or containsFirst[1]) //@variable The name of the current bar's opening weekday. string weekdayName = switch dayofweek dayofweek.sunday => "Sunday" dayofweek.monday => "Monday" dayofweek.tuesday => "Tuesday" dayofweek.wednesday => "Wednesday" dayofweek.thursday => "Thursday" dayofweek.friday => "Friday" dayofweek.saturday => "Saturday" //@variable A custom "string" representing the bar's opening date, including the weekday name. string openDateText = weekdayName + str.format_time(time, ", MMM d, yyyy") // Draw a green label when the bar opens on the first day of the month. if opensOnFirst string labelText = "Bar opened on\n" + openDateText label.new(time, open, labelText, xloc.bar_time, color = color.green, textcolor = color.white, size = size.large) // Draw a blue label when the bar opens and closes in different months. if containsFirst string labelText = "Bar includes the first day,\nbut opened on\n" + openDateText label.new(time, open, labelText, xloc.bar_time, color = color.blue, textcolor = color.white, size = size.large) // Draw a red label when the chart skips the first day of the month. if skipsFirst string labelText = "Chart doesn't include the first day.\nBar opened on\n" + openDateText label.new(time, open, labelText, xloc.bar_time, color = color.red, textcolor = color.white, size = size.large) // Highlight the background when the conditions occur. bgcolor(opensOnFirst ? color.new(color.green, 70) : na, title = "`opensOnFirst` condition highlight") bgcolor(containsFirst ? color.new(color.blue, 70) : na, title = "`containsFirst` condition highlight") bgcolor(skipsFirst ? color.new(color.red, 70) : na, title = "`skipsFirst` condition highlight")

Note that:

- The script calls the month() function with time_close as the `time` argument to calculate each bar's *closing* month for the `containsFirst` condition.
- The `dayofweek.*` namespace contains variables that hold each possible dayofweek value, e.g., dayofweek.sunday has a constant value of 1 and dayofweek.saturday has a constant value of 7. The script compares dayofweek to these variables in a switch structure to determine the weekday name shown inside each label.
- To detect the *first opening time* in a monthly timeframe, not strictly the first day in a calendar month, use `ta.change(time("1M")) > 0` or `timeframe.change("1M")` instead of conditions based on these variables. See the Testing for changes in higher timeframes section to learn more.

`last_bar_time`

The last_bar_time variable holds a UNIX timestamp representing the *last* available bar's opening time. It is similar to last_bar_index, which references the latest bar index. On historical bars, last_bar_time consistently references the time value of the last bar available when the script first *loads* on the chart. The only time the variable's value updates across script executions is when a new realtime bar opens.

The following script uses the last_bar_time variable to get the opening timestamp of the last chart bar during its execution on the *first bar*. It displays the UNIX timestamp and a formatted date and time using a single-cell table created only on that bar. When the script executes on the last available bar, it creates a label to show the bar's time value and its formatted representation for visual comparison.

As the chart below shows, both drawings display *identical* times, verifying that last_bar_time correctly references the last bar's time value on previous historical bars:

//@version=6 indicator("`last_bar_time` demo", overlay = true) if barstate.isfirst //@variable A single-cell `table`, created only on the *first* bar, showing the *last* available bar's opening time. table displayTable = table.new(position.bottom_right, 1, 1, color.aqua) //@variable A "string" containing the `last_bar_time` UNIX timestamp and a custom date and time representation. string lastBarTimeText = "`last_bar_time`: " + str.tostring(last_bar_time) + "\nDate and time (exchange): " + str.format_time(last_bar_time, "dd/MM/yy HH:mm:ss") // Initialize the `displayTable` cell with the `lastBarTimeText`. displayTable.cell(0, 0, lastBarTimeText, text_color = color.white, text_size = size.large, text_halign = text.align_left) //@variable Is `true` only on the first occurrence of `barstate.islast`, `false` otherwise. // This condition occurs on the bar whose `time` the `last_bar_time` variable refers to on historical bars. bool isInitialLastBar = barstate.islast and not barstate.islast[1] if isInitialLastBar //@variable A "string" containing the last available bar's `time` value and a custom date and time representation. // Matches the `lastBarTimeText` from the first bar because `last_bar_time` equals this bar's `time`. string openTimeText = "`time`: " + str.tostring(time) + "\nDate and time (exchange): " + str.format_time(time, "dd/MM/yy HH:mm:ss") // Draw a label anchored to the bar's `time` to display the `openTimeText`. label.new(time, high, openTimeText, xloc.bar_time, color = color.purple, textcolor = color.white, size = size.large, textalign = text.align_left) // Highlight the background when `isInitialLastBar` is `true` for visual reference. bgcolor(barstate.islast ? color.rgb(155, 39, 176, 80) : na, title = "Initial last bar highlight")

Note that:

- The script creates the label only on the *first* bar with the barstate.islast state because that bar's time value is what last_bar_time equals on all historical bars. On subsequent bars, the last_bar_time value *updates* to represent the latest realtime bar's opening time.

- Updates to last_bar_time on realtime bars do not affect the values on historical bars as the script executes. However, the variable's series *repaints* when the script restarts because last_bar_time always references the latest available bar's opening time.
- This script expresses dates using the `"dd/MM/yy"` format, meaning the two-digit day appears before the two-digit month, and the month appears before the two-digit representation of the year. See this section below for more information.

Visible bar times

The chart.left_visible_bar_time and chart.right_visible_bar_time variables reference the opening UNIX timestamps of the chart's leftmost (first) and rightmost (last) *visible bars* on every script execution. When a script uses these variables, it responds dynamically to visible chart changes, such as users scrolling across bars or zooming in/out. Each time the visible window changes, the script *re-executes* automatically to update the variables' values on all available bars.

The example below demonstrates how the chart.left_visible_bar_time and chart.right_visible_bar_time variables work across script executions. The script draws labels anchored to the visible bars' times to display the UNIX timestamps. In addition, it draws two single-cell tables showing corresponding dates and times in the standard ISO 8601 format. The script creates these drawings only when it executes on the first bar. As the script continues to execute on subsequent bars, it identifies each bar whose time value equals either visible bars' timestamp and colors it on the chart:

```
//@version=6 indicator("Visible bar times demo", overlay = true) // Create strings on the first chart bar that contain the first and last visible bars' UNIX timestamps. var string leftTimestampText = str.format("UNIX timestamp: {0,number,#}", chart.left_visible_bar_time) var string rightTimestampText = str.format("UNIX timestamp: {0,number,#}", chart.right_visible_bar_time) // Create strings on the first bar that contain the exchange date and time of the visible bars in ISO 8601 format. var string leftDateTimeText = "Date and time: " + str.format_time(chart.left_visible_bar_time) var string rightDateTimeText = "Date and time: " + str.format_time(chart.right_visible_bar_time) //@variable A `label` object anchored to the first visible bar's time. Shows the `leftTimestampText`. var label leftTimeLabel = label.new( chart.left_visible_bar_time, 0, leftTimestampText, xloc.bar_time, yloc.abovebar, color.purple, label.style_label_lower_left, color.white, size.large ) //@variable A `label` object anchored to the last visible bar's time. Shows the `rightTimestampText`. var label rightTimeLabel = label.new( chart.right_visible_bar_time, 0, rightTimestampText, xloc.bar_time, yloc.abovebar, color.teal, label.style_label_lower_right, color.white, size.large ) //@variable A single-cell `table` object showing the `leftDateTimeText`. var table leftTimeTable = table.new(position.middle_left, 1, 1, color.purple) //@variable A single-cell `table` object showing the `rightDateTimeText`. var table rightTimeTable = table.new(position.middle_right, 1, 1, color.teal) // On the first bar, initialize the `leftTimeTable` and `rightTimeTable` with the corresponding date-time text. if barstate.isfirst leftTimeTable.cell(0, 0, leftDateTimeText, text_color = color.white, text_size = size.large) rightTimeTable.cell(0, 0, rightDateTimeText, text_color = color.white, text_size = size.large) //@variable Is purple at the left visible bar's opening time, teal at the right bar's opening time, `na` otherwise. color barColor = switch time chart.left_visible_bar_time => color.purple chart.right_visible_bar_time => color.teal // Color the leftmost and rightmost visible bars using the `barColor`. barcolor(barColor, title = "Leftmost and rightmost visible bar color")
```

Note that:

- The chart.left_visible_bar_time and chart.right_visible_bar_time values are consistent across all executions, which allows the script to identify the visible bars' timestamps on the *first* bar and check

when the time value equals them. The script restarts on any chart window changes, updating the variables' series to reference the new timestamps on every bar.
- The str.format_time() function uses ISO 8601 format by default when the call does not include a `format` argument because it is the *international standard* for expressing dates and times. See the Formatting dates and times section to learn more about time string formats.

`syminfo.timezone`

The syminfo.timezone variable holds a time zone string representing the current symbol's *exchange* time zone. The "string" value expresses the time zone as an *IANA identifier* (e.g., `"America/New_York"`). The overloads of time functions that include a `timezone` parameter use syminfo.timezone as the default argument.

Because this variable is the default `timezone` argument for all applicable time function overloads, it is unnecessary to use as an explicit argument, except for stylistic purposes. However, programmers can use the variable in other ways, such as:

- Displaying the "string" in Pine Logs or drawings to inspect the exchange time zone's IANA identifier.
- Comparing the value to other time zone strings to create time zone-based conditional logic.
- Requesting the exchange time zones of other symbols with `request.*()` function calls.

The following script uses the timenow variable to retrieve the UNIX timestamp of its latest execution. It formats the timestamp into date-time strings expressed in the main symbol's exchange time zone and a requested symbol's exchange time zone, which it displays along with the IANA identifiers in a table on the last chart bar:

//@version=6 indicator("`syminfo.timezone` demo", overlay = true) //@variable The symbol to request exchange time zone information for. string symbolInput = input.symbol("NSE:BANKNIFTY", "Requested symbol") //@variable A `table` object displaying the exchange time zone and the time of the script's execution. var table t = table.new(position.bottom_right, 2, 3, color.yellow, border_color = color.black, border_width = 1) //@variable The IANA identifier of the exchange time zone requested for the `symbolInput` symbol. var string requestedTimezone = na if barstate.islastconfirmedhistory // Retrieve the time zone of the user-specified symbol's exchange. requestedTimezone := request.security(symbolInput, "", syminfo.timezone, calc_bars_count = 2) // Initialize the `t` table's header cells. t.cell(0, 0, "Exchange prefix and time zone string", text_size = size.large) t.cell(1, 0, "Last execution date and time", text_size = size.large) t.cell(0, 1, syminfo.prefix(syminfo.tickerid) + " (" + syminfo.timezone + ")", text_size = size.large) t.cell(0, 2, syminfo.prefix(symbolInput) + " (" + requestedTimezone + ")", text_size = size.large) if barstate.islast //@variable The formatting string for all `str.format_time()` calls. var string formatString = "HH:mm:ss 'on' MMM dd, YYYY" // Initialize table cells to display the formatted text. t.cell(1, 1, str.format_time(timenow, formatString), text_size = size.large) t.cell(1, 2, str.format_time(timenow, formatString, requestedTimezone), text_size = size.large)

Note that:

- Pine scripts execute on realtime bars only when new updates occur in the data feed, and timenow updates only on script executions. As such, when no realtime updates are available, the timenow timestamp does not change. See this section above for more information.
- The default `symbolInput` value is `"NSE:BANKNIFTY"`. NSE is in the "Asia/Kolkata" time zone, which is 9.5 hours ahead of the main symbol's exchange time zone ("America/New_York") at the time of the

screenshot. Although the *local* time representations differ, both refer to the same *absolute* time that the timenow timestamp represents.
- Pine v6 scripts use dynamic `request.*()` calls by default, which allows the script to call request.security() dynamically inside the if structure's *local scope*. See the Dynamic requests section of the Other timeframes and data page to learn more.

Time functions

Pine Script features several built-in functions that scripts can use to retrieve, calculate, and express time values:

- The time() and time_close() functions allow scripts to retrieve UNIX timestamps for the opening and closing times of bars within a session on a specified timeframe, without requiring `request.*()` function calls.
- The year(), month(), weekofyear(), dayofmonth(), dayofweek(), hour(), minute(), and second() functions calculate calendar-based values, expressed in a specified time zone, from a UNIX timestamp.
- The timestamp() function calculates a UNIX timestamp from a specified calendar date and time.
- The str.format_time() function formats a UNIX timestamp into a human-readable date/time "string", expressed in a specified time zone. The Formatting dates and times section below provides detailed information about formatting timestamps with this function.
- The input.time() function returns a UNIX timestamp corresponding to the user-specified date and time, and the input.session() function returns a valid time-based session string corresponding to the user-specified start and end times. See the Time input and Session input sections of the Inputs page to learn more about these functions.

`time()` and `time_close()` functions

The time() and time_close() functions return UNIX timestamps representing the opening and closing times of bars on a specified timeframe. Both functions can *filter* their returned values based on a given session in a specific time zone. They each have the following signatures:

```
functionName(timeframe, session, bars_back, timeframe_bars_back) → series int
functionName(timeframe, session, timezone, bars_back, timeframe_bars_back) → se
```

Where:

- `functionName` is the function's identifier.
- The `timeframe` parameter accepts a timeframe string that defines the timeframe for the calculation. The function uses the script's main timeframe if the argument is timeframe.period or an empty string `""`.
- The optional `session` parameter accepts a time-based session string that defines the session's start and end times (e.g., `"0930-1600"`) and the days for which it applies (e.g., `":23456"` means Monday - Friday). If the value does not specify the days, the session applies to *all* trading days automatically. The function returns UNIX timestamps only for the bars *within* the session. It returns na if a bar's time is *outside* the session. If the `session` argument is an empty string or not specified, the function uses the symbol's session information.

- The optional `timezone` parameter accepts a valid time zone string that defines the time zone of the specified `session`. It does **not** change the meaning of returned UNIX timestamps, as they are *time zone-agnostic*. If the `timezone` argument is not specified, the function uses the exchange time zone (syminfo.timezone).
- The optional `bars_back` parameter accepts an "int" value specifying the bar offset on the script's main timeframe. If the value is positive, the function finds the bar that is N bars before the current bar on the main timeframe, then retrieves the timestamp of the corresponding bar on the timeframe specified by the `timeframe` argument. If the value is a negative number from -1 to -500, the function calculates the expected timestamp of the `timeframe` bar corresponding to N bars after the current bar on the main timeframe. The default is 0. See the Calculating timestamps at bar offsets section to learn more about this parameter.
- The optional `timeframe_bars_back` parameter accepts an "int" value specifying the additional bar offset on the timeframe specified by the `timeframe` argument. If the value is positive, the function retrieves the timestamp of the `timeframe` bar that is N `timeframe` bars before the one corresponding to the `bars_back` offset. If the value is a negative number from -1 to -500, the function calculates the expected timestamp of the `timeframe` bar that is N `timeframe` bars after the one corresponding to the `bars_back` offset. The default is 0. See the Calculating timestamps at bar offsets section to learn more about this parameter.

Similar to the time and time_close variables, these functions behave differently on *time-based* and *non-time-based* charts.

Time-based charts have bars that open and close at *predictable* times, whereas the bars on tick charts and all non-standard charts, excluding Heikin Ashi, open and close at irregular, *unpredictable* times. Consequently, time_close() cannot calculate the expected closing time of an open realtime bar on non-time-based charts, so it returns na on that bar. Similarly, the time() function with a negative bar offset cannot accurately calculate the expected opening time of a future realtime bar on these charts. See the *second example* in this section above. That example script exhibits the same behavior on a price-based chart if it uses a `time_close("")` call instead of the time_close variable.

On non-time-based charts, a new bar's closing time is available in the realtime data feed once the bar closes. Therefore, time_close() can retrieve valid closing timestamps for *confirmed realtime bars* without needing to restart the script to load them historically. See the *third example* in this section above. Replacing the `time_close[1]` variable in the example script with a `time_close("", 1)` call achieves the same result to retrieve the closing time of the elapsed realtime bar.

Typical use cases for the time() and time_close() functions include:

- Testing for bars that open or close in specific sessions defined by the `session` and `timezone` parameters.
- Testing for changes or measuring time differences on specified higher timeframes.
- Calculating timestamps at bar offsets on the script's main timeframe, a specified timeframe, or both.

Testing for sessions

The time() and time_close() functions' `session` and `timezone` parameters define the sessions for which they can return *non-na* values. If a call to either function references a bar that opens/closes within the defined session in a given time zone, it returns a UNIX timestamp for that bar. Otherwise, it returns na. Programmers

can pass the returned values to the na() function to identify which bars open or close within specified intervals, which is helpful for session-based calculations and logic.

This simple script identifies when a bar on the chart's timeframe opens at or after 11:00 and before 13:00 in the exchange time zone on any trading day. It calls time() with timeframe.period as the `timeframe` argument and the `"1100-1300"` session string as the `session` argument, and then verifies whether the returned value is na with the na() function. When the value is **not** na, the script highlights the chart's background to indicate that the bar opened in the session:

//@version=6 indicator("Testing for session bars demo", overlay = true) //@variable Checks if the bar opens between 11:00 and 13:00. Is `true` if the `time()` function does not return `na`. bool inSession = not na(time(timeframe.period, "1100-1300")) // Highlight the background of the bars that are in the session "11:00-13:00". bgcolor(inSession ? color.rgb(155, 39, 176, 80) : na)

Note that:

- The `session` argument in the time() call represents an interval in the *exchange* time zone because syminfo.timezone is the default `timezone` argument.
- The session string expresses the start and end times in the `"HHmm-HHmm"` format, where `"HH"` is the two-digit *hour* and `"mm"` is the two-digit *minute*. Session strings can also specify the *weekdays* a session applies to. However, the time() call's `session` argument (`"1100-1300"`) does not include this information, which is why it considers the session valid for *every* day. See the Sessions page to learn more.

When using session strings in time() and time_close() calls, it's crucial to understand that such strings define start and end times in a *specific* time zone. The local hour and minute in one region may not correspond to the same point in UNIX time as that same hour and minute in another region. Therefore, calls to these functions with different `timezone` arguments can return non-na timestamps at *different* times, as the specified time zone string changes the meaning of the local times represented in the `session` argument.

This example demonstrates how the `timezone` parameter affects the `session` parameter in a time() function call. The script calculates an `opensInSession` condition that uses a time() call with arguments based on inputs. The session input for the `session` argument includes four preset options: `"0000-0400"`, `"0930-1400"`, `"1300-1700"`, and `"1700-2100"`. The string input that defines the `timezone` argument includes four IANA time zone options representing different offsets from UTC: `"America/Vancouver"` (UTC-7/-8), `"America/New_York"` (UTC-4/-5), `"Asia/Dubai"` (UTC+4), and `"Austrailia/Sydney"` (UTC+10/+11).

For any chosen `sessionInput` value, changing the `timezoneInput` value changes the specified session's time zone. The script highlights *different bars* with each time zone choice because, unlike UNIX timestamps, the *absolute* times that local hour and minute values correspond to *varies* across time zones:

//@version=6 indicator("Testing session time zones demo", overlay = true) //@variable The timeframe of the analyzed bars. string timeframeInput = input.timeframe("60", "Timeframe") //@variable The session to check. Features four interval options. string sessionInput = input.session("1300-1700", "Session", ["0000-0400", "0930-1400", "1300-1700", "1700-2100"]) //@variable An IANA identifier representing the time zone of the `sessionInput`. Fetures four preset options. string timezoneInput = input.string("America/New_York", "Time zone", ["America/Vancouver", "America/New_York", "Asia/Dubai", "Australia/Sydney"]) //@variable Is `true` if a `timeframeInput` bar opens within the `sessionInput` session in the `timezoneInput` time zone. // The condition detects the session on different bars, depending on the chosesn time zone, because identical // local times in different time zones refer to different absolute points in UNIX time. bool opensInSession = not

na(time(timeframeInput, sessionInput, timezoneInput)) // Highlight the background when `opensInSession` is `true`. bgcolor(opensInSession ? color.rgb(33, 149, 243, 80) : na, title = "Open in session highlight")

Note that:

- This script uses *IANA notation* for all time zone strings because it is the recommended format. Using an IANA identifier allows the time() call to automatically adjust the session's UTC offset based on a region's local time policies, such as daylight saving time.

Testing for changes in higher timeframes

The `timeframe` parameter of the time() and time_close() functions specifies the timeframe of the bars in the calculation, allowing scripts to retrieve opening/closing UNIX timestamps from *higher timeframes* than the current chart's timeframe without requiring `request.*()` function calls.

Programmers can use the opening/closing timestamps from higher-timeframe (HTF) bars to detect timeframe changes. One common approach is to call time() or time_close() with a consistent `timeframe` argument across all executions on a time-based chart and measure the one-bar change in the returned value with the ta.change() function. The result is a *nonzero* value only when an HTF bar opens. One can also check whether the data has a time gap at that point by comparing the time() value to the previous bar's time_close() value. A gap is present when the opening timestamp on the current bar is greater than the closing timestamp on the previous bar.

The script below uses the call `time("1M")` to get the opening UNIX timestamp of the current bar on the "1M" timeframe, then assigns the result to the `currMonthlyOpenTime` variable. It detects when bars on that timeframe open by checking when the one-bar change in the `currMonthlyOpenTime` value is above zero. On each occurrence of the condition, the script detects whether the HTF bar opened after a gap by checking if the new opening timestamp is greater than the "1M" closing timestamp from the previous chart bar (`time_close("1M")[1]`).

The script draws labels containing formatted "1M" opening times to indicate the chart bars that mark the start of monthly bars. If a monthly bar opens without a gap from the previous closing time, the script draws a blue label. If a monthly bar starts after a gap, it draws a red label. Additionally, if the "1M" opening time does not match the opening time of the chart bar, the script displays that bar's formatted time in the label for comparison:

//@version=6 indicator("Detecting changes in higher timeframes demo", overlay = true) //@variable The opening UNIX timestamp of the current bar on the "1M" timeframe. int currMonthlyOpenTime = time("1M") //@variable The closing timestamp on the "1M" timeframe as of the previous chart bar. int prevMonthlyCloseTime = time_close("1M")[1] //@variable Is `true` when the opening time on the "1M" timeframe changes, indicating a new monthly bar. bool isNewTf = ta.change(currMonthlyOpenTime) > 0 if isNewTf // Initialize variables for the `text` and `color` properties of the label drawing. string lblText = "New '1M' opening time:\n" + str.format_time(currMonthlyOpenTime) color lblColor = color.blue //@variable Is `true` when the `currMonthlyOpenTime` exceeds the `prevMonthlyCloseTime`, indicating a time gap. bool hasGap = currMonthlyOpenTime > prevMonthlyCloseTime // Modify the `lblText` and `lblColor` based on the `hasGap` value. if hasGap lblText := "Gap from previous '1M' close.\n\n" + lblText lblColor := color.red // Include the formatted `time` value if the `currMonthlyOpenTime` is before the first available chart bar's `time`. if time > currMonthlyOpenTime lblText += "\nFirst chart bar has a later time:\n" + str.format_time(time) // Draw a `lblColor` label anchored to the `time` to display the `lblText`. label.new(time, high, lblText,

xloc.bar_time, color = lblColor, style = label.style_label_lower_right, textcolor = color.white, size = size.large
)

Note that:

- Using ta.change() on a time() or time_close() call's result is not the *only* way to detect changes in a higher timeframe. The timeframe.change() function is an equivalent, more convenient option for scripts that do not need to *use* the UNIX timestamps from HTF bars in other calculations, as it returns a "bool" value directly without extra code.
- The detected monthly opening times do not always correspond to the first calendar day of the month. Instead, they correspond to the first time assigned to a "1M" bar, which can be *after* the first calendar day. For symbols with overnight sessions, such as "USDJPY" in our example chart, a "1M" bar can also open *before* the first calendar day.
- Sometimes, the opening time assigned to an HTF bar might *not* equal the opening time of any chart bar, which is why other conditions such as `time == time("1M")` cannot detect new monthly bars consistently. For example, on our "USDJPY" chart, the "1M" opening time `2023-12-31T17:00:00-0500` does not match an opening time on the "1D" timeframe. The first available "1D" bar after that point opened at `2024-01-01T17:00:00-0500`.

Calculating timestamps at bar offsets

The `bars_back` and `timeframe_bars_back` parameters of the time() and time_close() functions control *bar offsets* in the calculations, enabling the functions to compute the opening/closing UNIX timestamps for bars that are *before* or *after* the current bar, on a given timeframe, without requiring `request.*()` calls or history-referencing operations.

The `bars_back` parameter controls the bar offset on the script's **main timeframe**, and the `timeframe_bars_back` parameter controls the bar offset on the **separate timeframe** specified by the `timeframe` argument. A call to either function evaluates *both* offsets in succession to determine the returned timestamp:

1. The call evaluates the `bars_back` offset. If the value is a positive number from 1 to 5000, the call finds the bar that is the specified number of bars *before* the current one on the main timeframe, then retrieves the time of the corresponding `timeframe` bar. If the value is a negative number from -1 to -500, the call calculates the *expected* time of the `timeframe` bar corresponding to N main-timeframe bars *after* the current bar. If the value is 0 (default), the call retrieves the time of the *current* `timeframe` bar.

2. The call evaluates the `timeframe_bars_back` offset for the specified timeframe to determine the final timestamp. If the value is positive and less than or equal to 5000, the call returns the timestamp for the `timeframe` bar that is the specified number of periods *before* the one corresponding to the `bars_back` offset. If the value is negative and greater than or equal to -500, the call returns the *expected* timestamp of the `timeframe` bar that is N periods *after* the one corresponding to the `bars_back` offset. If the value is 0 (default), the call does not apply an additional timeframe-based offset to the calculation.

The following example shows how the `bars_back` and `timeframe_bars_back` parameters work together in timestamp calculations. The script below creates three drawings on the last historical bar, anchored to the

results from different time() and time_close() calls that return timestamps based on a specified chart bar offset (`chartOffsetInput`) and a higher-timeframe (HTF) offset (`tfOffsetInput`):

- The first drawing is a vertical line indicating the chart bar at the `bars_back` offset (`chartOffsetInput`), which the other drawings use in their timestamp calculations. The line uses the value from `time("", bars_back = chartOffsetInput)` as the `x1` and `x2` coordinates.
- The second is a blue box showing the range of the higher-timeframe (HTF) bar that *contains* the offset chart bar indicated by the line. It uses the value of `time(tfInput, bars_back = chartOffsetInput)` as the `left` coordinate, and the value of `time_close(tfInput, bars_back = chartOffsetInput)` as the `right` coordinate.
- The third is a purple box that shows the range of the HTF bar that is `tfOffsetInput` periods *away* from the HTF bar represented by the blue box. The time() and time_close() calls for this box use the same `timeframe` and `bars_back` arguments as those for the blue box, and they include `timeframe_bars_back = tfOffsetInput` to apply the additional HTF bar offset.

Both boxes also display formatted opening and closing timestamps, expressed in the exchange time zone, to show the time ranges covered by their respective periods:

//@version=6 indicator("Calculating timestamps at bar offsets demo", overlay = true, behind_chart = false) //@variable Sets the timeframe for the box timestamp calculations. string tfInput = input.timeframe("1D", "Higher timeframe") //@variable The offset on the script's main timeframe (chart timeframe in this example). int chartOffsetInput = input.int(0, "Chart bar offset", -500) //@variable The offset on the higher timeframe specified by `tfInput`. int tfOffsetInput = input.int(1, "HTF offset", -500) // Create persistent arrays to store opening time, high, and low values for the chart drawings. var array<int> times = array.new<int>() var array<float> highs = array.new<float>() var array<float> lows = array.new<float>() //@variable The opening time of the current bar on the `tfInput` timeframe, with no offset. int tfOpen0 = time(tfInput) switch // Add a new element to the end of all three arrays if the `tfOpen0` value changes, indicating a new HTF bar. ta.change(tfOpen0) > 0 => highs.push(high), lows.push(low), times.push(tfOpen0) // Otherwise, set the last element in the `highs` and `lows` arrays to track the period's highest and lowest values. times.size() > 0 => highs.set(-1, math.max(highs.last(), high)), lows.set(-1, math.min(lows.last(), low)) if barstate.islastconfirmedhistory and times.size() > 0 // Get the opening timestamp for the *chart bar* that is `chartOffsetInput` bars back, relative to the current bar. int chartTFOpen = time("", bars_back = chartOffsetInput) // Draw a line to show the offset bar's position. line.new(chartTFOpen, low[math.max(chartOffsetInput, 0)], chartTFOpen, high[math.max(chartOffsetInput, 0)], xloc.bar_time, extend.both, width = 3) // Get the timestamps of the HTF bar that *contains* the chart bar at `chartOffsetInput`. int tfOpen1 = time(tfInput, bars_back = chartOffsetInput) int tfClose1 = time_close(tfInput, bars_back = chartOffsetInput) // Get the `times` index for `tfOpen1` to retrieve the first offset HTF bar's high and low values. int offsetIndex1 = times.binary_search_leftmost(tfOpen1) // Draw a box to show the HTF bar that opens at `tfOpen1` and closes at `tfClose1`. box.new(tfOpen1, highs.get(offsetIndex1), tfClose1, lows.get(offsetIndex1), color.blue, xloc = xloc.bar_time, bgcolor = #2196f399, text_color = chart.fg_color, text = "Open: " + str.format_time(tfOpen1) + "\n\nClose: " + str.format_time(tfClose1)) // Get the timestamps of the HTF bar that is `tfOffsetInput` *HTF periods back*, relative to the one that // opens at `tfOpen1` and closes at `tfClose1`. int tfOpen2 = time(tfInput, bars_back = chartOffsetInput, timeframe_bars_back = tfOffsetInput) int tfClose2 = time_close(tfInput, bars_back = chartOffsetInput, timeframe_bars_back = tfOffsetInput) // Get the `times` index for `tfOpen2` to retrieve the second offset HTF bar's high and low values. int offsetIndex2 = times.binary_search_leftmost(tfOpen2) // Draw a box to show the HTF bar that opens at `tfOpen2` and closes at `tfClose2`. box.new(tfOpen2, highs.get(offsetIndex2), tfClose2, lows.get(offsetIndex2), color.purple, xloc = xloc.bar_time, bgcolor = #9c27b099, text_color = chart.fg_color,

text = "Open: " + str.format_time(tfOpen2) + "\n\nClose: " + str.format_time(tfClose2)) // Color the background of the last historical bar for visual reference. bgcolor(barstate.islastconfirmedhistory ? color.new(chart.fg_color, 80) : na, title = "Last historical bar highlight")

Note that:

- The script tracks high, low, and opening time values for each HTF period in arrays to determine the price ranges of the boxes. The arrays contain one element for each successive change in the value of `time(tfInput)`. The script searches through the `times` array and finds the indices of the earliest elements that match the calculated HTF opening times, then uses the `highs` and `lows` array elements at those indices to set the top and bottom edges of the boxes.

In the image below, we applied the script to a chart with the “240” (4 hour) timeframe, using the default settings. The vertical line is at the last historical bar because the “Chart bar offset” input’s value is 0, and the blue box displays the HTF range that contains that chart bar. The purple box shows the range that is one HTF period *before* the blue box, because the “HTF offset” input’s value is 1:

If we change the chart bar offset to -4, the vertical line moves four bars *forward* on the chart. In the following image, that chart bar belongs to the *next* HTF period, not the current one. Therefore, the blue box now displays the *expected* time range of the future HTF bar. With the same HTF offset of 1, the purple box now shows the *current* HTF bar’s range, because its timestamps are always one HTF period behind those for the blue box:

If we increase the HTF offset to 3, the distance from the purple box to the blue box is three HTF periods, regardless of how we change the chart bar offset. In this image, the purple box now shows the range from two HTF bars before the current one, because our chart offset still refers to the upcoming HTF bar:

Calendar-based functions

The year(), month(), weekofyear(), dayofmonth(), dayofweek(), hour(), minute(), and second() functions calculate *calendar-based* “int” values from a UNIX timestamp. Unlike the calendar-based variables, which always hold exchange calendar values based on the current bar’s opening timestamp, these functions can return calendar values for any valid timestamp and express them in a chosen time zone.

Each of these calendar-based functions has the following two signatures:

```
functionName(time) → series int
functionName(time, timezone) → series int
```

Where:

- `functionName` is the function’s identifier.
- The `time` parameter accepts an “int” UNIX timestamp for which the function calculates a corresponding calendar value.
- The `timezone` parameter accepts a time zone string specifying the returned value’s time zone. If the `timezone` argument is not specified, the function uses the exchange time zone (syminfo.timezone).

In contrast to the functions that return UNIX timestamps, a calendar-based function returns different “int” results for various time zones, as calendar values represent parts of a *local time* in a *specific region*.

For instance, the simple script below uses two calls to dayofmonth() to calculate each bar's opening day in the exchange time zone and the "Australia/Sydney" time zone. It plots the results of the two calls in a separate pane for comparison:

//@version=6 indicator("`dayofmonth()` demo", overlay = false) //@variable An "int" representing the current bar's opening calendar day in the exchange time zone. // Equivalent to the `dayofmonth` variable. int openingDay = dayofmonth(time) //@variable An "int" representing the current bar's opening calendar day in the "Australia/Sydney" time zone. int openingDaySydney = dayofmonth(time, "Australia/Sydney") // Plot the calendar day values. plot(openingDay, "Day of Month (Exchange)", linewidth = 6, color = color.blue) plot(openingDaySydney, "Day of Month (Sydney)", linewidth = 3, color = color.orange)

Note that:

- The first dayofmonth() call calculates the bar's opening day in the exchange time zone because it does not include a `timezone` argument. This call returns the same value that the dayofmonth variable references.
- Our example symbol's exchange time zone is "America/New_York", which follows UTC-5 during standard time and UTC-4 during daylight saving time (DST). The "Australia/Sydney" time zone follows UTC+10 during standard time and UTC+11 during DST. However, Sydney observes DST at *different* times of the year than New York. As such, its time zone is 14, 15, or 16 hours ahead of the exchange time zone, depending on the time of year. The plots on our "1D" chart diverge when the difference is at least 15 hours because the bars open at 09:30 in exchange time, and 15 hours ahead is 00:30 on the *next* calendar day.

It's important to understand that although the `time` argument in a calendar-based function call represents a single, absolute point in time, each function returns only *part* of the date and time information available from the timestamp. Consequently, a calendar-based function's returned value does **not** directly correspond to a *unique* time point, and conditions based on individual calendar values can apply to *multiple* bars.

For example, this script uses the timestamp() function to calculate a UNIX timestamp from a date "string", and it calculates the calendar day from that timestamp, in the exchange time zone, with the dayofmonth() function. The script compares each bar's opening day to the calculated day and highlights the background when the two are equal:

//@version=6 indicator("`dayofmonth()` demo", overlay = false) //@variable The UNIX timestamp corresponding to August 29, 2024 at 00:00:00 UTC. const int fixedTimestamp = timestamp("29 Aug 2024") //@variable The day of the month calculated from the `fixedTimestamp`, expressed in the exchange time zone. // If the exchange time zone has a negative UTC offset, this variable's value is 28 instead of 29. int dayFromTimestamp = dayofmonth(fixedTimestamp) //@variable An "int" representing the current bar's opening calendar day in the exchange time zone. // Equivalent to the `dayofmonth` variable. int openingDay = dayofmonth(time) // Plot the `openingDay`. plot(openingDay, "Opening day of month", linewidth = 3) // Highlight the background when the `openingDay` equals the `dayFromTimestamp`. bgcolor(openingDay == dayFromTimestamp ? color.orange : na, title = "Day detected highlight")

Note that:

- The timestamp() call treats its argument as a *UTC* calendar date because its `dateString` argument does not specify time zone information. However, the dayofmonth() call calculates the day in the *exchange time zone*. Our example symbol's exchange time zone is "America/New_York" (UTC-4/-5). Therefore, the returned value on this chart is 28 instead of 29.

- The script highlights *any* bar on our chart that opens on the 28th day of *any* month instead of only a specific bar because the dayofmonth() function's returned value does **not** represent a specific point in time on its own.
- This script highlights the bars that *open* on the day of the month calculated from the timestamp. However, some months on our chart have no trading activity on that day. For example, the script does not highlight when the July 28, 2024 occurs on our chart because NASDAQ is closed on Sundays.

Similar to calendar-based variables, these functions are also helpful when testing for dates/times and detecting calendar changes on the chart. The example below uses the year(), month(), weekofyear(), and dayofweek() functions on the time_close timestamp to create conditions that test if the current bar is the first bar that closes in a new year, quarter, month, week, and day. The script plots shapes, draws labels, and uses background colors to visualize the conditions on the chart:

//@version=6 indicator("Calendar changes demo", overlay = true, max_labels_count = 500) // Calculate the year, month, week of year, and day of week corresponding to the `time_close` UNIX timestamp. // All values are expressed in the exchange time zone. int closeYear = year(time_close) int closeMonth = month(time_close) int closeWeekOfYear = weekofyear(time_close) int closeDayOfWeek = dayofweek(time_close) //@variable Is `true` when the change in `closeYear` exceeds 0, marking the first bar that closes in a new year. bool closeInNewYear = ta.change(closeYear) > 0 //@variable Is `true` when the difference in `closeMonth` is not 0, marking the first bar that closes in a new month. bool closeInNewMonth = closeMonth - closeMonth[1] != 0 //@variable Is `true` when `closeMonth - 1` becomes divisible by 3, marking the first bar that closes in a new quarter. bool closeInNewQuarter = (closeMonth[1] - 1) % 3 != 0 and (closeMonth - 1) % 3 == 0 //@variable Is `true` when the change in `closeWeekOfYear` is not 0, marking the first bar that closes in a new week. bool closeInNewWeek = ta.change(closeWeekOfYear) != 0 //@variable Is `true` when the `closeDayOfWeek` changes, marking the first bar that closes in the new day. bool closeInNewDay = closeDayOfWeek != closeDayOfWeek[1] //@variable Switches between `true` and `false` after every `closeInNewDay` occurrence for background color calculation. var bool alternateDay = true if closeInNewDay alternateDay := not alternateDay // Draw a label above the bar to display the `closeWeekOfYear` when `closeInNewWeek` is `true`. if closeInNewWeek label.new(time, 0, "W" + str.tostring(closeWeekOfYear), xloc.bar_time, yloc.abovebar, color.purple, textcolor = color.white, size = size.normal) // Plot label shapes at the bottom and top of the chart for the `closeInNewYear` and `closeInNewMonth` conditions. plotshape(closeInNewYear, "Close in new year", shape.labelup, location.bottom, color.teal, text = "New year", textcolor = color.white, size = size.huge) plotshape(closeInNewMonth, "Close in new month", shape.labeldown, location.top, text = "New month", textcolor = color.white, size = size.large) // Plot a triangle below the chart bar when `closeInNewQuarter` occurs. plotshape(closeInNewQuarter, "Close in new quarter", shape.triangleup, location.belowbar, color.maroon, text = "New quarter", textcolor = color.maroon, size = size.large) // Highlight the background in alternating colors based on occurrences of `closeInNewDay`. bgcolor(alternateDay ? color.new(color.aqua, 80) : color.new(color.fuchsia, 80), title = "Closing day change")

Note that:

- This script's conditions check for the first bar that closes after each calendar unit changes its value. The bar where each condition is `true` varies with the data available on the chart. For example, the `closeInNewMonth` condition can be `true` *after* the first calendar day of the month if a chart bar did not close on that day.
- To detect when new bars start on a specific *timeframe* rather than strictly calendar changes, check when the ta.change() of a time() or time_close() call's returned value is nonzero, or use the timeframe.change() function. See this section above for more information.

`timestamp()`

The timestamp() function calculates a UNIX timestamp from a specified calendar date and time. It has the following three signatures:

```
timestamp(year, month, day, hour, minute, second) → simple/series int
timestamp(timezone, year, month, day, hour, minute, second) → simple/series int
timestamp(dateString) → const int
```

The first two signatures listed include `year`, `month`, `day`, `hour`, `minute`, and `second` parameters that accept "int" values defining the calendar date and time. A timestamp() call with either signature must include `year`, `month`, and `day` arguments. The other parameters are optional, each with a default value of 0. Both signatures can return either *"simple"* or *"series"* values, depending on the qualified types of the specified arguments.

The primary difference between the first two signatures is the `timezone` parameter, which accepts a time zone string that determines the time zone of the *date and time* specified by the other parameters. If a timestamp() call with "int" calendar arguments does not include a `timezone` argument, it uses the exchange time zone (syminfo.timezone) by default.

The third signature listed has only *one* parameter, `dateString`, which accepts a "string" representing a valid calendar date (e.g., `"20 Aug 2024"`). The value can also include the time of day and time zone (e.g., `"20 Aug 2024 00:00:00 UTC+0"`). If the `dateString` argument does not specify the time of day, the timestamp() call considers the time 00:00 (midnight).

Unlike the other two signatures, the default time zone for the third signature is **GMT+0**. It does **not** use the exchange time zone by default because it interprets time zone information from the `dateString` directly. Additionally, the third signature is the only one that returns a *"const int"* value. As shown in the Time input section of the Inputs page, programmers can use this overload's returned value as the `defval` argument in an input.time() function call.

When using the timestamp() function, it's crucial to understand how time zone information affects its calculations. The *absolute* point in time represented by a specific calendar date *depends* on its time zone, as an identical date and time in various time zones can refer to **different** amounts of time elapsed since the *UNIX Epoch*. Therefore, changing the time zone of the calendar date and time in a timestamp() call *can change* its returned UNIX timestamp.

The following script compares the results of four different timestamp() calls that evaluate the date 2021-01-01 in different time zones. The first timestamp() call does not specify time zone information in its `dateString` argument, so it treats the value as a *UTC* calendar date. The fourth call also evaluates the calendar date in UTC because it includes `"UTC0"` as the `timezone` argument. The second timestamp() call uses the first signature listed above, meaning it uses the exchange time zone, and the third call uses the second signature with `"America/New_York"` as the `timezone` argument.

The script draws a table with rows displaying each timestamp() call, its assigned variable, the calculated UNIX timestamp, and a formatted representation of the time. As we see on the "NASDAQ:MSFT" chart below, the first and fourth table rows show *different* timestamps than the second and third, leading to different formatted strings in the last column:

```
//@version=6 indicator("`timestamp()` demo", overlay = false) //@variable A `table` that displays the different `timestamp()` calls, their returned timestamps, and formatted results. var table displayTable = table.new( position.middle_center, 4, 5, color.white, border_color = color.black, border_width = 2 ) //@function Initializes a `displayTable` cell showing the `displayText` with an optional `specialFormat`. printCell(int colID, int rowID, string displayText, string specialFormat = "") => displayTable.cell(colID, rowID, displayText, text_size = size.large) switch specialFormat "header" => displayTable.cell_set_bgcolor(colID, rowID, color.rgb(76, 175, 79, 70)) "code" => displayTable.cell_set_text_font_family(colID, rowID, font.family_monospace) displayTable.cell_set_text_size(colID, rowID, size.normal) displayTable.cell_set_text_halign(colID, rowID, text.align_left) if barstate.islastconfirmedhistory //@variable The UNIX timestamp corresponding to January 1, 2021 in the UTC+0 time zone. int dateTimestamp1 = timestamp("2021-01-01") //@variable The UNIX timestamp corresponding to January 1, 2021 in the exchange time zone. int dateTimestamp2 = timestamp(2021, 1, 1, 0, 0) //@variable The UNIX timestamp corresponding to January 1, 2021 in the "America/New_York" time zone. int dateTimestamp3 = timestamp("America/New_York", 2021, 1, 1, 0, 0) //@variable The UNIX timestamp corresponding to January 1, 2021 in the "UTC0" (UTC+0) time zone. int dateTimestamp4 = timestamp("UTC0", 2021, 1, 1, 0, 0) // Initialize the top header cells in the `displayTable`. printCell(0, 0, "Variable", "header") printCell(1, 0, "Function call", "header") printCell(2, 0, "Timestamp returned", "header") printCell(3, 0, "Formatted date/time", "header") // Initialize a table row for `dateTimestamp1` results. printCell(0, 1, "`dateTimestamp1`", "header") printCell(1, 1, "`timestamp(\"2021-01-01\")`", "code") printCell(2, 1, str.tostring(dateTimestamp1)) printCell(3, 1, str.format_time(dateTimestamp1, "yyyy.MM.dd HH:mm (Z)")) // Initialize a table row for `dateTimestamp2` results. printCell(0, 2, "`dateTimestamp2`", "header") printCell(1, 2, "`timestamp(2021, 1, 1, 0, 0)`", "code") printCell(2, 2, str.tostring(dateTimestamp2)) printCell(3, 2, str.format_time(dateTimestamp2, "yyyy.MM.dd HH:mm (Z)")) // Initialize a table row for `dateTimestamp3` results. printCell(0, 3, "`dateTimestamp3`", "header") printCell(1, 3, "`timestamp(\"America/New_York\", 2021, 1, 1, 0, 0)`", "code") printCell(2, 3, str.tostring(dateTimestamp3)) printCell(3, 3, str.format_time(dateTimestamp3, "yyyy.MM.dd HH:mm (Z)")) // Initialize a table row for `dateTimestamp4` results. printCell(0, 4, "`dateTimestamp4`", "header") printCell(1, 4, "`timestamp(\"UTC0\", 2021, 1, 1, 0, 0)`", "code") printCell(2, 4, str.tostring(dateTimestamp4)) printCell(3, 4, str.format_time(dateTimestamp4, "yyyy.MM.dd HH:mm (Z)"))
```

Note that:

- The formatted date-time strings express results in the exchange time zone because the str.format_time() function uses syminfo.timezone as the default `timezone` argument. The formatted values on our example chart show the offset string "`-0500`" because NASDAQ's time zone ("America/New_York") follows UTC-5 during *standard time*.
- The formatted strings on the first and fourth rows show the date and time five hours *before* January 1, 2021, because the timestamp() calls evaluated the date in *UTC* and the str.format_time() calls used a time zone five hours *behind* UTC.
- On our chart, the second and third rows have matching timestamps because both corresponding timestamp() calls evaluated the date in the "America/New_York" time zone. The two rows would show different results if we applied the script to a symbol with a different exchange time zone.

Formatting dates and times

Programmers can format UNIX timestamps into human-readable dates and times, expressed in specific time zones, using the str.format_time() function. The function has the following signature:

```
str.format_time(time, format, timezone) → series string
```

Where:

- The `time` parameter specifies the “int” UNIX timestamp to express as a readable time.
- The `format` parameter accepts a “string” consisting of *formatting tokens* that determine the returned information. If the function call does not include a `format` argument, it uses the ISO 8601 standard format: `"yyyy-MM-dd'T'HH:mm:ssZ"`. See the table below for a list of valid tokens and the information they represent.
- The `timezone` parameter determines the time zone of the formatted result. It accepts a time zone string in UTC or IANA notation. If the call does not specify a `timezone`, it uses the exchange time zone (syminfo.timezone).

The general-purpose str.format() function can also format UNIX timestamps into readable dates and times. However, the function **cannot** express time information in *different* time zones. It always expresses dates and times in **UTC+0**. In turn, using this function to format timestamps often results in *erroneous* practices, such as mathematically modifying a timestamp to try and represent the time in another time zone. However, a UNIX timestamp is a unique, **time zone-agnostic** representation of a specific point in time. As such, modifying a UNIX timestamp changes the *absolute time* it represents rather than expressing the same time in a different time zone.

The str.format_time() function does not have this limitation, as it can calculate dates and times in *any* time zone correctly without changing the meaning of a UNIX timestamp. In addition, unlike str.format(), it is optimized specifically for processing time values. Therefore, we recommend that programmers use str.format_time() instead of str.format() to format UNIX timestamps into readable dates and times.

A str.format_time() call’s `format` argument determines the time information its returned value contains. The function treats characters and sequences in the argument as *formatting tokens*, which act as *placeholders* for values in the returned date/time “string”. The following table outlines the most commonly used formatting tokens and explains what each represents:

Token	Represents	Remarks and examples
`"y"`	Year	Use `"yy"` to include the last two digits of the year (e.g., `"00"`), or `"yyyy"` to include the complete year number (e.g., `"2000"`).
`"M"`	Month	Uppercase `"M"` for the month, not to be confused with *lowercase* `"m"` for the minute. Use `"MM"` to include the two-digit month number with a leading zero for single-digit values (e.g., `"01"`), `"MMM"` to include the three-letter abbreviation of month (e.g., `"Jan"`), or `"MMMM"` for the full month name (e.g., `"January"`).
`"d"`	Day of the **month**	Lowercase `"d"`. Includes the numeric day of the month (`"1"` to `"31"`). Use `"dd"` for the two-digit day number with a leading zero for single-digit values. It is *not* a placeholder for the day number of the *week* (1-7). Use dayofweek() to calculate that value.
`"D"`	Day of the **year**	Uppercase `"D"`. Includes the numeric day of the year (`"1"` to `"366"`). Use `"DD"` or `"DDD"` for the two-digit or three-digit day number with leading zeros.

Token	Represents	Remarks and examples
`"E"`	Day of the **week**	Includes the abbreviation of the weekday *name* (e.g., `"Mon"`). Use `"EEEE"` for the weekday's full name (e.g., `"Monday"`)
`"w"`	Week of the **year**	Lowercase `"w"`. Includes the week number of the year (`"1"` to `"53"`). Use `"ww"` for the two-digit week number with a leading zero for single-digit values.
`"W"`	Week of the **month**	Uppercase `"W"`. Includes the week number of the month (`"1"` to `"5"`).
`"a"`	AM/PM postfix	Lowercase `"a"`. Includes `"AM"` if the time of day is before noon, `"PM"` otherwise.
`"h"`	Hour in the **12-hour** format	Lowercase `"h"`. The included hour number from this token ranges from `"0"` to `"11"`. Use `"hh"` for the two-digit hour with a leading zero for single-digit values.
`"H"`	Hour in the **24-hour** format	Uppercase `"H"`. The included hour number from this token ranges from `"0"` to `"23"`. Use `"HH"` for the two-digit hour with a leading zero for single-digit values.
`"m"`	Minute	Lowercase `"m"` for the minute, not to be confused with *uppercase* `"M"` for the month. Use `"mm"` for the two-digit minute with a leading zero for single-digit values.
`"s"`	Second	Lowercase `"s"` for the second, not to be confused with *uppercase* `"S"` for fractions of a second. Use `"ss"` for the two-digit second with a leading zero for single-digit values.
`"S"`	Fractions of a second	Uppercase `"S"`. Includes the number of milliseconds in the fractional second (`"0"` to `"999"`). Use `"SS"` or `"SSS"` for the two-digit or three-digit millisecond number with leading zeros.
`"Z"`	Time zone (**UTC offset**)	Uppercase `"Z"`. Includes the hour and minute UTC offset value in `"HHmm"` format, preceded by its sign (e.g., `"−0400"`).
`"z"`	Time zone (**abbreviation or name**)	Lowercase `"z"`. A single `"z"` includes the abbreviation of the time zone (e.g., `"EDT"`). Use `"zzzz"` for the time zone's name (e.g., `"Eastern Daylight Time"`). It is not a placeholder for the *IANA identifier*. Use syminfo.timezone to retrieve the exchange time zone's IANA representation.
`":"`, `"/"`, `"−"`, `"."`, `","`, `"("`, `")"`, `" "`	Separators	These characters are separators for formatting tokens. They appear as they are in the formatted text. (e.g., `"01/01/24"`, `"12:30:00"`, `"Jan 1, 2024"`). Some other characters can also act as separators. However, the ones listed are the most common.
`"'"`	Escape character	Characters enclosed within *two single quotes* appear as they are in the result, even if they otherwise act as formatting tokens. For example, `" 'Day' "` appears as-is in the resulting "string" instead of listing the day of the year, AM/PM postfix, and year.

The following example demonstrates how various formatting tokens affect the str.format_time() function's result. The script calls the function with different `format` arguments to create date/time strings from time, timenow, and time_close timestamps. It displays each `format` value and the corresponding formatted result in a table on the last bar:

```
//@version=6 indicator("Formatting dates and times demo", overlay = false) //@variable A `table` that displays different date/time `format` strings and their results. var table displayTable = table.new( position.middle_center, 2, 15, bgcolor = color.white, frame_color = color.black, frame_width = 1, border_width = 1 ) //@function Initializes a `displayTable` row showing a `formatString` and its formatted result for a specified // `timeValue` and `timezoneValue`. displayText(rowID, formatString, timeValue = time, timezoneValue = syminfo.timezone) => //@variable Is light blue if the `rowID` is even, white otherwise. Used to set alternating table row colors. color rowColor = rowID % 2 == 0 ? color.rgb(33, 149, 243, 75) : color.white // Display the `formatString` in the row's first cell. displayTable.cell( 0, rowID, formatString, text_color = color.black, text_halign = text.align_left, bgcolor = rowColor ) // Show the result of formatting the `timeValue` based on the `formatString` and `timezoneValue` in the second cell. displayTable.cell( 1, rowID, str.format_time(timeValue, formatString, timezoneValue), text_color = color.black, text_halign = text.align_right, bgcolor = rowColor ) if barstate.islast // Initialize the table's header cells. displayTable.cell(0, 0, "FORMAT STRINGS") displayTable.cell(1, 0, "FORMATTED DATE/TIME OUTPUT") // Initialize a row to show the default date-time "string" format and its result for `time`. displayTable.cell( 0, 1, "(Default `str.format_time()` format)", text_color = color.black, text_halign = text.align_left, bgcolor = color.yellow) displayTable.cell( 1, 1, str.format_time(time), text_color = color.black, text_halign = text.align_right, bgcolor = color.yellow) // Initialize rows to show different formatting strings and their results for `time`, `time_close`, and `timenow`. displayText(2, "dd/MM/yy") displayText(3, "MMMM dd, yyyy") displayText(4, "hh:mm:ss.SS a", timenow) displayText(5, "HH:mm 'UTC'Z") displayText(6, "H:mm a (zzzz)") displayText(7, "my day / 'my day' ('escaped')") displayText(8, "'Month' M, 'Week' w, 'Day' DDD") displayText(9, "'Bar expected closing time': ha", time_close) displayText(10, "'Current date/time': MMM-d-y HH:mm:ss z", timenow) displayText(11, "'New Time zone': zzzz", timezoneValue = "Australia/Sydney") displayText(12, "'Time zone change': MMM-d-y HH:mm:ss z", timenow, "Australia/Sydney")
```

Expressing time differences

Every UNIX timestamp represents a specific point in time as the absolute *time difference* from a fixed historical point (epoch). The specific epoch all UNIX timestamps reference is *midnight UTC on January 1, 1970*. Programmers can format UNIX timestamps into readable date-time strings with the str.format_time() function because it uses the time difference from the UNIX Epoch in its date and time calculations.

In contrast, the difference between two nonzero UNIX timestamps represents the number of milliseconds elapsed from one absolute point to another. The difference does not directly refer to a specific point in UNIX time if neither timestamp in the operation has a value of 0 (corresponding to the UNIX Epoch).

Programmers may want to express the millisecond difference between two UNIX timestamps in *other time units*, such as seconds, days, etc. Some might assume they can use the difference as the `time` argument in a str.format_time() call to achieve this result. However, the function always treats its `time` argument as the time elapsed from the *UNIX Epoch* to derive a *calendar date/time* representation in a specific time zone. It **does not** express time differences directly. Therefore, attempting to format timestamp *differences* rather than timestamps with str.format_time() leads to unintended results.

For example, the following script calculates the millisecond difference between the current execution time (timenow) and the “1M” bar’s closing time (`time_close("1M")`) for a monthly countdown timer display. It attempts to express the time difference in another format using str.format_time(). It displays the function call’s result in a table, along with the original millisecond difference (`timeLeft`) and formatted date-time representations of the timestamps.

As we see below, the table shows correct results for the formatted timestamps and the `timeLeft` value. However, the formatted time difference appears as `"1970-01-12T16:47:10-0500"`. Although the `timeLeft` value is supposed to represent a difference between timestamps rather than a specific point in time, the str.format_time() function still treats the value as a **UNIX timestamp**. Consequently, it creates a "string" expressing the value as a *date and time* in the UTC-5 time zone:

//@version=6 indicator("Incorrectly formatting time difference demo", overlay = true) //@variable A table that displays monthly close countdown information. var table displayTable = table.new(position.top_right, 1, 4, color.rgb(0, 188, 212, 60)) if barstate.islast //@variable A UNIX timestamp representing the current time as of the script's latest execution. int currentTime = timenow //@variable A UNIX timestamp representing the expected closing time of the current "1M" bar. int monthCloseTime = time_close("1M") //@variable The number of milliseconds between the `currentTime` and the `monthCloseTime`. // This value is NOT intended as a UNIX timestamp. int timeLeft = monthCloseTime - currentTime //@variable A "string" representing the `timeLeft` as a date and time in the exchange time zone, in ISO 8601 format. // This format is INCORRECT for the `timeLeft` value because it's supposed to represent the time between // two nonzero UNIX timestamps, NOT a specific point in time. string incorrectTimeFormat = str.format_time(timeLeft) // Initialize `displayTable` cells to initialize the `currentTime` and `monthCloseTime`. displayTable.cell(0, 0, "Current time: " + str.format_time(currentTime, "HH:mm:ss.S dd/MM/yy (z)"), text_size = size.large, text_halign = text.align_right) displayTable.cell(0, 1, "`1M` Bar closing time: " + str.format_time(monthCloseTime, "HH:mm:ss.SS dd/MM/yy (z)"), text_size = size.large, text_halign = text.align_right) // Initialize a cell to display the `timeLeft` millisecond difference. displayTable.cell(0, 2, "`timeLeft` value: " + str.tostring(timeLeft), text_size = size.large, bgcolor = color.yellow) // Initialize a cell to display the `incorrectTimeFormat` representation. displayTable.cell(0, 3, "Time left (incorrect format): " + incorrectTimeFormat, text_size = size.large, bgcolor = color.maroon, text_color = color.white)

To express the difference between timestamps in other time units correctly, programmers must write code that *calculates* the number of units elapsed instead of erroneously formatting the difference as a specific date or time.

The calculations required to express time differences depend on the chosen time units. The sections below explain how to express millisecond differences in weekly and smaller units, and monthly and larger units.

Weekly and smaller units

Weeks and smaller time units (days, hours, minutes, seconds, and milliseconds) cover *consistent* blocks of time. These units have the following relationship:

- One week equals seven days.
- One day equals 24 hours.
- One hour equals 60 minutes.
- One minute equals 60 seconds.
- One second equals 1000 milliseconds.

Using this relationship, programmers can define the span of these units by the number of *milliseconds* they contain. For example, since every hour has 60 minutes, every minute has 60 seconds, and every second has 1000 milliseconds, the number of milliseconds per hour is `60 * 60 * 1000`, which equals `3600000`.

Programmers can use *modular arithmetic* based on the milliseconds in each unit to calculate the total number of weeks, days, and smaller spans covered by the difference between two UNIX timestamps. The process is as follows, starting from the *largest* time unit in the calculation:

1. Calculate the number of milliseconds in the time unit.
2. Divide the remaining millisecond difference by the calculated value and round down to the nearest whole number. The result represents the number of *complete* time units within the interval.
3. Use the *remainder* from the division as the new remaining millisecond difference.
4. Repeat steps 1-3 for each time unit in the calculation, in *descending* order based on size.

The following script implements this process in a custom `formatTimeSpan()` function. The function accepts two UNIX timestamps defining a start and end point, and its “bool” parameters control whether it calculates the number of weeks or smaller units covered by the time range. The function calculates the millisecond distance between the two timestamps. It then calculates the numbers of complete units covered by that distance and formats the results into a “string”.

The script calls `formatTimeSpan()` to express the difference between two separate time input values in selected time units. It then displays the resulting “string” in a table alongside formatted representations of the start and end times:

//@version=6 indicator("Calculating time span demo") // Assign the number of milliseconds in weekly and smaller units to "const" variables for convenience. const int ONE_WEEK = 604800000 const int ONE_DAY = 86400000 const int ONE_HOUR = 3600000 const int ONE_MINUTE = 60000 const int ONE_SECOND = 1000 //@variable A UNIX timestamp calculated from the user-input start date and time. int startTimeInput = input.time(timestamp("1 May 2022 00:00 -0400"), "Start date and time", group = "Time between") //@variable A UNIX timestamp calculated from the user-input end date and time. int endTimeInput = input.time(timestamp("7 Sep 2024 20:37 -0400"), "End date and time", group = "Time between") // Create "bool" inputs to toggle weeks, days, hours, minutes, seconds, and milliseconds in the calculation. bool weeksInput = input.bool(true, "Weeks", group = "Time units", inline = "A") bool daysInput = input.bool(true, "Days", group = "Time units", inline = "A") bool hoursInput = input.bool(true, "Hours", group = "Time units", inline = "B") bool minutesInput = input.bool(true, "Minutes", group = "Time units", inline = "B") bool secondsInput = input.bool(true, "Seconds", group = "Time units", inline = "B") bool millisecondsInput = input.bool(true, "Milliseconds", group = "Time units", inline = "B") //@function Calculates the difference between two UNIX timestamps as the number of complete time units in // descending order of size, formatting the results into a "string". The "int" parameters accept timestamps, // and the "bool" parameters determine which units the function uses in its calculations. formatTimeSpan(int startTimestamp, int endTimestamp, bool calculateWeeks, bool calculateDays, bool calculateHours, bool calculateMinutes, bool calculateSeconds, bool calculateMilliseconds) => //@variable The milliseconds between the `startTimestamp` and `endTimestamp`. int timeDifference = math.abs(endTimestamp - startTimestamp) //@variable A "string" representation of the interval in mixed time units. string formattedString = na // Calculate complete units within the interval for each toggled unit, reducing the `timeDifference` by those sizes. if calculateWeeks int totalWeeks = math.floor(timeDifference / ONE_WEEK) timeDifference %= ONE_WEEK formattedString += str.tostring(totalWeeks) + (totalWeeks == 1 ? " week " : " weeks ") if calculateDays int totalDays = math.floor(timeDifference / ONE_DAY) timeDifference %= ONE_DAY formattedString += str.tostring(totalDays) + (totalDays == 1 ? " day " : " days ") if calculateHours int totalHours = math.floor(timeDifference / ONE_HOUR) timeDifference %= ONE_HOUR formattedString += str.tostring(totalHours) + (totalHours == 1 ? " hour " : " hours ") if calculateMinutes int totalMinutes = math.floor(timeDifference / ONE_MINUTE) timeDifference %= ONE_MINUTE formattedString += str.tostring(totalMinutes) + (totalMinutes == 1 ? " minute " : " minutes ") if calculateSeconds int totalSeconds = math.floor(timeDifference / ONE_SECOND) timeDifference %= ONE_SECOND formattedString += str.tostring(totalSeconds) + (totalSeconds == 1 ? " second " : " seconds ") if calculateMilliseconds // `timeDifference` is in milliseconds already, so add it to the `formattedString` directly. formattedString +=

```
str.tostring(timeDifference) + (timeDifference == 1 ? " millisecond" : " milliseconds") // Return the `formattedString`. formattedString if barstate.islastconfirmedhistory //@variable A table that that displays formatted start and end times and their custom-formatted time difference. var table displayTable = table.new(position.middle_center, 1, 2, color.aqua) //@variable A "string" containing formatted `startTimeInput` and `endTimeInput` values. string timeText = "Start date and time: " + str.format_time(startTimeInput, "dd/MM/yy HH:mm:ss (z)") + "\n End date and time: " + str.format_time(endTimeInput, "dd/MM/yy HH:mm:ss (z)") //@variable A "string" representing the span between `startTimeInput` and `endTimeInput` in mixed time units. string userTimeSpan = formatTimeSpan( startTimeInput, endTimeInput, weeksInput, daysInput, hoursInput, minutesInput, secondsInput, millisecondsInput ) // Display the `timeText` in the table. displayTable.cell(0, 0, timeText, text_color = color.white, text_size = size.large, text_halign = text.align_left) // Display the `userTimeSpan` in the table. displayTable.cell(0, 1, "Time span: " + userTimeSpan, text_color = color.white, text_size = size.large, text_halign = text.align_left, bgcolor = color.navy)
```

Note that:

- The user-defined function uses math.floor() to round each divided result down to the nearest “int” value to get the number of *complete* units in the interval. After division, it uses the modulo assignment operator (%=) to get the *remainder* and assign that value to the `timeDifference` variable. This process repeats for each selected unit.

The image above shows the calculated time difference in mixed time units. By toggling the “bool” inputs, users can also isolate specific units in the calculation. For example, this image shows the result after enabling only the “Milliseconds” input:

Monthly and larger units

Unlike weeks and smaller units, months and larger units *vary* in length based on calendar rules. For example, a month can contain 28, 29, 30, or 31 days, and a year can contain 365 or 366 days.

Some programmers prefer to use the modular arithmetic outlined in the previous section, with *approximate lengths* for these irregular units, to calculate large-unit durations between UNIX timestamps. With this process, programmers usually define the units in either of the following ways:

- Using *common* lengths, e.g., a common year equals 365 days, and a common month equals 30 days.
- Using the *average* lengths, e.g., an average year equals 365.25 days, and an average month equals 30.4375 days.

Calculations involving approximate units produce *rough estimates* of the elapsed time. Such estimates are often practical when expressing relatively short durations. However, their precision diminishes with the size of the difference, drifting further away from the actual time elapsed.

Therefore, expressing time differences in monthly and larger units with precision requires a different calculation than the process outlined above. For a more precise estimate of months, years, and larger units elapsed, the calculations should use the *actual* span of each individual unit rather than approximations, meaning it must account for *leap years* and *variations* in month sizes.

The advanced example below contains a custom `formatTimeDifference()` function that calculates the years and months, in addition to days and smaller units, elapsed between two UNIX timestamps.

The function uses the process outlined in the previous section to calculate the daily and smaller units within the interval. For the monthly and yearly units, which have *irregular* lengths, the function uses a while loop to iterate across calendar months. On each iteration, it increments monthly and yearly counters and subtracts the number of days in the added month from the day counter. After the loop ends, the function adjusts the year, month, and day counters to account for partial months elapsed between the timestamps. Finally, it uses the counters in a str.format() call to create a formatted “string” containing the calculated values.

The script calls this `formatTimeDifference()` function to calculate the years, months, days, hours, minutes, seconds, and milliseconds elapsed between two separate time input values and displays the result in a label:

```
//@version=6 indicator("Calculating time span for larger units demo") //@variable The starting date and time of the time span, input by the user. int startTimeInput = input.time(timestamp("3 Apr 2022 20:00 -0400"), "Start date", group = "Time between") //@variable The ending date and time of the time span, input by the user. int endTimeInput = input.time(timestamp("3 Sep 2024 15:45 -0400"), "End date", group = "Time between") //@function Returns the number of days in the `monthNumber` month of the `yearNumber` year. daysPerMonth(int yearNumber, int monthNumber) => //@variable Is `true` if the `yearNumber` represents a leap year. bool leapYear = (yearNumber % 4 == 0 and yearNumber % 100 != 0) or (yearNumber % 400 == 0) //@variable The number of days calculated for the month. int result = switch monthNumber == 2 => leapYear ? 29 : 28 => 31 - (monthNumber - 1) % 7 % 2 //@function Calculates the relative time difference between two timestamps, covering monthly and larger units. formatTimeDifference(int timestamp1, int timestamp2) => // The starting time and ending time. int startTime = math.min(timestamp1, timestamp2), int endTime = math.max(timestamp1, timestamp2) // The year, month, and day of the `startTime` and `endTime`. int startYear = year(startTime), int startMonth = month(startTime), int startDay = dayofmonth(startTime) int endYear = year(endTime), int endMonth = month(endTime), int endDay = dayofmonth(endTime) // Calculate the total number of days, hours, minutes, seconds, and milliseconds in the interval. int milliseconds = endTime - startTime int days = math.floor(milliseconds / 86400000), milliseconds %= 86400000 int hours = math.floor(milliseconds / 3600000), milliseconds %= 3600000 int minutes = math.floor(milliseconds / 60000), milliseconds %= 60000 int seconds = math.floor(milliseconds / 1000), milliseconds %= 1000 // Calculate the number of days in the `startMonth` and `endMonth`. int daysInStartMonth = daysPerMonth(startYear, startMonth), int daysInEndMonth = daysPerMonth(endYear, endMonth) //@variable The number of days remaining in the `startMonth`. int remainingInMonth = daysInStartMonth - startDay + 1 // Subtract `remainingInMonth` from the `days`, and offset the `startDay` and `startMonth`. days -= remainingInMonth, startDay := 1, startMonth += 1 // Set `startMonth` to 1, and increase the `startYear` if the `startMonth` exceeds 12. if startMonth > 12 startMonth := 1, startYear += 1 // Initialize variables to count the total number of months and years in the interval. int months = 0, int years = 0 // Loop to increment `months` and `years` values based on the `days`. while days > 0 //@variable The number of days in the current `startMonth`. int daysInMonth = daysPerMonth(startYear, startMonth) // Break the loop if the number of remaining days is less than the `daysInMonth`. if days < daysInMonth break // Reduce the `days` by the `daysInMonth` and increment the `months`. days -= daysInMonth, months += 1 // Increase the `years` and reset the `months` to 0 when `months` is 12. if months == 12 months := 0, years += 1 // Increase the `startMonth` and adjust the `startMonth` and `startYear` if its value exceeds 12. startMonth += 1 if startMonth > 12 startMonth := 1, startYear += 1 // Re-add the `remainingInMonth` value to the number of `days`. Adjust the `days`, `months`, and `years` if the // new value exceeds the `daysInStartMonth` or `daysInEndMonth`, depending on the `startDay`. days += remainingInMonth if days >= (startDay < daysInStartMonth / 2 ? daysInStartMonth : daysInEndMonth) months += 1 if months == 12 months := 0, years += 1 days -= remainingInMonth // Format the calculated values into a "string" and return the result. str.format( " Years:   {0}\n Months: {1}\n Days:    {2}\n Hours:   {3}\n Minutes: {4}\n Seconds: {5}\n
```

Milliseconds: {6}", years, months, days, hours, minutes, seconds, milliseconds) if barstate.islastconfirmedhistory //@variable A "string" representing the time between the `startTimeInput` and `endTimeInput` in mixed units. string userTimeSpan = formatTimeDifference(startTimeInput, endTimeInput) //@variable Text shown in the label. string labelText = "Start time: " + str.format_time(startTimeInput, "dd/MM/yy HH:mm:ss (z)") + "\n End time: " + str.format_time(endTimeInput, "dd/MM/yy HH:mm:ss (z)") + "\n---------\nTime difference:\n" + userTimeSpan label.new(bar_index, high, labelText, color = #ff946e, textcolor = #363a45, size = size.large, textalign = text.align_left, style = label.style_label_center)

Note that:

- The script determines the number of days in each month with the user-defined `daysPerMonth()` function. The function identifies whether a month has 28, 29, 30, or 31 days based on its month number and the year it belongs to. Its calculation accounts for leap years. A leap year occurs when the year is divisible by 4 or 400 but not by 100.
- Before the while loop, the function subtracts the number of days in a partial starting month from the initial day count, aligning the counters with the beginning of a new month. It re-adds the subtracted days after the loop to adjust the counters for partial months. It adjusts the month and year counters based on the days in the `startMonth` if the `startDay` is less than halfway through that month. Otherwise, it adjusts the values based on the days in the `endMonth`.

Timeframes

Introduction

The *timeframe* of a chart is sometimes also referred to as its *interval* or *resolution*. It is the unit of time represented by one bar on the chart. All standard chart types use a timeframe: “Bars”, “Candles”, “Hollow Candles”, “Line”, “Area” and “Baseline”. One non-standard chart type also uses timeframes: “Heikin Ashi”.

Programmers interested in accessing data from multiple timeframes will need to become familiar with how timeframes are expressed in Pine Script®, and how to use them.

Timeframe strings come into play in different contexts:

- They must be used in request.security() when requesting data from another symbol and/or timeframe. See the page on Other timeframes and data to explore the use of request.security().
- They can be used as an argument to time() and time_close() functions, to return the time of a higher timeframe bar. This, in turn, can be used to detect changes in higher timeframes from the chart's timeframe without using request.security(). See the Testing for changes in higher timeframes section to see how to do this.
- The input.timeframe() function provides a way to allow script users to define a timeframe through a script's “Inputs” tab (see the Timeframe input section for more information).
- The indicator() declaration statement has an optional `timeframe` parameter that can be used to provide multi-timeframe capabilities to simple scripts without using request.security().
- Many built-in variables provide information on the timeframe used by the chart the script is running on. See the Chart timeframe section for more information on them, including timeframe.period which returns a string in Pine Script's timeframe specification format.

Timeframe string specifications

Timeframe strings follow these rules:

- They are composed of the multiplier and the timeframe unit, e.g., “1S”, “30” (30 minutes), “1D” (one day), “3M” (three months).
- The unit is represented by a single letter, with no letter used for minutes: “T” for ticks, “S” for seconds, “D” for days, “W” for weeks, and “M” for months.
- When no multiplier is used, 1 is assumed: “S” is equivalent to “1S”, “D” to “1D”, etc. If only “1” is used, it is interpreted as 1 minute, since no unit letter identifier is used for minutes.
- There is no “hour” unit; “1H” is **not** valid. The correct format for one hour is “60” (remember no unit letter is specified for minutes).
- The valid multipliers vary for each timeframe unit:
    - For ticks, only the discrete 1, 10, 100, and 1000 multipliers are valid.
    - For seconds, only the discrete 1, 5, 10, 15, 30, and 45 multipliers are valid.
    - For minutes, 1 to 1440.
    - For days, 1 to 365.

- For weeks, 1 to 52.
- For months, 1 to 12.

Comparing timeframes

It can be useful to compare different timeframe strings to determine, for example, if the timeframe used on the chart is lower than the higher timeframes used in the script.

Converting timeframe strings to a representation in fractional minutes provides a way to compare them using a universal unit. This script uses the timeframe.in_seconds() function to convert a timeframe into float seconds and then converts the result into minutes:

//@version=6 indicator("Timeframe in minutes example", "", true) string tfInput = input.timeframe(defval = "", title = "Input TF") float chartTFInMinutes = timeframe.in_seconds() / 60 float inputTFInMinutes = timeframe.in_seconds(tfInput) / 60 var table t = table.new(position.top_right, 1, 1) string txt = "Chart TF: " + str.tostring(chartTFInMinutes, "#.##### minutes") + "\nInput TF: " + str.tostring(inputTFInMinutes, "#.##### minutes") if barstate.isfirst table.cell(t, 0, 0, txt, bgcolor = color.yellow) else if barstate.islast table.cell_set_text(t, 0, 0, txt) if chartTFInMinutes > inputTFInMinutes runtime.error("The chart's timeframe must not be higher than the input's timeframe.")

Note that:

- We use the built-in timeframe.in_seconds() function to convert the chart timeframe and the timeframe selected by the user into seconds, then divide by 60 to convert into minutes.
- We use two calls to the timeframe.in_seconds() function in the initialization of the `chartTFInMinutes` and `inputTFInMinutes` variables. In the first instance, we do not supply an argument for its `timeframe` parameter, so the function returns the chart's timeframe in seconds. In the second call, we supply the timeframe selected in the timeframe input.
- Next, we validate the timeframes to ensure that the input timeframe is equal to or higher than the chart's timeframe. If it is not, the script generates a custom runtime error.
- We finally print the two timeframe values converted to minutes.

The condition of the "X" statement must evaluate to a "bool" value

This compilation error occurs if one or more of the *conditions* that control the flow of a conditional structure (an if or switch statement) returns a value that is *not* of the "bool" type. These structures *cannot* use values other than `true` and `false` as conditions.

The following sections explain the usual causes of this error and the typical solutions.

> TipA similar restriction also applies to the ternary operator: its *first operand* must be a "bool" value. Using a value of another type causes a different error, but the solutions to fix that error are the same as those below.

Using numeric conditions

In some programming languages, including earlier versions of Pine Script® up to v5, numeric values of the "int" and "float" types are implicitly converted to `true` or `false` when the script passes them to expressions

that require “bool” values. The typical rules for such conversions are as follows:

- A value of `0`, `0.0`, or na converts to `false`.
- *Any* other nonzero, non-na value converts to `true`.

Although this logic might offer a marginal amount of convenience for experienced programmers in some rare cases, it can also easily lead to unintended results. Therefore, as of v6, Pine Script no longer implicitly casts numeric values to “bool” values in operations or function calls.

To work around this limitation, programmers can do either of the following:

- Use the bool() function to *explicitly* cast “int” or “float” values to the “bool” type based on the above rules.
- Write conditional expressions that *compare* numeric values and return “bool” results.

For example, the script below does not compile because it attempts to use the `newMonth` variable, which holds a *“series int”* value, as the condition in an if statement:

//@version=6 indicator("Invalid numeric condition demo") //@variable Holds the one-bar change in the value of the `month` variable ("series int"). // The value is nonzero only on bars where the month changed in the exchange time zone. newMonth = ta.change(month) // This code causes the error. The `if` statement requires "bool" conditions for its control criteria. // It cannot accept an "int" value or a value of any other type. if newMonth label.new(bar_index, high, "New month started")

To resolve the error, we can pass the `newMonth` variable to a bool() function call, and then use that call’s “series bool” result as the if statement’s condition. For example:

//@version=6 indicator("Explicitly casting a numeric condition demo") //@variable Holds the one-bar change in the value of the `month` variable ("series int"). // The value is nonzero only on bars where the month changed in the exchange time zone. newMonth = ta.change(month) // This code does not cause the error. The `bool()` call casts the `newMonth` value to the "bool" type. // The call returns `true` if the variable's value is a nonzero number, and `false` otherwise. if bool(newMonth) label.new(bar_index, high, "New month started")

Note that:

- The bool() call casts the *value* retrieved from the variable to `true` or `false`, but it does *not* change the *variable’s* type to “bool”. The script can still use the `newMonth` variable in other code that accepts “series int” values.

Implicitly testing for `na` values

Values or references of most types can be na, which means *undefined*. Programmers sometimes tested for na in Pine v5 and earlier versions by using implicit “bool” casting behaviors in conditional logic. Such logic does not compile in Pine v6. Furthermore, such tests do not distinguish between numeric values that are na or 0.

Programmers can explicitly test for na instances of most available types and retrieve a “bool” result by using the na() function. A call to the function returns `true` if its argument is na, and `false` otherwise.

> TipIt is often useful for an expression to return `true` instead of `false` if the tested value is *not* na. To achieve this effect, use the not operator on the na() call's result.

Consider the following script, which tries to use the result of a ta.pivothigh() function call to control an if statement. The call returns a "float" price value if it confirms a pivot high point, or na if no pivot high is confirmed. This script does not compile, because the if statement cannot use a "float" value or any na value as a condition:

//@version=6 indicator("Invalid test for `na` demo", overlay = true) //@variable Holds a non-na price value ("series float") if a pivot high is detected, and `na` otherwise. pivot = ta.pivothigh(10, 10) // This code causes the error. The `if` statement requires a "bool" condition. It can't use a "float" value. // It also cannot use `na` in any case because "bool" values are never `na`. if pivot label.new(bar_index[10], pivot, "Pivot High")

We can resolve the error and achieve our script's intended result by replacing `pivot` in the if statement with the expression `not na(pivot)`, which returns `true` if the variable's value is not na, and `false` otherwise. For example:

//@version=6 indicator("Valid test for `na` demo", overlay = true) //@variable Holds a non-na price value ("series float") if a pivot high is detected, and `na` otherwise. pivot = ta.pivothigh(10, 10) // This code does not cause the error, because the expression used as the condition returns a "bool" result. if not na(pivot) label.new(bar_index[10], pivot, "Pivot High")

Compiled code contains too many tokens

This compilation error indicates that a script's compiled code is *too large* for the runtime system to execute.

Before a script executes on a dataset, it is compiled by the Pine Script compiler. During this process, the compiler translates the written source code into a tokenized *intermediate language (IL)*, then generates the final compiled code from the script's IL form. Translating a script into an IL representation enables the compiler to perform various internal transformations to optimize the script's structure.

Pine Script limits the maximum size of each indicator, strategy, and library for resource efficiency. This limit applies to the total size of the script's *IL* translation *after* optimization. It does **not** apply to the number of characters or code lines in the original written code. This behavior explains why one script might fail with the "too many tokens" error during compilation, while another with the same number of characters or lines compiles and runs successfully. It also explains why some kinds of manual code changes, such as removing comments or shortening variable names, *do not* affect the size of a compiled script.

> NoteThere is no direct way to inspect the size of a script's IL form if the script compiles successfully. The only time that the size is visible is when the "too many tokens" error occurs.

The maximum possible size of a script's IL translation is 100,256 tokens. There are several ways to reduce a translated script's size if it exceeds this limit, depending on the original script's structure and the kinds of code it contains.

The sections on this page explain some of the typical optimizations that the compiler performs before measuring a translated script's size, various types of manual code changes that have little to no effect on a

script's translation, and the most common techniques that can help reduce script size and resolve this error.

Ineffective code changes

Before the compiler measures the size of a translated script, it performs several code transformations to eliminate unnecessary tokens and optimize the script's structure. Because of these transformations, several types of source code changes a programmer might consider to resolve the "too many tokens" error have *little to no impact* on the size of the compiled code. Common types of ineffective code changes include the following:

- Revising or removing comments
- Removing whitespaces and merging code lines
- Changing code identifiers
- Swapping constant expressions and values
- Reducing identical expressions and functions
- Removing unused code

The following sections explain why these common source code changes do not help resolve this error. To learn about code changes that can meaningfully reduce the size of a translated script, refer to the Effective techniques to reduce script size section below.

Revising or removing comments

Programmers might assume that reducing the length of the comments in their scripts or removing them altogether reduces the size of the compiled code. However, the compiler *discards* all comments during translation, because their main purpose is to provide extra information in the source code for script readers. Therefore, reducing or removing comments has *no effect* on the compiled code.

For example, the simple script below contains multiple comments that document what each part of the code does:

//@version=6 indicator("Removing comments demo") // Define the input length factor for the RSI and RMA calculations. int lengthInput = input.int(14, "Length", minval = 2) // Calculate the RSI. float rsi = ta.rsi(close, lengthInput) // Calculate the RMA of the RSI using the same length. float ma = ta.rma(rsi, lengthInput) // Add the RMA of the difference between the MA and the RSI for the final smoothed output. float smoothed = ma + ta.rma(rsi - ma, lengthInput) // Plot the RSI and the smoothed RSI in a separate pane. plot(rsi, "RSI", color.blue) plot(smoothed, "Smoothed RSI", color.orange)

If we remove all the comments from the script, the source code becomes visibly shorter. However, because the compiler excludes comments automatically, the version of the code below translates to the *same* compiled result as the previous code:

//@version=6 indicator("Removing comments demo") int lengthInput = input.int(14, "Length", minval = 2) float rsi = ta.rsi(close, lengthInput) float ma = ta.rma(rsi, lengthInput) float smoothed = ma + ta.rma(rsi - ma, lengthInput) plot(rsi, "RSI", color.blue) plot(smoothed, "Smoothed RSI", color.orange)

> NoteThe same principle applies to compiler annotations. Although annotations are special comments that can specify extra information for a script, the compiler does not include annotation text directly in a

> script's IL translation. The information from annotations is processed separately. Therefore, it does not contribute to a script's compiled size.

Removing whitespaces and merging code lines

Pine Script code uses whitespace characters, including standard spaces and line terminators, to separate expressions and statements, organize code on multiple text lines, and visually delimit code blocks. However, the IL translation of a script uses a *compact format*, without most whitespace delimiters, to represent the script's structure. Therefore, removing spaces between items in the source code *does not* affect the script's compiled size. Likewise, the size is not affected by merging multiple statements onto a single code line and separating them with commas.

For example, the following script lists statements across multiple lines of code. The source code includes space separators, line-wrapped expressions, and multiple blank lines:

```
//@version=6 indicator("Removing whitespaces and merging code lines demo", overlay = true) int lengthInput1 = input.int( 14, "Fast MA length", minval = 2 ) int lengthInput2 = input.int( 28, "Slow MA length", minval = 2 ) float ma1 = ta.sma(close, lengthInput1) float ma2 = ta.sma(close, lengthInput2) bool condition = ( ma1 > ma2 and ma1 > ma1[1] and ma2 > ma2[1] ) plot(ma1, "Fast MA", color.orange) plot(ma2, "Slow MA", color.purple) bgcolor( condition ? color.new(color.green, 80) : na, title = "Condition highlight" )
```

The modified script version below removes multiple space separators and blank lines, eliminates line wrapping, and groups the statements onto fewer lines. Although the code is visually shorter and more compact, its IL form is *identical* to that of the previous version, because the compiler already discards the spaces and line terminators automatically while translating the code:

```
//@version=6 indicator("Removing whitespaces and merging code lines demo",overlay=true) int lengthInput1=input.int(14,"Fast MA length",minval=2),int lengthInput2=input.int(28,"Slow MA length",minval=2) float ma1=ta.sma(close,lengthInput1),float ma2=ta.sma(close,lengthInput2) bool condition=ma1>ma2 and ma1>ma1[1] and ma2>ma2[1],plot(ma1,"Fast MA",color.orange),plot(ma2,"Slow MA",color.purple) bgcolor(condition?color.new(color.green,80):na,title="Condition highlight")
```

Note that:

- Automatic whitespace removal does *not* apply to literal strings, because each character in a string is part of the saved value. The values of literal strings and other "string" constants are *embedded* directly into a script's IL translation and *do* affect the total size. See the Shortening or removing constant strings section below to learn more.

Changing code identifiers

When the compiler translates a script to an IL form, it automatically *replaces* the identifiers (names) of declared variables, user-defined functions, function parameters, user-defined types, and type fields with compact *internal* identifiers based on the code's structure. These identifiers do not vary with the variable, function, and type names specified by the programmer in the source code. Therefore, changing identifiers in a script does *not* affect the script's compiled size.

Similarly, *named* arguments in function calls (e.g., `f(x = 1)`) are automatically converted to *positional* arguments (e.g., `f(1)`) when possible. Therefore, manually converting named arguments to positional

arguments typically has little to no effect on the compiled code's size.

The following example script contains a user-defined function with multiple verbose identifiers in its signature and body. The script calls this function and the plot() function using named arguments. It also declares multiple variables with lengthy identifiers:

```
//@version=6 indicator("Changing code identifiers demo") calculateExponentialMovingAverage(float sourceSeries, int lengthFactor) => float smoothingFactor = 2.0 / (lengthFactor + 1.0) float exponentialMovingAverage = na exponentialMovingAverage := ( nz(exponentialMovingAverage[1], sourceSeries) * (1 - smoothingFactor) + sourceSeries * smoothingFactor ) float fastExponentialMovingAverage = calculateExponentialMovingAverage(sourceSeries = close, lengthFactor = 12) float slowExponentialMovingAverage = calculateExponentialMovingAverage(sourceSeries = close, lengthFactor = 26) float movingAverageConvergenceDivergence = fastExponentialMovingAverage - slowExponentialMovingAverage float signalMovingAverage = calculateExponentialMovingAverage(movingAverageConvergenceDivergence, lengthFactor = 9) float histogramLineValue = movingAverageConvergenceDivergence - signalMovingAverage plot(series = movingAverageConvergenceDivergence, title = "MACD", color = color.blue) plot(series = signalMovingAverage, title = "Signal", color = color.orange) plot( series = histogramLineValue, title = "Histogram", color = histogramLineValue > 0 ? color.green : color.red, style = plot.style_columns )
```

The script version below reduces the verbosity of all the script's identifiers and changes the function calls to use positional arguments instead of named arguments. Although the code is now visibly more compact, it still translates to the *same IL form* as the previous version:

```
//@version=6 indicator("Changing code identifiers demo") ema(src, len) => sf = 2.0 / (len + 1.0) float ma = na ma := nz(ma[1], src) * (1 - sf) + src * sf maFast = ema(src = close, len = 12) maSlow = ema(src = close, len = 26) macd = maFast - maSlow sig = ema(macd, len = 9) hist = macd - sig plot(macd, "MACD", color.blue) plot(sig, "Signal", color.orange) plot(hist, "Histogram", hist > 0 ? color.green : color.red, style = plot.style_columns)
```

Note that:

- We also removed type keywords from most of the variable and parameter declarations. If a declaration does not include a type keyword, the compiler automatically infers the appropriate type based on how the script uses the variable or parameter. Therefore, omitting type keywords typically does not affect the size of the IL translation.

> NoteUnlike most user-defined identifiers, the names of enum members *can* impact the size of a compiled script. If an enum member does not have an assigned *title*, the member's title is the "const string" representation of its *name*, and that value is embedded directly into the script's IL form if the script uses it. See the Shortening or removing constant strings section for more information.

Swapping constant expressions and values

When possible, the compiler automatically evaluates expressions that return *constant* values during translation. Rather than including tokens to represent each part of a constant expression in the script's IL form, the compiler embeds the expression's *final value* into the translated code. This optimization prevents

the script from repeatedly recalculating the result at runtime. Therefore, replacing a constant expression with a literal value in the source code typically does not affect the size of the compiled script.

Similarly, if an if or switch structure or a conditional expression relies on a constant `true` or `false` condition, the compiler often simplifies the logic by replacing it with tokens for only the calculations that the script consistently executes. Therefore, manually simplifying such logic has little to no effect on the compiled script's size.

The following example script assigns an expression that returns a "const int" value to a variable. It then checks the expression `fixedValue == 6` in an if structure and adds 10 to the variable's value if the expression evaluates to `true`:

//@version=6 indicator("Constant expressions demo") int fixedValue = (1 + 2) * 10 / 5 if fixedValue == 6 fixedValue += 10 else fixedValue := 0 plot(fixedValue, "16")

The above script plots a constant value of 16. The `fixedValue` variable is initially assigned a value of 6, so the `fixedValue == 6` condition *always* evaluates to `true`, thus causing the variable's final value to change to 16 on each bar. The compiler recognizes this pattern in the code and simplifies the logic to `fixedValue = 16`. Therefore, the script has the *same* IL translation as the script below:

//@version=6 indicator("Constant expressions demo") int fixedValue = 16 plot(fixedValue, "16")

Reducing identical expressions and functions

If the compiler detects that multiple expressions are structurally *identical* and consistently evaluate to the *same* value, it automatically *reduces* the duplicate expressions to avoid repetitive calculations when possible. Rather than including tokens for each repetition, the compiler assigns the translated expression to an internal variable, then uses that variable in all code that requires the expression's result. This optimization helps conserve compiled tokens and potentially reduce the script's runtime.

Therefore, manually reducing the repetition of expressions that the compiler can automatically replace has little to no effect on the size of a compiled script.

> NoteThis optimization does not apply to *all* expressions that might appear identical in the source code. For example, it does not affect repeated expressions that rely on reference types, math.random() calls, or most imported library functions. Likewise, it does not typically affect equivalent expressions that list operations, values, or variables in *different* orders. Therefore, manually reducing such expressions *can* help reduce the compiled size.

The following script shows a scenario in which removing repeated expressions does not affect the size of the compiled code. The script defines a `posPercent()` function, which calculates the percentage of positive values in a series over a specified length. The script plots the results of seven calls to the function, each using identical arguments:

//@version=6 indicator("Identical expressions demo") //@function Calculates the percentage of positive values in the `source` series over a specified length. posPercent(float source, int length) => float posSum = math.sum(source > 0 ? 1 : 0, length) float result = 100 * posSum / length // Plot the result of the same `posPercent()` call seven times. plot(posPercent((close - open) / (high - low), 10), color = color.red, linewidth = 7) plot(posPercent((close - open) / (high - low), 10), color = color.orange, linewidth = 6)

plot(posPercent((close - open) / (high - low), 10), color = color.yellow, linewidth = 5) plot(posPercent((close - open) / (high - low), 10), color = color.green, linewidth = 4) plot(posPercent((close - open) / (high - low), 10), color = color.teal, linewidth = 3) plot(posPercent((close - open) / (high - low), 10), color = color.blue, linewidth = 2) plot(posPercent((close - open) / (high - low), 10), color = color.purple, linewidth = 1)

As the compiler translates the script, it verifies that all seven `posPercent()` expressions use identical value-type arguments, perform operations in the *same order*, and return the same result. It then restructures the script to execute only *one* call to the function and *reuse* the result in all the plot() calls. Therefore, the script has the *same* IL translation as the following script, which assigns one `posPercent()` call to a single variable and plots the variable's value seven times:

//@version=6 indicator("Identical expressions demo") //@function Calculates the percentage of `source` values that are positive over a specified length. posPercent(float source, int length) => float posSum = math.sum(source > 0 ? 1 : 0, length) float result = 100 * posSum / length //@variable The percentage of positive body-to-wick ratios over 10 bars. float percentPositive = posPercent((close - open) / (high - low), 10) // Plot the `percentPositive` value seven times. plot(percentPositive, color = color.red, linewidth = 7) plot(percentPositive, color = color.orange, linewidth = 6) plot(percentPositive, color = color.yellow, linewidth = 5) plot(percentPositive, color = color.green, linewidth = 4) plot(percentPositive, color = color.teal, linewidth = 3) plot(percentPositive, color = color.blue, linewidth = 2) plot(percentPositive, color = color.purple, linewidth = 1)

Similar to the above, if the compiler identifies user-defined functions or methods that are structurally identical, have the same parameter types, and consistently return matching results when passed the same arguments, it automatically *removes* the duplicate function definitions from the translated code rather than including tokens for each one, when possible. The compiled script then treats each call to a duplicate function as an *alias* for a call to the *first* function. Therefore, manually removing such duplicate functions from the code does not typically affect the compiled script's size.

> NoteThis optimization applies only to *non-exported* functions that can be verified as identical at compile time. It does not affect equivalent definitions that use rearranged calculations, reference types, or code that produces varying results. *Exported* functions and methods *always* add tokens to a compiled library, even if two or more of the functions are structurally identical. Programmers can often check whether a duplicate function is removed during compilation by profiling the script. See the Insignificant, unused, and redundant code section of the Profiling and optimization page to learn more. If two or more functions are structurally different but produce equivalent results, programmers can remove the redundant functions manually to reduce the compiled size.

To illustrate this behavior, the following script includes the `posPercent()` function from our previous example, along with a separate function named `f()` that follows the *same* structure and differs only in its *identifiers*. The script executes one `posPercent()` call and one `f()` call, then plots the results:

//@version=6 indicator("Removing identical functions demo") //@function Calculates the percentage of positive values in the `source` series over a specified length. posPercent(float source, int length) => float posSum = math.sum(source > 0 ? 1 : 0, length) log.info(str.tostring(posSum)) float result = 100 * posSum / length //@function A copy of `posPercent()`. Follows an identical structure, but uses different identifiers. f(float s, int l) => float ps = math.sum(s > 0 ? 1 : 0, l) log.info(str.tostring(ps)) float r = 100 * ps / l //@variable The ratio of the bar's body range to its wick range. float barRatio = (close - open) / (high - low) // Plot the

results from calls to both functions. plot(posPercent(barRatio, 10)) plot(f(barRatio, 20)) // The compiled script treats this `f()` call as a call to `posPercent()`.

Aside from differences in identifiers, both functions in the script are *identical*. They have the same required parameters, rely on the same value types, perform calculations in the same order, and return matching values when passed the same arguments. The compiler recognizes this pattern and includes tokens for only the *first* function definition in the IL translation. After this transformation, the `f()` call is treated as a call to `posPercent()`. Therefore, the above script has the **same** compiled form as the script below:

//@version=6 indicator("Removing identical functions demo") //@function Calculates the percentage of positive values in the `source` series over a specified length. posPercent(float source, int length) => float posSum = math.sum(source > 0 ? 1 : 0, length) log.info(str.tostring(posSum)) float result = 100 * posSum / length //@variable The ratio of the bar's body range to its wick range. float barRatio = (close - open) / (high - low) // Plot the results from two calls to the function. plot(posPercent(barRatio, 10)) plot(posPercent(barRatio, 20))

Note that:

- If we profile the previous script that contains an `f()` call, the Pine Profiler does *not* display any performance data next to the lines in that function's definition. Instead, it shows combined performance results for *both* function calls next to the `posPercent()` definition, confirming that the duplicate function is *removed* during translation.

Removing unused code

As the compiler translates a script, it verifies which parts of the code affect the script's *outputs*, such as plots, drawings, strategy orders, alerts, and Pine Logs. If the script contains code that does not affect any outputs, the compiler recognizes that the code is *unused* and automatically *discards* tokens for it in the IL translation. Therefore, manually removing unused code *does not* reduce a script's compiled size.

> NoteThis optimization also applies to unused inputs. However, some of the data from `input.*()` function calls is processed separately from a script's IL translation. Therefore, if an input is unused, it still *appears* in the script's "Settings/Inputs" tab, but it does *not* affect the script's IL token count.

The following example script contains a user-defined function that creates an array of simple moving averages over a range of specified lengths. It calls the function to create an array containing averages of close values over 2 to 50 bars, then uses the results to calculate a custom oscillator. However, the script *does not use* this code in any outputs; the only output from this script is a single `plot(close)` call:

//@version=6 indicator("Removing unused code demo") //#region // None of the code in this region contributes to any script output, such as a plot, drawing, or log. // Therefore, the compiler automatically *discards* it while translating the script. //@function Calculates simple moving averages of a `source` series over a range of lengths. //@param source The series of values to process. //@param minLength The length of the shortest SMA. //@param maxLength The length of the longest SMA. //@returns The ID of an array containing each calculated SMA, from shortest to longest. calcSMAs(float source, float minLength, float maxLength) => var array<float> result = array.new<float>() float total = ta.cum(source) result.clear() for length = minLength to maxLength float sum = total - total[length] float sma = sum / length result.push(sma) result //@variable References an array of SMAs over 2 to 50 bars. array<float> smas = calcSMAs(close, 2,

50) //@variable The average of all SMA values in the array. float avgSMA = smas.avg() //@variable The standard deviation of all values in the array. float dev = smas.stdev() //@variable The distance from the shortest SMA to the average SMA, relative to the standard deviation. float osc = (smas.first() - avgSMA) / dev //#endregion // This `plot()` call is the *only* code that generates an output, and it does not depend on the code above. plot(close)

The compiler recognizes that the user-defined function and all related calculations are unused by the script's outputs, so it excludes tokens for that code in the IL translation. Consequently, after this transformation, the script above has the *same* compiled result as the following script:

//@version=6 indicator("Removing unused code demo") plot(close)

Effective techniques to reduce script size

The following sections list some of the most common types of code changes that can help meaningfully reduce the size of a compiled script. These techniques are effective because they involve code revisions that the compiler *cannot* perform automatically during translation. The techniques covered below include the following:

- Shrinking or removing embedded data
- Shortening or removing constant strings
- Replacing unrolled statements with loops
- Encapsulating code in functions
- Replacing custom code with built-ins
- Splitting the script into parts

Shrinking or removing embedded data

One of the most common causes of the "too many tokens" error is a large quantity of *embedded data* in the translated code. A typical cause of large embedded data is the use of numerous *constant* values in calls to functions that support arbitrary numbers of arguments, such as array.from().

The Pine Script compiler embeds every literal "int", "float", "bool", and "string" value used in a source code into the script's IL translation. Likewise, it automatically evaluates constant expressions of these types, then embeds their results in the translated code to prevent recalculation at runtime. Additionally, if a script uses constant variables or expressions in function or method calls, the compiler *inlines* their values directly into each call site.

Therefore, hard-coding numerous constants into a script can significantly increase its compiled size, especially if the script uses them as arguments in function calls. For instance, a single array.from() call with thousands of constant arguments can consume the entire size budget on its own, depending on the saved values.

To reduce the amount of embedded data in a compiled script:

- Swap large constants with concise expressions that compute the same values at *runtime* where possible.
- Replace function calls and other expressions that use long lists of constants with logic that yields equivalent results at runtime. For instance, rather than defining an array.from() call with numerous

constant arguments, create a loop that iteratively *calculates* the elements and *inserts* them into the array using functions such as array.set() or array.push().
- If using a lengthy function call or other expression with many hard-coded values is *unavoidable*, consider moving the call into a user-defined function and *exporting* that function from a separate library. See the Splitting the script into parts section to learn more.

The following example script calculates a weighted moving average with custom precomputed weights. The script uses a single array.from() call with multiple literal “float” arguments to create an array of weights. It populates a single-row matrix with the array’s elements, calls the matrix.mult() method to calculate a weighted sum, then divides the result by the sum of the weights to compute the moving average. The array.from() call in this script significantly increases the size of the compiled code, because each literal value used as an argument is embedded directly into the IL translation. Adding multiple similar calls to this script can easily cause the “too many tokens” error:

//@version=6 indicator("Shrinking embedded data demo", overlay = true) //@variable References an array of precomputed weights. Each weight is a square number. // Every literal value defined in this `array.from()` call is *embedded* into the compiled code. var array<float> fixedWeights = array.from(1.0, 4.0, 9.0, 16.0, 25.0, 36.0, 49.0, 64.0, 81.0, 100.0, 121.0, 144.0, 169.0, 196.0, 225.0, 256.0, 289.0, 324.0, 361.0, 400.0, 441.0, 484.0, 529.0, 576.0, 625.0, 676.0, 729.0, 784.0, 841.0, 900.0, 961.0, 1024.0, 1089.0, 1156.0, 1225.0, 1296.0, 1369.0, 1444.0, 1521.0, 1600.0, 1681.0, 1764.0, 1849.0, 1936.0, 2025.0, 2116.0, 2209.0, 2304.0, 2401.0, 2500.0, 2601.0, 2704.0, 2809.0, 2916.0, 3025.0, 3136.0, 3249.0, 3364.0, 3481.0, 3600.0, 3721.0, 3844.0, 3969.0, 4096.0, 4225.0, 4356.0, 4489.0, 4624.0, 4761.0, 4900.0, 5041.0, 5184.0, 5329.0, 5476.0, 5625.0, 5776.0, 5929.0, 6084.0, 6241.0, 6400.0, 6561.0, 6724.0, 6889.0, 7056.0, 7225.0, 7396.0, 7569.0, 7744.0, 7921.0, 8100.0, 8281.0, 8464.0, 8649.0, 8836.0, 9025.0, 9216.0, 9409.0, 9604.0, 9801.0, 10000.0, 10201.0, 10404.0, 10609.0, 10816.0, 11025.0, 11236.0, 11449.0, 11664.0, 11881.0, 12100.0, 12321.0, 12544.0, 12769.0, 12996.0, 13225.0, 13456.0, 13689.0, 13924.0, 14161.0, 14400.0, 14641.0, 14884.0, 15129.0, 15376.0, 15625.0, 15876.0, 16129.0, 16384.0, 16641.0, 16900.0, 17161.0, 17424.0, 17689.0, 17956.0, 18225.0, 18496.0, 18769.0, 19044.0, 19321.0, 19600.0, 19881.0, 20164.0, 20449.0, 20736.0, 21025.0, 21316.0, 21609.0, 21904.0, 22201.0, 22500.0, 22801.0, 23104.0, 23409.0, 23716.0, 24025.0, 24336.0, 24649.0, 24964.0, 25281.0, 25600.0, 25921.0, 26244.0, 26569.0, 26896.0, 27225.0, 27556.0, 27889.0, 28224.0, 28561.0, 28900.0, 29241.0, 29584.0, 29929.0, 30276.0, 30625.0, 30976.0, 31329.0, 31684.0, 32041.0, 32400.0, 32761.0, 33124.0, 33489.0, 33856.0, 34225.0, 34596.0, 34969.0, 35344.0, 35721.0, 36100.0, 36481.0, 36864.0, 37249.0, 37636.0, 38025.0, 38416.0, 38809.0, 39204.0, 39601.0, 40000.0, 40401.0, 40804.0, 41209.0, 41616.0, 42025.0, 42436.0, 42849.0, 43264.0, 43681.0, 44100.0, 44521.0, 44944.0, 45369.0, 45796.0, 46225.0, 46656.0, 47089.0, 47524.0, 47961.0, 48400.0, 48841.0, 49284.0, 49729.0, 50176.0, 50625.0, 51076.0, 51529.0, 51984.0, 52441.0, 52900.0, 53361.0, 53824.0, 54289.0, 54756.0, 55225.0, 55696.0, 56169.0, 56644.0, 57121.0, 57600.0, 58081.0, 58564.0, 59049.0, 59536.0, 60025.0, 60516.0, 61009.0, 61504.0, 62001.0, 62500.0, 63001.0, 63504.0, 64009.0, 64516.0, 65025.0, 65536.0, 66049.0, 66564.0, 67081.0, 67600.0, 68121.0, 68644.0, 69169.0, 69696.0, 70225.0, 70756.0, 71289.0, 71824.0, 72361.0, 72900.0, 73441.0, 73984.0, 74529.0, 75076.0, 75625.0, 76176.0, 76729.0, 77284.0, 77841.0, 78400.0, 78961.0, 79524.0, 80089.0, 80656.0, 81225.0, 81796.0, 82369.0, 82944.0, 83521.0, 84100.0, 84681.0, 85264.0, 85849.0, 86436.0, 87025.0, 87616.0, 88209.0, 88804.0, 89401.0, 90000.0, 90601.0, 91204.0, 91809.0, 92416.0, 93025.0, 93636.0, 94249.0, 94864.0, 95481.0, 96100.0, 96721.0, 97344.0, 97969.0, 98596.0, 99225.0, 99856.0, 100489.0, 101124.0, 101761.0, 102400.0, 103041.0, 103684.0, 104329.0, 104976.0, 105625.0, 106276.0, 106929.0, 107584.0, 108241.0, 108900.0, 109561.0, 110224.0, 110889.0, 111556.0, 112225.0, 112896.0, 113569.0, 114244.0, 114921.0, 115600.0, 116281.0, 116964.0, 117649.0, 118336.0, 119025.0, 119716.0, 120409.0, 121104.0, 121801.0, 122500.0, 123201.0, 123904.0, 124609.0, 125316.0, 126025.0, 126736.0, 127449.0, 128164.0, 128881.0, 129600.0,

130321.0, 131044.0, 131769.0, 132496.0, 133225.0, 133956.0, 134689.0, 135424.0, 136161.0, 136900.0, 137641.0, 138384.0, 139129.0, 139876.0, 140625.0, 141376.0, 142129.0, 142884.0, 143641.0, 144400.0, 145161.0, 145924.0, 146689.0, 147456.0, 148225.0, 148996.0, 149769.0, 150544.0, 151321.0, 152100.0, 152881.0, 153664.0, 154449.0, 155236.0, 156025.0, 156816.0, 157609.0, 158404.0, 159201.0, 160000.0, 160801.0, 161604.0, 162409.0, 163216.0, 164025.0, 164836.0, 165649.0, 166464.0, 167281.0, 168100.0, 168921.0, 169744.0, 170569.0, 171396.0, 172225.0, 173056.0, 173889.0, 174724.0, 175561.0, 176400.0, 177241.0, 178084.0, 178929.0, 179776.0, 180625.0, 181476.0, 182329.0, 183184.0, 184041.0, 184900.0, 185761.0, 186624.0, 187489.0, 188356.0, 189225.0, 190096.0, 190969.0, 191844.0, 192721.0, 193600.0, 194481.0, 195364.0, 196249.0, 197136.0, 198025.0, 198916.0, 199809.0, 200704.0, 201601.0, 202500.0, 203401.0, 204304.0, 205209.0, 206116.0, 207025.0, 207936.0, 208849.0, 209764.0, 210681.0, 211600.0, 212521.0, 213444.0, 214369.0, 215296.0, 216225.0, 217156.0, 218089.0, 219024.0, 219961.0, 220900.0, 221841.0, 222784.0, 223729.0, 224676.0, 225625.0, 226576.0, 227529.0, 228484.0, 229441.0, 230400.0, 231361.0, 232324.0, 233289.0, 234256.0, 235225.0, 236196.0, 237169.0, 238144.0, 239121.0, 240100.0, 241081.0, 242064.0, 243049.0, 244036.0, 245025.0, 246016.0, 247009.0, 248004.0, 249001.0, 250000.0, 251001.0, 252004.0, 253009.0, 254016.0, 255025.0, 256036.0, 257049.0, 258064.0, 259081.0, 260100.0, 261121.0, 262144.0, 263169.0, 264196.0, 265225.0, 266256.0, 267289.0, 268324.0, 269361.0, 270400.0, 271441.0, 272484.0, 273529.0, 274576.0, 275625.0, 276676.0, 277729.0, 278784.0, 279841.0, 280900.0, 281961.0, 283024.0, 284089.0, 285156.0, 286225.0, 287296.0, 288369.0, 289444.0, 290521.0, 291600.0, 292681.0, 293764.0, 294849.0, 295936.0, 297025.0, 298116.0, 299209.0, 300304.0, 301401.0, 302500.0, 303601.0, 304704.0, 305809.0, 306916.0, 308025.0, 309136.0, 310249.0, 311364.0, 312481.0, 313600.0, 314721.0, 315844.0, 316969.0, 318096.0, 319225.0, 320356.0, 321489.0, 322624.0, 323761.0, 324900.0, 326041.0, 327184.0, 328329.0, 329476.0, 330625.0, 331776.0, 332929.0, 334084.0, 335241.0, 336400.0, 337561.0, 338724.0, 339889.0, 341056.0, 342225.0, 343396.0, 344569.0, 345744.0, 346921.0, 348100.0, 349281.0, 350464.0, 351649.0, 352836.0, 354025.0, 355216.0, 356409.0, 357604.0, 358801.0, 360000.0, 361201.0, 362404.0, 363609.0, 364816.0, 366025.0, 367236.0, 368449.0, 369664.0, 370881.0, 372100.0, 373321.0, 374544.0, 375769.0, 376996.0, 378225.0, 379456.0, 380689.0, 381924.0, 383161.0, 384400.0, 385641.0, 386884.0, 388129.0, 389376.0, 390625.0, 391876.0, 393129.0, 394384.0, 395641.0, 396900.0, 398161.0, 399424.0, 400689.0, 401956.0, 403225.0, 404496.0, 405769.0, 407044.0, 408321.0, 409600.0, 410881.0, 412164.0, 413449.0, 414736.0, 416025.0, 417316.0, 418609.0, 419904.0, 421201.0, 422500.0, 423801.0, 425104.0, 426409.0, 427716.0, 429025.0, 430336.0, 431649.0, 432964.0, 434281.0, 435600.0, 436921.0, 438244.0, 439569.0, 440896.0, 442225.0, 443556.0, 444889.0, 446224.0, 447561.0, 448900.0, 450241.0, 451584.0, 452929.0, 454276.0, 455625.0, 456976.0, 458329.0, 459684.0, 461041.0, 462400.0, 463761.0, 465124.0, 466489.0, 467856.0, 469225.0, 470596.0, 471969.0, 473344.0, 474721.0, 476100.0, 477481.0, 478864.0, 480249.0, 481636.0, 483025.0, 484416.0, 485809.0, 487204.0, 488601.0, 490000.0, 491401.0, 492804.0, 494209.0, 495616.0, 497025.0, 498436.0, 499849.0, 501264.0, 502681.0, 504100.0, 505521.0, 506944.0, 508369.0, 509796.0, 511225.0, 512656.0, 514089.0, 515524.0, 516961.0, 518400.0, 519841.0, 521284.0, 522729.0, 524176.0, 525625.0, 527076.0, 528529.0, 529984.0, 531441.0, 532900.0, 534361.0, 535824.0, 537289.0, 538756.0, 540225.0, 541696.0, 543169.0, 544644.0, 546121.0, 547600.0, 549081.0, 550564.0, 552049.0, 553536.0, 555025.0, 556516.0, 558009.0, 559504.0, 561001.0, 562500.0, 564001.0, 565504.0, 567009.0, 568516.0, 570025.0, 571536.0, 573049.0, 574564.0, 576081.0, 577600.0, 579121.0, 580644.0, 582169.0, 583696.0, 585225.0, 586756.0, 588289.0, 589824.0, 591361.0, 592900.0, 594441.0, 595984.0, 597529.0, 599076.0, 600625.0, 602176.0, 603729.0, 605284.0, 606841.0, 608400.0, 609961.0, 611524.0, 613089.0, 614656.0, 616225.0, 617796.0, 619369.0, 620944.0, 622521.0, 624100.0, 625681.0, 627264.0, 628849.0, 630436.0, 632025.0, 633616.0, 635209.0, 636804.0, 638401.0, 640000.0) //@variable References a single-row matrix containing the elements of the `fixedWeights` array. // The script uses this matrix to calculate the dot product for the average's numerator. var matrix<float> weightMatrix = matrix.new<float>() // Add the `fixedWeights` array's elements to the matrix on the first bar. if barstate.isfirst

weightMatrix.add_row(0, fixedWeights) //@variable References an array of consecutive `close` values. var array<float> prices = array.new<float>(fixedWeights.size()) // Queue the current `close` value into the `prices` array on each bar. prices.push(close) prices.shift() // Calculate and plot the weighted average. float weightedAvg = weightMatrix.mult(prices).first() / fixedWeights.sum() plot(weightedAvg, "Weighted average")

The array in this example contains a series of *square numbers*, where the first element is 1 (`1 * 1`), the second is 4 (`2 * 2`), and so on. In other words, each element is the square of `i + 1`, where `i` refers to the element's index. Therefore, we can reduce our script's size by calculating the array's values within a *loop* rather than defining them directly in the code.

The script version below creates a persistent array using an `array.new<float>()` call, then uses a loop to populate it on the first bar. On each iteration, the script calculates a square value, then calls the array.push() function to push the value into the array. This version calculates the same results as the previous script, but its compiled size is substantially smaller because we avoided defining a long list of literal values. Additionally, this source code is visibly shorter and simpler to maintain:

//@version=6 indicator("Shrinking embedded data demo", overlay = true) //@variable References an array of weights calculated on the first bar. Each weight is a square number. var array<float> fixedWeights = array.new<float>() //@variable References a single-row matrix containing the elements of the `fixedWeights` array. // The script uses this matrix to calculate the dot product for the average's numerator. var matrix<float> weightMatrix = matrix.new<float>() // Populate the weight array and matrix on the first bar. if barstate.isfirst // Although this loop executes over a fixed range and uses consistent operations, // the compiler does not embed all the calculated values into the translated code. // It includes only necessary tokens for the loop's logic, so the compiled size is much smaller. for i = 1 to 800 float weight = i * i fixedWeights.push(weight) weightMatrix.add_row(0, fixedWeights) //@variable References an array of consecutive `close` values. var array<float> prices = array.new<float>(fixedWeights.size()) // Queue the current `close` value into the `prices` array on each bar. prices.push(close) prices.shift() // Calculate and plot the weighted average. float weightedAvg = weightMatrix.mult(prices).first() / fixedWeights.sum() plot(weightedAvg, "Weighted average")

> NoteWhen modifying code to reduce its compiled size, it's also important to consider the impact that such changes have on a script's *runtime*. For instance, populating an array using a loop requires more runtime resources than initializing an array from a list of constants. However, in our example above, we ensure that the script initializes and populates the array on the *first bar* only, thus minimizing the runtime impact. See the Profiling and optimization page to learn more about writing performant code.

Shortening or removing constant strings

As explained in the section Shrinking or removing embedded data above, the compiler embeds constant values, including strings, into a script's IL translation where possible. Consequently, using lengthy constant strings in a script can significantly increase the size of the compiled code, especially if the script passes the strings as multiple arguments in function calls.

If a script with lengthy or multiple constant strings causes the "too many tokens" error, try any of the following to reduce the script's compiled size:

- Remove all lengthy constant strings that the script's logic does not absolutely require, and try to reduce the length of all others.

- Avoid using a long constant string for multiple arguments in function calls. Each instance of the argument's value is *inlined* at the call site when possible. Therefore, using the value for multiple arguments can *multiply* its effect on the compiled code's size.
- Check for repetitive expressions that use the same constant string. If possible, replace the repeated expressions with a single expression that executes in a loop. For instance, if the script contains multiple str.format() calls with the same formatting string, replace them with a loop that executes a *single* str.format() call multiple times.
- Shorten the titles of enum members used by the script's logic. The titles of enum members are embedded into parts of the code that require their values. By default, the title of each member is the "const string" representation of its *name*. Therefore, if an enum member does not have a specified title, shorten the member's name or assign it a short title.
- If the script requires a specific lengthy string, but does not require it to be a constant, create a user-defined function that returns a "simple" or "series" version of the value, or the ID of a collection that contains the value, then *export* the function from a separate library. See the Splitting the script into parts section to learn more.

Replacing unrolled statements with loops

Loop unrolling is a common technique in which the programmer writes a set of repetitive statements and expressions to perform a sequence of calculations without a loop. This technique can marginally improve a script's runtime in some cases. However, each instance of repeated code that the compiler cannot reduce *adds tokens* to the script's IL translation. Consequently, writing several pieces of unrolled code can impact the size of a compiled script.

Therefore, if a script contains multiple unrolled statements or expressions, defining an equivalent, concise loop to execute the necessary calculations can help reduce the script's compiled size.

For example, the following script calculates the number of close values that are greater than the current value over the past 50 bars. The script calculates the value using 50 written statements that perform the same operations using different values. All of the statements are unique and cannot be reduced by the compiler. Therefore, each one adds tokens to the script's IL translation:

//@version=6 indicator("Replacing unrolled code with loops demo") //@variable The number of past `close` values, over the latest 50 bars, that are greater than the current `close` value. int higherCloses = 0 // Add 1 to the `higherCloses` value for each past `close` value that is greater than the current value. // All of these statements add separate tokens to the translated code. higherCloses += close[1] > close ? 1 : 0 higherCloses += close[2] > close ? 1 : 0 higherCloses += close[3] > close ? 1 : 0 higherCloses += close[4] > close ? 1 : 0 higherCloses += close[5] > close ? 1 : 0 higherCloses += close[6] > close ? 1 : 0 higherCloses += close[7] > close ? 1 : 0 higherCloses += close[8] > close ? 1 : 0 higherCloses += close[9] > close ? 1 : 0 higherCloses += close[10] > close ? 1 : 0 higherCloses += close[21] > close ? 1 : 0 higherCloses += close[22] > close ? 1 : 0 higherCloses += close[23] > close ? 1 : 0 higherCloses += close[24] > close ? 1 : 0 higherCloses += close[25] > close ? 1 : 0 higherCloses += close[26] > close ? 1 : 0 higherCloses += close[27] > close ? 1 : 0 higherCloses += close[28] > close ? 1 : 0 higherCloses += close[29] > close ? 1 : 0 higherCloses += close[30] > close ? 1 : 0 higherCloses += close[31] > close ? 1 : 0 higherCloses += close[32] > close ? 1 : 0 higherCloses += close[33] > close ? 1 : 0 higherCloses += close[34] > close ? 1 : 0 higherCloses += close[35] > close ? 1 : 0 higherCloses += close[36] > close ? 1 : 0 higherCloses += close[37] > close ? 1 : 0 higherCloses += close[38] > close ? 1 : 0 higherCloses += close[39] > close ? 1 : 0 higherCloses += close[40] > close ? 1 : 0 higherCloses += close[41] > close ? 1 : 0 higherCloses += close[42] > close ? 1 : 0 higherCloses += close[43] > close ? 1 : 0 higherCloses += close[44] > close ? 1 : 0

higherCloses += close[45] > close ? 1 : 0 higherCloses += close[46] > close ? 1 : 0 higherCloses += close[47] > close ? 1 : 0 higherCloses += close[48] > close ? 1 : 0 higherCloses += close[49] > close ? 1 : 0 higherCloses += close[50] > close ? 1 : 0 plot(higherCloses)

We can reduce our script's token count by replacing all the repetitive statements with a single loop that executes the required logic iteratively. The script version below uses a for loop that increments its counter variable (`i`) from 1 to 50. The single statement in the loop's local block increments the `higherCloses` value on each iteration where the value of `close[i]` is greater than the close value on the current bar. This version of the script achieves the same result as the previous version, but its compiled size is significantly smaller:

//@version=6 indicator("Replacing unrolled code with loops demo") //@variable The number of past `close` values, over the latest 50 bars, that are greater than the current `close` value. int higherCloses = 0 // Add 1 to the `higherCloses` value for each past `close` value that is greater than the current value. // This loop consumes fewer tokens than an equivalent set of unrolled statements, and it is easier to maintain. for i = 1 to 50 higherCloses += close[i] > close ? 1 : 0 plot(higherCloses)

> Note
>
> If a set of unrolled statements depend on past values of variables or expressions defined within a *local block* or a *function call scope*, keeping them *out* of a loop is typically necessary for correct results. For example, a statement containing a call to the ta.rsi() function causes compiler warning CW10003 when used inside a loop, because the function call references past values from its scope internally.
>
> If lengthy unrolled statements cannot be condensed into a loop without affecting their results, try encapsulating the logic in a *function* and replacing the statements with concise function calls. See the Encapsulating code in functions section below to learn more.

Encapsulating code in functions

Function calls in Pine Script are *not inlined*, meaning the compiler does not replace each function call with a copy of the function's body during translation, unlike the compilers for some other languages. However, the compiler does typically include tokens for *each argument* in a function call, including the function's *default* arguments. Therefore, the size that each function call contributes to a compiled script often varies with the number of *parameters*.

Calls to functions with few parameters typically have low impact on the size of the compiled code, unless the arguments are lengthy constant strings. By contrast, calls to functions with many parameters, or to functions that support an arbitrary number of arguments, can significantly impact a compiled script's size in some cases. See the Shrinking or removing embedded data section above for an example.

If a script contains multiple large, repetitive statements or expressions, or calls to functions with many parameters, encapsulating that code in a separate user-defined function containing as few parameters as possible can help reduce the size of a compiled script.

> Tip*Exporting* an encapsulating function from a library can help further reduce a script's compiled size. Libraries are compiled *separately* from the scripts that import them, and Pine Script applies *separate*

> *limits* to imported code. See the Splitting the script into parts section to learn more.

The following example script calculates an average Efficiency Ratio across several periods that are powers of two. The script declares a variable with an initial value of 0, then uses nine addition assignment operations that use the same expressions but different length values. It then divides the value by 9 and plots the result. Each of the repetitive statements in this script contributes multiple tokens to the script's IL translation. We cannot simplify the script by replacing the statements with a loop, because the ta.change() and math.sum() calls require *one* execution on *every bar* for correct results:

//@version=6 indicator("Encapsulating code in functions demo") //@variable The average Efficiency Ratio measured across multiple periods that are powers of 2. float avgEfficiency = 0.0 // Calculate each period's value as the ratio of absolute change to the total bar-by-bar change over the period, // and add it to the total. Each repetitive expression here contributes to the script's compiled size. avgEfficiency += math.abs(ta.change(close, 8)) / math.sum(math.abs(close - close[1]), 8) avgEfficiency += math.abs(ta.change(close, 16)) / math.sum(math.abs(close - close[1]), 16) avgEfficiency += math.abs(ta.change(close, 32)) / math.sum(math.abs(close - close[1]), 32) avgEfficiency += math.abs(ta.change(close, 64)) / math.sum(math.abs(close - close[1]), 64) avgEfficiency += math.abs(ta.change(close, 128)) / math.sum(math.abs(close - close[1]), 128) avgEfficiency += math.abs(ta.change(close, 256)) / math.sum(math.abs(close - close[1]), 256) avgEfficiency += math.abs(ta.change(close, 512)) / math.sum(math.abs(close - close[1]), 512) avgEfficiency += math.abs(ta.change(close, 1024)) / math.sum(math.abs(close - close[1]), 1024) avgEfficiency += math.abs(ta.change(close, 2048)) / math.sum(math.abs(close - close[1]), 2048) // Divide the result by 9 to calculate the average. avgEfficiency /= 9 plot(avgEfficiency, "Average ER")

We can reduce our script's size by encapsulating the structure of the repetitive expressions within a user-defined function, then replacing each expression with a concise function call. The script version below defines a `calcER()` function, which calculates the Efficiency Ratio of close values over a specified length. It then uses a call to this function instead of the repeated expressions in each addition assignment operation. Each `calcER()` call is not inlined; the compiler adds tokens only for the function's reference and its single argument. Therefore, in addition to being visibly more compact, this version has a smaller compiled size:

//@version=6 indicator("Encapsulating code in functions demo") //@function Calculates the Kaufman's Efficiency Ratio for `close` values over a specified length. // The compiler does not inline this function's body into each function call. The calculations in the // function's body are defined only once in the compiled code. calcER(int length) => math.abs(ta.change(close, length)) / math.sum(math.abs(close - close[1]), length) //@variable The average Efficiency Ratio measured across multiple periods that are powers of 2. float avgEfficiency = 0.0 // Calculate each period's value as the ratio of absolute change to the total bar-by-bar change over the period, // and add it to the total. // The concise `calcER()` calls below add tokens only for the function's reference and each "int" argument. // Therefore, each repetitive statement here has a lower impact on the compiled size. avgEfficiency += calcER(8) avgEfficiency += calcER(16) avgEfficiency += calcER(32) avgEfficiency += calcER(64) avgEfficiency += calcER(128) avgEfficiency += calcER(256) avgEfficiency += calcER(512) avgEfficiency += calcER(1024) avgEfficiency += calcER(2048) // Divide the result by 9 to calculate the average. avgEfficiency /= 9 plot(avgEfficiency, "Average ER")

Replacing custom code with built-ins

Pine Script features many built-in functions and variables that help simplify code and streamline script creation. Each written call to a built-in function produces compiled tokens for only the function's reference

and the call's arguments. Similarly, many built-in variables produce minimal tokens for referencing data calculated outside the script.

Sometimes, programmers create custom code that mirrors built-ins or yields equivalent results. Custom code implementations provide extra freedom to modify calculations as needed. However, they also typically contribute *more tokens* to a script's IL translation. Therefore, if a script performs calculations that can be done with built-ins, replacing those calculations with built-ins can often help reduce a script's compiled size.

> TipIn many cases, using built-ins instead of custom code can also help improve a script's *runtime* performance. See the Using built-ins section of the Profiling and optimization page to learn more.

The following example script computes a weighted moving average of hlc3 values over an input length, and it calculates the daily volume-weighted average price. The script uses custom functions to compute both values rather than using equivalent built-ins. All the code defined within those functions adds tokens to the script's IL translation, thus increasing its size:

//@version=6 indicator("Replacing custom code with built-ins demo", overlay = true) //@variable The number of bars in the weighted moving average. int lengthInput = input.int(20, "WMA length", 2) //@function A custom function that calculates the weighted moving average of a series over a fixed length. // This function is equivalent to using `ta.wma()` with a consistent `length` argument. wma(float source, simple int length) => var float sum = 0.0 var float numerator = 0.0 var float denominator = 0.5 * length * (length + 1) sum += source - nz(source[length]) numerator += length * source - nz(sum[1]) bar_index < length ? na : numerator / denominator //@function A custom function that calculates the daily volume-weighted average price. // This function produces the same result stored by the `ta.vwap` variable. calcDailyVWAP() => var float numerator = 0.0 var float denominator = 0.0 if timeframe.change("1D") numerator := 0.0 denominator := 0.0 numerator += hlc3 * volume denominator += volume numerator / denominator //@variable The 20-bar weighted average of `hlc3` values. float wmaSeries = wma(hlc3, 20) //@variable The daily VWAP. float vwapSeries = calcDailyVWAP() // Plot both series on the chart. plot(wmaSeries, "WMA", wmaSeries > vwapSeries ? color.green : color.red, 3) plot(vwapSeries, "VWAP", color.orange)

Implementing custom functions for our script's calculations is *unnecessary*. We can use the built-in ta.wma() function to calculate the weighted moving average, and use the ta.vwap variable to retrieve the daily volume-weighted average price. The script version below implements these changes. This version computes the same results, is visibly shorter, and has a smaller compiled size:

//@version=6 indicator("Replacing custom code with built-ins demo", overlay = true) //@variable The number of bars in the weighted moving average. int lengthInput = input.int(20, "WMA length", 2) //@variable The 20-bar weighted average of `hlc3` values. float wmaSeries = ta.wma(hlc3, 20) //@variable The daily VWAP. float vwapSeries = ta.vwap // Plot both series on the chart. plot(wmaSeries, "WMA", wmaSeries > vwapSeries ? color.green : color.red, 3) plot(vwapSeries, "VWAP", color.orange)

Splitting the script into parts

If a script causes the "too many tokens" error, and no manual source code revisions resolve it without compromising the script's logic, the typical solution is to split the code into two or more smaller scripts or offload reusable code components into libraries. Distributing a script's logic across multiple scripts directly reduces the compiled tokens from a single source code.

Programmers can try any of the following to split a large script's logic, depending on the types of code it uses and the tasks that it performs:

- If the script contains multiple large function or method definitions, consider exporting them from a separate library, and importing that library into the script. Likewise, if the script contains several lengthy statements, consider defining exported library functions to encapsulate those statements, if possible. Moving functions to libraries often fixes the problem, because libraries are compiled *independently* and their code does *not* directly affect the compiled size of the scripts that import them. A single script can import multiple libraries, provided their *combined* size does not exceed *1 million* tokens. Refer to the Libraries page to learn more about library scripts and how to use them.
- If the script performs multiple independent tasks, create *independent scripts* for each task instead. For example, if a script calculates multiple complex indicators for no reason other than to consolidate them into a single source code, move each indicator's logic into a separate script to keep compiled sizes low.
- If the script performs a lengthy set of related calculations, consider creating separate indicators for *intermediate stages* of those calculations, then use plots and source inputs to pass data between them. To learn more about connecting scripts to other scripts, refer to the How to apply an indicator or strategy to another indicator article in our Help Center. Note that charts allow up to *24* total indicator-on-indicator connections, and a maximum of *11* indicators connected *sequentially* (i.e., `indicator_1 -> indicator_2 -> ... indicator_11`). See this related article in our Help Center for more information.

The function "X" should be called on each calculation for consistency. It is recommended to extract the call from this scope.

This compiler warning occurs if a call to a built-in function or user-defined function (or method) inside a conditional structure or loop retrieves data from its calculations on *past bars* by using the `[]` history-referencing operator or other functions that rely on history internally. History-dependent function calls that execute either conditionally or iteratively can cause **unintended results**. A similar warning also occurs if a ternary or and/or operation executes a history-dependent function call conditionally.

As a script runs a dataset, Pine's runtime system successively *commits (saves)* the data for the script's variables and expressions on each closed bar to internal time series structures. For the system to save *consistent* time series data for a function call, the call must execute **once** on **each** bar's close. If the call does not execute on each bar, or if it executes more than once on some bars, the time series for the call contains an **inconsistent** history of past data.

This behavior affects the results of history-referencing operations that access the history of a function call's parameters, variables, or expressions. To ensure consistent historical references in a function call, move the call to the script's **global scope** and *outside* conditional expressions.

> NoteReferencing the history of a local variable declared inside conditional structures or loops causes a slightly different compiler warning, but the solution is similar. Move the variable declaration to the *global scope* to ensure consistent history-referencing operations on that variable.

For example, the following script defines a `previousValue()` function that uses the [] operator with an offset of 1 to access the last saved value of its `source` parameter as of the previous bar. The script calls the function once every three bars in an if structure to retrieve a past value of the bar_index variable and plots

the result on the chart. Because the call executes only on each third bar, its `source` series contains data for only those bars. Consequently, the `source[1]` expression *does not* retrieve a value from one bar back in this case; it retrieves the `source` value for the last bar on which the call executed – three bars ago:

//@version=6 indicator("Conditional history-dependent call demo") //@function Returns the last saved value of the `source` parameter as of the previous bar. // // This function requires one execution on each bar's close for consistent results, because it uses the // `[]` operator to access past values from its `source` series. // // If a written call does not execute on a bar, nothing is saved to the time series, and the `[]` // operator cannot consistently retrieve a value corresponding to only one bar back. previousValue(source) => source[1] //@variable Stores the result from calling `previousValue()` once every three bars, and `na` on other bars. float pastValue = na if bar_index % 3 == 0 // The `previousValue()` call below causes the warning. // The call's `source` series contains only `bar_index` values for each third bar, and not other bars. // Therefore, the `source[1]` expression executed by the call does not retrieve the `bar_index` value from one bar back. // Instead, it retrieves the value from *three* bars back, because that's the last bar on which the call executed. pastValue := previousValue(source = bar_index) // Plot the result alongside the bar index. plot(bar_index, "Bar index") plot(pastValue, "Previous value from local call", chart.fg_color, 5, plot.style_circles)

Note that:

- This script has the same behaviors we describe below if it uses the statement `float pastValue = bar_index % 3 == 0 ? previousValue(source = bar_index) : na` instead of the if structure.
- The compiler displays the warning on line 21, where the local `previousValue()` call occurs, to inform the user that the call might cause inconsistent results.

If the conditions that control the executions of a function call vary, the historical references in the call's scope access data from a *varying* number of bars back, even if the specified offset is a constant. This behavior is a common cause of unexpected results. For example, in the following script version, we changed the if structure's condition to `close > open`. With this change, the plotted results now show the bar_index value from an inconsistent number of bars back, because the `previousValue()` call builds a series of `source` data for only the bars whose closing price exceeds their opening price:

//@version=6 indicator("History-dependent call across varying intervals demo") //@function Returns the last saved value of the `source` parameter as of the previous bar. // // This function requires one execution on each bar's close for consistent results, because it uses the // `[]` operator to access past values from its `source` series. // // If a written call does not execute on a bar, nothing is saved to the time series, and the `[]` // operator cannot consistently retrieve a value corresponding to only one bar back. previousValue(source) => source[1] //@variable Stores the result from calling `previousValue()` on bars that close above the opening price. float pastValue = na if close > open // The `previousValue()` call below causes the warning. // The call's `source` series contains `bar_index` values only for bars whose close exceeds their open. // The bars back that the `source[1]` expression represents *varies* in this case. pastValue := previousValue(source = bar_index) // Plot the result alongside the bar index. plot(bar_index, "Bar index") plot(pastValue, "Previous value from local call", chart.fg_color, 5, plot.style_circles)

To ensure that the `previousValue()` call consistently returns a value from *one* bar back, regardless of conditions, we must move the call to the *global* scope, outside conditional expressions, so that its `source` series stores values for *consecutive bars*.

For example, this script version executes the `previousValue()` call in the global scope and reassigns the result to the `pastValue` variable inside the if structure. Now, on bars where the plot appears, the result consistently shows the bar_index value from one bar back:

```
//@version=6 indicator("Consistent history-dependent call demo") //@function Returns the last saved value of the `source` parameter as of the previous bar. // // This function requires one execution on each bar's close for consistent results, because it uses the // `[]` operator to access past values from its `source` series. // // If a written call does not execute on a bar, nothing is saved to the time series, and the `[]` // operator cannot consistently retrieve a value corresponding to only one bar back. previousValue(source) => source[1] //@variable Stores the previous bar's `bar_index` value if `close > open`, and `na` otherwise. float pastValue = na //@variable Stores the previous bar's `bar_index` value on each bar. // This call executes on each bar, so its history stores consecutive past values. float globalPrev = previousValue(source = bar_index) if close > open // This code does not cause a warning. // Reassigning the call's result to another variable does not change the call's behavior. pastValue := globalPrev // Plot the result alongside the bar index. plot(bar_index, "Bar index") plot(pastValue, "Previous value from global call", chart.fg_color, 5, plot.style_circles)
```

Inconsistent time series storage can significantly change the results from calls to built-in functions that reference history internally, such as those in the `ta` namespace. For instance, the ta.sma() function calculates the average of a specified number of recent values from a series. If a script does not execute a call to this function once on *every* bar's closing tick, the call builds an *inconsistent* history for its calculations, and its results can change significantly.

The following example executes two calls to ta.sma() – one in the script's global scope, and the other inside an if structure – and then plots the result on the chart. As shown below, the two plots display very different values. The *global* ta.sma() call executes on each bar, so its `source` series records values for *consecutive* bars to use in the resulting 20-bar average. By contrast, the local call does *not* use consecutive bars in its average. Instead, the call calculates the average of close values from the latest 20 bars on which it executes:

```
//@version=6 indicator("Consistent vs. inconsistent built-in call demo", overlay = true) //@variable The 20-bar average of consecutive values from the `close` series. float consistentSMA = ta.sma(source = close, length = 20) //@variable The average of `close` values over the latest 20 bars where `close > close[1]` returns `true`. // On bars where the condition is `false`, the value is `na`. float inconsistentSMA = if close > close[1] // This call causes a warning, and returns a very different result, // because it uses inconsistent historical references internally. ta.sma(close, 20) // Plot the two results for comparison. plot(consistentSMA, "Consistent SMA", color.blue, 3) plot(inconsistentSMA, "Inconsistent SMA", color.orange, 3)
```

> TipFor more *advanced* details about this behavior, refer to the Time series in scopes section of the Execution model page.

Why this behavior?

This behavior is necessary because forcing a function call to execute once on each bar, regardless of scope, would cause erroneous or unexpected results, especially in functions that produce *side effects*, such as drawings, alerts, and Pine Logs, or functions that use persistent variable declaration modes.

For example, forcing calls to the label.new() function inside an if structure to execute on each bar would cause a new label drawing to appear on every bar instead of only when the condition's value is `true`. Likewise, forcing a local call to a function that modifies var variables to execute once on each bar can cause unintended values to persist in calculations.

Exceptions

Not all functions use previous values from their scopes in their calculations. Calls to such functions *do not* require execution on every bar for correct results. For example, the built-in math.max() function returns the maximum value from its specified arguments. It does not use past values from its scope in its calculations. Therefore, calling the function conditionally or iteratively does *not* affect the function's behavior.

If the use of a function call in a local block does not cause a compiler warning, it is typically safe to use the call in that block without affecting the call's calculations. However, if the warning occurs, move the function call to the global scope and outside ternary or and/or operations to ensure consistency. If you choose to keep a function call within a local block or a conditional expression despite encountering a warning, debug the script carefully to avoid unintended results.

The inputs with the `inline` argument `"X"` might not align correctly in the "Settings/Inputs" tab

This warning occurs if two or more inputs with the same `inline` and `group` arguments also specify `title` arguments. Placing multiple inputs with titles on the same line can cause *inconsistent* input alignment in the "Settings/Inputs" tab.

Historically in Pine Script®, all `input*()` function calls with non-empty `inline` arguments did not horizontally align with other inputs in the "Settings/Inputs" tab. Non-inline inputs applied a consistent amount of horizontal spacing between the title and the input field, while inline inputs applied more minimal spacing. This misalignment often negatively affected input readability and aesthetics. Programmers typically attempted to work around this issue by inserting combinations of Unicode spaces into input titles. While that solution helped reduce misalignment, it did not usually produce precise and consistent alignment behavior.

Pine Script now uses an improved model to align all script inputs. With this model, programmers can achieve consistent horizontal alignment for inline inputs *without* using workarounds such as padding titles with invisible characters. However, the updated alignment rules apply only if all inputs except for the first one on each line have an *empty title*. If multiple inputs on a line include non-empty titles, those inputs might not align as intended.

The image below demonstrates the differences in old and current input alignment behaviors. The left side shows the old behaviors, and the right shows the current behaviors. On the left side, the inline inputs align their fields differently from other inputs in the tab, and they do not align with any of the non-inline inputs. By contrast, most inline inputs on the right side do align the with the non-inline inputs. However, the inputs on the line with the "Pivot Stats" checkbox still follow the same alignment as before, because each of those inputs includes a separate *title*:

To ensure consistent horizontal input alignment in the "Settings/Inputs" tab, follow these steps:

1. Use a *non-empty string* as the `title` argument for only the **first** `input*()` call that uses specific `inline` and `group` arguments.
2. For all other `input*()` calls that use the same `inline` and `group` arguments, include an *empty string* ("") as the `title` argument.

> TipTo promote input readability, set the title on each line to clarify the meaning of *all* inputs on the line, and use tooltips as needed for additional context. If multiple inputs require distinct titles, place them on *separate lines* to ensure they align cleanly in the tab.

The following example script demonstrates how inline input fields align differently if multiple inputs on the same line have titles. In the first group of inline input.int() calls, each call includes a *non-empty* `title` argument. In the second group of inline inputs, only the *first* call specifies a non-empty title, while the others use an empty title. The inputs in the second group align as intended in the “Settings/Inputs” tab. The inputs in the first group do not, and they cause this compiler warning:

//@version=6 indicator("Inline input alignment demo") string TT = "Set the MACD calculation's slow MA length, fast MA length, and signal length, respectively." //@variable The source series for the MACDs. float sourceInput = input.source(close, "Source", "Source series for MACD calculations.") // Group of inputs for the first MACD. Each input on the line has a title, causing incorrect alignment. int fastInput1 = input.int(12, "Slow", inline = "0", group = "First MACD") int slowInput1 = input.int(26, "Fast", inline = "0", group = "First MACD") int sigInput1 = input.int(9, "Signal", inline = "0", group = "First MACD", tooltip = TT) // Group of inputs for the second MACD. Only the first input on the line has a title, resulting in consistent alignment. // The tooltip on the line already clarifies the meaning of the inputs, so specifying multiple titles is unnecessary. int fastInput2 = input.int(24, "Lengths", inline = "0", group = "Second MACD") int slowInput2 = input.int(52, "", inline = "0", group = "Second MACD") int sigInput2 = input.int(18, "", inline = "0", group = "Second MACD", tooltip = TT) // Calculate the two MACDs and plot their difference. [macd1, _, _] = ta.macd(sourceInput, fastInput1, slowInput1, sigInput1) [macd2, _, _] = ta.macd(sourceInput, fastInput2, slowInput2, sigInput2) plot(macd1 - macd2)

Note that:

- The last input for each group includes a `tooltip` argument to clarify the meaning of all inputs on the line. We can move the argument to the other input.int() calls for the same line, and the tooltip still appears at the end of the line. However, if multiple inputs on the same line have a specified tooltip, only the last one appears in the script’s settings. See the Input function parameters section of the Inputs page to learn more about `tooltip`, `inline`, and other parameters for inputs.
- Although all the integer inputs use the *same* `inline` argument, both sets of three inputs use *different* `group` arguments, so they appear on *separate lines* in the “Settings/Inputs” tab.

Memory limits exceeded

The most common cause of this error is the retrieval of custom objects and collections from `request.*()` functions such as request.security(). Other possible causes include unnecessary drawing updates, excess historical buffer capacity, or inefficient use of max_bars_back().

The sections below explain these issues and potential solutions to fix them.

Requesting collections and other objects with `request.*()` calls

The “Memory limits exceeded” error most often occurs when a script calls request.security() or other `request.*()` functions to create custom objects or collections on the dataset for another symbol or timeframe.

When a script executes a data request, the retrieved data is copied and stored in memory on *each bar*, enabling the script to reference that data on subsequent bars as it executes across the main dataset. Storing requested results for each bar can consume a lot of memory, depending on the type of data. Requesting the references (IDs) of large collections on each bar can easily lead to excessive memory consumption.

For example, the script below calls request.security() to request the result of a user-defined function call evaluated on the "1D" timeframe. The custom function (`dataFunction()`) creates a "float" array and assigns its ID to a persistent variable declared using the `var` keyword. On each bar, the function pushes a new balance of power (BOP) value into its array and returns the array's ID. Each time that the request.security() call evaluates the `dataFunction()` call, it creates a *new copy* of the array and returns that copy's ID. Storing a new copy of the array on each bar requires a lot of memory. Consequently, this script can *exceed* the memory limits when it runs on datasets that have a lengthy history:

```
//@version=6 indicator("Request exceeding memory limits demo") //@variable User-input length for calculating average of BOP values. int avgLengthInput = input.int(5, "Average BOP Length", minval = 1) //@function Inserts a BOP value into a persistent array and returns the array's ID. dataFunction() => //@variable Stores the ID of a persistent "float" array. var array<float> dataArray = array.new<float>(0) // Push a new BOP percentage into the array. float bop = (close - open) / (high - low) * 100 dataArray.push(bop) // Return the array's ID. dataArray // This request executes a `dataFunction()` call on data for the "1D" timeframe and returns the ID of a copied array. // Storing a copy of the requested array for each bar requires a lot of memory. Therefore, this request can easily // cause the script to exceed its memory limit. array<float> reqData = request.security(syminfo.tickerid, "1D", dataFunction()) //@variable The latest BOP value from the `reqData` array, or `na` if the requested ID is `na` or the array is empty. float latestValue = na //@variable The average of the latest specified number of BOP values. float avgBOP = na if not na(reqData) // Assign the last value stored in the `reqData` array to the `latestValue` variable. latestValue := reqData.last() //@variable The number of elements in the `reqData` array. int dataSize = reqData.size() //@variable References a slice containing the recent BOP values to use in the average. array<float> lastValues = reqData.slice(math.max(dataSize - avgLengthInput, 0), dataSize) // Calculate the average of the values accessed by the `lastValues` slice. avgBOP := lastValues.avg() // Display the latest BOP value and the average in a separate pane. hline(0, "Zero line", color.gray, hline.style_dotted) plot(latestValue, "BOP", latestValue >= 0 ? color.aqua : color.orange, style = plot.style_columns) plot(avgBOP, "Avg BOP", color.purple, linewidth = 3)
```

How do I fix this?

Optimize the script's data requests and limit the data returned by the requests to ensure that only the *minimum required* data is preserved in memory.

The typical solution to minimize the saved data from a request depends on the script's requirements:

- If the script must access only the *latest state* of a requested collection or other object, structure the request to return that object's ID on the *last bar* only.
- If the script must access a requested object in the main context only when specific conditions occur, structure the request to not return the object's ID when it is not required.
- If the script performs specific calculations on data from a requested collection or other object, and it does *not* require access to that object in other parts of the main context, structure the request to perform those calculations internally and return the *result* rather than returning the object's ID.

The sections below explore these scenarios using the initial example above.

Return the last state only

If a script requires only the *latest* state of a requested collection, use a conditional structure or expression with barstate.islast as the condition to limit retrieving a copy of that collection to the *last* bar where the latest state is available.

For example, suppose we want our initial script to plot the BOP value and display the current average in a table. Users cannot see the past results displayed by a table as the script loads on historical bars. Only the *final* result as of the latest bar is visible to users. Therefore, to minimize the script's memory demands, we can structure the script's request to return a usable ID on the last bar only.

Below, we modified the script's custom function to return a two-item tuple. On the last bar, the function returns the BOP value and the array's ID. On all previous bars, it returns only the BOP value and an na array ID. The script requests the result of a call to this revised function. It plots the retrieved BOP value on each bar, then calculates and displays the requested array's average in a table on the last bar if the retrieved array ID is not na:

//@version=6 indicator("Return the last state only demo") //@variable User-input length for calculating average of BOP values. int avgLengthInput = input.int(5, "Average BOP Length", minval = 1) //@function Inserts a BOP value into a persistent array and returns a tuple. The tuple's second item is the // ID of a BOP array on the last bar, and `na` on other bars. dataFunction() => //@variable Stores the ID of a persistent "float" array. var array<float> dataArray = array.new<float>(0) // Push a new BOP percentage into the array. float bop = (close - open) / (high - low) * 100 dataArray.push(bop) // Return a tuple. The first item is the BOP value, and the second is the `dataArray` ID on the last bar only. [bop, barstate.islast ? dataArray : na] // This request returns a valid array ID only when `barstate.islast == true` on the "1D" timeframe. On all previous bars, // the `reqData` ID is `na`, and no copy of the `dataFunction()` call's internal array is stored in the main context. [latestValue, reqData] = request.security(syminfo.tickerid, "1D", dataFunction()) // Calculate the average and draw a table on the last chart bar if the `reqData` ID is not `na`. if barstate.islast and not na(reqData) // Assign the last value stored in the `reqData` array to the `latestValue` variable. latestValue := reqData.last() //@variable The number of elements in the `reqData` array. int dataSize = reqData.size() //@variable References a slice containing the recent BOP values to use in the average. array<float> lastValues = reqData.slice(math.max(dataSize - avgLengthInput, 0), dataSize) // Calculate the average of the values accessed by the `lastValues` slice. float avgBOP = lastValues.avg() // Display latest average value in a single-cell table. var table displayTable = table.new(position.bottom_right, 1, 1, color.purple) displayTable.cell(0, 0, "Avg of last " + str.tostring(avgLengthInput) + " BOPs: " + str.tostring(avgBOP, "##.##") + "%", text_color = color.white) // Display the latest BOP value and the average in a separate pane. hline(0, "Zero line", color.gray, hline.style_dotted) plot(latestValue, "BOP", latestValue >= 0 ? color.aqua : color.orange, style = plot.style_columns)

Return IDs on some bars

If a script needs to access a requested collection or other object in its main context, but *not* on *every bar*, structure the request to use conditional structures or expressions that return the object's ID on only necessary bars, and na on other bars. The logic in the script's main context can then handle the na gaps in the requested series as necessary.

Suppose we want to modify our initial script to calculate the average, high, and low daily BOP for each *completed month*. We can structure the script's request to collect BOP values across the days in each month and return an array ID for the necessary calculations only after a month ends. The `dataFunction()`

function in the following script version pushes BOP values for each bar in a month into an array and returns a two-item tuple. The tuple contains the BOP value and the array's ID only on the first bar of a month. On other bars, the tuple contains the BOP value and an na ID. The script uses a call to this updated function in its data request. On bars where the requested array ID is not na, the script calculates the average, high, and low values from that array, then assigns the results to persistent variables for plotting:

//@version=6 indicator("Return IDs on some bars demo") //@function Inserts a BOP value into a persistent array and returns a tuple. The tuple's second item is the // ID of a BOP array on the first bar of each month, and `na` on other bars. dataFunction() => //@variable Stores the ID of a persistent "float" array. var array<float> dataArray = array.new<float>(0) //@variable Temporarily saves the BOP value from the opening bar of each month. var float openBOP = na //@variable Is `true` on the first bar of each month, and `false` on other bars. bool isNewMonth = timeframe.change("1M") // Clear the array and push the `openBOP` value into it on the month's second bar. if isNewMonth[1] dataArray.clear() if not na(openBOP) dataArray.push(openBOP) openBOP := na // Push a new BOP percentage into the array. float bop = (close - open) / (high - low) * 100 // Save the `bop` value to the `openBOP` variable on the month's first bar, and to the array on other bars. switch isNewMonth => openBOP := bop => dataArray.push(bop) // Return a tuple. The first item is the BOP value. The second is the `dataArray` ID on the first bar of the month. [bop, isNewMonth ? dataArray : na] // This request returns a valid array ID as its second item only on the first daily bar of each month. // On other bars, the second item is `na`, so no copy of the `dataFunction()` call's array is saved in the main context. [latestValue, reqData] = request.security(syminfo.tickerid, "1D", dataFunction()) // Declare persistent variables to store average, high, and low values calculated from the `reqData` array. var float avgBOP = na, var float hiBOP = na, var float loBOP = na // Update the variables on each bar where the `reqData` ID is not `na`. if not na(reqData) avgBOP := reqData.avg(), hiBOP := reqData.max(), loBOP := reqData.min() // Display the latest BOP value alongside the average, high, and low in a separate pane. hline(0, "Zero line", color.gray, hline.style_dotted) plot(latestValue, "BOP", latestValue >= 0 ? color.aqua : color.orange, style = plot.style_columns) plot(avgBOP, "Avg BOP", color.purple, linewidth = 3) plot(hiBOP, "High BOP", color.green) plot(loBOP, "Low BOP", color.red)

Return calculated results

If a script requires the *result* of a calculation involving a collection or other large object, and the script does *not* need to access that object in its main context, define a function that performs the calculation and returns the result directly rather than returning the underlying object's ID. Then, use a call to that function as the `expression` argument in the `request.*()` call. Avoiding returning object references from requested contexts altogether when they are unnecessary helps minimize memory demands.

For example, we can directly resolve the error in our initial script by structuring the custom `dataFunction()` function to perform the average calculation and return the result without returning an array ID on any bar. The function in the script version below returns a tuple containing only the BOP and average value. The script requests the result of a call to the updated function and directly plots the calculated values. With this change, the request does not cause any copies of the `dataFunction()` call's internal array to persist in memory in the main context:

//@version=6 indicator("Return calculated results demo") //@variable User-input length for calculating average of BOP values. int avgLengthInput = input.int(5, "Average BOP Length", minval = 1) //@function Calculates and returns the BOP and average BOP values. // The function uses an array internally but does not return its ID. dataFunction() => //@variable Stores the ID of a persistent "float" array. var array<float> dataArray = array.new<float>(0) // Push a new BOP percentage into the array. float bop = (close - open) /

(high - low) * 100 dataArray.push(bop) //@variable The number of elements in the `reqData` array. int dataSize = dataArray.size() //@variable References a slice containing the recent BOP values to use in the average. array<float> lastValues = dataArray.slice(math.max(dataSize - avgLengthInput, 0), dataSize) // Return a tuple containing only the BOP and average values. [bop, lastValues.avg()] // This request returns the BOP and average directly. It does not return an array ID. // Therefore, no copies of the `dataFunction()` call's internal array are saved in the main context. [latestValue, avgBOP] = request.security(syminfo.tickerid, "1D", dataFunction()) // Display the latest BOP value and the average in a separate pane. hline(0, "Zero line", color.gray, hline.style_dotted) plot(latestValue, "BOP", latestValue >= 0 ? color.aqua : color.orange, style = plot.style_columns) plot(avgBOP, "Avg BOP", color.purple, linewidth = 3)

Other possible error sources and their fixes

There are a few other factors that can significantly impact a script's memory demands, including:

- An excessive use of `request.*()` function calls.
- An unnecessary or improper use of the `max_bars_back` parameter.
- Excessive historical buffer calculations.
- Inefficient drawing management.
- The generation of many strategy orders and trades.

The sections below explain the best practices for reducing these issues to optimize a script's memory use.

Minimize `request.*()` calls

The `request.*()` functions can be computationally expensive to call, because they retrieve data from additional datasets and perform extra calculations on that data. Data requests often require significant runtime and memory resources. Excessive or inefficient data requests can easily cause a script to reach its memory limit.

This memory consumption is especially substantial for scripts that request data from lower timeframes with request.security_lower_tf(), because the function returns the IDs of arrays that contain intrabar data for *each* bar in the script's main dataset. For example, a script that requests one-minute data from a "1D" chart creates a new array containing hundreds of intrabar data points on each bar. The system must allocate sufficient memory to store each of those large arrays so that the script can access them later in the main context. Maintaining that much data in memory requires a significant amount of resources.

Programmers can reduce the memory requirements of a script's requests by:

- Removing *unnecessary* `request.*()` function calls from the script.
- Changing the timeframe of a request to a higher timeframe to reduce the number of retrieved data points.
- Combining multiple requests to the *same* dataset into a single `request.*()` call.
- Using the `request.*()` function's `calc_bars_count` parameter to limit the number of historical bars in the requested dataset.

See the Minimizing `request.*()` calls section of the Profiling and optimization page to learn more about optimizing data requests.

Use `max_bars_back` only when necessary

The `max_bars_back` parameter of an indicator or strategy script sets the size of the historical buffers for *all* series in a script. Each buffer determines the number of historical data points that the system *maintains in memory* for the script's variables and expressions.

By default, the Pine Script® runtime system attempts to automatically define an appropriate buffer for each variable and expression. Therefore, using the `max_bars_back` parameter or max_bars_back() function is typically necessary only in the rare case where the system *cannot* determine a sufficient buffer size. See the error page The requested historical offset (X) is beyond the historical buffer's limit (Y) to learn about the potential causes of this issue.

If you encounter a buffer error and must manually set the sizes of a script's historical buffers by using the `max_bars_back` parameter or the max_bars_back() function, choose the *smallest* buffer sizes that accommodate your script's historical references. Setting buffer sizes to store *more* data than necessary causes an *excessive* use of memory resources.

Minimize historical buffer calculations

As explained above, the Pine Script runtime system creates *historical buffers* for all variables and expressions in a script. It determines the size of each buffer based on the *historical references* that the script performs via the [] history-referencing operator or the functions that reference history internally.

As a script loads on a dataset, historical references to distant points in the dataset can cause the system to *reload* the script and *increase* the size of necessary historical buffers. Each increase to historical buffer sizes leads to increased memory consumption. In some cases, excessive buffer resizing can cause the script to exceed the memory limits. Therefore, ensure a script references only *necessary* historical data in its calculations. When possible, modify the script's logic to avoid referencing very distant points in history.

Specifying a `calc_bars_count` argument in the indicator() or strategy() declaration statement can help reduce memory issues, because it *restricts* the number of *historical bars* that the script can use for its calculations. Similarly, using the max_bars_back() function to manually define the appropriate size for a buffer can help reduce buffer calculations. When using this function to specify the size of historical buffers, choose the **smallest** possible size that accommodates the script's historical references, to avoid unnecessary memory use.

To learn more about historical buffer calculations and how to optimize them, see the Minimizing historical buffer calculations section of the Profiling and optimization page.

Reduce drawing updates for tables

Tables display only their *latest state* as of the last available bar on the chart. The previous states of a table on historical bars are *not* visible to script users. Therefore, updating and displaying tables across historical bars is typically *unnecessary*. To use the least memory in table drawings, prevent unneccessary updates by defining logic that creates the table object once and then populates it on only the *last bar*.

To create a table object on only one bar, assign the result of the table.new() call to a variable declared with the var keyword. Then, populate the table on the last bar by placing calls to table.cell() or the `table.cell_set_*()` functions inside a conditional structure that uses barstate.islast as its condition. See the Tables page to learn more.

Do not update drawings on historical bars

As with tables, all other drawing objects, such as lines and labels, show only their *latest* states after the script loads on the chart. Users cannot see the visuals update as the script executes across history; users see only the drawing updates that occur on *realtime bars*.

To help reduce a script's runtime and memory use, eliminate updates to drawings on historical bars wherever possible. Refer to the Reducing drawing updates section of the Profiling and optimization page for more information.

Minimize the total drawings stored for a chart

Drawing objects, such as lines and labels, can consume a lot of memory, especially if a script *recreates* drawings unnecessarily.

For example, if a script draws a line between two points, and then needs to update the line's coordinates later, newcomers to Pine might opt to delete the existing line and create a new one with the updated coordinates. However, such an approach is *inefficient* and leads to unnecessary resource consumption. The more efficient approach is to use the built-in *setter functions* for the drawing type, such as line.set_x2(), to modify the coordinates of the original drawing without creating a new object.

In general, to help minimize the resource usage of drawings, optimize them by using any of the following approaches:

- Eliminate unnecessary redrawing by assigning a drawing object's ID to a variable declared with the var keyword and modifying that object with the available setter functions.
- Remove unnecessary drawings with the `*.delete()` functions (e.g., line.delete() and label.delete()) to release resources.
- Reduce a script's drawing limits by specifying arguments for the `max_lines_count`, `max_labels_count`, `max_boxes_count`, or `max_polylines_count` parameters of the indicator() or strategy() declaration statement.

Filter dates in strategies

The total number of trades or orders simulated by strategies can impact memory consumption. When running strategy scripts that generate frequent orders on large datasets, reduce the number of unnecessary historical orders and trades that persist in memory by limiting the *starting point* of the strategy.

To limit the starting point of a strategy, a simple and effective approach is to use a conditional structure that activates the strategy's order placement commands only when the bar's opening or closing time comes after a specified date.

See the How do I filter trades by a date or time range? portion of our Strategies FAQ page for an example of this technique.

The requested historical offset (X) is beyond the historical buffer's limit (Y)

In Pine Script®, a single script executes from start to end on each bar of the chart. After each execution on a confirmed bar, Pine's runtime system *commits (saves)* data for a script's variables and expressions on that bar to *fixed-sized* historical buffers. The script can retrieve past bar values from these buffers by using the [] history-referencing operator or the functions that reference history internally. For example, the expression `myVar[500]` retrieves the last saved value of the `myVar` variable as of 500 bars back.

By default, the runtime system automatically sizes a script's buffers to store only what the script requires for the historical references that it performs across historical bars:

- The system first checks the historical references that the script executes on the first 244 bars of the dataset, then sets an initial buffer size based on those references. If the script uses constant historical offsets, or if the historical offsets used on subsequent bars do not exceed the buffer sizes, the script's historical references work without issues.
- If the script requests data from beyond an initial buffer's limit while executing across other historical bars, the system *increases* that buffer's size and *restarts* the script. This process repeats as necessary until the system reaches the restart limit or the script loads successfully without exceeding a buffer's boundaries.

This runtime error can occur in some rare cases if the system fails to determine the appropriate size of a buffer on historical bars after restarting a script several times. However, the error more commonly occurs when a script executes out-of-bounds historical references on realtime bars, because the system *does not* continue to resize historical buffers after the script loads.

For example, the following script plots the value of close variable from 500 bars back on historical bars, then attempts to plot the value from *1000* bars back on realtime bars. The script initially loads successfully across the chart's history. However, the script halts and raises the runtime error once a *new realtime tick* becomes available. The error is also reproducible by enabling Bar Replay and pressing "Forward" once. The automatic historical buffer for the close series in this example contains only **500** past values. It does *not* contain 1000, because the script does not reference values from 1000 bars back on any *historical* bars, and the system does not resize the buffer on realtime bars. Consequently, the historical offset of 1000 on the first realtime bar is *beyond* the historical buffer's limit:

//@version=6 indicator("Historical buffer error on realtime bars demo") // This code works as expected on historical bars, but it fails with an error on the first *realtime tick*. // On historical bars, the maximum historical offset that the script requires is 500 bars back, so the system sets the // buffer size for the `close` series to include only 500 past values. As such, an offset of 1000 is *out of bounds*. float myVar = close[barstate.ishistory ? 500 : 1000] plot(myVar)

To resolve such errors, programmers must ensure that their scripts' historical buffers have a sufficient size to accommodate historical references on all bars. See below to learn more.

> NoticeThe maximum possible buffer size for most variables is **5000**, and referring further into the past causes a *different* runtime error. To prevent a script from referencing more than 5000 previous bars of data, use an expression such as `math.min(myOffset, 5000)` as the offset in history-referencing operations.

Potential fixes

The following sections explain the different ways to explicitly define appropriate buffer sizes to accommodate historical references. Using these techniques is necessary only if a script's historical references cause this runtime error. In most other cases, we typically recommend relying on automatic buffer sizing.

> NoticeWhen defining buffer sizes manually using any of these techniques, ensure that you specify the **smallest** possible buffer sizes that the script requires for its historical references. Buffers that are larger than what a script requires can significantly impact the script's performance. For example, setting a buffer's size to include 5000 past values when a script requires only 700 bars back causes an excessive use of memory. In some cases, buffers that are too large can push a script to its memory limit.

Use the `max_bars_back()` function

The built-in max_bars_back() function explicitly sets the initial size of the historical buffer for a specified variable. If automatic buffer sizing fails, or if a script references deeper history on only realtime bars, programmers can use this function to specify appropriate buffers for specific series to accommodate historical references for up to 5000 bars back.

For example, to resolve the error in our initial script above, we can add the call `max_bars_back(close, 1000)` to specify that the buffer for the close series stores at least the latest 1000 past values. With the appropriate buffer size explicitly defined, the script performs its historical references successfully on realtime bars without raising an error:

//@version=6 indicator("Use `max_bars_back()` demo") //@variable The `close` value from 500 bars back on historical bars, and 1000 bars back on realtime bars. float myVar = close[barstate.ishistory ? 500 : 1000] plot(myVar) // This call prevents the runtime error. It specifies that the buffer for the `close` series stores data for // 1000 previous bars, which is the *minimum* historical depth that the script requires on realtime bars. max_bars_back(close, 1000)

Note that:

- Scripts can include a max_bars_back() function call at any location in the script. In our example above, the call occurs *after* the historical reference on the close variable.

Use the `max_bars_back` parameter

The `max_bars_back` parameter of the indicator() and strategy() functions is an alternative to max_bars_back() that defines the initial size of *all* series in a script if it has a specified argument. This parameter can offer convenience in cases where multiple series require manually defined buffers. However, it's crucial to understand that increasing the size of every historical buffer in a script can *negatively* impact its runtime performance and memory consumption. Therefore, for resource efficiency, we typically recommend using the max_bars_back() function to size only specific buffers instead of sizing every buffer with this parameter. The only exception is if *all* or *most* of the series in the script actually require historical buffers with a specific size and the runtime system fails to determine that size automatically.

Use the larger historical reference on early bars

An alternative way to increase the size of a specific historical buffer is to use the *largest* required offset in historical references on *early bars*, regardless of whether the script requires that offset on those bars. The runtime system analyzes that historical offset while the script loads and sets the buffer's size accordingly.

For example, the script version below includes the offset of 1000 in the history-referencing operation on the *first bar*, where the barstate.isfirst value is `true`. This change makes no practical difference in the historical output, because the operation returns na for any nonzero offset on the first bar. However, it resolves the

error because it causes the script to request data for 1000 bars back on a *historical bar*, forcing the system to create a buffer that maintains the latest 1000 past close values:

//@version=6 indicator("Larger historical reference on early bars demo") // This code works as expected on historical and realtime bars. // Setting the offset to 1000 on the first bar forces the system to create a 1000-bar buffer for the `close` series. float myVar = close[barstate.isfirst ? 1000 : barstate.ishistory ? 500 : 1000] plot(myVar)

Historical buffer errors in realtime drawings

A common cause of the historical offset error is the creation of drawing objects that anchor to the *past* while the script executes on *realtime bars*. All Pine drawings that rely on chart coordinates convert their x-coordinates into *timestamps* by referencing the time series internally, even if the programmer defines those coordinates using bar_index values.

Therefore, if a script draws into the past on realtime bars and raises this error, the typical solution is to set an explicit buffer size for the time variable.

For example, the script below draws a horizontal line from the bar at `bar_index - 500` to the current bar while executing on realtime bars only. The script loads successfully across historical bars, but it quickly halts and raises the historical offset error on the first *realtime tick*. Although the script does not explicitly use the [] operator, the line drawing itself references the time series internally to convert its `x1` and `x2` coordinates. Converting the `x1` coordinate to a timestamp requires access to the time value from *500 bars back*. However, the default buffer for that series stores fewer than 500 past values, because the script does not request the value from 500 bars back while it *loads* across the historical dataset:

//@version=6 indicator("Realtime buffer error from a drawing demo") if barstate.isrealtime // This call causes an error on the first realtime tick. // Pine drawings internally convert index-based x-coordinates into *timestamps* by referencing values // from the `time` series. The coordinate `bar_index - 500` converts to `time[500]` internally, but the default // buffer in this example stores data for *fewer* than 500 past `time` values. line.new(x1 = bar_index - 500, y1 = close, x2 = bar_index, y2 = close)

To resolve the error, programmers must use any of the techniques described in the Potential fixes section to explicitly set the size of the historical buffer for the time series. The simplest solution is to use the max_bars_back() function to set the appropriate buffer size.

In the script version below, we added the call `max_bars_back(time, 500)` to specify that the buffer for the time series stores values for the latest 500 past bars. This change prevents the script from requesting a value that is outside the buffer's boundaries when converting the `bar_index - 500` coordinate into a timestamp:

//@version=6 indicator("Preventing time-related buffer errors in realtime drawings demo") if barstate.isrealtime // Draws a horizontal line from the bar 500 bars back to the current bar. line.new(x1 = bar_index - 500, y1 = close, x2 = bar_index, y2 = close) // This call ensures that the buffer for the `time` series contains a sufficient history for the line drawing to convert // its `x1` coordinate into a timestamp. With an appropriate buffer size defined, no error occurs. max_bars_back(time, 500)

Overview

Introduction

Pine Script® uses *runtime errors*, *compilation errors*, and *compiler warnings* to help prevent unintended or erroneous script behaviors:

- Runtime errors occur under specific conditions as a script runs on a dataset. If a script encounters a runtime error while it executes on the chart, it *stops* executing and displays a red “exclamation point” icon in the *status line*. The user can click the icon to view the error message and the bar index on which the error occurred. Pine includes built-in runtime errors for invalid arguments in function calls, history-referencing operations with invalid offsets, and various other error conditions. Programmers can also define custom runtime errors for specific conditions in their scripts by using the runtime.error() function.
- Compilation errors occur at compile time, *before* a script begins to run on a dataset. If a script contains *invalid syntax* or other issues that *prevent* compilation, the Pine Editor highlights the problematic code in red and displays an error message to indicate the cause. If the script is on the chart, users can also access the error message from the script’s status line.
- Compiler warnings also occur before a script begins to run. In contrast to compilation errors, these warnings inform users about syntax that does *not* prevent compilation but can cause *unintended* results or behaviors. They can also occur for other reasons, such as using deprecated features or old Pine Script versions. The Pine Editor highlights code that causes a warning in orange, then displays a message outlining the issue and potential solutions. We recommend following the suggestions in compiler warnings to help ensure that a script works as intended.

This page provides an overview of the common runtime errors, compilation errors, and warnings described in the User Manual’s “Errors and warnings” section.

> TipYou can also find information about other errors and warnings for specific features on the pages that describe those features in the User Manual. For example, the sections in the Type system page include details about common type-related errors.

Error reference table

The reference table below lists the current errors and warnings documented in this section of the User Manual. The codes in the “Error/warning code” column link to the pages that contain detailed information about the issues and possible solutions.

> NoteThis list is not exhaustive. New pages for other common errors and warnings will likely be added over time.

Error/warning code	Message	Solution
CE10101	The condition of the “X” statement must evaluate to a “bool” value.	Use only expressions that return values of the “bool” type as the conditions in if and switch statements.
CE10117	Compiled code contains too many tokens.	This error occurs if the compiled form of the script is *too large*. Typical fixes include reducing embedded data, shortening or removing constant strings, replacing repetitive code with loops or concise function calls, distributing logic across separate scripts, or moving code into libraries.
CW10003	The function “X” should be called on each calculation for consistency. It is recommended to extract the call from this scope.	The function call might cause *unintended results* when executing inside a conditional structure or loop because it relies on data from *past bars*. Move the call to the *global scope*, and outside conditional expressions, to ensure consistent history-based calculations.
CW10026	The inputs with the `inline` argument "X" might not align correctly in the “Settings/Inputs” tab.	Multiple inputs with matching `inline` and `group` arguments include non-empty `title` arguments, causing misalignment in the tab. To ensure consistent horizontal alignment, use a non-empty title for the *first input* on each line, and an empty title for all others.
RE10139	Memory limits exceeded.	There are multiple possible causes and solutions. A common cause is using `request.*()` calls to request large collections of data across bars. The usual solution for that case is to optimize the requests to return collection IDs only when **necessary**. Consult the error page to learn more.
RE10143	The requested historical offset (X) is beyond the historical buffer’s limit (Y).	This error occurs if a script references the history of a variable or expression from *too many* bars back. A typical solution is to use the max_bars_back() function to specify how many past bars of data to include in the historical buffer for the referenced series.

Alerts

How do I make an alert available from my script?

In indicator scripts, there are two ways to define triggers for alerts:

- Using the alertcondition() function
- Using the alert() function

In strategy scripts, there are also two ways to define alert triggers:

- Using the alert() function
- Using order fill events

These methods make alert triggers available but do not *create* alerts directly. Users must create alerts using a script's alert triggers by selecting the appropriate trigger in the "Condition" dropdown of the "Create Alert" dialog box.

Programmers can define multiple alert triggers of one or more types in a script.

How are the types of alerts different?

Usability

Any script can include calls to the alertcondition() and alert() functions within their code. However, alertcondition() calls have no effect unless the script is an *indicator*. Libraries can *export* functions containing alert() calls, but they cannot issue alert triggers directly.

Order fill alert triggers are available only from strategies.

Options for creating alerts

Each alertcondition() call in an indicator script defines one distinct trigger and one corresponding option in the "Condition" dropdown menu of the "Create Alert" dialog box. If the user wants multiple alerts, they must create each one *separately*.

By contrast, if a script includes one or more alert() function calls, only *one* option appears in the "Condition" dropdown menu, titled "Any alert() function call". Selecting this option creates a *single alert* that activates based on the occurrences of *any* executed alert() call.

Similarly, for strategy scripts, the "Order fills and and alert() function calls" or "Order fills only" option in the "Condition" dropdown menu creates an alert that fires when *any* order fill event occurs.

How alerts activate

The alertcondition() function operates exclusively in an indicator's global scope. Scripts cannot include calls to this function within any *local block*, such as the *indented* code within an if structure. The function triggers an alert when its specified `condition` is `true`. Users can set the allowed *frequency* of the alert trigger using the "Frequency" field in the "Create Alert" dialog box.

The alert() function has no `condition` parameter. Scripts trigger the alerts on any alert() call based on each call's `freq` argument. Therefore, programmers typically include such calls within the local scopes of conditional structures to control when they execute.

Order fill alert triggers are available from strategies *automatically* without requiring extra code. However, programmers can customize the default alert messages. These alerts fire on order fill events, which occur when the broker emulator fills a strategy's *orders*.

Messages

The `message` parameter of the alertcondition() function populates the "Message" field of the "Create Alert" dialog box with a default message, which script users can customize to suit their alert needs. It accepts a "const string" argument, meaning its value cannot change after compilation. However, the argument can include placeholders to make the message's information *dynamic*.

The `message` parameter of the alert() function accepts a "series string" argument, allowing programmers to create *dynamic messages* that can include "string" representations of a script's calculated values. Unlike alertcondition(), this function does not populate a "Message" field in the "Create Alert" dialog box, and it does not process placeholders. Programmers can allow users to customize alert() messages by creating inputs in the script's settings.

Order fill alerts have a default message that describes a strategy's order fill event. This default message contains strategy-specific placeholders, which the alert replaces with current strategy information each time it fires. Programmers can override the default message using the `//@strategy_alert_message` compiler annotation, which allows text and strategy placeholders, but *not* script variables. Script users can edit the default message from the "Message" field in the "Create Alert" dialog.

The `alert_message` parameter in a strategy's order placement commands allows programmers to define distinct messages for each order fill event. The parameter accepts "series string" values that can change on each event. To use the values from this parameter in a strategy's order fill alerts, include the `{{strategy.order.alert_message}}` placeholder in the `//@strategy_alert_message` annotation, or include it in the "Message" field when creating an alert.

Limitations

The alertcondition() function has some limitations:

- Each active alert condition counts toward the total number of alerts the user's plan allows.
- Every alertcondition() call contributes to the script's plot count.
- Only indicators can issue alert triggers with this function. Other script types do not raise a compilation error when they include calls to this function in their code, but each call has **no effect**.

By contrast, all calls to the alert() function count as **one** alert, regardless of the number of calls in the code. In addition, alert() calls *do not* contribute to a script's plot count.

Similarly, if a user creates a strategy alert based on order fill events, it counts as **one** alert, even though it can fire multiple times with distinct messages from different order executions.

Example `alertcondition()` alert

The script below demonstrates a simple alertcondition() call that triggers alerts when the current bar's close is above the value from the previous chart bar. It also uses a plotshape() call to indicate each bar where the `triggerCondition` occurred:

//@version=6 indicator("Simple alert demo", overlay = true) // Create condition to trigger alert. bool triggerCondition = close > close[1] // Use `triggerCondition` for the `condition` parameter. // Define a title for the alert in the menu and a message to send with the alert. alertcondition(condition = triggerCondition, title = "Example `alertcondition` Alert", message = "The example `alertcondition` alert was triggered.") // Plot a shape when `triggerCondition` is true to visually mark where alerts occur. plotshape(triggerCondition, "Trigger Condition", shape.xcross, color = color.fuchsia)

See this section of the Alerts page to learn more.

Example `alert()` alert

This example uses the `vstop()` function from our ta library to calculate a volatility stop value and trend information based on the Average True Range (ATR). The `stopValue` trails behind the chart's close to form a trend-following system.

The script triggers an alert with an alert() call each time the trend direction changes. The alert's message is a "series string" that shows the trend's new direction and the current stop value. An additional alert occurs whenever the `stopValue` moves in the current trend direction, with a message containing the updated value:

//@version=6 indicator("Vstop alert demo", overlay = true) import TradingView/ta/7 as TVta // Calculate ATR trailing stop and determine trend direction. [stopValue, trendUp] = TVta.vStop(close, 20, 2) // Round the stop value to mintick for accuracy in comparison operators. float stop = math.round_to_mintick(stopValue) // Check for trend changes. bool trendReversal = trendUp != trendUp[1] bool trendToDn = trendReversal and not trendUp bool trendToUp = trendReversal and trendUp // Create color variables for the plot display. color plotColor = trendUp ? color.green : color.red color lineColor = trendReversal ? color(na) : plotColor // Plot the stop value on the chart. Plot a circle on trend changes. plot(stop, "V-Stop", lineColor) plot(trendReversal ? stop : na, "Trend Change Circle", plotColor, 3, plot.style_circles) // Convert the stop value to string for use in the alert messages. string stopStr = str.tostring(stop) // If the trend changed to up, send a long alert with the initial stop value. if trendToUp alert("Long alert. Stop @ " + stopStr, alert.freq_once_per_bar_close) // If the trend changed to down, send a short alert with the initial stop value. if trendToDn alert("Short alert. Stop @ " + stopStr, alert.freq_once_per_bar_close) // If the stop value has progressed, send an alert to update the stop value. if (trendUp and stop > stop[1] or not trendUp and stop < stop[1]) and not trendReversal alert('Update stop to ' + stopStr, alert.freq_once_per_bar_close)

See this section of the Alerts page for more information.

Example strategy alert

This example strategy places a market order with strategy.entry() and a stop-loss and take-profit (bracket) order with strategy.exit() when a 5-bar moving average crosses over a 10-bar moving average. The stop-loss price is 1% below the current close and the take-profit price is 2% above the close. Order fill alerts occur when the broker emulator fills an entry or exit order. Both order placement commands include unique `alert_message` arguments that combine placeholders and "string" representations of the `limit` and `stop` values to output details like the trade action, position size, chart symbol, and order prices:

```
//@version=6 // This annotation auto-populates the alert dialogue with the `alert_message` string. // @strategy_alert_message {{strategy.order.alert_message}} strategy("Alert message demo", overlay = true) // Declare two moving averages to use for the entry condition. float fastMa = ta.sma(close, 5) float slowMa = ta.sma(close, 10) // Declare two persistent variables that will hold our stop-loss and take-profit values. var float limit = na var float stop = na // If `fastMa` has crossed over `slowMa` and we are not already in a position, // place an entry and exit order. // • Set the `limit` to 2% above the close and the stop to 1% below. // • Use a combination of script variables and placeholders in the alert strings. // • The exit alert shows the order direction, position size, ticker, and order price. // • The entry alert includes the same values plus the stop and limit price. if ta.crossover(fastMa, slowMa) and strategy.position_size == 0 limit := close * 1.02 stop := close * 0.99 string exitString = "{{strategy.order.action}} {{strategy.position_size}} {{ticker}} @ {{strategy.order.price}}" string entryString = exitString + " TP: " + str.tostring(limit, format.mintick) + " SL: " + str.tostring(stop, format.mintick) strategy.entry("Buy", strategy.long, alert_message = entryString) strategy.exit("Exit", "Buy", stop = stop, limit = limit, alert_message = exitString) // Plot the moving averages, stop, and limit values on the chart. plot(fastMa, "Fast Moving Average", color.aqua) plot(slowMa, "Slow Moving Average", color.orange) plot(strategy.position_size > 0 ? limit : na, "Limit", color.green, style = plot.style_linebr) plot(strategy.position_size > 0 ? stop : na, "Stop", color.red, style = plot.style_linebr)
```

For more information about order fill events, see this section of the Alerts page. To learn more about how strategy scripts work, see the Strategies page.

If I change my script, does my alert change?

No, not without creating a new alert.

When a user creates an alert using the “Create Alert” dialog box, that action saves a “snapshot” of the script, its inputs, and the current chart’s context on TradingView’s servers. This snapshot acts as an independent *copy* of the script instance and chart. Therefore, any changes to the script, its inputs, or the user’s chart **do not** affect that created alert. To update an alert after making changes, *delete* the existing alert and *create* a new one.

Why aren’t my alerts working?

Here are some common reasons why alerts might not work as expected, and how to solve them:

Make sure the alert is active and has not expired

Scripts that include alert triggers **do not** directly create alerts. Users must create alerts in the “Create Alert” dialog box, where they specify the “Condition” that triggers the alert and the “Expiration” time. Created alerts do not fire after they expire. See this Help Center article on Setting up alerts.

Check the alert logs

An alert can fire without a notification, depending on the alert’s settings. Check the logs in the alert manager to see whether an alert occurred. To set up notifications for an alert, use the options in the “Notifications” tab of the “Create/Edit Alert” dialog box.

Check for repainting

If an alert fires at a different time than expected, *repainting* might be the cause. Refer to the Repainting page for more information.

Limit the frequency of alerts

If more than 15 alerts occur within three minutes, the system automatically *halts* further alerts. This frequency limit helps prevent excessive notifications and potential server overload.

Debug script errors

If a script instance raises a *runtime error* at some point during its executions, alerts from that instance **cannot** fire because the error stops the script from continuing to execute its code. Some common issues that can halt alerts include:

- Attempting to store more than 100,000 elements within a collection
- Trying to access an item from a collection at an *out-of-bounds* index
- Referencing historical values of a time series outside its allocated memory buffer
- Using loops that take longer than 500 ms to complete their iterations

See this page for additional details about common error messages and troubleshooting tips.

Why is my alert firing at the wrong time?

Sometimes, alerts may fire when users do not expect according to what their script displays on the chart. Repainting is the typical cause of such issues.

A chart's realtime and historical bars often rely on *different* data feeds. Data providers may retroactively adjust the reported values on realtime bars, which the displayed data reflects *after* users refresh their charts or restart their scripts. Such adjustments can cause discrepancies where a triggered alert's timing may not align with the script's output after reloading it.

Scripts may also behave differently on historical and realtime bars, which can lead to repainting. On historical bars, scripts execute once per bar close, whereas on realtime bars, where alerts fire, scripts execute once for *each new tick* from the data feed. Therefore, if a script behaves differently on those bars, users may see differences between its signals and triggered alerts after reloading the chart.

Below are some common repainting issues that can affect a script's alerts:

Alerts firing before bar close

Most scripts have fluid data values that update after new ticks during an unconfirmed realtime bar and finalize after the bar closes. Consequently, an alert that fires on an open bar may not reflect the *final state* of the condition after the bar's confirmation. Set the alert's frequency to "Once Per Bar Close" to avoid this issue.

Using `calc_on_every_tick` in strategies

When a strategy script includes `calc_on_every_tick = true` in its declaration statement or the user selects the "On every tick" option in the "Recalculate" section of the strategy's properties, it recalculates on *every* price update in the realtime data. This behavior can cause strategies to repaint because historical bars do not contain the same information as realtime bars. See this section of the Strategies page to learn more.

Incorrect usage of `request.security()` calls

Using request.security() calls to fetch data from alternative timeframes can cause discrepancies on historical bars that scripts **cannot** reproduce on realtime bars. Ensure you follow the best practices for *non-repainting* data requests to avoid such discrepancies, especially with higher-timeframe data. See the Avoiding repainting section of the Other timeframes and data page and the Higher-timeframe requests publication from PineCoders for more information.

Can I use variable messages with alertcondition()?

The `message` parameter of the alertcondition() function requires a "const string" argument, which **cannot change** after compilation. However, the "string" can include placeholders, which an alert substitutes with corresponding dynamic values from a script each time it fires.

The script below demonstrates two alertcondition() calls whose `message` arguments include placeholders for dynamic values. Each time alerts from these triggers occur, the message displays information about the current chart's exchange, symbol, price, and volume:

//@version=6 indicator("Placeholder demo", overlay = false) [macdLine, signalLine, histLine] = ta.macd(close, 12, 26, 9) plot(macdLine, "MACD", color.blue) plot(signalLine, "Signal", color.orange) plot(histLine, "Hist.", color.red, style = plot.style_histogram) bool crossUp = ta.crossover(macdLine, signalLine) bool crossDown = ta.crossunder(macdLine, signalLine) alertcondition(crossUp, "MACD Cross Up", "MACD cross up on {{exchange}}:{{ticker}}\nprice = {{close}}\nvolume = {{volume}}") alertcondition(crossDown, "MACD Cross Down", "MACD cross down on {{exchange}}:{{ticker}}\nprice = {{close}}\nvolume = {{volume}}")

How can I include values that change in my alerts?

The method for including dynamic values in alert messages varies with the type of alert trigger:

- The alertcondition() function accepts a "const string" `message` argument that can contain placeholders for dynamic values. See Can I use variable messages with `alertcondition()`? for more information.
- The alert() function accepts "series string" `message` arguments, which allows the convenient creation of dynamic messages that use a script's calculated values. See this section for an example.
- Order fill alerts can use "series string" values and placeholders. Refer to the example here.

How can I get custom alerts on many symbols?

To manage alerts across multiple symbols using a custom script, one option is to set an individual alert on each symbol. There is no automated method to set the same alert across many symbols simultaneously in a single action. It's also important to note that the TradingView screener uses built-in filters and does not support custom Pine Script® code.

Scripts can retrieve data from other *contexts* (symbols, timeframes, and modifiers such as non-standard chart calculations and extended sessions) using the functions in the `request` namespace. With these functions, programmers can design scripts that retrieve data from up to 40 or 64 unique contexts, depending on their plan. Search for "screener" in the Community scripts for in-depth examples.

Here is an example incorporating three symbols. The `checkForAlert()` function calls request.security() to fetch data from a specified context and evaluate the user-defined `checkForRsiConditions()` function using

that data. Then, the function calls alert() using the result to create an alert trigger. The script calls this function three times, creating distinct alert triggers for each specified symbol:

//@version=6 indicator("Screener demo", overlay = true) // Declare inputs for the alert symbols and the timeframe to run the alerts on. The default is the current chart timeframe. string tfInput = input.timeframe("", "Timeframe") string symbol1Input = input.symbol("BINANCE:ETHUSDT", "Symbol 1") string symbol2Input = input.symbol("BINANCE:BATUSDT", "Symbol 2") string symbol3Input = input.symbol("BINANCE:SOLUSDT", "Symbol 3") // @function Generates alert messages for RSI crossing over or under 50, and crosses of price and the 50 EMA. // @returns (string) Formatted alert messages with values for each crossover and crossunder event. checkForRsiConditions() => float rsi = ta.rsi(close, 14) float ema = ta.ema(close, 50) string alertMessage = "" if ta.crossover(rsi, 50) alertMessage += str.format("RSI ({0}) crossed over 50 for {1} on {2} timeframe.\n", rsi, syminfo.ticker, timeframe.period) if ta.crossunder(rsi, 50) alertMessage += str.format("RSI ({0}) crossed under 50 for {1} on {2} timeframe.\n", rsi, syminfo.ticker, timeframe.period) if ta.crossover(close, ema) alertMessage += str.format("Crossover of 50 EMA for {0} on {1} timeframe. Price is {2}", syminfo.ticker, timeframe.period, close) if ta.crossunder(close, ema) alertMessage += str.format("Crossunder of 50 EMA for {0} on {1} timeframe. Price is {2}", syminfo.ticker, timeframe.period, close) // @function Calls the `checkForRsiConditions()` function for the provided symbol and timeframe. // Triggers an alert if the function returns a message. // @param symbol (simple string) The symbol to check. // @param tf (simple string) The timeframe to check. // @param freq (const string) The frequency of the alert. Optional. Default is `alert.freq_once_per_bar`. // @returns (void) The function has no explicit return, but triggers an alert with the message if the // conditions defined within the `checkForRsiConditions()` function are met. checkForAlert(simple string symbol, simple string tf, const string freq = alert.freq_once_per_bar) => string msg = request.security(symbol, tf, checkForRsiConditions()) if msg != msg[1] and str.length(msg) > 0 alert(msg, freq) // Check for alerts on the input symbols and timeframe. checkForAlert(symbol1Input, tfInput) checkForAlert(symbol2Input, tfInput) checkForAlert(symbol3Input, tfInput) // Add calls for additional symbols up to your plan's limit...

Note that:

- A script can execute up to 40 *unique* `request.*()` calls, or up to 64 if the user has the Ultimate plan. A `request.*()` call is typically *not* unique if a script already calls the same function with the same arguments. See the `request.*()` calls section of the Limitations page for more information.
- This script uses the alert() function because alertcondition() is not allowed within *local scopes*.

How can I trigger an alert for only the first instance of a condition?

Firing an alert only on its first occurrence can help avoid redundant notifications and isolate specific conditions or state changes, which is beneficial in several use cases. For instance, if a user relies on alerts to automate order placement, restricting redundant alerts to their first occurrence can help avoid accidentally placing excessive orders.

For alerts with alertcondition() triggers, setting them to fire once using the “Only Once” option in the “Create Alert” dialog box is not an optimal solution because it requires *manual* reactivation each time an alert occurs. Alerts from the alert() function do not have an “Only Once” frequency option. The programmer must use conditional logic to ensure the call executes at the appropriate time.

There are two primary ways to code repeating alerts that fire on only the first instance of a condition:

Using stricter criteria

Rather than relying on a continuous condition like `close > ma`, which may remain `true` for multiple consecutive bars, try using a more strict condition like `ta.crossover(close, ma)`. For simple cases, this is the easiest method.

Using state control

More complex scenarios might require controlling and tracking *states*, which entails setting flags or specific values to signify certain conditions.

The example script below manages separate bullish and bearish states, and it colors the background to represent each state. When a bullish or bearish state first occurs, an alert() call executes and the script plots a triangle on the chart. It also plots smaller triangles to show where other signals occur within a state, which do not trigger additional alerts:

//

//@version=6 indicator("Single alert demo", overlay = true) // ————— Calculations: Determine highest/lowest values over last `lengthInput` bars. int lengthInput = input.int(20, "Length") float highest = ta.highest(lengthInput) float lowest = ta.lowest(lengthInput) // ————— Trigger conditions: Define bull and bear signals. Bull signal is triggered by a new high, and bear by a new low. bool bullSignal = high == highest bool bearSignal = low == lowest // ————— State change flags: Set true on state transition bars only. bool changeToBull = false bool changeToBear = false // ————— State tracking: `isBull` is set to true for bull state, false for bear. It's set only at the initial switch to the opposite condition. // This variable's state is retained from bar to bar because we use the `var` keyword to declare it. var bool isBull = false // ————— State transitions: Allow a switch from bull to bear or bear to bull; ignore repeated signals in current state. // Set the state change flags to true only on the first bar where a new signal appears. if bullSignal and not isBull isBull := true changeToBull := true else if bearSignal and isBull isBull := false changeToBear := true // Plot highest and lowest values. plot(highest, "Highest", color.new(color.green, 80), 2) plot(lowest, "Lowest", color.new(color.red, 80), 2) // Background color: Green for bull, none for bear. bgcolor(isBull ? color.new(color.green, 90) : na) // State change markers: Display "ALERT" text on bars where a state change occurs and an alert would trigger. plotchar(changeToBull, "Change to Bull state", "▲", location.belowbar, color.new(color.lime, 30), size = size.small, text = "BULL\nALERT") plotchar(changeToBear, "Change to Bear state", "▼", location.abovebar, color.new(color.red, 30), size = size.small, text = "BEAR\nALERT") // Signal markers: Display for repeated signals within the current state. // These signals would trigger redundant alerts if not for the state tracking flag preventing them. plotchar(bullSignal and not changeToBull, "Bull signal", "▲", location.belowbar, color.green, size = size.tiny) plotchar(bearSignal and not changeToBear, "Bear signal", "▼", location.abovebar, color.maroon, size = size.tiny) // Alerts: Trigger on state changes only. if changeToBull alert("Change to bull state") if changeToBear alert("Change to bear state")

How can I run my alert on a timer or delay?

It is possible to program logic to delay alert triggers so that they occur *after* the initial condition. However, because Pine scripts execute on realtime bars only after new *price updates*, and an alert only fires when a script *executes*, it is difficult to predict the exact time of a delayed alert.

There are no price updates in a closed market, meaning an alert with a delay will not fire until the market opens again. Similarly, thinly traded securities may have very infrequent price updates in unpredictable intervals, which can cause a larger delay than intended.

The Pine script below implements a *time-delayed* alert, which is subject to the limitations above. When the current close is higher than a moving average, a delay counter starts. After the delay passes, the alert fires once, and another alert *cannot* fire until the timer resets. Users can specify whether the timer resets on each bar using the script's `resetInput`:

//@version=6 indicator("Delayed alert demo", overlay = true) import PineCoders/Time/4 as PCtime string TIME_TT = "The delay's duration and units. This specifies the continuous duration for which the condition must be true before triggering the alert." string RESET_TT = "When checked, the duration will reset every time a new realtime bar begins." enum TimeUnit seconds minutes hours int durationInput = input.int(20, "Condition must last", minval = 1, inline = "00") TimeUnit timeUnitInput = input.enum(TimeUnit.seconds, "", inline="00") bool resetInput = input.bool(false, "Reset timing on new bar", tooltip = RESET_TT) int maLengthInput = input.int(9, "MA length") // Calculate and plot a SMA with `maLengthInput` length. float ma = ta.sma(close, maLengthInput), plot(ma, "MA") // Check whether the close is greater than the SMA. bool cond = close > ma // Time the duration for which the condition has been true. int secSince = PCtime.secondsSince(cond, resetInput and barstate.isnew) // Check if the duration is greater than the input timer. bool timeAlert = secSince > (PCtime.timeFrom("bar", durationInput, str.tostring(timeUnitInput)) - time) / 1000 // Format a time string for the timer label. string alertTime = str.format_time(secSince * 1000, "mm:ss") // Set the contents for the label depending on the stage of the alert timer. string alertString = switch timeAlert => "Timed Alert Triggered\n\n" + alertTime cond => "Condition Detected...\n\nTimer count\n" + alertTime => "Waiting for condition..." // Display alert timer using a label. Declare a basic label once and update location, color, and text on the last bar for efficiency. if barstate.islast var label condTime = label.new(na, na, yloc = yloc.abovebar, style = label.style_label_lower_left, textcolor = chart.fg_color) label.set_x(condTime, bar_index) label.set_text(condTime, alertString) label.set_color(condTime, color.new(timeAlert ? color.green : cond ? color.orange : color.red, 50)) // Create a flag to ensure alert is triggered only once each time the delay timer is exceeded. varip bool isFirstOccurrence = true // Fire alert if timer is triggered. if timeAlert and isFirstOccurrence alert(str.format("{0} {1} Delayed Alert Triggered", durationInput, str.tostring(timeUnitInput)), alert.freq_all) // Toggle the flag to `false` when alert triggers, and reset when the condition clears. isFirstOccurrence := not timeAlert

Note that:

- The `secondsSince()` function from the PineCoders' time library determines the duration, in seconds, for which a certain condition remains continuously `true`. The duration can be tracked within bars because it uses the varip keyword.
- The timing starts when the condition first becomes `true`. If the condition becomes `false` or an optional resetting condition occurs, the timer restarts. If "Reset timing on new bar" is enabled in the "Settings/Inputs" tab, the function restarts its timing at the start of a new bar.
- A colored label shows what state the script is in:
    1. **Red** - The condition has not occurred yet.
    2. **Orange** - The condition occurred and the delay timer is active.
    3. **Green** - The timer has surpassed the set duration, simulating a delayed alert.

This script relies on variables declared with the varip keyword, which do not revert to their last committed states during realtime bar calculations. See this section of the User Manual to learn more about using this keyword. To learn about how *rollback* works, see the Execution model page.

How can I create JSON messages in my alerts?

Alerts can send messages containing JavaScript Object Notation (JSON) to webhooks. Pine Script does not include any built-in functions to produce JSON, but programmers can create JSON messages in Pine by constructing “string” representations.

When constructing JSON representations, ensure the keys and values intended as strings in the JSON-formatted text use *double quotes*, not single quotes.

The following example shows three ways to construct JSON strings in Pine Script:

1. **Static JSON Strings**

   Define separate alerts with predefined JSON-formatted strings. This method is the simplest.

2. **Placeholders**

   Use placeholders in the alert message, such as `{{close}}` and `{{volume}}`, to add *dynamic* values to the JSON. The alert instance replaces the placeholders with corresponding values when it fires. This method can create richer alerts, especially for strategies, which have extra placeholders for their calculated values. See this section above for an example.

3. **Dynamic strings**

   Use the functions in the `str.*()` namespace and “string” concatenation to create dynamic JSON-formatted text. This method is the most customizable and advanced. Our script below shows a simple, straightforward example of this approach. When using dynamic string formatting to construct JSON strings, ensure the resulting JSON is *valid* for all the combined values.

//@version=6 indicator("JSON example", overlay = true) // Define EMA cross conditions to trigger alerts, and plot the ema on the chart. float ema = ta.ema(close, 21) bool crossUp = ta.crossover(close, ema) bool crossDown = ta.crossunder(close, ema) plot(ta.ema(close, 21)) // ———— Method 1 - Separate alerts with static messages. string alertMessage1a = '{"method": 1, "action": "buy", "direction": "long", "text": "Price crossed above EMA"}' string alertMessage1b = '{"method": 1, "action": "sell", "direction": "short", "text": "Price crossed below EMA"}' alertcondition(crossUp, "Method 1 - Cross up", alertMessage1a) alertcondition(crossDown, "Method 1 - Cross down", alertMessage1b) // Rendered alert: // { // "method": 1, // "action": "buy", // "direction": "long", // "text": "Price crossed above EMA" // } // ———— Method 2 - Using placeholders for dynamic values. string alertMessage2 = '{"method": 2, "price": {{close}}, "volume": {{volume}}, "ema": {{plot_0}}}' alertcondition(crossUp, "Method 2 - Cross Up", alertMessage2) // Rendered alert: // { // "method": 2, // "price": 2066.29, // "volume": 100.859, // "ema": 2066.286 // } // ———— Method 3 - String concatenation using dynamic values. string alertMessage3 = '{"method": 3, "price": ' + str.tostring(close) + ', "volume": ' + str.tostring(volume) + ', "ema": ' + str.tostring(ema) + '}' if crossUp alert(alertMessage3, alert.freq_once_per_bar_close) // Rendered alert: // { // "method": 3, // "price": 2052.27, // "volume": 107.683, // "ema": 2052.168 // }

Before using the JSON-formatted string in alerts for real-world applications, such as sending messages to place orders, *test* and *validate* the JSON message to ensure it works as intended:

- Send alerts to an email address to see how the JSON message appears.
- Copy the alert message from the email into an online JSON validation tool.
- Use an API client application to check the server response to the request.

Refer to this Wikipedia page to learn more about JSON format. To learn more about how alerts send information using webhooks, see the Help Center article on webhooks.

How can I send alerts to Discord?

Sending alerts from a Pine script to a Discord chat room is possible using webhooks.

The message for Discord communication requires JSON format. The *minimum* requirement for a valid message is `{"content": "Your message here"}`.

The script example below uses placeholders to dynamically populate alert messages with script values, including the new high or low price, and the chart's symbol and timeframe:

//@version=6 indicator("Discord demo", overlay = true) // Calculate a Donchian channel using the TV ta library. import TradingView/ta/7 as TVta int lengthInput = input.int(10, "Channel length") [highest, lowest, middle] = TVta.donchian(lengthInput) // Create conditions checking for a new channel high or low. bool isNewHi = high > highest[1] bool isNewLo = low < lowest[1] // Plot the Donchian channel and fill between the midpoint and the upper and lower halves. hi = plot(highest, "Channel high", color.new(color.fuchsia, 70)) mid = plot(middle, "Channel mid.", color.new(color.gray, 70)) lo = plot(lowest, "Channel low", color.new(color.lime, 70)) fill(mid, hi, color.new(color.fuchsia, 95)) fill(mid, lo, color.new(color.lime, 95)) // Plot shapes to mark new highs and lows to visually identify where alert trigger conditions occur. plotshape(isNewHi, "isNewHi", shape.arrowup, location.abovebar, color.new(color.lime, 70)) plotshape(isNewLo, "isNewLo", shape.arrowdown, location.belowbar, color.new(color.fuchsia, 70)) // Create two alert conditions, one for new highs, and one for new lows. // Format the message for Discord in the following JSON format: {"content": "Your message here"} alertcondition(isNewHi, "New High (Discord Alert Demo)", '{"content": "New high ({{high}}) on {{ticker}} on {{interval}} chart!"}') alertcondition(isNewLo, "New Low (Discord Alert Demo)", '{"content": "New low ({{low}}) on {{ticker}} on {{interval}} chart!"}') // The following test alert condition fires immediately. Set this alert frequency to "Only Once". alertcondition(true, "Test (Discord Alert Demo)", '{"content": "This is a test alert from TradingView to Discord."}')

To send these alert messages to Discord, follow these steps:

1. Create a Discord webhook

- Create a new webhook in a server using an account with webhook creation and management permissions. Refer to Discord's Intro to Webhooks article for instructions.
- Copy the Webhook URL. This URL represents the address where the alert sends a POST request.

2. Set up an alert on TradingView

- Add the above "Discord demo" script to a chart and open the "Create Alert" dialog box.
- Choose one of the script's alert conditions as the "Condition" in the dialog. If you select the "New High" or "New Low" alerts, choose the "Once Per Bar Close" option in the "Frequency" field to avoid triggering alerts for new highs or lows on an unconfirmed bar. When using the "Test" alert, choose "Only Once" as the "Frequency" option.
- In the "Notifications" tab of the "Create Alert" dialog, select "Webhook URL" and paste the URL of the Discord webhook.

3. Test the integration

- Check that alerts appear in the alert log on TradingView.

- Use the “Test” alert to check whether the webhook works as expected. After the alert fires, check your Discord to see if it received the message.
- If the alert message does not appear in Discord, check whether the pasted webhook URL is correct.
- If the message does not display correctly in Discord, check the JSON format. The minimum required format is `{"content": "Your message here"}`.

Consult Discord’s Webhook Resource to learn about advanced JSON message configurations.

For more information about dynamic values in alert messages, refer to How can I include values that change in my alerts?.

To learn about using JSON format in script alerts, see How can I create JSON messages in my alerts?.

How can I send alerts to Telegram?

Sending TradingView alerts directly to Telegram is challenging due to protocol differences and formatting requirements. One solution is to use an intermediary service, which receives webhook alerts from TradingView, formats them as required by Telegram, and then forwards them to a Telegram bot.

1. Choose a platform like Zapier, Integromat, or Pipedream. Alternatively, programmers can consider developing a custom server script using Node.js or Python.
2. In TradingView, set up alerts to send webhook requests to the intermediary service’s provided URL.
3. Configure the intermediary service to reformat TradingView’s incoming requests for Telegram’s API and send the formatted message to a Telegram bot using the `sendMessage` method.

See the Telegram Bot API documentation for detailed technical information.

Data structures

What data structures can I use in Pine Script®?

Pine data structures resemble those in other programming languages, with some important differences:

- **Tuple**: An arbitrary—and temporary—grouping of values of one or more types.
- **Array**: An ordered sequence of values of a single type.
- **Matrix**: A two-dimensional ordered sequence of values of a single type.
- **Object**: An arbitrary—and persistent—collection of values of one or more types.
- **Map**: An *unordered* sequence of key-value pairs, where the keys are of a single type and the values are of a single type.

The following sections describe each data structure in more detail.

Tuples

A tuple in Pine Script is a list of values that is returned by a function, method, or local block. Unlike in other languages, tuples in Pine serve no other function. Tuples do not have names and cannot be assigned to variables. Apart from the fact that the values are requested and returned together, the values have no relation to each other, in contrast to the other data structures described here.

To define a tuple, enclose a comma-separated list of values in square brackets.

Using a tuple to request several values from the same symbol and timeframe using a request.security() call is more efficient than making several calls. For instance, consider a script that contains separate request.security() calls for the open, high, low, and close prices:

float o = request.security(syminfo.tickerid, "D", open) float h = request.security(syminfo.tickerid, "D", high) float l = request.security(syminfo.tickerid, "D", low) float c = request.security(syminfo.tickerid, "D", close)

Using a tuple can consolidate these calls into a single request.security() function call, reducing performance overhead:

[o, h, l, c] = request.security(syminfo.tickerid, "D", [open, high, low, close])

See the Tuples section in the User Manual for more information.

Arrays

Arrays store multiple values of the same type in a single variable. Each *element* in an array can be efficiently accessed by its *index*—an integer corresponding to its position within the array.

Arrays can contain an arbitrary number of elements. Scripts can loop through arrays, testing each element in turn for certain logical conditions. There are also many built-in functions to perform different operations on arrays. This flexibility makes arrays very versatile data structures.

Arrays can be created with either the array.new<type>() or array.from() function. In this simple example, we store the last five closing prices in an array and display it in a table:

//@version=6 indicator("Array example") // Declare an array with 5 `na` values on the first bar. var array<float> pricesArray = array.new<float>(5) // On each bar, add a new value to the end of the array and remove the first (oldest) element. array.push(pricesArray, close) array.shift(pricesArray) // Display the array and its contents in a table. var table displayTable = table.new(position.middle_right, 1, 1) if barstate.islast table.cell(displayTable, 0, 0, str.tostring(pricesArray), text_color = chart.fg_color)

See the Arrays section in the User Manual for more information.

Matrices

A matrix is a two-dimensional array, made of rows and columns, like a spreadsheet. Matrices, like arrays, store values of the same built-in or user-defined type.

Matrices have many built-in functions available to organize and manipulate their data. Matrices are useful for modeling complex systems, solving mathematical problems, and improving algorithm performance.

This script demonstrates a simple example of matrix addition. It creates a 3x3 matrix, calculates its transpose, then calculates the matrix.sum() of the two matrices. This example displays strings representing the original matrix, its transpose, and the resulting sum matrix in a table on the chart:

//@version=6 indicator("Matrix sum example") //@variable An empty matrix of type "float". m = matrix.new<float>() // Add rows to the matrix containing data. m.add_row(0, array.from(1, 2, 3)) m.add_row(1, array.from(0, 4, 2)) m.add_row(2, array.from(3, 1, 2)) var table displayTable = table.new(position.middle_right, 5, 2) if barstate.islast matrix<float> t = m.transpose() table.cell(displayTable, 0, 0, "A", text_color = chart.fg_color) table.cell(displayTable, 0, 1, str.tostring(m), text_color = chart.fg_color) table.cell(displayTable, 1, 1, "+", text_color = chart.fg_color) table.cell(displayTable, 2, 0, "Aᵀ", text_color = chart.fg_color) table.cell(displayTable, 2, 1, str.tostring(t), text_color = chart.fg_color) table.cell(displayTable, 3, 1, "=", text_color = chart.fg_color) table.cell(displayTable, 4, 0, "A + Aᵀ", text_color = color.green) table.cell(displayTable, 4, 1, str.tostring(matrix.sum(m, t)), text_color = color.green)

See the Matrices section in the User Manual for more information.

Objects

Pine Script objects are containers that group together multiple fields into one logical unit.

Objects are *instances* of user-defined types (UDTs). UDTs are similar to *structs* in traditional programming languages. They define the rules for what an object can contain. Scripts first create a UDT by using the type keyword and then create one or more objects of that type by using the UDT's built-in `new()` method.

UDTs are *composite* types; they contain an arbitrary number of fields that can be of any type. A UDT's field type can even be another UDT, which means that objects can contain other objects.

Our example script creates a new `pivot` object each time a new pivot is found, and draws a label using each of the object's fields:

//@version=6 indicator("Object example", overlay = true) // Create the pivot type with 3 fields: the x coordinate, the y coordinate, and a formatted time string. type pivot int x float y string pivotTime // Check for new pivots. `ta.pivotHigh` returns the price of the pivot. float pivFound = ta.pivothigh(10, 10) // When a pivot

is found, create a new pivot object and generate a label using the values from its fields. if not na(pivFound) pivot pivotObject = pivot.new(bar_index - 10, pivFound, str.format_time(time[10], "yyyy-MM-dd HH:mm")) label.new(pivotObject.x, pivotObject.y, pivotObject.pivotTime, textcolor = chart.fg_color)

See the User Manual page on Objects to learn more about working with UDTs.

Maps

Maps in Pine Script are similar to *dictionaries* in other programming languages, such as dictionaries in Python, objects in JavaScript, or HashMaps in Java. Maps store elements as key-value pairs, where each key is unique. Scripts can access a particular value by looking up its associated key.

Maps are useful because they can access data directly without searching through each element, unlike arrays. For example, maps can be more performant and simpler than arrays for associating specific attributes with symbols, or dates with events.

The following example illustrates the practical application of maps for managing earnings dates and values as key-value pairs, with dates serving as the keys:

//@version=6 indicator("Earnings map", overlay = true) // Get the earnings value if present. We use `barmerge.gaps_on` to return `na` unless earnings occurred. float earnings = request.earnings(syminfo.tickerid, earnings.actual, barmerge.gaps_on) // Declare a map object for storing earnings dates and values. var map<string, float> earningsMap = map.new<string, float>() // If `request.security()` returned data, add an entry to the map with the date as the key and earnings as the value. if not na(earnings) map.put(earningsMap, str.format_time(time, "yyyy-MM-dd"), earnings) // On the last historical bar, loop through the map in the insertion order, writing the key-value pairs to the logs. if barstate.islastconfirmedhistory string logText = "\n" for [key, value] in earningsMap logText += str.format("{0}: {1}\n", key, value) log.info(logText)

Here, we use request.earnings() with the barmerge parameter set to barmerge.gaps_on to return the earnings value on bars where earnings data is available, and return na otherwise. We add non-na values to the map, associating the dates that earnings occurred with the earnings numbers. Finally, on the last historical bar, the script loops through the map, logging each key-value pair to display the map's contents.

To learn more about working with maps, refer to the Maps section in the User Manual.

What's the difference between a series and an array?

In Pine Script, "series" variables are calculated on each bar. Historical values cannot change. Series values can change during the open realtime bar, but when the bar closes, the value for that bar becomes fixed and immutable. These fixed values are automatically indexed for each bar. Scripts can access values from previous bars by using the [] history-referencing operator to go back one or more bars.

Where "series" variables are strictly time-indexed, and the historical values are created automatically, arrays are created, filled, and manipulated arbitrarily by a script's logic. Programmers can change the size of arrays dynamically by using functions that insert or remove elements. Any element in an array can also be altered using the array.set() function.

The concept of time series is a fundamental aspect of Pine Script. Its series-based execution model processes scripts bar-by-bar. This built-in behavior mimics looping, allowing a series to track values,

accumulate totals, or perform calculations across a sequence of data on each bar.

Simple calculations can thus be done efficiently using “series” variables. Using arrays for similar tasks requires manually creating a dataset, managing its size, and using loops to process the array’s contents, which can be far less efficient.

Arrays, of course, can do many things that series variables cannot. Scripts can use arrays to store a fixed set of values, collect complex data such as objects of user-defined types, manage drawing instances for visual display, and more. In general, use arrays to handle data that doesn’t fit the time series model, or for complex calculations. Arrays can also mimic series by creating custom datasets, as in the getSeries library.

> NoteAn array itself is part of a series. Scripts can reference the previous committed states of any array by using the history-referencing operator. See the History referencing section of the Arrays page for more information.

How do I create and use arrays in Pine Script?

Pine Script arrays are one-dimensional collections that can hold multiple values of a single type.

Declaring arrays

Declare an array by using one of the following functions: array.new<type>(), array.from(), or array.copy(). Arrays can be declared with the var keyword to have their values persist from bar to bar, or without it, so that the values initialize again on each bar. For more on the differences between declaring arrays with or without var, see this section of this FAQ.

Adding and removing elements

Pine Script provides several functions for dynamically adjusting the size and contents of arrays.

Adding elements

- array.unshift() inserts a new element at the beginning of an array (index 0) and increases the index values of any existing elements by one.
- array.insert() inserts a new element at the specified index and increases the index of existing elements at or after the insertion index by one. It accepts both positive and negative indices, which reference an element’s position starting from the beginning of the array or from the end, respectively.
- array.push() adds a new element at the end of an array.

Removing elements

- array.remove() removes the element at the specified index and returns that element’s value. It accepts both positive and negative indices, which reference an element’s position starting from the beginning of the array or from the end, respectively.
- array.shift() removes the first element from an array and returns its value.
- array.pop() removes the last element of an array and returns its value.
- array.clear() removes all elements from an array. Note that clearing an array won’t delete any objects that were referenced by its elements. To delete objects contained by an array, loop through the array and delete the objects first, and then clear the array.

The flexibility afforded by these functions supports various data management strategies, such as queues or stacks, which are useful for custom datasets or sliding window calculations. Read more about implementing a stack or queue in this FAQ entry.

Calculations on arrays

Because arrays are not time series data structures, performing operations across an array's elements requires special functions designed for arrays. Programmers can write custom functions to perform calculations on arrays. Additionally, built-in functions enable computations like finding the maximum, minimum, or average values within an array. See the Calculation on arrays section of the User Manual for more information.

Script example

This script example demonstrates a practical application of arrays by tracking the opening prices of the last five sessions. The script declares a float array to hold the prices using the var keyword, allowing it to retain its values from bar to bar.

At the start of each session, we update the array by adding the new opening price and removing the oldest one. This process, resembling a queue, keeps the array's size constant while maintaining a moving window of the session opens for the last five days. Built-in array functions return the highest, lowest, and average opening price over the last five sessions. We plot these values to the chart.

//@version=6 indicator("Array demo", overlay = true) // Create an input to determine the number of session opens to track, with a default value of 5. int numOpensInput = input.int(5, "Number of opens to track") // Create an array to store open prices. Using `var` ensures the array retains its values from bar to bar. // Initially, the array is filled with placeholder values (`na`), which are later updated with actual open prices. var array<float> opensArray = array.new<float>(numOpensInput) // On the first bar of each session, update the array: add the current open price and remove the oldest entry. if session.isfirstbar_regular array.push(opensArray, open) array.shift(opensArray) // Plot the highest, lowest, and average open prices from the tracked sessions plot(array.max(opensArray), "Highest open in n sessions", color.lime) plot(array.min(opensArray), "Lowest open in n sessions", color.fuchsia) plot(array.avg(opensArray), "Avg. open of the last n sessions", color.gray) // Change the background color on the first bar of each session to visually indicate session starts. bgcolor(session.isfirstbar_regular ? color.new(color.gray, 80) : na)

For more information about arrays, see the Arrays page in the User Manual.

What's the difference between an array declared with or without `var`?

Using the var keyword, a script can declare an array variable in a script that is initialized only once, during the first iteration on the first chart bar.

Persistent arrays

When an array is declared with var, it is initialized only once, at the first execution of the script. This allows the array to retain its contents and potentially grow in size across bars, making it ideal for cumulative data collection or tracking values over time.

Non-persistent arrays

Arrays declared without var are reinitialized on every new bar, effectively resetting their content. This behavior suits scenarios where calculations are specific to the current bar, and historical data retention is unnecessary.

Example script

Here, we initialize two arrays. Array `a` is declared without using the var keyword, while array `b` is declared with var, allowing us to observe and compare their behavior. Throughout the runtime, we incrementally add an element to each array on each bar. We use a table to present and compare both the sizes of these arrays and the number of chart bars, effectively illustrating the impact of different declaration methods on array behavior:

//@version=6 indicator("Using `var` with arrays") //@variable An array that initializes on every bar. a = array.new<float>() array.push(a, close) //@variable An array that expands its size by 1 on each bar. var b = array.new<float>(0) array.push(b, close) // Populate a table on the chart's last bar to display the sizes of the arrays and compare it to the number of chart bars. if barstate.islast var table displayTable = table.new(position.middle_right, 2, 3) table.cell(displayTable, 0, 0, "Array A size:", text_color = chart.fg_color, text_halign = text.align_right) table.cell(displayTable, 1, 0, str.tostring(a.size()), text_color = chart.fg_color, text_halign = text.align_left) table.cell(displayTable, 0, 1, "Array B size:", text_color = chart.fg_color, text_halign = text.align_right) table.cell(displayTable, 1, 1, str.tostring(b.size()), text_color = chart.fg_color, text_halign = text.align_left) table.cell(displayTable, 0, 2, "Number of chart bars:", text_color = chart.fg_color, text_halign = text.align_right) table.cell(displayTable, 1, 2, str.tostring(bar_index + 1), text_color = chart.fg_color, text_halign = text.align_left)

Results

- **Array A (Non-Persistent):** This array is reset at the beginning of each new bar. As a result, despite adding elements on each bar, its size remains constant, reflecting only the most recent addition.
- **Array B (Persistent):** This array retains its elements and accumulates new entries across bars, mirroring the growing count of chart bars. This persistent nature of the array shows its ability to track or aggregate data over the script's runtime.

For further details, consult the sections concerning variable declaration modes and their use in array declarations in the User Manual.

What are queues and stacks?

Scripts can use arrays to create queues and stacks.

Stacks

A stack uses the "last in, first out" (LIFO) principle, where the most recently added item is the first to be taken away. Think of this like a stack of plates, where you can only place a new plate on top or remove the top plate. To use an array as a stack, add elements to the end of the array using array.push() and remove elements from the end of the array using array.pop().

Queues

A queue uses the "first in, first out" (FIFO) principle, where the first item to be added is the first to be removed. This kind of queue in code is like a queue in real life, such as in a coffee shop, where no matter how many people join the end of the queue, the first person still gets served first. To use an array as a

queue, add elements to the end of the array using array.push() and remove them from the beginning using array.shift().

Stacks are particularly useful for accessing the most recent data, such as for tracking price levels. Queues are used for sequential data processing tasks, like event handling. Two example scripts follow, to illustrate these different usages.

Example: Arrays as stacks

This script uses arrays as stacks to manage pivot points. It draws lines from the pivot points and extends the lines with each new bar until price intersects them. When the script detects a pivot point, it adds (pushes) a new line to the stack. With each new bar, the script extends the end point of each line in the stack. It then checks whether price has intersected the high or low pivot lines at the top of the stack. If so, the script removes (pops) the intersected line from the stack, meaning that it will no longer be extended with new bars. Note that we do not need to iterate through the arrays to check all the lines, because price is always between only the high and low pivot lines at the end of each array.

//@version=6 indicator("Array as a stack", overlay = true) // @function Adds a new horizontal line to an array of lines at a specified pivot level. // @param id (array<line>) The array to which to add the new line. // @param pivot (float) The price level at which to draw the horizontal line. // @param lineColor (color) The color of the line. // @returns (void) The function has no explicit return. stackLine(array<line> id, float pivot, color lineColor) => if not na(pivot) array.push(id, line.new(bar_index - 10, pivot, bar_index, pivot, color = lineColor)) // @function Extends the endpoint (`x2`) of each line in an array to the current `bar_index`. // @param id (array<line>) The array containing the line objects to update. // @returns (void) The function has no explicit return. extendLines(array<line> id) => for eachLine in id eachLine.set_x2(bar_index) // @function Removes line objects from an array if they are above or below the current bar's high or low. // @param id (array<line>) The array from which to remove line objects. // @param isBull (bool) If true, remove bullish pivot lines below the high price; // if false, remove bearish pivot line above the low price. // @returns (void) The function has no explicit return. removeLines(array<line> id, bool isBull) => if array.size(id) > 0 float linePrice = line.get_price(array.last(id), bar_index) if isBull ? high > linePrice : low < linePrice array.pop(id) line(na) // Find the pivot high and pivot low prices. float pivotLo = ta.pivotlow(10, 10), float pivotHi = ta.pivothigh(10, 10) // Initialize two arrays on the first bar to stack our lines in. var array<line> pivotHiArray = array.new<line>() var array<line> pivotLoArray = array.new<line>() // If a pivot occurs, draw a line from the pivot to the current bar and add the line to the stack. stackLine(pivotHiArray, pivotHi, color.orange) stackLine(pivotLoArray, pivotLo, color.aqua) // Extend all lines in each array to the current bar on each bar. extendLines(pivotHiArray) extendLines(pivotLoArray) // Check the final element of each array to see if price exceeded the pivot lines. // Pop the line off the stack if it was exceeded. removeLines(pivotHiArray, true) removeLines(pivotLoArray, false)

Example: Arrays as queues

This script uses arrays as queues to track pivot points for monitoring recent support and resistance levels. It dynamically updates lines extending from the four most recent pivot highs and lows to the current bar with each new bar. When the script detects a new pivot high or low, it adds a line that represents this pivot to the respective queue. To maintain the queue's size at a constant four items, the script removes the oldest line in the queue whenever it adds a new line.

//@version=6 indicator("Array as a queue", overlay = true) int PIVOT_LEGS = 10 // @function Queues a new `value` at the end of the `id` array and removes // the first element if the array size exceeds the specified `maxSize`. // @param id (<any array type>) The array in which to queue the element. // @param

maxSize (int) The maximum allowed number of elements in the array. // If the array exceeds this size, the first element is removed. // @param value (<type of the array>) The new element to add to the array. // @returns (<type of the array>) The removed element. arrayQueue(id, int maxSize, value) => id.push(value) if id.size() > maxSize id.shift() // @function Adds a new horizontal line to an array at a certain pivot level and removes the oldest line. // @param id (array<line>) The array to which to add the new line. // @param pivot (float) The price level at which to draw the horizontal line. // @param numLines (int) The number of lines to keep in the queue. // @param lineColor (color) The color of the line to draw. // @returns (void) The function has no explicit return. queueLine(array<line> id, float pivot, int numLines, color lineColor) => if not na(pivot) arrayQueue(id, numLines, line.new(bar_index - PIVOT_LEGS, pivot, bar_index, pivot, color = lineColor)) // @function Extends the endpoint (`x2`) of each line in an array to the current `bar_index`. // @param id (array<line>) The array containing the line objects to update. // @returns (void) The function has no explicit return. extendLines(array<line> id) => for eachLine in id eachLine.set_x2(bar_index) // Find the pivot high and pivot low price. float pivotLo = ta.pivotlow(PIVOT_LEGS, PIVOT_LEGS) float pivotHi = ta.pivothigh(PIVOT_LEGS, PIVOT_LEGS) // Initialize two arrays on the first bar to queue our lines in. var array<line> pivotHiArray = array.new<line>() var array<line> pivotLoArray = array.new<line>() // If a pivot occurs, draw a line from the pivot to the current bar, add it to the queue, and remove the oldest line. queueLine(pivotHiArray, pivotHi, 4, color.orange) queueLine(pivotLoArray, pivotLo, 4, color.aqua) // Extend all lines in each array to the current bar on each bar. extendLines(pivotHiArray) extendLines(pivotLoArray)

For more information on manipulating arrays, see the Arrays section in the User Manual.

How can I perform operations on all elements in an array?

In Pine Script, there are no built-in functions to apply operations across the entire array at once. Instead, scripts need to iterate through the array, performing the operation on each element one at a time.

The easiest way to retrieve each element in an array is by using a for…in structure. This type of loop retrieves each element in turn, without the need for specifying the number of iterations.

The simple form of the loop has the format `for element in array`, where `element` is a variable that is assigned the current array element being accessed.

If the script's logic requires the position of the element in the array, use the two-argument form: `for [index, element] in array`. This form returns both the current element and its index in a tuple.

Example: retrieving array elements

This first example script uses an array as a queue to store lines representing the latest four pivot highs and lows. The for…in loop performs two tasks:

- It adjusts the `x2` endpoint of each line to the current bar_index.
- It changes the colors of the lines to blue for support or orange for resistance, based on their position relative to the close price.

Note that neither of these operations requires knowing the index of the array element.

//@version=6 indicator("Example: `for...in` loop", overlay = true) // @function Queues a new `value` at the end of the `id` array and removes // the first element if the array size exceeds the specified `maxSize`. // @param id (<any array type>) The array in which to queue the element. // @param maxSize (int) The maximum allowed number of elements in the array. // If the array exceeds this size, the first element is

removed. // @param value (<type of the array>) The new element to add to the array. // @returns (<type of the array>) The removed element. arrayQueue(id, int maxSize, value) => id.push(value) if id.size() > maxSize id.shift() // @function Adds a new horizontal line to an array at a certain pivot level and removes the oldest line. // @param id (array<line>) The array to which to add the new line. // @param pivot (float) The price level at which to draw the horizontal line. // @param numLines (int) The number of lines to keep in the queue. // @param lineColor (color) The color of the line to draw. // @returns (void) The function has no explicit return. queueLine(array<line> id, float pivot, int numLines, color lineColor) => if not na(pivot) arrayQueue(id, numLines, line.new(bar_index - 10, pivot, bar_index, pivot, color = lineColor)) // @function Extends the endpoint (`x2`) of each line in an array to the current `bar_index`. // @param id (array<line>) The array containing the line objects to update. // @returns (void) The function has no explicit return. extendLines(array<line> id) => for eachLine in id eachLine.set_x2(bar_index) // @function Adjusts the color of each line in an array. If the `close` is above the line, the line is // set to `bullColor` (support), else, `bearColor` (resistance). // @param id (array<line>) The array containing the line objects. // @param bullColor (color) The color to apply to the line if `close` is equal to or higher than the line's price. // @param bearColor (color) The color to apply to the line if `close` is below the line's price. // @returns (void) The function has no explicit return. colorLines(array<line> id, color bullColor, color bearColor) => for eachLine in id if close >= eachLine.get_price(bar_index) eachLine.set_color(bullColor) else eachLine.set_color(bearColor) // Find the pivot high and pivot low prices. float pivotLo = ta.pivotlow(10, 10) float pivotHi = ta.pivothigh(10, 10) // Initialize two arrays on the first bar to queue our lines in. var array<line> pivotHiArray = array.new<line>(), var array<line> pivotLoArray = array.new<line>() // If a pivot occurs, draw a line from the pivot to the current bar, add it to the queue, and remove the oldest line. queueLine(pivotHiArray, pivotHi, 4, color.orange), queueLine(pivotLoArray, pivotLo, 4, color.aqua) // Extend all lines in each array to the current bar on each bar. extendLines(pivotHiArray), extendLines(pivotLoArray) // Set the color of lines as support or resistance by checking if the closing price is above or below the lines. colorLines(pivotHiArray, color.aqua, color.orange) colorLines(pivotLoArray, color.aqua, color.orange)

Example: retrieving array elements and indices

In our second script, we use the two-argument variant of the for…in loop to access elements and their indices in an array. This method facilitates operations that depend on element indices, such as managing parallel arrays or incorporating index values into calculations. The script pairs a boolean array with an array of positive and negative random integers. The boolean array flags whether each corresponding integer in the primary array is positive.

//@version=6 indicator("Example: `for...in` loop with index") // Create an array of random integers above and below 0. var valuesArray = array.from(4, -8, 11, 78, -16, 34, 7, 99, 0, 55) // Create an array to track the positive state of each integer. var isPos = array.new<bool>(10, false) // Iterate over the valuesArray using a `for...in` loop and update each corresponding element in the bool array to true // if the value is above 0, or false if it is below 0. for [i, eachValue] in valuesArray if eachValue > 0 array.set(isPos, i, true) // Print both arrays in a label on the last historical bar. if barstate.islastconfirmedhistory label.new(bar_index +1, high, str.tostring(valuesArray) + "\n" + str.tostring(isPos), style = label.style_label_left, textcolor = chart.fg_color)

What’s the most efficient way to search an array?

The obvious way to search for an element in an array is to use a loop to check each element in turn. However, there are more efficient ways to search, which can be useful in different situations. Some of the following functions return only the index of a value. Programmers can then use array.get() if the script needs the actual value.

Checking if a value is present in an array

If all the script needs to do is to check whether a certain value is present in an array or not, use the array.includes() function. If the element is found, the function returns true; otherwise, it returns false. This method does not return the index of the element.

The following example script checks if the value `3` is present in the `values` array, and displays either “found” or “not found” in a label.

//@version=6 indicator("Example: Find whether an array element is present") array<int> values = array.from(1, 3, 5) int searchValue = input(3, "Value to Search For") bool valuePresent = array.includes(values, searchValue) if barstate.islast label.new(bar_index, low, valuePresent ? "Search value found" : "Search value not found", textcolor = color.white)

Finding the position of an element

If the script requires the *position* of an element, programmers can use the array.indexof() function. This function returns the index of the *first* occurrence of a value within an array. If the value is not found, the function returns `-1`. This method does not show whether there are multiple occurrences of the search value in the array. Depending on the script logic, this method might not be suitable if the array contains values that are not unique.

The following script searches for the first occurrence of `101.2` in the `prices` array and displays “found” and the value’s index in a label, or “not found” otherwise.

//@version=6 indicator("Example: Find index of array element") array<float> prices = array.from(100.5, 101.2, 102.8, 100.5) float searchValue = input(101.2, "Value to Search For") int indexFound = array.indexof(prices, searchValue) if barstate.islast string lblString = switch indexFound < 0 => "Search value: not found" => "Search value: found\n Index: " + str.tostring(indexFound) label.new(bar_index, high, lblString, textcolor = color.white, textalign = text.align_left, text_font_family = font.family_monospace)

Binary search

If the script requires the position of the element in a sorted array, the function array.binary_search() returns the index of a value more efficiently than array.indexof(). The performance improvement is significant for large arrays. If the value is not found, the function returns `-1`.

> NoticeThe array.binary_search() function requires arrays of “int” or “float” values, and the values must be sorted in *ascending order* for correct results.

This script uses a binary search to find the value `100.5` within an array of prices. The script displays the original array, the sorted array, the target value (100.5), and the result of the search. If the value is found, it displays “found”, along with the index of the value. If the value is not found, it displays “not found”.

//@version=6 indicator("Example: Binary search in sorted array") array<float> sortedPrices = array.from(100.5, 102.3, 98.7, 99.2) string originalArrayString = str.tostring(sortedPrices) float searchValue = input(100.5) // Ensure that the array is sorted (order is ascending by default); this step is crucial for binary search. array.sort(sortedPrices) string sortedArrayString = str.tostring(sortedPrices) int searchValueIndex = array.binary_search(sortedPrices, searchValue) bool valueFound = searchValueIndex >= 0 if barstate.islast

string lblTxt = str.format("Original array: {0}\n Sorted Array: {1}\n Search value: {2}\n Value found: {3}\n Position: {4}", originalArrayString, sortedArrayString, searchValue, valueFound, searchValueIndex) label.new(bar_index, high, lblTxt, textcolor = color.white, textalign = text.align_left, text_font_family = font.family_monospace)

If a script does not need the exact value, the functions array.binary_search_leftmost() and array.binary_search_rightmost() provide an effective way to locate the nearest index to a given value in sorted arrays. These functions return the index of the value, if it is present. If the value is not present, they return the index of the element that is closest to the search value on the left (smaller) or right (larger) side.

How can I debug arrays?

To debug arrays, scripts need to display the contents of the array at certain points in the script. Techniques that can display the contents of arrays include using plots, labels, tables, and Pine Logs.

For information about commonly encountered array-related errors, refer to the array Error Handling section in the User Manual.

Plotting

Using the plot() function to inspect the contents of an array can be helpful because this function can show numerical values on the script's status line, the price scale, and the Data Window. It is also easy to review historical values.

Limitations of this approach include:

- Arrays must be of type "float" or "int".
- The number of plots used for debugging counts towards the plot limit for a script.
- Plot calls must be in the global scope and scripts cannot call them conditionally. Therefore, if the size of the array varies across bars, using this technique can be impractical.

Here we populate an array with the open, high, low and close (OHLC) prices on each bar. The script retrieves all the elements of the array and plots them on the chart.

//@version=6 indicator("Plot array elements") array<float> ohlc = array.from(open, high, low, close) plot(ohlc.get(0), "Open", color.red) plot(ohlc.get(1), "High", color.yellow) plot(ohlc.get(2), "Low", color.blue) plot(ohlc.get(3), "Close", color.green)

Using labels

Using labels to display array values on certain bars is particularly useful for non-continuous data points or to view all elements of an array simultaneously. Scripts can create labels within any local scope, including functions and methods. Scripts can also position drawings at any available chart location, irrespective of the current bar_index. Unlike plots, labels can display the contents of a variety of array types, including boolean and string arrays.

Limitations of using labels include:

- Pine labels display only in the chart pane.
- Scripts can display only up to a maximum number of labels.

In the following example script, we monitor the close price at the last four moving average (MA) crosses in a queued array and use a label to display this array from a local scope whenever a cross occurs:

//@version=6 indicator("Array elements in a label", overlay = true) var array<float> crossPrices = array.new<float>(4) float fastMa = ta.ema(close, 9) float slowMa = ta.ema(close, 21) if ta.cross(fastMa, slowMa) crossPrices.push(close) crossPrices.shift() label.new(bar_index, high, str.tostring(crossPrices), textcolor = color.white) plot(fastMa, "Fast MA", color.aqua) plot(slowMa, "Slow MA", color.orange)

For more information, see the Labels section of the Debugging page in the User Manual.

Using label tooltips

If programmers want to be able to inspect the values in an array on every bar, displaying the contents of the array in a label is not convenient, because the labels overlap and become difficult to read. In this case, displaying the array contents in a label tooltip can be visually clearer. This method has the same advantages and limitations as using labels in the section above.

This example script plots a fast and a slow moving average (MA). It maintains one array of the most recent three values of the fast MA, and one array for the slow MA. The script prints empty labels on each bar. The tooltip shows the values of the MA arrays and whether or not the MAs crossed this bar. The labels are displayed in a semi-transparent color, and the tooltip is visible only when the cursor hovers over the label.

//@version=6 indicator("Array elements in a label tooltip", overlay = true) // Create two arrays to hold the MA values. var array<float> fastMaValues = array.new<float>(3) var array<float> slowMaValues = array.new<float>(3) // Calculate the MAs. float fastMa = ta.ema(close, 9) float slowMa = ta.ema(close, 21) // Load the current MA values into the arrays. fastMaValues.push(math.round(fastMa,2)), slowMaValues.push(math.round(slowMa,2)) // Remove the first element to keep the arrays at the same size. fastMaValues.shift(), slowMaValues.shift() // Define the string to print in the label tooltip. string labelString = str.format("Fast MA array: {0}\n Slow MA array: {1}\n Crossed this bar? {2}", str.tostring(fastMaValues), str.tostring(slowMaValues), ta.cross(fastMa, slowMa)) //Print the labels. label.new(bar_index, high, text="", color=color.new(chart.fg_color,90), textcolor = chart.fg_color, tooltip=labelString) plot(fastMa, "Fast MA", color.aqua) plot(slowMa, "Slow MA", color.orange)

Using tables

Using tables for debugging offers a more organized and scalable alternative to labels. Tables can display multiple "series" strings in a clear format that remains unaffected by the chart's scale or the index of the bars.

Limitations of using tables for debugging include that, unlike labels, the state of a table can only be viewed from the most recent script execution, making it hard to view historical data. Additionally, tables are computationally more expensive than other debugging methods and can require more code.

In the following example script, we create and display two unrelated arrays, to show how flexible this approach can be. The first array captures the times of the last six bars where a Golden Cross occurred. The second array records the last eight bar indices where the Relative Strength Index (RSI) reached new all-time highs within the chart's history. We use the `whenSince()` function from the PineCoders' getSeries library to create and update the arrays. This function treats the arrays as queues, and limits their size.

//@version=6 indicator("Debugging arrays with tables", overlay = true) // Import the `getSeries` PineCoders library to build fixed-size arrays populated on specific conditions. //

https://www.tradingview.com/v/Bn7QkdZR/ import PineCoders/getSeries/1 as PCgs // Calculate MAs and create cross condition. float ma50 = ta.sma(close, 50) float ma200 = ta.sma(close, 200) bool goldenCross = ta.cross(ma50, ma200) // Calculate the RSI and determine if it's hitting a new all-time high. float myRsi = ta.rsi(close, 20) bool newRsiAth = myRsi == ta.max(myRsi) // Create two arrays using the imported `whenSince()` function. array<float> goldenCrossesTimes = PCgs.whenSince(time_close, goldenCross, length = 6) array<float> barIndicesOfHiRSIs = PCgs.whenSince(bar_index, newRsiAth, length = 8) // Plot the MAs for cross reference. plot(ma50, "50 MA", color.aqua) plot(ma200, "200 MA", color.orange) // On the last historical bar, display the date and time of the last crosses. if barstate.islast // Declare our MA table to display the Golden Cross times. var table maTable = table.new(position.top_right, 2, 8, color.new(color.black, 100), color.gray, 1, color.gray, 1) // Create a title cell for the MA table and merge cells to form a banner two cells wide. table.cell(maTable , 0, 0, "Golden Cross Times", text_color = color.black, bgcolor = #FFD700) table.merge_cells(maTable , 0, 0, 1, 0) // Loop the array and write cells to the MA table containing the cross time for each element of the array. Number each element in the left row. // Format the UNIX time value to a formatted time string using `str.format_time()`. for [i, timeValue] in goldenCrossesTimes table.cell(maTable, 0, i + 1, str.tostring(i + 1), text_color = #FFD700) table.cell(maTable, 1, i + 1, str.format_time(int(timeValue), "yyyy.MM.dd 'at' HH:mm:ss z"), text_color = chart.fg_color) // Create a second table to display the indices of the last eight RSI all-time highs. var table rsiTable = table.new(position.bottom_right, 1, 1, color.new(color.black, 100), color.gray, 1, color.gray, 1) table.cell(rsiTable, 0, 0, "Bar indices of RSI ATHs\n" + str.tostring(barIndicesOfHiRSIs), text_color = chart.fg_color)

Using Pine Logs

Pine Logs are messages that display in the Pine Logs pane, along with a timestamp when the logging function was called. Scripts can create log messages at specific points during the execution of a script. Programmers can use the `log.*()` functions to create Pine Logs from almost anywhere in a script – including inside the *local scopes* of user-defined functions, conditional structures, and loops.

By logging messages to the console whenever there is a modification to the array, programmers can track the logical flow of array operations in much more detail than by using other approaches.

The script below updates a previous example script from the section on queues and stacks to add logging. It uses arrays as stacks to track lines drawn from pivot points. When a pivot occurs, the script adds a new line to the stack and continues to extend the lines on each bar until an intersection with price occurs. If an intersection is found, the script removes (pops) the intersected line from the stack, meaning it will no longer be extended with new bars.

The messages in the Pine Logs pane are time stamped and offer detailed information about when elements are added to and removed from the arrays, the current size of the arrays, and the specific prices at which elements were added.

//@version=6 indicator("Array as a stack", overlay = true) // @function Adds a new horizontal line to an array of lines at a specified pivot level. // @param id (array<line>) The array to which to add the new line. // @param pivot (float) The price level at which to draw the horizontal line. // @param lineColor (color) The color of the line. // @returns (void) The function has no explicit return. stackLine(array<line> id, float pivot, color lineColor) => if not na(pivot) array.push(id, line.new(bar_index - 10, pivot, bar_index, pivot, color = lineColor)) if barstate.isconfirmed log.info("\nNew line added at {0}\nArray size: {1}", pivot, id.size()) // @function Extends the endpoint (`x2`) of each line in an array to the current `bar_index`. // @param id (array<line>) The array containing the line objects to update. // @returns (void) The function has no explicit return. extendLines(array<line> id) => for eachLine in id eachLine.set_x2(bar_index) // @function Removes

line objects from an array if they are above or below the current bar's high or low. // @param id (array<line>) The array from which to remove line objects. // @param isBull (bool) If true, remove bullish pivot lines below the high price; // if false, remove bearish pivot line above the low price. // @returns (void) The function has no explicit return. removeLines(array<line> id, bool isBull) => if array.size(id) > 0 float linePrice = line.get_price(array.last(id), bar_index) if isBull ? high > linePrice : low < linePrice array.pop(id) if barstate.isconfirmed log.warning("\nLine removed from {0} array.\nPrice breached {1}\nArray size: {2}", isBull ? "Highs" : "Lows", linePrice, id.size()) // Find the pivot high and pivot low prices. float pivotLo = ta.pivotlow(10, 10), float pivotHi = ta.pivothigh(10, 10) // Initialize two arrays on the first bar to stack our lines in. var array<line> pivotHiArray = array.new<line>() var array<line> pivotLoArray = array.new<line>() // If a pivot occurs, draw a line from the pivot to the current bar and add the line to the stack. stackLine(pivotHiArray, pivotHi, color.orange), stackLine(pivotLoArray, pivotLo, color.aqua) // Extend all lines in each array to the current bar on each bar. extendLines(pivotHiArray), extendLines(pivotLoArray) // Check the final element of each array. If price exceeded the pivot lines, pop the line off the stack. removeLines(pivotHiArray, true), removeLines(pivotLoArray, false)

Can I use matrices or multidimensional arrays in Pine Script?

Pine Script does not directly support multidimensional arrays; however, it provides matrices and user-defined types (UDTs). Programmers can use these data structures to create and manipulate complex datasets.

Matrices

Pine Script matrices are like two-dimensional arrays. They organize data in a rectangular grid, facilitating operations like transformations, linear algebra, and other complex calculations. They are particularly useful for quantitative modeling, such as portfolio optimization, correlation matrix analysis, and more. Just as in arrays, all elements in a matrix must be of the same type, which can be a built-in or a user-defined type. Pine Script provides a range of functions for manipulating and performing calculations on matrices, including addition, subtraction, multiplication, and more.

Using UDTs for multidimensional structures

Programmers can achieve similar functionality to multidimensional arrays through defining user-defined types (UDTs). For example, a script can define a UDT that includes an array as one of its fields. UDTs themselves can be contained in arrays. In this way, scripts can effectively have arrays of arrays.

For more information, see the sections on Matrices, Maps, and Objects in the User Manual.

How can I debug objects?

To debug objects, create custom functions that break down an object into its constituent fields and convert these fields into strings. See the Debugging section of the User Manual for information about methods to display debug information. In particular, Pine Logs can display extensive and detailed debug information. See the FAQ section about debugging arrays using Pine Logs for an explanation of using logs for debugging.

In our example script, we create a user-defined type (UDT) named `openLine`, which includes fields such as `price`, `openTime`, and a line object called `level`. On the first bar of each session, the script initializes a new `openLine` instance. This object tracks the session's opening price and time, and it draws a line at the open

price, extending from the session’s start to its close. An array stores each `openLine` object. A custom function `debugOpenLine()` breaks an `openLine` object into its individual fields, converts the fields to strings, and then logs a message that displays these strings in the console.

```
//@version=6 indicator("Debugging objects", overlay = true) // Define the user-defined type. type openLine float price int openTime line level // @function Queues a new `arrayElement` at the end of the `id` array and removes // the first element if the array size exceeds the specified `maxSize`. // @param id (<any array type>) The array in which the element is queued. // @param maxSize (int) The maximum allowed number of elements in the array. // If the array exceeds this size, the first element is removed. // @param arrayElement (<array type) The new element to add to the array. // @returns (<array type>) The removed element. arrayQueue(id, int maxSize, value) => id.push(value) if id.size() > maxSize id.shift() // @function Logs detailed information about an open line object for debugging purposes. // @param ol (openLine) The open line object to log. // @returns (void) Function has no explicit return. debugOpenLine(openLine ol) => if barstate.isconfirmed log.info( "\nprice: {0}\nopenTime: {1}\nlevel line coords:\nx1: {2}\ny1: {3}\nx2: {4}\ny2: {5}", ol.price, ol.openTime, str.format_time(ol.level.get_x1()), ol.level.get_y1(), str.format_time(ol.level.get_x2()), ol.level.get_y2() ) // Create an empty `openLine` array. var openLineArray = array.new<openLine>() // On session start, create a new `openLine` object and add it to the array. // Use the custom debug function to print the object's fields to the Pine Logs pane. if session.isfirstbar_regular openLine ol = openLine.new(open, time) ol.level := line.new(time, open, time_close("D"), open, xloc.bar_time, color = color.aqua) arrayQueue(openLineArray, 4, ol) debugOpenLine(ol)
```

Functions

Can I use a variable length in functions?

Many built-in technical analysis (TA) functions have a `length` parameter, such as `ta.sma(source, length)`. A majority of these functions can process "series" lengths, i.e., lengths that can change from bar to bar. Some functions, however, only accept "simple" integer lengths, which must be known on bar zero and not change during the execution of the script.

Check the Reference Manual entry for a function to see what type of values a function can process.

Additional resources

For more advanced versions of functions that support "series" lengths, or for extra technical analysis tools explore the ta library on the TradingView profile. This library offers a range of extended TA-related capabilities and custom implementations.

User-defined functions

For built-in functions that do not accept "series" lengths and for which the functionality is not available in the ta library, consider creating a user-defined function.

How can I calculate values depending on variable lengths that reset on a condition?

To calculate certain values that are dependent on varying lengths, which also reset under specific conditions, the ta.barssince() function can be useful. This function counts the number of bars since the last occurrence of a specified condition, automatically resetting the count each time this condition is met. There are, however, some considerations to take into account when using this function for this purpose.

Firstly, before the condition is met for the first time in a chart's history, ta.barssince() returns na. This value is not usable as a length for functions and can cause errors, especially during execution on a chart's early bars. For a more robust version, use nz() to replace the na return of ta.barssince() with zero for early bars.

Secondly, when the condition is met, ta.barssince() returns zero for that bar, since zero bars have elapsed since the condition was last true.

Since lengths cannot be zero, it is necessary to add one to a returned value of zero, ensuring that the length is always at least one.

Here's an example of how to use these principles for a practical purpose. The following example script calculates the highest and lowest price points since the start of a new day. We use timeframe.change() to detect the start of a new day, which is our condition. The ta.barssince() function calculates the number of bars that elapsed since this condition was last met. The script passes this number, or "lookback", to the ta.lowest() and ta.highest() functions, which determine the highest and lowest points since the start of the new day:

//@version=6 indicator("Highest/lowest since new day", "", true) // Identify the start of a new day and calculate the number of bars since then. bool newDay = timeframe.change("D") int lookback = nz(ta.barssince(newDay)) + 1 // Calculate the highest and lowest point since the new day began. float

lowestSinceNewDay = ta.lowest(lookback) float highestSinceNewDay = ta.highest(lookback) // Plot the high/low level since the start of a new day. plot(lowestSinceNewDay, "High today", color.orange) plot(highestSinceNewDay, "Low today", color.aqua) // Change the background color to indicate the start of a new day. bgcolor(newDay ? color.new(color.gray, 80) : na) // Display the varying lookback period in Data Window. plot(lookback, "Lookback", display = display.data_window)

> NoticeIf a script uses a dynamic value as the argument for a built-in function parameter that defines a lookback length, such as the `length` parameter of ta.sma(), an error can occur if the value increases unpredictably. This behavior is most common on realtime bars, but it can also happen on historical bars in some cases. Refer to the section The requested historical offset (X) is beyond the historical buffer's limit (Y) in the Error messages page for more information about the error and its causes.

How can I round a number to x increments?

Rounding numbers to specific increments is useful for tasks like calculating levels for grid trading, dealing with fractional shares, or aligning trading parameters to specific pip values.

In this example, the `roundToIncrement()` function accepts a value and an increment as parameters. It divides the value by the increment, rounds the result, then multiplies it by the increment to give the rounded value. To demonstrate the function, the closing price is rounded to the nearest increment defined in the user menu:

//@version=6 indicator("Round to x increment demo", overlay = true) float incrementInput = input.float(0.75, "Increment", step = 0.25) // @function Rounds a value to the nearest multiple of a specified increment. // @param value The value to round. // @param increment The increment to round the value to. // @returns The rounded value. roundToIncrement(value, increment) => math.round(value / increment) * increment plot(series = roundToIncrement(close, incrementInput), color = chart.fg_color)

How can I control the precision of values my script displays?

The `precision` and `format` arguments in the indicator() or strategy() declaration statement control the number of decimals in the values that a script displays.

By default, scripts use the precision of the price scale. To display more decimal places, specify a `precision` argument that exceeds the value of the current price scale.

How can I control the precision of values used in my calculations?

The `math.round(number, precision)` variation of the math.round() function rounds values according to a specified precision. Alternatively, the math.round_to_mintick() function rounds values to the nearest tick precision of the chart's symbol.

How can I round to ticks?

To round values to the tick precision of a chart's symbol, use the function math.round_to_mintick(). To convert the resulting number to a string, use `str.tostring(myValue, format.mintick)` to first round the number to tick precision and then return its string representation, where `myValue` is the number to convert into a rounded string.

How can I abbreviate large values?

There are different ways to abbreviate large numerical values, such as volume. For instance, the number 1,222,333.0 can be simplified to 1.222M. Here are some methods to accomplish this:

Apply a global setting

Use the argument `format = format.volume` within either the indicator() or strategy() statements. Using this setting, displays all values in the script in their abbreviated forms.

Abbreviate specific values

To abbreviate only certain values for string display, use the `str.tostring(value, format.volume)` function.

Use a custom function

To specify a custom precision or abbreviate values up to trillions, use a custom function. In the following example script, the user-defined function `abbreviateValue()` divides the `value` by a power of ten based on its magnitude, and adds an abbreviation letter (K, M, B, or T) to represent the magnitude of the original value. The function also adds a subtle space between the value and the magnitude letter. The `print()` function displays the value on the chart for visualization.

//@version=6 indicator("Value abbreviation example") // @function Converts a numeric value into a readable string representation featuring the appropriate order // of magnitude abbreviation (K, M, B, T). // @param value (float) The value to format. // @param precision (string) The numerical precision of the result. ("" for none, ".00" for two digits, etc.) // @returns (string) The formatted value as a string with the appropriate abbreviation suffix. abbreviateValue(float value, string precision) => float digitsAmt = math.log10(math.abs(value)) string formatPrecision = "#" + precision string result = switch digitsAmt > 12 => str.tostring(value / 1e12, formatPrecision + "T") digitsAmt > 9 => str.tostring(value / 1e9, formatPrecision + "B") digitsAmt > 6 => str.tostring(value / 1e6, formatPrecision + "M") digitsAmt > 3 => str.tostring(value / 1e3, formatPrecision + "K") => str.tostring(value, "#" + formatPrecision) print(formattedString) => var table t = table.new(position.middle_right, 1, 1) table.cell(t, 0, 0, formattedString, bgcolor = color.yellow) print(abbreviateValue(volume, ".00"))

How can I calculate using pips?

You can use the custom function `calcBaseUnit()` in the following example script to retrieve the expected pip value for Forex symbols, or the minimum tick size for other symbols:

//@version=6 indicator("Pip calculation example") // @function Calculates the chart symbol's base unit of change in asset prices. // @returns (float) A ticks or pips value of base units of change. calcBaseUnit() => bool isForexSymbol = syminfo.type == "forex" bool isYenQuote = syminfo.currency == "JPY" bool isYenBase = syminfo.basecurrency == "JPY" float result = isForexSymbol ? isYenQuote ? 0.01 : isYenBase ? 0.00001 : 0.0001 : syminfo.mintick // Call the function and plot the result in a label var label baseUnitLabel = na if barstate.islast baseUnitLabel := label.new(x=bar_index + 1, y=open, text="Base Unit: " + str.tostring(calcBaseUnit(), "#.######"), style=label.style_label_left, color=color.new(color.blue, 0), textcolor=color.white) label.delete(baseUnitLabel[1])

> NoteThis function might not address all potential scenarios. Therefore, we recommend confirming this function's results with the pip values shown by your broker.

How do I calculate averages?

The method of calculating averages depends on the type of values to average.

Distinct variables

To find the average of a small number of discrete variables, use the function math.avg(). Simply pass each of the variables as an argument to this function.

Bar prices

To find the average price of a single bar, use the built-in variables hl2, hlc3, and ohlc4.

Series values

To compute the average of the last *n* values in a series, use the function ta.sma().

Custom datasets

To average a custom set of values, organize them into an array and use array.avg(). For complex datasets, programmers can use the matrix.avg() function to average the contents of a matrix. For a deeper understanding of averaging custom datasets, refer to this conditional averages publication.

How can I calculate an average only when a certain condition is true?

The usual methods of calculating averages, which were discussed in the calculating averages section above, apply across *all* data points in a range. To calculate averages of only those values that occur under specific conditions, calculate *conditional averages* using custom functions.

The example script below imports a library called ConditionalAverages and uses two of its functions:

- The `avgWhen()` function calculates the average volume of session opening bars across the entire dataset.
- The `avgWhenLast()` function averages the opening volumes for the last five session opening bars.

The condition for these conditional averages is *session opening bars*, which we determine using the session.isfirstbar_regular variable.

//@version=6 indicator("Average session opening volume") import PineCoders/ConditionalAverages/2 as PCca // Color aqua for the session's opening bar, otherwise distinct colors for up/down volume columns. color volumeColor = switch session.isfirstbar_regular => color.aqua close > open => color.new(#D1D4DC, 65) => color.new(#787B86, 65) // Plot the volume columns. plot(volume, "volume", volumeColor, 4, plot.style_histogram) // Average volume over *all* session opening bars in the dataset. plot(PCca.avgWhen(source = volume, condition = session.isfirstbar_regular), "avg. When", #FF00FF) // Average volume over the last five opening bars. plot(PCca.avgWhenLast(source = volume, condition = session.isfirstbar_regular, count = 5), "avgWhenInLast()", #00FF00)

> TipSome built-in functions, such as ta.sma() *ignore* the bars with na values in their calculations. Therefore, it is possible to perform some condition-based calculations using these functions. For example, the call `ta.sma(session.isfirstbar_regular ? volume : na, 5)` returns the same result as the `PCca.avgWhenLast()` call in the example above, because its calculation includes only the volume values from the latest five bars where the value of session.isfirstbar_regular is `true`.

How can I generate a random number?

Use the math.random() function to generate pseudorandom numbers. This example script creates a circle plot with random RGB color values and a random y value between 0 and 1:

//@version=6 indicator("Random demo", overlay = false) // Generate a pseudorandom price value (the default range is 0 to 1). float y = math.random() // Generate a color with red, green, and blue values as separate pseudorandom values between 0 and 255. color plotColor = color.rgb(math.random(0, 255), math.random(0, 255), math.random(0, 255)) plot(series = y, title = "Random number", color = plotColor, linewidth = 2, style = plot.style_circles)

How can I evaluate a filter I am planning to use?

To evaluate a filter, insert your filter code into the Filter Information Box - PineCoders FAQ script. This script conducts an impulse response analysis and shows the filter's characteristics in a label on the chart.

For further details and a guide on integrating your filter into the code, refer to the publication's description.

What does nz() do?

The nz() function replaces any na values with zero, or with a user-defined value if the `replacement` argument is specified. This function helps to prevent na values from interfering with calculations.

The following example script shows an exaggerated failure as a result of a single na value. The `barRangeRaw` variable is na only once, on the first bar, because it references a bar that does not exist, using the history-referencing operator. The alternative variable `barRangeWithNz` uses nz() to prevent an na value from ever occurring.

The `dependentCalculation` variable takes one of these values and uses it to calculate a crude average of the bar range. If the input to this calculation is ever na, the series will be na forever after that.

Choose between the two values for bar range using the input setting, and the range either displays or not. In the latter case, the Data Window shows that the value of `dependentCalculation` is `0`, meaning na.

//@version=6 indicator("`na` values on first bar demo") bool useNzInput = input.bool(true, "Use `nz` to ensure value is never na") // This variable is na on the first bar. float barRangeRaw = close - close[1] // This variable is never na. float barRangeWithNz = close - nz(close[1], open) // Choose the value to use based on the input float barRange = useNzInput ? barRangeWithNz : barRangeRaw // Perform a calculation that depends on the barRange var float dependentCalculation = 0 dependentCalculation := ((dependentCalculation + barRange)/2) // Plot the results plot(dependentCalculation, title="Average Bar Range")

The nz() function is also useful to protect against any potential divide-by-zero errors. It guarantees a return value even when an equation unintentionally features a zero in the denominator. Consider the following code snippet that intentionally creates a divide-by-zero scenario by setting the denominator to zero. Without the nz() function, this expression would return na, instead of zero:

```
float dbzTest = nz(close / (close - close))
```

FAQ

Get real OHLC price on a Heikin Ashi chart

Suppose, we have a Heikin Ashi chart (or Renko, Kagi, PriceBreak etc) and we've added a Pine script on it:

//@version=6 indicator("Visible OHLC", overlay=true) c = close plot(c)

You may see that variable `c` is a Heikin Ashi *close* price which is not the same as real OHLC price. Because `close` built-in variable is always a value that corresponds to a visible bar (or candle) on the chart.

So, how do we get the real OHLC prices in Pine Script® code, if current chart type is non-standard? We should use `request.security` function in combination with `ticker.new` function. Here is an example:

//@version=6 indicator("Real OHLC", overlay = true) t = ticker.new(syminfo.prefix, syminfo.ticker) realC = request.security(t, timeframe.period, close) plot(realC)

In a similar way we may get other OHLC prices: *open*, *high* and *low*.

Get non-standard OHLC values on a standard chart

Backtesting on non-standard chart types (e.g. Heikin Ashi or Renko) is not recommended because the bars on these kinds of charts do not represent real price movement that you would encounter while trading. If you want your strategy to enter and exit on real prices but still use Heikin Ashi-based signals, you can use the same method to get Heikin Ashi values on a regular candlestick chart:

//@version=6 strategy("BarUpDn Strategy", overlay = true, default_qty_type = strategy.percent_of_equity, default_qty_value = 10) maxIdLossPcntInput = input.float(1, "Max Intraday Loss(%)") strategy.risk.max_intraday_loss(maxIdLossPcntInput, strategy.percent_of_equity) needTrade() => close > open and open > close[1] ? 1 : close < open and open < close[1] ? -1 : 0 trade = request.security(ticker.heikinashi(syminfo.tickerid), timeframe.period, needTrade()) if trade == 1 strategy.entry("BarUp", strategy.long) if trade == -1 strategy.entry("BarDn", strategy.short)

Plot arrows on the chart

You may use plotshape with style `shape.arrowup` and `shape.arrowdown`:

//@version=6 indicator('Ex 1', overlay = true) condition = close >= open plotshape(condition, color = color.lime, style = shape.arrowup, text = "Buy") plotshape(not condition, color = color.red, style = shape.arrowdown, text = "Sell")

You may use the `plotchar` function with any unicode character:

//@version=6 indicator('buy/sell arrows', overlay = true) condition = close >= open plotchar(not condition, char='↓', color = color.lime, text = "Buy") plotchar(condition, char='↑', location = location.belowbar, color = color.red, text = "Sell")

Plot a dynamic horizontal line

There is the function `hline` in Pine Script, but it is limited to only plot a constant value. Here is a simple script with a workaround to plot a changing hline:

//@version=6 indicator("Horizontal line", overlay = true) plot(close[10], trackprice = true, offset = -9999) // `trackprice = true` plots horizontal line on close[10] // `offset = -9999` hides the plot plot(close, color = #FFFFFFFF) // forces display

Plot a vertical line on condition

//@version=6 indicator("Vertical line", overlay = true, scale = scale.none) // scale.none means do not resize the chart to fit this plot // if the bar being evaluated is the last baron the chart (the most recent bar), then cond is true cond = barstate.islast // when cond is true, plot a histogram with a line with height value of 100,000,000,000,000,000,000.00 // (10 to the power of 20) // when cond is false, plot no numeric value (nothing is plotted) // use the style of histogram, a vertical bar plot(cond ? 10e20 : na, style = plot.style_histogram)

Access the previous value

//@version=6 //... s = 0.0 s := nz(s[1]) // Accessing previous values if (condition) s := s + 1

Get a 5-days high

Lookback 5 days from the current bar, find the highest bar, plot a star character at that price level above the current bar

//@version=6 indicator("High of last 5 days", overlay = true) // Milliseconds in 5 days: millisecs * secs * mins * hours * days MS_IN_5DAYS = 1000 * 60 * 60 * 24 * 5 // The range check begins 5 days from the current time. leftBorder = timenow - time < MS_IN_5DAYS // The range ends on the last bar of the chart. rightBorder = barstate.islast // ————— Keep track of highest `high` during the range. // Intialize `maxHi` with `var` on bar zero only. // This way, its value is preserved, bar to bar. var float maxHi = na if leftBorder if not leftBorder[1] // Range's first bar. maxHi := high else if not rightBorder // On other bars in the range, track highest `high`. maxHi := math.max(maxHi, high) // Plot level of the highest `high` on the last bar. plotchar(rightBorder ? maxHi : na, "Level", "—", location.absolute, size = size.normal) // When in range, color the background. bgcolor(leftBorder and not rightBorder ? color.new(color.aqua, 70) : na)

Count bars in a dataset

Get a count of all the bars in the loaded dataset. Might be useful for calculating flexible lookback periods based on number of bars.

//@version=6 indicator("Bar Count", overlay = true, scale = scale.none) plot(bar_index + 1, style = plot.style_histogram)

Enumerate bars in a day

//@version=6 indicator("My Script", overlay = true, scale = scale.none) isNewDay() => d = dayofweek na(d[1]) or d != d[1] plot(ta.barssince(isNewDay()), style = plot.style_cross)

Find the highest and lowest values for the entire dataset

//@version=6 indicator("", "", true) allTimetHi(source) => var atHi = source atHi := math.max(atHi, source) allTimetLo(source) => var atLo = source atLo := math.min(atLo, source) plot(allTimetHi(close), "ATH", color.green) plot(allTimetLo(close), "ATL", color.red)

Query the last non-na value

You can use the script below to avoid gaps in a series:

//@version=6 indicator("") series = close >= open ? close : na vw = fixnan(series) plot(series, style = plot.style_linebr, color = color.red) // series has na values plot(vw) // all na values are replaced with the last non-empty value

Indicators

Can I create an indicator that plots like the built-in Volume or Volume Profile indicators?

The Volume and Visible Range Volume Profile indicators (along with some other built-in indicators) are written in Java. They display data on the main chart pane in a unique way:

- The bars are anchored to the bottom or right edge of the chart, not to an absolute x or y value.
- The length of the bars is a relative percentage of the available space and is not an absolute price or number of bars.
- The length of the bars adjusts automatically according to the data from the range of bars that are visible on the chart. The lengths of the bars are normalized so as never to appear too small or too large.
- The width of the bars adjusts automatically to fit the visible space.

It is difficult for Pine Script® indicators to plot values in the same way.

Limitations of `plot.style_columns`

If volume, or another series, plotted as columns, it is anchored to the bottom of the chart, and the width and length of the bars can adjust dynamically. However, the tops of the bars are defined by absolute price values. This means that it is not possible for the series to be plotted on the main chart without distorting the price scale. Also, plots must be defined during processing of the bar they are plotted on, and cannot be plotted retroactively.

Limitations of drawings

Drawing objects such as lines and boxes are anchored to an absolute price scale, not to the edge of the chart. Drawing objects do not adjust their length automatically. Lines do not adjust their width automatically. Although boxes can be drawn exactly one bar wide, and so adjust their width automatically, they cannot be drawn so as to fit exactly in one bar; they always draw from the middle of one bar to the middle of another.

The following example script demonstrates some techniques for approximating the way that the built-in Volume indicator displays.

- We use the chart.right_visible_bar_time and chart.left_visible_bar_time built-in variables, through the PineCoders' VisibleChart library, to define the bars that are visible. Then we calculate the highest and lowest price, and the highest volume, for that period.
- We plot lines retroactively, after the visible window and all related values are known.
- We anchor the lines below the lowest visible price, so that it looks as if they are anchored to the bottom edge of the chart.
- We scale the length of all the volume bars so that the longest bar in the set is approximately 30% of the screen height, similar to the built-in Volume indicator.
- We adjust the width of the lines depending on how many bars are visible.

> TipThe bottom margin of the chart must be set to zero in order for the lines to start from the bottom edge of the chart. To set the margin, right-click the chart background and click "Settings…" then "Canvas", and

> set the “Bottom” margin in the “Margins” section. To preserve the same space at the bottom of the chart, add a bottom margin in the script settings.

```
//@version=6 indicator("Dynamically scaled volume", overlay=true, max_lines_count=500) // Import the PineCoders' VisibleChart library import PineCoders/VisibleChart/5 as visibleChart const float RELATIVE_HEIGHT = 0.3 // 30% matches the built-in volume indicator. const string BOTTOM_TTIP = ("Copy the bottom margin % from your chart settings to here, and then set the bottom" + " margin to *zero* on the chart settings.") int bottomInput = input.int(title = "Bottom Margin %", defval = 10, minval = 0, maxval = 100, tooltip = BOTTOM_TTIP) // Get the highest volume, and highest and lowest price points, by calculating on each bar during the visible window. var float hiVol = na var float hiPrice = na var float loPrice = na if visibleChart.barIsVisible() hiVol := na(hiVol) ? volume : math.max(hiVol, volume) hiPrice := na(hiPrice) ? high : math.max(hiPrice, high) loPrice := na(loPrice) ? low : math.min(loPrice, low) int bars = visibleChart.bars() // Calculate the thickness for the lines based on how many bars are displayed. int lineWidth = math.ceil(1000/bars) // Draw the lines once, when the visible window ends. if time == chart.right_visible_bar_time // Calculate the bottom y coordinate for all lines once. float priceDifference = hiPrice - loPrice float scale = (priceDifference / hiVol) * RELATIVE_HEIGHT float bottomY = loPrice - (bottomInput / 100) * priceDifference // Loop through the visible window using the historical operator. for i = bars - 1 to 0 // Calculate the top y coordinate for each line. float topY = bottomY + (volume[i] * scale) // Draw the line. line.new(x1 = bar_index - i, y1 = bottomY, x2 = bar_index - i, y2 = topY, color = close[i] >= open[i] ? color.new(color.green, 50) : color.new(color.red, 50), width = lineWidth)
```

This script has some other limitations:

- The lines do not begin from the bottom of the chart if other indicators display plots or drawings below that level.
- In common with any script that uses the chart.right_visible_bar_time or chart.left_visible_bar_time built-in variables, the script must refresh each time the chart is moved or a new bar appears.
- There is a maximum limit of 500 lines per script.
- The width of the lines is calculated based on how many bars are visible. However, a Pine script has no way of knowing how much blank space there is to the right of the final bar. If the user scrolls to the right, the lines can appear too wide and overlap each other.

Can I use a Pine script with the TradingView screener?

The TradingView screener uses only its built-in filters, and cannot use a Pine script. However, the Pine Screener can use any Pine script containing plot() or alertcondition() calls to scan and filter symbols. Add a personal, built-in, or community indicator to your favorites to use it in the Pine Screener. See this Help Center article for more information.

Alternatively, search for “screener” in the Community Collection to find scripts that use the request.security() function to screen pre-set lists of symbols.

See also this FAQ entry for an example script that generates alerts on multiple symbols.

How can I use the output from one script as input to another?

Scripts with an input of type input.source() can take a plot from another script (up to a maximum of ten) as an input. Select the script and plot to take as input in the script’s “Settings” tab. If the user removes the script from the chart and adds it again, they must select the correct inputs again.

The sources used as external inputs must originate from indicators; they cannot originate from strategies. However, plots originating from indicators *can* be used in strategies.

For further information, refer to this blog post and the Source input section in the User Manual.

Can my script draw on the main chart when it's running in a separate pane?

Scripts that have the `overlay` parameter in the indicator() or strategy() functions set to `false` appear in a separate pane to the main chart. Such scripts can affect the display of the main chart in only two ways:

- Changing bar colors, using the barcolor() function.
- Forcing plots to overlay, using `force_overlay = true` in the plotting function. The `force_overlay` parameter is available in most functions that draw on the chart.

Is it possible to export indicator data to a file?

The option "Export chart data…" in the dropdown menu at the top right corner of the chart exports a comma-separated values (CSV) file that includes time, OHLC data, and any plots generated by your script. This option can also export strategy data.

To include specific information in the CSV file, ensure that it is plotted by the script. If this extra information is far from the symbol's price and the existing indicator plots, and plotting it on the chart could distort the scale of the script, or if you prefer not to display certain plots, consider using the `display` parameter in the plot() function.

Here is an example plot that displays the close only in the Data Window. The plot title "No chart display" becomes the column header for this value in the CSV file.

```
plot(close * 0.5, "No chart display", display = display.data_window)
```

Alternatively, the "Scale price chart only" in the chart settings maintains the script's scale. To access these settings, right-click on the chart's price scale.

To determine if a condition is true or false, use the plotshape() function, which records a 1 (for true) or 0 (for false) in the CSV file.

Other data and timeframes

What kinds of data can I get from a higher timeframe?

Generally speaking, the request.security() function can get the same kinds of data from another timeframe that is available on the chart timeframe. Scripts can retrieve built-in variables like open, high, low, close, volume, and bar_index.

The request.security() function can also evaluate built-in or user-defined functions in the requested *context* (timeframe and symbol). For example, the following example script retrieves the Average True Range (ATR) value from the daily (`1D`) timeframe by passing the ta.atr() function as the `expression` argument.

//@version=6 indicator("HTF ATR") float higherTfAtr = request.security(symbol = syminfo.tickerid, timeframe = "1D", expression = ta.atr(14)) plot(higherTfAtr)

> NoticeWhile the above script executes on realtime bars, its requested data includes *unconfirmed* values from developing daily bars. Only the *confirmed* values for the closed bars and the latest value for the open bar remain available after the script reloads. To learn how to request *non-repainting* values from another context, see the Avoiding repainting section of the Other timeframes and data page.

Which `request.*()` function should I use for lower timeframes?

The request.security() function is intended for accessing data at timeframes that are equal to or higher than the chart's current timeframe. It is *possible* to retrieve lower-timeframe (LTF) data using this function. However, the function returns the value from only *one* LTF bar within the chart's current bar (the last bar, by default).

If the script supplies the `expression` as a variable or simple calculation, directly or within a function, the data that request.security() returns from a lower timeframe is generally of limited use (see the first script in this section for an example). It is possible, however, to construct a function that performs meaningful calculations on the LTF bars and then returns the result on the last bar. The following example script counts the number of LTF bars in a chart bar and returns this value on the last LTF bar. For simplicity, the timeframes are hardcoded to `"1D"` and `"1W"` and the script should therefore be run from a chart on the weekly timeframe.

//@version=6 indicator("Counting intrabars using `request.security()`") // @function Calculates the quantity of 1D bars in a week of trading. // @returns (int) The number of intrabars within the current weekly bar up to the current moment. qtyIntrabars() => var int count = 0 count := timeframe.change("W") ? 1 : count + 1 int qtyIntrabars = request.security(syminfo.tickerid, "1D", qtyIntrabars()) plot(qtyIntrabars, "qtyIntrabars", style=plot.style_histogram)

When using the request.security() function on a lower timeframe, all calculations that reference individual LTF bars must be done *within the requested context*, and only the *result* of the calculation is returned. Using the request.security_lower_tf() function for intrabar analysis is usually easier and more powerful, because it returns an array of data from *all* available intrabars within a chart bar. Returning the data for each bar allows scripts to perform calculations on specific bars or all bars in the main script context.

In the following version of our example script, we use request.security_lower_tf() to perform the same calculations. With this approach, we do not need to explicitly define the current chart's timeframe, nor do we need a custom function.

//@version=6 indicator("Counting intrabars using `request.security_lower_tf()`") // Count the number of elements in the array of close prices for each LTF bar in the current chart's bar. int qtyIntrabars = array.size(request.security_lower_tf(syminfo.tickerid, "1D", close)) plot(qtyIntrabars, "qtyIntrabars", style=plot.style_histogram)

> NoteAlthough this approach is simpler to implement than the previous, it is also more computationally expensive, because it retrieves a *new array* instead of a single value on each execution. If either approach produces the same result – as is the case in our examples above – it is often more optimal to use the first approach, especially if the script is large and performs many intrabar calculations.

See the sections in the User Manual page "Other timeframes and Data" about `request.security_lower_tf()` and using `request.security()` on lower timeframes to learn more about the differences between running these functions on a lower timeframe.

How to avoid repainting when using the `request.security()` function?

Repainting can be a problem when retrieving data from higher or lower timeframes using request.security().

Retrieving data from a different symbol on the chart's timeframe does not risk repainting. Requesting data from the chart's own symbol and timeframe does not result in repainting either, but it is usually unnecessary to use request.security() rather than simply referencing the chart's own values (except when modifying the chart's ticker using `ticker.*()` functions). When using the chart's timeframe, there is no need to offset the data, change the default `lookahead` value, or use barmerge.lookahead_on in order to avoid repainting.

Higher timeframes

Values from a *higher timeframe* (HTF) often repaint because a historical bar on the chart might include data from a realtime bar on the HTF. Realtime values can change throughout the bar; for example, the close price reflects the *latest* price update in a realtime bar. When the HTF bar closes and its values become fixed, the relevant historical chart bars *change* to adjust to the fixed HTF values. This behavior is described in the Historical and realtime behavior section of the User Manual. Users expect historical bars not to change, which is one reason why repainting is such a concern.

To prevent repainting, use confirmed values that remain consistent across all bars. The most robust method is to offset all expressions by 1. For example, instead of `close`, which is equivalent to `close[0]`, use `close[1]`. The request.security() call must also use barmerge.lookahead_on. This method returns data that is up to one HTF bar "late", and is thus not subject to change.

> NoticeNeglecting to offset the `expression` argument in an HTF request causes **lookahead bias** on historical bars. See the lookahead section of the Other timeframes and data page and the Repainting page's section about future leak to learn more.

The following example script demonstrates the use of a single bar offset to the `expression` argument and barmerge.lookahead_on in request.security() to ensure that the data behaves the same on historical and realtime bars. The script calls runtime.error() to trigger a custom runtime error if the chart's timeframe exceeds or matches the daily timeframe, to prevent the return of inaccurate values.

//@version=6 indicator("HTF close" , overlay = true) float dailyClose = request.security(syminfo.tickerid, "1D", close[1], lookahead = barmerge.lookahead_on) plot(dailyClose) if timeframe.in_seconds() >= timeframe.in_seconds("1D") runtime.error("Chart timeframe must be less than 1D.")

See the Avoiding repainting section of the User Manual for more information.

Lower timeframes

Although the request.security() function is intended to operate on timeframes greater than or equal to the chart timeframe, it *can* request data from a lower timeframe (LTF), with limitations. When accessing data from a LTF, the function evaluates the given expression in the LTF context and returns the result from a *single* LTF bar per chart bar. The specific LTF bar returned depends on the lookahead parameter:

- barmerge.lookahead_on returns the *first* intrabar of the period historically, but the *last* intrabar in realtime.
- barmerge.lookahead_off always returns the last intrabar for both historical and realtime data. To prevent repainting (in this case, inconsistent results between realtime and historical data) use barmerge.lookahead_off for lower timeframe data requests.

In most cases, using the request.security_lower_tf() function is more suitable for lower timeframes, as it returns an array containing data from *all* available intrabars within a chart bar. See the section on `request.security_lower_tf()` to learn more.

How can I convert the chart's timeframe into a numeric format?

The timeframe.in_seconds() function converts a timeframe specified in timeframe.period format into an equivalent number of seconds. Having the timeframe in a numeric format means that scripts can calculate the number of time units within a specific timeframe, or perform operations that adjust the timeframe used in HTF calls in relation to the chart's timeframe, as described in this FAQ entry.

In this script example, we use the timeframe.in_seconds() function to determine the chart's timeframe, measured in seconds. Since no specific `timeframe` argument is specified, the function defaults to using `timeframe.period`, which returns the chart's current timeframe. The script then converts the timeframe in seconds into various other units of time, including minutes, hours, and days, and displays the original string and converted numeric values in a table:

//@version=6 indicator("Timeframe to value") tfInSec = timeframe.in_seconds() tfInMin = tfInSec / 60 tfInHrs = tfInMin / 60 tfInDays = tfInHrs / 24 if barstate.islastconfirmedhistory var table displayTable = table.new(position.top_right, 2, 5, na, color.gray, 1, color.gray, 1) table.cell(displayTable, 0, 0, "Original TF string", text_color = chart.fg_color) table.cell(displayTable, 1, 0, "\"" + timeframe.period + "\"", text_color = chart.fg_color) table.cell(displayTable, 0, 1, "Timeframe in seconds", text_color = chart.fg_color) table.cell(displayTable, 1, 1, str.tostring(tfInSec), text_color = chart.fg_color) table.cell(displayTable, 0, 2, "Timeframe in minutes", text_color = chart.fg_color) table.cell(displayTable, 1, 2, str.tostring(tfInMin), text_color = chart.fg_color) table.cell(displayTable, 0, 3, "Timeframe in hours", text_color = chart.fg_color) table.cell(displayTable, 1, 3, str.tostring(tfInHrs), text_color = chart.fg_color) table.cell(displayTable, 0, 4,

"Timeframe in days", text_color = chart.fg_color) table.cell(displayTable, 1, 4, str.tostring(tfInDays), text_color = chart.fg_color)

How can I convert a timeframe in “float” minutes into a string usable with `request.security()`?

The built-in function timeframe.from_seconds() function converts a number of seconds into a timeframe string that is compatible with request.security().

The example script below converts a user-defined number of minutes into a timeframe string using the timeframe.from_seconds() function. The script then requests the close price from that timeframe using request.security() and plots it. Additionally, we display the resulting timeframe string in a table on the chart’s top right corner:

//@version=6 indicator("Target TF in string from float minutes", "", true) float tfInMinInput = input.float(1440, "Minutes in target timeframe (<= 0.0167 [1 sec.])", minval = 0.0167) // Convert target TF in minutes from input into string. string targetTfString = timeframe.from_seconds(int(tfInMinInput * 60)) // Fetch target timeframe's close. float targetTfClose = request.security(syminfo.tickerid, targetTfString, close) // Plot target timeframe close. plot(targetTfClose, "Target TF close") // Display the target timeframe string in a table cell at the chart's top right. if barstate.islastconfirmedhistory var table displayTable = table.new(position.top_right, 1, 1, color.new(color.yellow, 70), color.gray, 1, color.gray, 1) table.cell(displayTable, 0, 0, str.format("Target TF (string): {0}", targetTfString), text_color = chart.fg_color)

How do I define a higher timeframe that is a multiple of the chart timeframe?

This example script uses the timeframe.in_seconds() and timeframe.from_seconds() functions to calculate a higher timeframe that is a fixed multiple of the chart’s current timeframe. Using the input for the multiplier, the user can define the ratio between the chart’s timeframe and the higher timeframe. The script then calculates the Relative Strength Index (RSI) for both the chart’s timeframe and the higher timeframe, plotting both in a separate pane for comparison. We display the calculated higher timeframe string in a table on the main chart pane by using `force_overlay`:

//@version=6 indicator("Multiple of current TF", overlay = false) // Provide an input to specify the multiple to apply to the chart's timeframe. float tfMult = input.float(4, minval = 1) // Get multiple of current timeframe. string targetTfString = timeframe.from_seconds(int(timeframe.in_seconds() * tfMult)) // Create RSI from the current timeframe. float myRsi = ta.rsi(close, 14) plot(myRsi, "Current TF RSI", color = color.silver) // Non-repainting HTF RSI. float myRsiHtf = request.security(syminfo.tickerid, targetTfString, myRsi[1], lookahead = barmerge.lookahead_on) plot(myRsiHtf, "Non-repainting HTF RSI", color = color.green) // Display the calculated timeframe at the top right of the main chart pane. if barstate.islastconfirmedhistory var table displayTable = table.new(position.top_right, 1, 1, color.new(color.yellow, 70), color.gray, 1, color.gray, 1, force_overlay = true) table.cell(displayTable, 0, 0, str.format("Target TF (string): {0}", targetTfString), text_color = chart.fg_color)

How can I plot a moving average only when the chart’s timeframe is 1D or higher?

To plot a moving average on a chart only if it has a timeframe of daily (“1D”) or higher, scripts can use the timeframe.in_seconds() function to convert the chart’s current timeframe into seconds. Since a day consists of 86400 seconds, any timeframe equal to or exceeding this value corresponds to a daily or longer duration.

The example script below calculates and plots a Simple Moving Average (SMA) of the closing prices over the last 200 bars. The script uses a ternary operator to return the moving average on timeframes of 1D or greater, or na if the timeframe is shorter than one day. Because plot() calls cannot be in a local scope, scripts cannot conditionally call this function. Passing an na value as the `series` argument is an effective way to not plot anything. Note that plotting an na value *does* count towards the script’s plot limit.

//@version=6 indicator("Timeframe-dependent MA", overlay = true) bool tfIsDailyOrGreater = timeframe.in_seconds() >= 86400 float ma = ta.sma(close, 200) plot(tfIsDailyOrGreater ? ma : na, "MA", color.aqua)

What happens if I plot a moving average from the 1H timeframe on a different timeframe?

The request.security() function can access data from a different *context*, such as a different symbol or timeframe. There are different considerations when accessing data from a timeframe *higher* or *lower* than the chart timeframe.

First, let’s consider an example of plotting data from a *lower* timeframe. The following script plots a 21-period Exponential Moving Average (EMA) derived from the 1-hour (1H) timeframe on any chart, irrespective of the timeframe of that chart:

//@version=6 indicator("1hr EMA", overlay = true) plot(request.security(syminfo.tickerid, "60", ta.ema(close, 21)), color = color.orange)

Assuming that we run this script on a chart with a daily timeframe, we encounter the following problems:

- For each daily bar, the chart can plot only 1 of the 24 MA values theoretically available. The plot misses out the intraday fluctuations and trends that a 1H moving average (MA) is typically used to identify.
- The script above displays only the EMA value calculated for the *final* 1-hour bar of each day. In realtime, the plot displays the most recently known value.

Unlike request.security(), the request.security_lower_tf() function is intended for use on lower timeframes. It returns an array containing data from all available intrabars within a chart bar. See this section of the User Manual to learn more.

We could rewrite the script to use request.security_lower_tf(), but plotting a moving average from a lower timeframe is still not very practical.

A more logical approach is to plot MAs from a *higher* timeframe. This strategy shows broader market trends within the context of shorter-term price movements. For example, plotting a daily MA on a 1H chart provides insights into how intraday prices are trending relative to the longer-term daily average.

In the following example script, we plot the 21 EMA calculated at the 1H timeframe, but only when the chart’s timeframe is *equal to or lower than* 1H. We call the request.security() function in the recommended way to avoid repainting.

```
//@version=6 indicator("HTF EMA", overlay = true) // Input to specify the timeframe for `request.security() call. string tfinput = input.timeframe("60", "Timeframe for MA") // @function A wrapper for the `request.security()` function for non-repainting calls to HTFs. // @param timeframe Timeframe of the requested data. // To use the chart's timeframe, use an empty string or the `timeframe.period` variable. // @param expression An expression to calculate and returne from the request.security() call's context. // @returns The result of the calculated expression. htfSecurity(string timeframe, expression) => result = request.security(syminfo.tickerid, timeframe, expression[1], lookahead = barmerge.lookahead_on) // Calculate the moving average in the chart context. float ma = ta.ema(close, 21) // Calculate the moving average in the specified `tfInput` timeframe. float htfMA = htfSecurity(tfinput, ma) // Check whether the requested timeframe is greater or less than the chart's timeframe. bool tfIsGreater = timeframe.in_seconds() < timeframe.in_seconds(tfinput) bool tfIsLess = timeframe.in_seconds() > timeframe.in_seconds(tfinput) // Plot the HTF MA, the chart MA, or nothing, depending on the timeframe. float maPlot = tfIsGreater ? htfMA : tfIsLess ? na : ma plot(maPlot, "Requested MA", color.orange) // Display a message in a table indicating that the requested timeframe is lower than the chart's timeframe, if applicable. if barstate.islastconfirmedhistory and tfIsLess var table displayTable = table.new(position.bottom_right, 1, 1, color.new(color.yellow, 70)) table.cell(displayTable, 0, 0, "Requested TF is lower than chart's TF\nNo MA displayed", text_color = color.red)
```

Why do intraday price and volume values differ from values retrieved with `request.security()` at daily timeframes and higher?

Intraday open, high, low, close, and volume (OHLCV) values can be different from those from request.security() at daily timeframes and higher for a number of reasons, including the following:

- **Different data feeds:** Certain trades (like block trades and OTC trades, especially in stocks) are recorded only at the end of the trading day, so their volume affects the End-of-Day (EOD) feed but not the intraday feed.
- **Price discrepancies:** There can be slight differences in prices between EOD and intraday data. For example, an EOD high might not match any intraday highs due to variations in data feeds.
- **Extended hours data:** EOD data feeds do not include information from trading outside regular hours, unlike some intraday feeds. For instance, the bars of an hourly chart might straddle the open of a session, mixing data from pre-market and regular trading.

For an extended list of factors with detailed explanations, refer to the Data feeds section in the User Manual.

Programming

What does “scope” mean?

The *scope* of a variable is the part of a script that defines the variable and in which it can be referenced. There are two main types of scope: *global* and *local*.

Global Scope: The global scope is all of the script that is not inside a function, if statement, or other conditional structure. Code from anywhere in the script can access global variables. There is only one global scope.

Local Scope: Code that is inside a function or in any local block (one that is inset by four spaces) defines a local scope. Only code that is in the same local scope can access a local variable. There can be many local scopes.

The following example script gives an “Undeclared identifier” error when we try to access a local variable from the global scope.

//@version=6 indicator("Scope demo") // Global scope int globalValue = close > open ? 1 : -1 if barstate.isconfirmed // Local scope int localValue = close > open ? 1 : -1 plot(localValue, "Local variable", chart.fg_color, 2)

To fix this error, we can declare the variable in the global scope, thus making it accessible from any scope in the script, and then conditionally modify it within a local block:

//@version=6 indicator("Scope demo") // Global scope int globalValue = close > open ? 1 : -1 int localValue = na if barstate.isconfirmed // Local scope localValue := close > open ? 1 : -1 plot(localValue, "Local variable", chart.fg_color, 2)

Similarly, the following script gives an “Undeclared identifier” error when we try to access a variable defined in one local scope from another local scope. In this case, local scope 1 *contains* local scope 2, but the same problem would be present if they were on the same level. When a scope contains another one, the inner scope can access variables declared in the outer one, but not vice versa.

//@version=6 indicator("Scope demo") bool isUpCandleWithLargerUpWick = false if barstate.isconfirmed // Local scope 1 bool upWickIsLarger = (high - math.max(open, close)) > (math.min(open,close) - low) if close > open // Local scope 2 bool isUpCandle = true isUpCandleWithLargerUpWick := upWickIsLarger and isUpCandle ? true : false plot(isUpCandleWithLargerUpWick, "Global variable depending on two local variables", chart.fg_color, 2)

For more information about scopes, see the Code section of the User Manual.

How can I convert a script to a newer version of Pine Script®?

See the Migration Guides section of the User Manual for instructions about upgrading the version of Pine that a script uses.

Can I access the source code of “invite-only” or “closed-source” scripts?

No; only *open* scripts have their source code visible. The source code of *protected* and *invite-only* scripts is hidden and can only be seen by the script author.

Refer to the Visibility types section of the Publishing scripts page to learn more about the differences between open-source, protected, and invite-only scripts. To learn about the difference between *public* and *private* scripts, see the Privacy types section on that page.

Is Pine Script an object-oriented language?

Although Pine Script is not strictly an object-oriented programming language, it incorporates some object-oriented features, notably user-defined types (UDTs). Scripts can create objects as instances of a UDT. These objects have one or more fields, which can store values of various data types.

Here is a simple example of how to use the type keyword to create an object:

//@version=6 indicator("Object demo") // Define a new type named `pivot`. type pivot int x float y bool isHigh // Create a new `pivot` with specific values. pivot newPivot = pivot.new(bar_index, close, true) // Plot the `y` component of `newPivot`. plot(newPivot.y)

In this example, we create an object `newPivot`, which is an instance of the user-defined type `pivot`. The script then plots the `y` field of `newPivot`.

How can I access the source code of built-in indicators?

There are two ways to access the source code of built-in indicators that are written in Pine:

Create a new indicator

In the Pine Script Editor, click the dropdown menu (the arrow in the upper-left corner of the editor pane) and choose the "*Create new*" > "*Built in…*" option. Select the built-in indicator that you want to work with.

Edit the code

With the indicator displayed on the chart, click on the curly braces `{}` next to the indicator name to open it in the Pine Editor. To edit the code, click the option to create a working copy.

Some built-in indicators, such as the Volume Profile or chart pattern indicators, are not written in Pine and so the code for these indicators is not accessible. These indicators are not included in the "*Built-in script*" menu, and curly braces are not displayed next to their names on the chart.

How can I examine the value of a string in my script?

Scripts can print strings to Pine Logs on any or every bar, along with messages about the logic of the script at that point. See the Pine Logs section of the User Manual for information about logging.

Scripts can also display string in labels or label tooltips. The following example script displays a string in a label on the last bar of the chart using a custom function.

//@version=6 indicator("print()", "", true) print(string txt) => // Create a persistent label var label myLabel = label.new(bar_index, na, txt, xloc.bar_index, yloc.price, color(na), label.style_label_left, chart.fg_color, size.large, text.align_left) // Update the label's x and y position, and the text it displays.

label.set_xy(myLabel, bar_index, open) label.set_text(myLabel, txt) if barstate.islast print("Timeframe = " + timeframe.period)

How can I visualize my script's conditions?

If a script contains complex logical conditions, it can be difficult to debug the output. Visualizing each condition separately can help to debug any problems. See the Plotting and coloring conditions section of the User Manual for an example.

How can I make the console appear in the editor?

To display the console in the editor, either press the keyboard shortcut Ctrl + ` (grave accent), or right-click within the editor and choose the "Toggle Console" option.

How can I plot numeric values so that they don't affect the indicator's scale?

Plotting numerical values on the main chart pane can distort the price scale if the values differ too much from the price.

One way around this is not to plot the values on the chart, but use the Data Window to inspect them. Add `display = display.data_window` to the plot() call, and the values are visible in the Data Window for any single historical or realtime bar that the cursor hovers over.

Another option is to set the script to display in a separate pane by using `overlay = false` in the indicator() declaration. The user needs to delete and re-add the script to the chart if this parameter is changed. Plot the numeric values to track in the separate pane, and draw the rest of the script visuals on the main chart pane by using the `force_overlay` parameter.

Additionally, right-clicking on the scale on the chart brings out the dropdown menu. The "Scale Price Chart Only" option there makes it so the Auto mode of the chart scale only takes the chart itself into account, without adjusting for plots or other graphics of all indicators that overlay that chart.

Strategies

Using Pine Script® strategy scripts, users can test *simulated* trades on historical and realtime data, to backtest and forward test trading systems. Strategies are similar to indicators, but with added capabilities such as placing, modifying, and canceling simulated orders and analyzing their results. Scripts that use the strategy() function as their declaration statement gain access to the `strategy.*` namespace, which contains functions and variables for simulating orders and retrieving strategy information.

When a user applies a strategy that uses order placement commands to the chart, the strategy uses the broker emulator to calculate simulated trades, and displays hypothetical performance results in a strategy report in the chart's bottom panel.

Strategies support various types of orders including market, limit, stop, and stop-limit orders, allowing programmers to simulate different trading scenarios. Strategy order commands can send alerts when order fill events occur. An *order fill event* is triggered by the broker emulator when it executes a simulated order in realtime.

For a thorough exploration of strategy features, capabilities, and usage, refer to the Strategies section in the User Manual.

Strategy basics

How can I turn my indicator into a strategy?

To convert an indicator to a strategy, begin by replacing the indicator() declaration with the strategy() declaration. This designates the script as a strategy.

Add order placement commands for simulating orders. Use logical conditions from the initial indicator to trigger the commands in the converted strategy.

The following example includes two scripts: an initial indicator script and a strategy script converted from the indicator. We use a simple RSI oscillator as a momentum indicator to gauge the direction of a market's momentum, with values above 50 indicating an upward (bullish) trend and values below 50 signaling a downward (bearish) trend:

The initial indicator colors the plot line and the bars on the chart in a lime color when the RSI is greater than 50 and fuchsia when less than 50. We use plotshape() to plot triangles at the top and bottom of the oscillator on bars where the RSI crosses over or under the 50 level.

//@version=6 indicator("Example RSI indicator") float rsi = ta.rsi(close, 14) plot(rsi, "RSI", rsi >= 50 ? color.lime : color.fuchsia) hline(50, "Middle line", linestyle = hline.style_solid) plotshape(ta.crossover(rsi, 50), "Cross up", shape.arrowup, location.bottom, color.lime) plotshape(ta.crossunder(rsi, 50), "Cross Down", shape.arrowdown, location.top, color.fuchsia) barcolor(rsi >= 50 ? color.lime : color.fuchsia)

In the converted strategy version, we maintain the same RSI crossover and crossunder conditions used in the indicator script. These conditions, which previously only drew the plotshape() triangles, now also trigger entry orders for long and short positions using the strategy.entry() function. A long entry is called when the RSI crosses over 50, and a short entry is initiated when it crosses under 50. A long entry cancels a short trade, and vice-versa.

//@version=6 strategy("Example RSI strategy") float rsi = ta.rsi(close, 14) plot(rsi, "RSI", rsi >= 50 ? color.lime : color.fuchsia) hline(50, "Middle line", linestyle = hline.style_solid) plotshape(ta.crossover(rsi, 50), "Cross up", shape.triangleup, location.bottom, color.lime) plotshape(ta.crossunder(rsi, 50), "Cross Down", shape.triangledown, location.top, color.fuchsia) barcolor(rsi >= 50 ? color.lime : color.fuchsia) if ta.crossover(rsi, 50) strategy.entry("Long", strategy.long, comment = "Long") if ta.crossunder(rsi, 50) strategy.entry("Short", strategy.short, comment = "Short")

How do I set a basic stop-loss order?

Stop losses are a risk management method that traders use to limit potential losses. The strategy.exit() function sets an order to exit a trade once it hits a specified price, thus preventing the loss from exceeding a predetermined amount.

To implement a basic stop loss in Pine Script, use the strategy.exit() function with either the `stop` or the `loss` parameter. The `stop` parameter specifies the *price* for the stop loss order, while the `loss` parameter sets the stop loss a certain number of *ticks* away from the entry order's price. Similarly, to set a take-profit level, use either the `limit` parameter, specifying the exact price for taking profit, or the `profit` parameter, defining the profit size in ticks from the entry price.

If a strategy.exit() call includes both the `stop` *and* `loss` parameters, or both the `limit` *and* `profit` parameters, the function uses the price level that is expected to trigger an exit first.

The following example script uses the tick-based `loss` parameter for long positions and the price-based `stop` parameter for short positions, and plots these stop levels on the chart. The script enters positions on the crossover or crossunder of two simple moving averages.

//@version=6 strategy("Stop using `loss` and `stop`", overlay = true) int lossTicksInput = input.int(60, "Stop loss in ticks (for longs)") float atrMultInput = input.float(1.0, "ATR multiplier (for shorts)", minval = 0) // Calculate the ATR value, adjusted by the multiplier, for setting dynamic stop loss levels on short positions. float atr = ta.atr(14) * atrMultInput // A persistent short stop loss level, updated based on short entry signals. var float shortStopLevel = na // Define conditions for entering long and short positions based on the crossover and crossunder of two SMAs. float ma1 = ta.sma(close, 14) float ma2 = ta.sma(close, 28) bool longCondition = ta.crossover(ma1, ma2) bool shortCondition = ta.crossunder(ma1, ma2) // On detecting a long condition, place a long entry. if longCondition strategy.entry("Long", strategy.long) // For a short condition, place a short entry and set the stop loss level by adding the ATR value to the closing price. if shortCondition strategy.entry("Short", strategy.short) shortStopLevel := close + atr // Apply a fixed-size stop loss for long positions using the specified input tick size in the `loss` parameter. strategy.exit(id = "Long Exit", from_entry = "Long", loss = lossTicksInput) // For short positions, set the stop loss at the calculated price level using the `stop` parameter. strategy.exit(id = "Short Exit", from_entry = "Short", stop = shortStopLevel) // Calculate the long stop loss price by subtracting the loss size from the average entry price. // Set the price to `na` if the strategy is not in a long position. float longStopPlot = strategy.position_size > 0 ? strategy.position_avg_price - lossTicksInput * syminfo.mintick : na // The short stop price is already calculated. Set to `na` if the strategy is not in a short position. float shortStopPlot = strategy.position_size < 0 ? shortStopLevel : na // Plot the moving averages and stop loss levels. plot(ma1, "MA 1", color.new(color.lime, 50)) plot(ma2, "MA 2", color.new(color.fuchsia, 50)) plot(longStopPlot, "Long Stop", color.red, style = plot.style_steplinebr) plot(shortStopPlot, "Short Stop", color.red, style = plot.style_steplinebr) // Color the background when long or short conditions are met. bgcolor(longCondition ? color.new(color.aqua, 80) : shortCondition ? color.new(color.orange, 90) : na)

Note that:

- In this example, we include `from_entry` arguments in the strategy.exit() calls so that each exit order closes only open trades with the corresponding entry ID. Without this argument, the exit intended for long positions would apply to both long *and* short positions, and the exit intended for short positions would likewise attempt to close any open position.

For more information, see the entry in the User Manual on strategy.exit().

How do I set an advanced stop-loss order?

Scripts can use different types of exits that are more advanced than simply closing the position at a predetermined level.

Bracket orders

A bracket order is a pair of orders that close the position if price moves far enough in either direction. Scripts can combine a stop-loss and take-profit order within a single strategy.exit() function call. See the FAQ entry about bracket orders for more details.

Trailing stop losses

A trailing stop loss is a stop loss that moves with price, but in the profitable direction only. To create a trailing stop, either adjust the stop price with each new bar, or use the built-in trailing stop parameters in the strategy.exit() function. Refer to the FAQ on implementing a trailing stop loss for information and examples.

Scaled exits

Scaled exits use multiple exit orders at varied price levels. When using tiered exit strategies, which progressively scale out of a position, ensure that the total quantity of all exit orders does not surpass the size of the initial entry position. Consult the FAQ on multiple exits for more information.

Moving a stop loss to breakeven

Adjusting a stop loss to the breakeven point once a specific condition is met can help in risk management. Details can be found in the FAQ on moving stop losses to breakeven.

Adjusting position size based on stop loss

Modify the position size relative to the stop loss to maintain a constant risk percentage of total equity. For more insights, see the FAQ on position sizing.

How can I save the entry price in a strategy?

Scripts can access the entry price for a *specific trade*, or the average entry price for a *position*.

Average entry price

The strategy.position_avg_price variable automatically updates to the average entry price of the current position. If the position consist of only one trade, the average price of the position is equal to the entry price of that single trade. If a strategy closes a market position that consists of multiple trades, trades are closed in the order they were opened, by default. Since the average price of the open position changes according

to which positions are still open, be aware of the order in which trades are closed, and if necessary, configure it using the `close_entries_rule` parameter of the strategy() declaration function.

Specific entry price

The strategy.opentrades.entry_price() function returns the entry price for a given trade ID. To find the entry price for the most recent open trade, and remembering that the trade indexes start at zero, use `float entryPrice = strategy.opentrades.entry_price(strategy.opentrades - 1)`.

How do I filter trades by a date or time range?

Using a date and time range filter in a strategy allows trades to be simulated only during a certain time period. Such filters can be useful to backtest specific historical periods, or to focus on particular times of the trading day.

Additionally, if the strategy sends signals for live trading, consider excluding all trades earlier than the trading start date and time, to ensure that the broker emulator starts in a neutral state.

The following example script restricts trading if a bar falls within a defined `startTime` and `endTime`, or outside of an optional intraday session window. The script colors the background red for bars that fall outside the time windows. On the screenshot, we've limited the trading range from June 1st 2024 to June 10th 2024, and additionally forbidden trading from 0000-0300 UTC:

//@version=6 strategy("Date/time filtering demo", "", true) // Timezone setting for date and time calculations. Adjust to the chart timezone. string TZ = "GMT+0" // Define the date window, an intraday time session to exclude, and the filtering to apply. bool useDateFilterInput = input.bool(true, "Allow trades only between the following dates (" + TZ + ")") int startTimeInput = input.time(timestamp("01 Jan 2000 00:00 " + TZ), " Start date", confirm = true) int endTimeInput = input.time(timestamp("01 Jan 2099 00:00 " + TZ), " End date", confirm = true) bool useTimeFilterInput = input.bool(false, "Restrict trades during the following times (" + TZ + ")") string sessionStringInput = input.session("0000-0300", "") // @function Determines whether the current bar falls within a specified date and time range. // @param startTime (int) A timestamp marking the start of the time window. // @param endTime (int) A timestamp marking the end of the time window. // @param useDateFilter (bool) Whether to filter between `startTime` and `endTime`. Optional. // @param useTimeFilter (bool) Whether to restrict trades in the time session. Optional. // @param timeSession (string) Session time range in 'HHMM-HHMM' format, used if `useTimeFilter` is true. // @param timeZone (string) Timezone for the session time, used if `useTimeFilter` is true. // @returns (bool) `true` if the current bar is within the specified date and time range. timeWithinAllowedRange(int startTime, int endTime, bool useDateFilter = true, bool useTimeFilter = false, string timeSession = "0000-0000", string timeZone = "GMT-0") => bool isOutsideTime = na(time(timeframe.period, timeSession, timeZone)) bool timeIsAllowed = useTimeFilter and isOutsideTime or not useTimeFilter bool dateIsAllowed = time >= startTime and time <= endTime or not useDateFilter bool result = timeIsAllowed and dateIsAllowed // Determine if each bar falls within the date window or outside the ignored time session. bool isWithinTime = timeWithinAllowedRange(startTimeInput, endTimeInput, useDateFilterInput, useTimeFilterInput, sessionStringInput, TZ) // Calculate RSI for simple trading signals. float rsi = ta.rsi(close, 14) // Generate trading signals based on RSI conditions, provided they occur within the permissible date/time range. bool enterLong = ta.crossover(rsi, 50) and isWithinTime bool enterShort = ta.crossunder(rsi, 50) and isWithinTime // Simulate trades only if they meet the filtering criteria. if enterLong strategy.entry("Long", strategy.long) if enterShort strategy.entry("Short", strategy.short) // Color the background red for bars falling outside the specified date/time range. bgcolor(isWithinTime ? na : color.new(color.red, 80), title = "Exempt times")

Note that:

- We use the time() function to calculate whether bars are outside the user-defined session times. For additional details on integrating session data in Pine Script, refer to the Sessions section in the User Manual.
- We set the `confirm` argument to `true` for the inputs that define the time range. When the script is first added to the chart, it prompts the user to confirm the values by clicking on the chart.
- We use a constant string `TZ` in our script to represent the time zone (set to `"GMT+0"` by default). Adjust this string to the local time zone or the exchange's time zone. We use a constant rather than an input so that we can include the time zone in input titles.

Order execution and management

Why are my orders executed on the bar following my triggers?

Each historical bar in a chart is composed of a single set of open, high, low and close (OHLC) data. Pine scripts execute on this data once per historical bar, at the **close** of the bar.

So that results are consistent between historical and realtime bars, strategies also execute at the close of realtime bars. The next possible moment for an order to be filled is the beginning of the next bar.

Users can alter a strategy's calculation behavior by configuring strategies to process orders at the close of the signal bar instead, by selecting the "Fill orders/On bar close" setting in the "Settings/Properties" tab. Programmers can do the same by setting the process_orders_on_close parameter to `true` in the strategy() declaration statement:

//@version=6 strategy("My Strategy", process_orders_on_close = true, ...)

An alternative method is to specify the `immediately` parameter as `true` in a strategy.close() or strategy.close_all() function call. This setting causes the broker emulator to close a position on the same tick that the strategy creates the close order – meaning, when bar closes instead of the beginning of the next one. The `process_orders_on_close` parameter affects all closing orders in the strategy, whereas the `immediately` parameter affects only the close order in which it is used.

However, processing orders on close might not give accurate results. For instance, if an alert occurs at the close of the session's last bar, the actual order can be executed only on the next trading day, since the bar is already closed. In contrast, the emulator would simulate the order being filled at the previous day's close. This discrepancy can lead to repainting, where the behavior of the strategy's simulation on historical bars differs from that seen in live trading.

How can I use multiple take-profit levels to close a position?

Setting up a strategy with multiple take profit levels enables traders to scale out of trades in segments to secure profits incrementally.

There are two main methods for scaling out at varying levels:

- Multiple strategy.exit() calls. This method is most suitable when each take-profit level has a corresponding stop loss.
- An OCA reduce group. This method is ideal for a different number of take-profit levels and stop losses.

Multiple `strategy.exit()` functions

Each strategy.exit() call can set a bracket order for a specific take-profit and stop-loss level. However, if a strategy uses multiple strategy.exit() functions with the **same** stop level, each function call triggers a *separate* order (and therefore multiple order alerts). If order alerts are configured to trigger real trades, ensure that the trade system handles multiple alerts at the same stop level appropriately.

The following example script uses two separate strategy.exit() functions, each with its own stop-loss and take-profit levels. The quantity for the first bracket order is set to 50% of the total position size. This combination of orders creates a scaled exit with distinct stop levels.

//@version=6 strategy("Multiple exit demo", overlay = true) int exitPercentInput = input.int(1, "Exit %", minval = 1, maxval = 99) float exitPercent = exitPercentInput / 100 //@variable Is `true` on every 100th bar. bool buyCondition = bar_index % 100 == 0 var float stopLoss1 = na, var float takeProfit1 = na // Exit levels for `Exit1` var float stopLoss2 = na, var float takeProfit2 = na // Exit levels for `Exit2` // Place orders when `buyCondition` is true and we are not in a position. if buyCondition and strategy.position_size == 0.0 stopLoss1 := close * (1 - exitPercent), takeProfit1 := close * (1 + exitPercent) // Update the levels based on the current price stopLoss2 := close * (1 - (2 * exitPercent)), takeProfit2 := close * (1 + (2 * exitPercent)) strategy.entry("Buy", strategy.long, qty = 2) strategy.exit("Exit1", "Buy", stop = stopLoss1, limit = takeProfit1, qty_percent = 50) strategy.exit("Exit2", "Buy", stop = stopLoss2, limit = takeProfit2) // Set `stopLoss1` and `takeProfit1` to `na` when price touches either. if low <= stopLoss1 or high >= takeProfit1 stopLoss1 := na takeProfit1 := na // Set `stopLoss2` and `takeProfit2` to `na` when price touches either. if low <= stopLoss2 or high >= takeProfit2 stopLoss2 := na takeProfit2 := na plot(stopLoss1, "SL1", color.red, style = plot.style_circles) plot(stopLoss2, "SL2", color.red, style = plot.style_circles) plot(takeProfit1, "TP1", color.green, style = plot.style_circles) plot(takeProfit2, "TP2", color.green, style = plot.style_circles)

Note that:

- We use persistent global variables for the take-profit and stop-loss levels so that we can plot them. Otherwise, declaring the variables in the first if block would be simpler.

Using `strategy.oca.reduce`

Creating exit orders as a group, using the strategy.oca.reduce type, ensures that when one exit order from the group is filled, the quantity of the remaining orders is reduced accordingly. This method is ideal in scripts that have an unequal number of take-profit levels to stops.

When using a group of orders whose OCA type is strategy.oca.reduce, we recommend ensuring that the total size of all exit orders, after any reductions, matches the size of the initial entry orders. Matching the sizes guarantees that the strategy closes the position entirely without leaving any part open or inadvertently opening a new position in the opposite direction.

The following example script uses two take-profit levels but only one stop level. All three sell orders have the same `oca_name`, which means they form a group. They have `oca_type = strategy.oca.reduce` set, so that filling one of the limit orders reduces the quantity of the remaining orders. The total quantity of the exit orders matches the entry order quantity, preventing the strategy from trading an excessive number of units and causing a reversal.

//@version=6 strategy("Multiple TP, one stop demo", overlay = true) int exitPercentInput = input.int(1, "Exit %", minval = 1, maxval = 99) float exitPercent = exitPercentInput / 100 var float stop = na var float limit1 = na

var float limit2 = na bool buyCondition = bar_index % 100 == 0 // Is `true` on every 100th bar. // Place orders when `buyCondition` is true and we are not in a position. if buyCondition and strategy.position_size == 0 stop := close * (1 - exitPercent) limit1 := close * (1 + exitPercent) limit2 := close * (1 + (2 * exitPercent)) strategy.entry("Long", strategy.long, 6) // All three sell orders use the "Bracket" OCA group; filling one order reduces the quantity of the remaining orders. strategy.order("Stop", strategy.short, stop = stop, qty = 6, oca_name = "Bracket", oca_type = strategy.oca.reduce) strategy.order("Limit 1", strategy.short, limit = limit1, qty = 3, oca_name = "Bracket", oca_type = strategy.oca.reduce) strategy.order("Limit 2", strategy.short, limit = limit2, qty = 6, oca_name = "Bracket", oca_type = strategy.oca.reduce) // Set `limit1` to `na` when price exceeds it. if high >= limit1 limit1 := na // Set `stop`, `limit11`, and `limit2` to `na` when price surpasses either the last take-profit, or the stop. if low <= stop or high >= limit2 stop := na, limit1 := na, limit2 := na plot(stop, "Stop", color.red, style = plot.style_linebr) plot(limit1, "Limit 1", color.green, style = plot.style_linebr) plot(limit2, "Limit 2", color.green, style = plot.style_linebr)

How can I execute a trade partway through a bar?

On historical bars, Pine scripts can access only a single set of open, high, low and close (OHLC) data per bar. Consequently, strategies are calculated once, at the close of each bar. This limitation means it's not possible to evaluate logical conditions that occur mid-bar, such as a price cross, on historical data.

Using `calc_on_every_tick`

Strategies running on realtime bars can simulate orders partway through a bar by enabling the calc_on_every_tick parameter. This setting allows the strategy to process each tick (incoming price update) and execute trades on the tick after a logical condition occurs.

> NoticeIn contrast to realtime bars, historical bars do not contain data for each incoming tick. Those bars contain only confirmed price data. Consequently, a strategy that enables calculation on every tick might repaint on elapsed realtime bars after reloading, because those bars become *historical* and no longer contain data for each tick before their close. Therefore, we recommend setting `calc_on_every_tick` to `false` while backtesting.

Using predefined prices

Stop or limit orders at predefined prices *can* execute orders partway through a bar, even when the strategy does not enable the `calc_on_every_tick` parameter. This method is effective on both realtime *and* historical data. Even though orders are processed on the close of historical bars, the broker emulator simulates an order fill at the predefined price level, if the broker determines that price has hit that level during the bar. For information about the assumptions that the broker emulator makes about price movements, see the Broker emulator section of the User Manual.

The following example script uses stop and limit orders to exit a trade partway through a bar. The script calls the strategy.exit() function with the `stop` and `limit` parameters, determining the specific price levels at which the trade will exit.

//@version=6 strategy("Predefined price exit demo", overlay = true) int exitPercentInput = input.int(1, "Exit %", minval = 1, maxval = 99) float exitPercent = exitPercentInput / 100 //@variable Is `true` on every 100th bar. bool buyCondition = bar_index % 100 == 0 var float stopLoss = na var float takeProfit = na // Place

orders when `buyCondition` is true and we are not in a position. if buyCondition and strategy.position_size == 0.0 stopLoss := close * (1 - exitPercent) takeProfit := close * (1 + exitPercent) strategy.entry("buy", strategy.long) strategy.exit("exit", "buy", stop = stopLoss, limit = takeProfit) // Set `stopLoss` and `takeProfit` to `na` when price touches either, i.e., when the strategy simulates an exit. if low <= stopLoss or high >= takeProfit stopLoss := na takeProfit := na plot(stopLoss, "SL", color.red, style = plot.style_linebr) plot(takeProfit, "TP", color.green, style = plot.style_linebr)

How can I exit a trade in the same bar as it opens?

Sometimes, traders want to enter and exit trades on the same bar. By default, if an exit condition occurs during the same bar that a trade is opened, the broker emulator closes the trade at the open of the *next* bar. To learn why this happens, refer to this FAQ entry.

To override this default behavior, either specify exit prices, or exit with a market order at the bar close.

Specifying exit prices

If the entry command also sets stop-loss or take-profit orders to trigger an exit when certain price levels are reached, then the trade can exit during the same bar that it opens.

In the following example script, the trade exits within the same bar *if* the price hits either of the defined profit or loss levels. Setting small profit and loss values increases the likelihood of triggering an exit within the entry bar, although the trade could hit those levels for the first time in a subsequent bar instead.

//@version=6 strategy("Exit on entry bar with specific price", overlay = true) int exitTickSizeInput = input.int(10, "Exit if price moves this many ticks", minval = 1) //@variable Is `true` on every 100th bar. bool buyCondition = bar_index % 10 == 0 // Place orders when `buyCondition` is true and we are not in a position. if buyCondition and strategy.position_size == 0.0 strategy.entry("buy", strategy.long) strategy.exit("exit", "buy", profit = exitTickSizeInput, loss = exitTickSizeInput)

Using a market order at bar close

Another method to exit a trade in the same bar that it opens is to use a market order at the bar's close, by setting the `immediately` argument to `true` in the strategy.close() function.

In the following example script, if the buy order is opened, the strategy closes the position at the end of the entry bar. Scripts can call the strategy.close() function conditionally within a local block if necessary. For simplicity, in this example we apply the command to every entry.

//@version=6 strategy("Exit on entry bar with market order", overlay = true) //@variable Is `true` on every 100th bar. bool buyCondition = bar_index % 10 == 0 // Place orders when `buyCondition` is true and we are not in a position. if buyCondition and strategy.position_size == 0.0 strategy.entry("buy", strategy.long) strategy.close("buy", immediately = true)

> NoticeThe `immediately` parameter operates in a similar way to process_orders_on_close, but it is specific to the strategy.close() and strategy.close_all() functions. The emulator calculates the close order using bar closing prices, but the same prices might not always be attainable in realtime trading. Additionally, this behavior can cause repainting.

Advanced order types and conditions

How can I set stop-loss and take-profit levels as a percentage from my entry point?

To set exit orders as a percentage from the entry price, the script needs the average entry price calculated by the broker emulator (which is affected by conditions including multiple entries and slippage). However, the built-in variable strategy.position_avg_price returns na until the close of the entry bar. This means that take-profit and stop-loss orders based on the entry price can only be placed during the *next* bar.

If programmers want strategies to be able to close trades on the same bar that they are opened, there are two workarounds, each of which have their own benefits and limitations: altering the emulator behavior and using a different, fixed value.

Using `calc_on_order_fills`

Setting the calc_on_order_fills argument of the strategy() declaration function to `true` recalculates the strategy immediately after simulating an order fill. This setting provides access to data such as the current average price of a position on an unconfirmed bar.

> NoticeEnabling `calc_on_order_fills` for some strategies might lead to unrealistic results on historical bars. During the extra script execution after an order fills, the script has access to the *confirmed* OHLC values for the historical bar, but those values would not be available in the real world until the bar's closing time. For an explanation of this form of *lookahead bias*, see this Help Center article.

The following example script sets take-profit and stop-loss orders during the entry bar, based on the entry price strategy.position_avg_price. The script uses the `calc_on_order_fills` setting to enable this behavior.

//@version=6 strategy("Exit demo using `calc_on_order_fills`", overlay = true, calc_on_order_fills = true) float stopSizeInput = input.float(1.0, "SL %", minval = 0.0) / 100.0 float profitSizeInput = input.float(1.0, "TP %", minval = 0.0) / 100.0 //@variable Is `true` on every 100th bar. bool buyCondition = bar_index % 100 == 0 //@variable Stop-loss price for exit commands. var float stopLoss = na //@variable Take-profit price for exit commands. var float takeProfit = na // Place orders when `buyCondition` is true and we are not in a position. if buyCondition and strategy.position_size == 0.0 strategy.entry("buy", strategy.long) // If we are in a position, set the exit orders. if strategy.position_size != 0.0 stopLoss := strategy.position_avg_price * (1.0 - stopSizeInput) takeProfit := strategy.position_avg_price * (1.0 + profitSizeInput) strategy.exit("exit", "buy", stop = stopLoss, limit = takeProfit) // Set `stopLoss` and `takeProfit` to `na` when price touches either, i.e., when the strategy simulates an exit. if low <= stopLoss or high >= takeProfit stopLoss := na takeProfit := na plot(stopLoss, "SL", color.red, style = plot.style_linebr) plot(takeProfit, "TP", color.green, style = plot.style_linebr)

Note that:

- If we change `calc_on_order_fills` to `false` in this script, the exit orders are placed on the bar *after* the entry bar, and can fill at very different levels depending on the movement of price.

Using predefined prices

The following example script calculates the stop and limit orders based on the *closing price* of the signal bar. The disadvantage of this approach is that the close price might not match the average opening price exactly. The advantage is that this method doesn't introduce potential *lookahead bias* like using `calc_on_order_fills`.

//@version=6 strategy("Exit demo using predefined prices", overlay = true) float stopSizeInput = input.float(1.0, "SL %", minval = 0.0) / 100.0 float profitSizeInput = input.float(1.0, "TP %", minval = 0.0) / 100.0 //@variable Is `true` on every 100th bar. bool buyCondition = bar_index % 100 == 0 //@variable Stop-loss price for exit commands. var float stopLoss = na //@variable Take-profit price for exit commands. var float takeProfit = na // Place orders when `buyCondition` is true and we are not in a position. if buyCondition and strategy.position_size == 0.0 stopLoss := close * (1.0 - stopSizeInput) takeProfit := close * (1.0 + profitSizeInput) strategy.entry("buy", strategy.long) strategy.exit("exit", "buy", stop = stopLoss, limit = takeProfit) // Set `stopLoss` and `takeProfit` to `na` when price touches either, i.e., when the strategy simulates an exit. if low <= stopLoss or high >= takeProfit stopLoss := na takeProfit := na plot(stopLoss, "SL", color.red, style = plot.style_linebr) plot(takeProfit, "TP", color.green, style = plot.style_linebr)

How do I move my stop-loss order to breakeven?

Moving a stop-loss order to breakeven can be a useful technique to manage risk.

The following example script sets a persistent `stopLoss` variable when the strategy enters a position. The script then updates the stop price to the entry price when the market price gets halfway to the take-profit level. The script calls the strategy.exit() function on every bar to ensure that the broker emulator receives any updates made to the `stopLoss` value. Lastly, it plots the average price according to the strategy.position_avg_price variable for reference.

//@version=6 strategy("Move stop to breakeven", overlay = true) float stopSizeInput = input.float(5.0, "SL %", minval = 0.0) / 100.0 float profitSizeInput = input.float(5.0, "TP %", minval = 0.0) / 100.0 float breakEvenInput = input.float(50, "BE %", minval = 0.0, maxval = 100) / 100.0 //@variable Is `true` on every 100th bar. bool buyCondition = bar_index % 100 == 0 //@variable Stop-loss price for exit commands. var float stopLoss = na //@variable Take-profit price for exit commands. var float takeProfit = na //@variable Price that, if breached, sets the stop to breakeven. var float breakEvenThreshold = na // Place orders when `buyCondition` is true and we are not in a position. if buyCondition and strategy.position_size == 0.0 stopLoss := close * (1.0 - stopSizeInput) takeProfit := close * (1.0 + profitSizeInput) // Set the breakeven threshold. breakEvenThreshold := close * (1.0 + profitSizeInput * breakEvenInput) strategy.entry("buy", strategy.long) // If the breakeven threshold is exceeded while in a position, set the stop to the entry price. if high >= breakEvenThreshold and strategy.position_size != 0 stopLoss := strategy.position_avg_price //@variable Is `true` on the bar on which a trade exits. bool isExitBar = strategy.closedtrades.exit_bar_index(strategy.closedtrades - 1) == bar_index //@variable Condition to determine when plots are displayed. bool showPlots = strategy.position_size != 0 or buyCondition or isExitBar // Plot the entry price, stop loss, take-profit, and the breakeven threshold. plot(strategy.position_avg_price, "BE", chart.fg_color, style = plot.style_linebr) plot(showPlots ? stopLoss : na, "SL", color.red, style = plot.style_linebr) plot(showPlots ? takeProfit : na, "TP", color.green, style = plot.style_linebr) plot(showPlots ? breakEvenThreshold : na, "TG", color.blue, style = plot.style_circles) // Place a bracket order using the `stopLoss` and `takeProfit` values. // We call it on every bar so that the stop level is updated when the breakeven threshold is exceeded. strategy.exit("exit", "buy", stop = stopLoss, limit = takeProfit)

Note that:

- This strategy uses strategy.position_avg_price as the breakeven level. However, the real breakeven price of a trade is affected by slippage and commission.

How do I place a trailing stop loss?

A trailing stop loss limits a trader's losses while allowing a position to remain open as long as the price moves favorably.

Strategies can create trailing stops either by using the built-in functionality of the strategy.exit() function or by creating custom trailing stop-loss logic.

Trailing stops set in the strategy.exit() use live price updates in realtime but assumed price movements for historical bars. These assumptions can cause repainting. This type of trailing stop is therefore potentially more responsive but less accurate.

Custom trailing stop values are typically updated at the close of each bar, and so do not capture realtime intrabar price movements with the same responsiveness. This delay helps to avoid repainting strategy results.

Using built-in trailing stop functionality

To set a trailing stop in the strategy.exit() function, specify both *when* the trail should activate and *how far* behind price it should trail.

Activation level

When price crosses this level, the trailing stop activates. The activation level can be set as a number of ticks past the entry price via the `trail_points` parameter, or as a price value via the `trail_price` parameter.

Trail offset

After it activates, the stop loss trails behind the bar's high or low price by this distance, defined in ticks using the `trail_offset` parameter.

In the following long-only example script, the strategy.exit() function uses the `trail_points` and `trail_offset` parameters to set a trailing stop. The stop-loss trails the high, minus the offset points, after it activates. The script creates and plots a separate `trailingStop` variable to visualize the trailing stop price that the function calculates internally, although this is not necessary for the trailing stop to function. We also set a separate stop-loss order to close trades that go too low before they trigger the trailing stop.

//@version=6 strategy("Trailing stop order demo", overlay = true) string TT_SO = "The trailing stop offset in ticks. Also used as the initial stop loss distance from the entry price." //@variable The activation level is this number of ticks above the entry price. int activationOffsetInput = input.int(1000, "Activation Level Offset (in ticks)") //@variable The trailing stop trails this many ticks below the high price. int stopOffsetInput = input.int(2000, "Stop Offset (in ticks)", tooltip = TT_SO) //@variable The price at which the trailing stop activates. float trailPriceActivationLevel = activationOffsetInput * syminfo.mintick + strategy.position_avg_price //@variable The price at which the trailing stop itself is located. var float trailingStop = na // Calculate a fast and slow Simple Moving Average. float ma1 = ta.sma(close, 14) float ma2 = ta.sma(close, 28) //@variable Is `true` when `ma1` crosses over `ma2` and we are not in a position. bool longCondition = ta.crossover(ma1, ma2) and strategy.position_size == 0 //@variable Is `true` on the bar that a trade exits. bool isExitBar = strategy.closedtrades.exit_bar_index(strategy.closedtrades - 1) ==

bar_index float exitPrice = strategy.closedtrades.exit_price(strategy.closedtrades - 1) // Generate a long market order when `longCondition` is `true`. // Set a static abd trailing stop loss. if longCondition strategy.entry("Long", strategy.long) strategy.exit("Stop", from_entry = "Long", trail_points = activationOffsetInput, trail_offset = stopOffsetInput, loss = stopOffsetInput) // If the high exceeds the activation level, set the `trailingStop` to whichever is higher: // the current high minus the price equivalent of `stopOffsetInput` or the previous `trailingStop` value. if high > trailPriceActivationLevel or isExitBar and exitPrice > trailingStop trailingStop := math.max(high - stopOffsetInput * syminfo.mintick, nz(trailingStop)) //@variable The price of the active stop price, using the trailing stop when activated, or a static stop loss otherwise. float stopLevel = na(trailingStop) ? strategy.position_avg_price - stopOffsetInput * syminfo.mintick : trailingStop // Visualize the movement of the trailing stop and the activation level. plot(stopLevel, "Stop Level", chart.fg_color, 2, plot.style_linebr) plot(trailPriceActivationLevel, "Activation level", color.aqua, 1, plot.style_linebr) // Display the two simple moving averages on the chart. plot(ma1, "MA 1", color.new(color.lime, 60)) plot(ma2, "MA 2", color.new(color.fuchsia, 60)) // Mark the point where the trailing stop is activated with a shape and text. plotshape(high > trailPriceActivationLevel and na(trailingStop)[1], "Trail Activated", shape.triangledown, size = size.small, color = color.aqua, text = "Trailing stop\nactivated", textcolor = color.aqua) // Set the trailing stop to `na` when not in a position. if strategy.position_size == 0 trailingStop := na

Coding a custom trailing stop

A custom trailing stop can use different activation conditions, and can trail in a different way, to the trailing stop built into the strategy.exit() function. To work correctly, a custom trailing stop must calculate the stop price on each bar that the stop is active, and call the strategy.exit() function on each bar to set the `stop` price.

The following example script triggers long and short trades based on crosses of two moving averages. A custom function calculates the trailing stop using the highest or lowest price from the last five bars, adjusted by an Average True Range (ATR) buffer. This method of distancing the stop by a measure of average price movement attempts to reduce premature stop triggers in volatile conditions.

//@version=6 strategy("ATR trailing stop demo", overlay = true) // Set the lookback period in bars to identify the highest or lowest point for trailing stop calculations. int SWING_LOOKBACK = 5 // @function Calculates a dynamic trailing stop by adjusting the highest // (bearish) or lowest (bullish) swing points over a set `length` // of bars using the ATR, for a stop distance proportional to average bar size. // @param calcStop (series bool) A condition that activates the trailing stop, e.g., being in a trade. // @param length (simple int) The number of bars to look back to determine the highest or lowest point for // the trailing stop calculation. // @param isLong (simple bool) Indicator of the trailing stop's orientation: true for long trades // (stop below price) and false for short trades (stop above price). // @param atrMultiplier (simple float) The multiplier applied to the ATR, adjusting the stop's distance from the // identified extreme price point. Optional. Default is 1.0, or 100% of the ATR value. // @returns (float) The trailing stop price, or `na` if `calcStop` is false. atrTrailingStop(series bool calcStop, simple int length, simple bool isLong, simple float atrMultiplier = 1.0) => var float trailPrice = na int m = isLong ? 1 : -1 float atr = ta.atr(14) * atrMultiplier float swingPoint = switch isLong => ta.lowest(length) - atr => ta.highest(length) + atr trailPrice := switch calcStop and not calcStop[1] => swingPoint calcStop[1] and not calcStop => na => math.max(trailPrice * m, swingPoint * m) * m // Calculate a fast and slow simple moving average. float ma1 = ta.sma(close, 14) float ma2 = ta.sma(close, 28) // Conditions for long/short entries on MA crossover/crossunder, if we are not in a position. bool longCondition = ta.crossover(ma1, ma2) and strategy.position_size == 0 bool shortCondition = ta.crossunder(ma1, ma2) and strategy.position_size == 0 // Determine when to calculate trailing stops for

long/short positions, based on entries and position. bool isExitBar = strategy.closedtrades.exit_bar_index(strategy.closedtrades - 1) == bar_index bool isLong = longCondition or strategy.position_size > 0 or isExitBar bool isBear = shortCondition or strategy.position_size < 0 or isExitBar // Use `atrTrailingStop()` to calculate trailing stops for both long and short positions. float longStop = atrTrailingStop(isLong, SWING_LOOKBACK, true) float shortStop = atrTrailingStop(isBear, SWING_LOOKBACK, false) // Place long entry order when `longCondition` occurs. if longCondition strategy.entry("long", strategy.long) // Place short entry order when `shortCondition` occurs. if shortCondition strategy.entry("short", strategy.short) // Create exit orders for long/short trades with ATR trailing stop, called on each bar to update to the latest price. strategy.exit("long exit", "long", stop = longStop) strategy.exit("short exit", "short", stop = shortStop) // Display the two simple moving averages and stop levels on the chart. plot(ma1, "MA 1", color.new(color.lime, 60)) plot(ma2, "MA 2", color.new(color.fuchsia, 60)) plot(isExitBar ? longStop[1] : longStop, "Long Stop", color.red, 2, plot.style_linebr) plot(isExitBar ? shortStop[1] : shortStop, "Short Stop", color.red, 2, plot.style_linebr)

Note that:

- Because strategies run once per bar, the trailing stop price in this example script updates at the close of each bar. During realtime bars the *previous* bar's stop value is used. This approach, while slightly delayed compared to using the built-in trailing stop described in the FAQ entry about how to place a trailing stop loss using built-in trailing stop functionality, ensures that the trailing stop price is not subject to assumptions about intrabar price movements, and thus avoids repainting.

How can I set a time-based condition to close out a position?

To close positions after a certain amount of time has passed, track the entry time for each trade and close the position using strategy.close() after the timeout.

Because strategies calculate at the close of each bar on historical data, time-based conditions can only be evaluated at the close, so **trade times are assessed in multiples of the chart bar's duration**. Further, if the timeout value is not divisible by the duration of a chart bar, each trade will last at least one additional chart bar. For instance, setting a timeout of 100 seconds on a 1-minute chart effectively means a minimum of two bars before a position can be closed.

In realtime, the same logic applies unless the strategy uses the calc_on_every_tick parameter, in which case the trade closes as soon as the first tick exceeds the timeout value. Remember that altering emulator behavior typically introduces repainting.

The following example script calculates the duration of each open trade by comparing the current time against the trade entry time. If a trade's duration exceeds the specified timeout, the script closes the trade and marks the event with a comment on the chart including the trade's duration in seconds.

//@version=6 strategy("Close position by timeout", overlay = true) // @function Automatically closes all positions that have been open for longer than a specified period. // @param timeoutInSeconds (int) The maximum allowed duration for an open trade, measured in seconds. // @returns (void) The function has no explicit return. closePositionsAfter(int timeoutInSeconds) => if strategy.opentrades > 0 for i = 0 to strategy.opentrades - 1 int timeNow = barstate.isrealtime ? timenow : time_close int tradeDurationInSeconds = (timeNow - strategy.opentrades.entry_time(i)) / 1000 if tradeDurationInSeconds >= timeoutInSeconds string entryName = strategy.opentrades.entry_id(i) string tradeComment = str.format("Close \"{0}\" by timeout {1}s", entryName, tradeDurationInSeconds) strategy.close(entryName, comment = tradeComment, immediately = true) // Create long and short conditions based on the crossover/under of 2 moving averages.

bool longCondition = ta.crossover(ta.sma(close, 14), ta.sma(close, 28)) bool shortCondition = ta.crossunder(ta.sma(close, 14), ta.sma(close, 28)) // Place long entry order upon `longCondition`. if longCondition strategy.entry("long", strategy.long) // Place short entry order upon `shortCondition`. if shortCondition strategy.entry("short", strategy.short) // Close positions after a configurable number of seconds. closePositionsAfter(input(1200, "Timeout (seconds)"))

Note that:

- The script uses either the time of the bar's close using the time_close variable, or the current time from the timenow variable (if the strategy uses the calc_on_every_tick parameter).
- The script uses the built-in functions strategy.opentrades.entry_time() and strategy.opentrades.entry_id() to measure trade duration and identify individual trades.
- The strategy.close() function uses the `immediately` argument to simulate trades at the end of the bar that exceeds the timer, rather than waiting for the opening of the next bar. Consequently, when a 120-second timeout is applied and the script runs on a 1-minute chart, it gives the appearance that trades last exactly two bars.

How can I configure a bracket order with a specific risk-to-reward (R:R) ratio?

To create a bracket order, define a stop-loss and a take-profit order using a single strategy.exit() call. To apply a specific risk-to-reward ratio, calculate the distance between the entry point and the stop-loss level. This stop distance represents the "risk". Then place the take-profit order a certain multiple of the stop distance away. The distance to the take-profit order represents the "reward", and the ratio between them is the risk:reward (R:R) ratio.

The following example script simulates long and short trades using inputs to define the stop distance in ticks and the R:R ratio. The `loss` parameter of the strategy.exit() function is simply the stop distance. The `profit` parameter is the stop distance multiplied by the R:R ratio. The script fills the areas between the entry and stop-loss points, and between the entry and take-profit points, to illustrate the risk and reward.

//@version=6 strategy("R:R demo", overlay = true) // Declare the stop size in ticks and the risk-to-reward ratio as inputs. int lossSizeInput = input.int(300, "Loss size (in ticks)", minval = 0) float riskRewardInput = input.float(2.0, "Risk/Reward multiple", minval = 0) // Create long and short entry conditions on MA crossover/crossunder, as long as we are not in a position. float ma1 = ta.sma(close, 14), float ma2 = ta.sma(close, 28) bool buyCondition = ta.crossover(ma1, ma2) and strategy.position_size == 0 bool sellCondition = ta.crossunder(ma1, ma2) and strategy.position_size == 0 // Place orders when `buyCondition` or `sellCondition` is true. if buyCondition strategy.entry("buy", strategy.long) if sellCondition strategy.entry("sell", strategy.short) // Define exit point for the entries based on a predefined loss size. // Calculate the profit target by multiplying the loss size with the user-defined risk-to-reward ratio. strategy.exit("exit", loss = lossSizeInput, profit = lossSizeInput * riskRewardInput) // Calculate the price equivalent of the profit and loss level. float tradeBias = math.sign(strategy.position_size) float stopLossPrice = strategy.position_avg_price - (tradeBias * lossSizeInput * syminfo.mintick) float takeProfitPrice = strategy.position_avg_price + (tradeBias * lossSizeInput * syminfo.mintick * riskRewardInput) // Plot the entry price, the stop price, and the price of the take-profit order. plotEntry = plot(strategy.position_avg_price, "Entry price", color.new(color.gray, 70), style = plot.style_linebr) plotStop = plot(stopLossPrice, "Stop-loss price", color.red, style = plot.style_linebr) plotTP = plot(takeProfitPrice, "Take-profit price", color.green, style = plot.style_linebr) // Highlight the R:R ratio by shading the area between the entry and the stop and the entry and the take-profit. fill(plotStop, plotEntry, color.new(color.red, 80)) fill(plotTP, plotEntry, color.new(color.green, 80))

How can I risk a fixed percentage of my equity per trade?

Adjusting the position size to risk a fixed percentage of equity normalizes risk exposure, regardless of equity fluctuations, and helps avoid disproportionate risks across a strategy's trading history.

Calculate the position size so that as the stop distance increases, the position size decreases, and vice-versa, to maintain a constant risk percentage:

1. Calculate monetary *risk per contract* by multiplying the stop distance in ticks by the monetary value of each tick (syminfo.mintick) and by the number of units each contract represents (syminfo.pointvalue).
2. Determine *risk amount* by multiplying the current equity (strategy.equity) by the percentage of equity that you want to risk.
3. Calculate position size by dividing the *risk amount* by the *risk per contract*.

> TipSmaller stop distances require larger position sizes to achieve a specific fixed risk. In some cases, the strategy might require increased *leverage* to achieve the required sizes. To prevent the strategy from entering trades with increased leverage, set the strategy's *margin* requirements to 100% by passing a value of 100 (default) to the `margin_long` and `margin_short` parameters of the strategy() declaration statement. Alternatively, set the "Margin for long/short positions" inputs to 100 in the script's "Settings/Properties" tab. To learn more about leverage and margin in strategies, see this Help Center article.

The following example script uses moving average crosses to generate long and short orders. The stop distance, risk:reward ratio, and percentage of equity to risk are all configurable via inputs. The script plots the current equity, the current value of a new position, and the percentage change in equity to the Data Window. Note that the actual exposure level can be less than intended if the available capital does not divide neatly by the unit value, particularly with small equity amounts, high unit prices, or assets such as stocks where trading partial shares is not possible.

Additionally, we display lines on the chart for the current total equity (in green) and the value of a position needed for the specified risk exposure at the current price (in blue). If the position value exceeds the total equity, the strategy requires leverage to achieve the required exposure, and the script colors the background red and displays the minimum leverage ratio needed in the data window.

//@version=6 strategy("Fixed risk", overlay = false, initial_capital = 100000) // Specify the desired stop distance (in ticks), the trade R:R ratio, and the percentage of equity to risk. int lossSizeInput = input.int(300, "Loss size (in ticks)", minval = 0) float riskRewardInput = input.float(2.0, "Risk/Reward multiple", minval = 0) float pctRiskInput = input.float(1.0, "% of equity to risk") / 100 // Create conditions for long/short entries on MA crossover/crossunder, if we are not in a position. float ma1 = ta.sma(close, 14), float ma2 = ta.sma(close, 28) bool buyCondition = ta.crossover(ma1, ma2) and strategy.position_size == 0 bool sellCondition = ta.crossunder(ma1, ma2) and strategy.position_size == 0 // Store the equity value at each trade entry, in order to calculate the percent change in equity. var float equityAtEntry = 0.0 // Calculate the risk per contract of the instrument. float riskPerContract = lossSizeInput * syminfo.mintick * syminfo.pointvalue // Calculate the amount of equity to risk. float equityToRisk = strategy.equity * pctRiskInput // Determine the position size necessary to risk the specified percentage of the equity. float positionSize = equityToRisk / riskPerContract // Place orders when `buyCondition` or `sellCondition` is true. if

buyCondition strategy.entry("buy", strategy.long, positionSize) equityAtEntry := strategy.equity // Set the `equityAtEntry` variable to the current equity on each entry. if sellCondition strategy.entry("sell", strategy.short, positionSize) equityAtEntry := strategy.equity // Stop-loss level is from the user input. Profit target is the multiple of the loss size with the risk-to-reward ratio. strategy.exit("exit", loss = lossSizeInput, profit = lossSizeInput * riskRewardInput) // Calculate the percent equity change between the current equity and the equity at entry. // On the exit bar of each trade, this value can be used to verify the percentage of equity risked. float equityChgPercent = 100 * (strategy.equity - equityAtEntry) / equityAtEntry color equityChgColor = equityChgPercent < 0 ? color.red : color.green, // Display current equity and current value of a new position on the chart, and % change in equity to the Data Window. plot(strategy.equity, "Current Total Equity", color.green, 2, display = display.all - display.pane) plot(positionSize * close, "Value of New Position at Current Price", color.aqua, 2, display = display.all - display.pane) plot(equityChgPercent, "% Change in Equity per Trade", equityChgColor, display = display.data_window, format = format.percent) // Color the background red if the calculated risk value exceeds the available equity (leverage required). bgcolor(strategy.equity < positionSize * close ? color.new(color.red, 80) : na) // Plot the minimum leverage multiple required to open the position, applicable only if leverage is necessary. plot(strategy.equity < positionSize * close ? positionSize * close / strategy.equity : na, "Leverage multiple required", display = display.data_window)

Note that:

- The stop distance in our example script is set to a constant value for demonstration purposes. In practice, the stop distance normally varies for each trade.

Strategy optimization and testing

Why did my trade results change dramatically overnight?

Strategy results can vary over time depending on where the historical data starts. The starting point of the data set aligns with the start of the nearest day, week, month or year, depending on the chart timeframe. Additionally, different TradingView plans provide access to varying amounts of historical bars. Refer to the User Manual entry on starting points for a discussion of these factors.

For strategies, this means the historical results seen today might change as the dataset's starting point moves. This can lead to a natural repainting of strategy results over time. To reduce the effect of these changes on backtesting, follow these tips:

Export strategy results

Regularly exporting strategy results to a file maintains a record of performance over time despite changes in historical data. Use the "Download data as XLSX" option from the context menu in the strategy report to export all the available report data. Alternatively, use the "Download" button in the upper-right corner of the panel's "Trades" tab to export only the list of trades data as a comma-separated values (CSV) file.

Use Deep Backtesting

Users with Premium and higher plans have access to the Deep Backtesting feature, which provides results from the entire available dataset of a symbol. Deep Backtesting results are displayed in the strategy report but are not visible on the chart.

Use Bar Replay

Use the Bar Replay feature on the first chart bar to extend the dataset backward, allowing a strategy to run on an additional full dataset prior to the current range. This process can be repeated a few times to analyze multiple datasets.

Why is backtesting on Heikin Ashi and other non-standard charts not recommended?

Non-standard charts like Heikin Ashi, Renko, Line Break, Kagi, Point & Figure, and Range Charts offer unique perspectives on price action. However, these chart types are not suited for strategy backtesting or automated trading systems execution, because the prices and time intervals do not match market prices and times.

Renko, Line Break, Kagi, Point & Figure, and Range Charts simplify price action, losing some price detail. Heikin Ashi charts calculate synthetic prices for each bar's open, high, low, and close (OHLC) values based on averages.

Further, all non-standard chart types with the exception of Heikin Ashi charts form new price units based on price movement only and omit the element of time.

Both the distortion of price data and the omission of time in non-standard charts lead to unrealistic and potentially misleading backtesting results.

Programmers can specify the `fill_orders_on_standard_ohlc` parameter of the strategy() declaration, which causes the strategy to calculate on standard chart data even if the current view is of Heikin Ashi candles. The user can do the same thing by by enabling the "Fill orders on standard OHLC" option in the strategy's properties. This option has no effect on other non-standard chart types, because they use non-standard time as well as price.

For a more detailed analysis of how non-standard chart types affect strategy results, refer to this script from the PineCoders account.

How can I backtest deeper into history?

Different TradingView plans give access to different amounts of historical information. To conduct more comprehensive backtesting in Pine Script, exploring further into an asset's historical data, use Bar Replay or Deep Backtesting.

Bar Replay

Starting the Bar Replay from the first chart bar in history effectively rolls back the dataset to an earlier point in time. Each iteration of the bar replay extends the dataset further back, offering analysis of multiple historical datasets. However, there is a limit to the number of times this process can be repeated. This method has the added benefit of visualizing the strategy's performance directly on the chart, which can be insightful for understanding trade entries, exits, and behavior during specific historical market conditions.

Deep Backtesting

For TradingView users with Premium and higher plans, the Deep Backtesting mode can execute the strategy across *all* historical data available for the selected symbol. The results from Deep Backtesting mode are displayed only in the strategy report, and the trade markers from using this mode are not visible on the chart; the trades shown on the chart are always calculated without Deep Backtesting. The strategy report

results from Deep Backtesting mode might be different from the results in regular mode for the same chart and strategy, as explained in this Help Center article.

How can I backtest multiple symbols?

Each Pine Script strategy runs on one dataset at a time. To evaluate a strategy across various markets or instruments:

- Apply the strategy to the chart and then switch the chart to the desired symbol.
- Use TradingView's watchlist feature to organize and quickly access different symbols.
- Export the results from the strategy report and use external tools such as spreadsheet software to compare the performance of a strategy on different symbols.

Advanced features and integration

Can my strategy script place orders with TradingView brokers?

Pine Script strategies and indicators cannot directly place orders on exchanges. Traders can use external tools or platforms that can interpret alert signals from Pine scripts using webhooks and execute trades accordingly.

How can I add a time delay between orders?

Adding a time delay between orders can help to prevent too many trades in a short time. Strategies can also prevent trading for a time after a series of losses. Here's how to set up a time delay between orders:

- Define the delay duration, whether in time units (minutes, hours, days) or a number of bars. For time-based delays, convert the chosen time unit into milliseconds, because Pine time variables use milliseconds.
- Check the time or bar_index of the last trade using strategy.closedtrades.exit_time() or strategy.closedtrades.exit_bar_index().
- If the difference between the current bar `time` or `bar_index` and that of the last trade's exit exceeds the delay duration, set a boolean flag to allow new orders. Make sure to include the flag in the strategy entry conditions.

The following example script provides two methods for delaying orders: a time-based delay or a specified number of bars. The strategy creates a long entry order when either the `time` of a bar or its `bar_index` exceeds the set delay from the last active trade bar. No other conditions are used for entry in this demonstration, but users can add their own logic to these conditions.

To keep the chart clean, the script calls the strategy.close() function to close active trades after they have been open for 10 bars. The script uses background shading, labels and arrows to illustrate the trade entries and exits.

//@version=6 strategy("Time-delayed orders", overlay=true, max_labels_count = 500, max_lines_count = 500) import PineCoders/Time/4 as PCtime // Constants string TU1 = "seconds", string TU2 = "minutes", string TU3 = "hours", string TU4 = "days" string TU5 = "weeks", string TU6 = "months", string DT1 = "bars", string DT2 = "time" // Tooltips for inputs string D_TT = "Delay orders for a specific number of bars or a specific duration of time since the last trade." string N_TT = "Specify the number of bars or time units for the delay." string U_TT = "Unit of time; relevant only if the delay type is 'time'." // User inputs for delay type,

number of units/bars, and time units. string delayTypeInput = input.string(DT2, "Delay type", tooltip = D_TT, options = [DT1, DT2]) int nInput = input.int(15, "Number of bars or time units", tooltip = N_TT) string unitsInput = input.string(TU2, "Time units", tooltip = U_TT, options = [TU1, TU2, TU3, TU4, TU5]) // Convert the time unit string input to a value in milliseconds for use in the time delay calculation. int mult = switch unitsInput TU1 => 1000 TU2 => 60000 TU3 => 3600000 TU4 => 86400000 TU5 => 604800000 => 2628003000 bool useTimeDelay = delayTypeInput == DT2 // Use time delay or not. int timeOfExit = strategy.closedtrades.exit_time(strategy.closedtrades - 1) // Time of last trade exit. int barOfExit = strategy.closedtrades.exit_bar_index(strategy.closedtrades - 1) // Bar index of last trade exit. int timeSinceExit = time - timeOfExit // Calculate the time since the last trade. int barsSinceExit = bar_index - barOfExit // Calculate the number of bars since the last trade. bool timeAllowed = (timeSinceExit >= nInput * mult or na(timeOfExit)) and useTimeDelay bool barAllowed = (bar_index - barOfExit >= nInput or na(barOfExit)) and not useTimeDelay // Allow entry of a trade if the delay has passed and we're not in a position. bool entryCondition = (timeAllowed or barAllowed) and strategy.position_size == 0 bool tradeExited = barOfExit == bar_index // Did the trade exit on the current bar? if entryCondition // Enter the trade if conditions allow. strategy.entry("Long", strategy.long) // Set label text: format time or show bar count since last trade. string labelTxt = useTimeDelay ? PCtime.formattedNoOfPeriods(timeSinceExit, unitsInput) : str.format("{0} bars", barsSinceExit) label.new(bar_index, low, labelTxt, color = color.new(color.lime, 80), textcolor = color.lime, style = label.style_label_up) line.new(timeOfExit, low, time, low, xloc.bar_time, color = color.new(color.lime, 50), style = line.style_arrow_left, width = 2) if bar_index % 10 == 0 // Close any open position on every tenth bar. strategy.close("Long") bgcolor(entryCondition ? color.new(color.lime, 85) : tradeExited ? color.new(color.fuchsia, 85) : na)

Consider the following limitations when adding time-based delays.

Historical bars

Strategies calculate at the close of each bar, so they can only evaluate time-based conditions at that moment. This constraint entails that on historical bars, **delay times are assessed in increments equal to the chart bar's duration**.

Session times

Strategies cannot evaluate delays when the market is closed, because there are no price updates to trigger script execution. This means that if a delay extends beyond the end of a trading session, the delay condition cannot be identified until the script runs again on the next session, resulting in a longer-than-anticipated time between orders.

Delay duration on different timeframes

If the delay value is not divisible by the duration of a chart bar, each delay lasts at least one additional chart bar. For instance, setting a delay of 100 seconds on a 1-minute chart effectively means a minimum of two bars before the delay is exceeded.

How can I calculate custom statistics in a strategy?

To track metrics other than the default metrics that the strategy report tracks, strategies can calculate custom statistics. These calculations might need to detect order executions, track closed trades, monitor entries into trades, and assess whether a trade is active. Changes in built-in variables such as strategy.opentrades and strategy.closedtrades can track the execution of orders.

The following example script uses a moving average crossover strategy to generate orders. It calculates custom metrics, including the price risk at entry, average position size, and the average percentage of bars involved in trades across the dataset, and plots the custom metrics and some built-in variables to the Data Window. Users can view the history of values plotted in the Data Window by moving the cursor over any bar. In contrast, the strategy report summarizes data over the entire testing period.

```
//@version=6 strategy("Custom strategy metrics", "", true, initial_capital = 10000, commission_type = strategy.commission.percent, commission_value = 0.075, max_bars_back = 1000, default_qty_type = strategy.percent_of_equity, default_qty_value = 100) // Calculate entry conditions. float c = math.round_to_mintick(close) // Round OHLC to chart prices. float maF = math.round_to_mintick(ta.sma(hlc3, 10)), float maS = math.round_to_mintick(ta.sma(hlc3, 60)) bool enterLong = ta.crossover(maF, maS), bool enterShort = ta.crossunder(maF, maS) // Entry conditions. float stopLong = ta.lowest(20)[1], float stopShort = ta.highest(20)[1] // Stop-loss order levels. // Enter a new position or reverse, unless stop could not be calculated yet. if enterLong and not na(stopLong) strategy.entry("Long", strategy.long, comment = "►Long") if enterShort and not na(stopShort) strategy.entry("Short", strategy.short, comment = "►Short") // Modify existing exit orders using the current stop value. strategy.exit("◄Long", "Long", stop = stopLong), strategy.exit("◄Short", "Short", stop = stopShort) // Generate custom statistics. float riskOnEntry = math.abs(c - (enterLong ? stopLong : enterShort ? stopShort : na)) // Trade risk at entry. int changeInClosedTrades = ta.change(strategy.closedtrades) int changeInOpenTrades = ta.change(strategy.opentrades) bool tradeWasClosed = changeInClosedTrades != 0 bool tradeWasEntered = changeInOpenTrades > 0 or (strategy.opentrades == strategy.opentrades[1] and tradeWasClosed) or changeInClosedTrades > 1 bool tradeIsActive = strategy.opentrades != 0 // Check if a trade is currently active. float barsInTradePct = 100 * ta.cum(tradeIsActive ? 1 : 0) / bar_index // Percentage of bars on which a trade was open. float tradesEntered = ta.cum(tradeWasEntered ? 1 : 0) float positionSize = math.abs(strategy.position_size) float avgPositionSize = ta.cum(nz(positionSize))[1] / tradesEntered // Calculate average position size. float positionValue = positionSize * close // Position monetary value float priceRiskPct = riskOnEntry / close // Risk percentage of trade relative to entry price. float tradeRiskPct = positionSize * riskOnEntry // Monetary risk of the trade. float stop = strategy.position_size > 0 ? stopLong : strategy.position_size < 0 ? stopShort : na // Plot the MAs, stop price, and markers for entries and exits to the chart. plot(maF,"MA Fast"), plot(maS, "MA Slow", color.silver), plot(stop, "Stop", color.fuchsia, 1, plot.style_circles) plotchar(tradeWasClosed, "tradeWasClosed", "—", location.bottom, color.fuchsia, size = size.tiny) plotchar(tradeWasEntered, "tradeWasEntered", "+", location.top, color.lime, size = size.tiny) // Highlight the background while long and short positions are active. bgcolor(strategy.position_size > 0 ? color.new(color.teal, 80) : strategy.position_size < 0 ? color.new(color.maroon, 80) : na) // Plot statistics to the Data Window. plot(na, "═══════ Built-ins", display = display.data_window) plot(strategy.opentrades, "strategy.opentrades", display = display.data_window) plot(strategy.closedtrades, "strategy.closedtrades", display = display.data_window) plot(strategy.position_size, "strategy.position_size", display = display.data_window) plot(strategy.equity, "Equity", display = display.data_window) plot(na, "═══════ Custom Metrics", display = display.data_window) plot(riskOnEntry, "Risk On Entry", display = display.data_window) plot(positionSize, "Position Size", display = display.data_window) plot(tradesEntered, "tradesEntered", display = display.data_window) plot(barsInTradePct, "barsInTradePct", display = display.data_window) plot(avgPositionSize, "avgPositionSize", display = display.data_window) plot(positionValue, "Position Value", display = display.data_window) plot(priceRiskPct, "Price Risk %", display = display.data_window) plot(tradeRiskPct, "Trade Risk Value", display = display.data_window)
```

Note that:

- The strategy incorporates trading costs. Failing to account for these costs can lead to an unrealistic perception of strategy performance and diminish the credibility of test results.
- We round the open, high, low and close (OHLC) built-in variables to the symbol's precision. This rounding ensures that any statistics the script calculates align within the strategy report and with strategy order-related built-in variables.
- The script creates global variables `changeInOpenTrades` and `changeInClosedTrades` for the changes in built-in variables for open and closed trades so that it calls the ta.change() function on every bar for consistency. See the Storing and using data from previous bars section of the Execution model page to learn more.

How do I incorporate leverage into my strategy?

Trading with *leverage* means borrowing capital from a broker to control larger position sizes than the amount of capital risked. This amplifies both potential profits and losses, making it a powerful but risky tool. The amount of the trader's capital that they risk is called the *margin*.

For example, setting a 20% margin ratio means that the trader's balance funds only 20% of the position's value, allowing positions up to five times the account balance. A margin ratio of 20% is therefore the same as 5:1 leverage. With an available balance of $10,000 and a 20% margin setting, a strategy can open positions up to $50,000 in value.

Pine Script strategies can simulate trading with leverage by specifying margin requirements for long and short positions. Users can adjust the "Margin for long positions" and Margin for short positions" in the strategy's "Properties" tab. Programmers can set the default margin in the script using the `margin_long` and `margin_short` parameters in the strategy() declaration function.

> NoticeIf a leveraged trade, or even a short trade with 1:1 leverage, incurs significant losses that cause the strategy's account balance to drop below the required margin, the broker emulator initiates a *margin call* event by liquidating *four times* the amount required to cover the loss. This behavior helps prevent constant margin calls on subsequent bars.

For more information on using leverage in strategies, see the Help Center article How do I simulate trading with leverage?

Can you hedge in a Pine Script strategy?

When traders offset the risk of one position by opening another position at the same time, this is called *hedging*.

The main ways to hedge an open position are:

- By opening a second position in a related asset that is expected to move in the opposite direction to the first asset.
- By opening a short position to offset a long position or vice-versa.
- By using derivatives such as options.

Strategies cannot use these methods, because Pine strategies can only have positions open in one direction at a time, either long or short. Pine strategies run on only the chart asset and cannot open positions in

different assets.

Can I connect my strategies to my paper trading account?

Pine Script does not support placing orders using the brokers integrated via the Trading Panel, or using TradingView's built-in paper trading account. The strategy report closely mimics a paper trading account by simulating orders and tracking theoretical positions and capital in a risk-free environment.

Strategies can customize order fill alerts to include detailed results and performance metrics in the alert strings, providing a record of the strategy's theoretical fills and overall performance in realtime.

> TipWhen configuring alerts for forward testing, it is often helpful to restrict the strategy's logic to remove the effects of historical trades by using a date filter set to today's date.

Troubleshooting and specific issues

Why are no trades executed after I add the strategy to the chart?

The strategy report, which displays a strategy's simulated trading results, becomes available when at least one strategy script is *active* (loaded and visible) on the chart. If a strategy that is running on the chart does not place any orders, the panel displays a message to inform the user that no trade data is available.

If a valid script that uses the strategy() declaration statement is running but is not placing any orders, consider the following potential problems and their solutions:

Lack of order placement commands

The strategy must use either the strategy.order() or strategy.entry() order placement commands to place orders. Add log.info() messages and review the Pine Logs to check whether the conditions to run those commands are met.

Insufficient capital

Verify that the strategy has enough initial capital to cover the position sizes it attempts to open. Remember, the cost of entering a futures contract position is the chart price multiplied by the syminfo.pointvalue, which can be significantly greater than the chart price. For a quick fix, increase the initial capital to a very high value in the strategy's "Settings/Properties" tab.

Runtime errors

Check for runtime errors indicated by a red exclamation mark on the chart pane next to the script's title. Resolve any issues by correcting the script as necessary.

By contrast, if a strategy produces too few trades, or executes over too short a duration, there might not be enough information to populate the full strategy report. In that case, the strategy report panel can generate a portion of its results, but some report sections might display no results for certain metrics or show the message "Not enough data to display" in place of certain visuals. To address this problem, adjust the chosen dataset, testing range, and strategy configuration to ensure that the strategy produces a reasonable number of simulated trades to assess its hypothetical performance (ideally 100 trades or more).

For more detailed guidance on this topic and troubleshooting tips, refer to the Help Center article I've successfully added a strategy to my chart, but it doesn't generate orders.

Why does my strategy not place any orders on recent bars?

If a strategy places one or more orders early in the testing range but then stops placing orders, check the following issues:

Total account loss

Check whether the simulated account balance experienced a total loss of equity earlier in the available history. As a result, the account might lack sufficient capital to continue trading the symbol and fail to show trades only in the chart's recent history.

No exit condition

Some programmers define entry conditions that rely on having no positions currently open. Make sure to explicitly close trades by specifying corresponding exit conditions for all trades. Without explicit instructions to close an open position using strategy.close() or strategy.exit() commands, the strategy might display only a single entry order early in the chart's history and in the strategy report's "Trades" tab. If trades are not closed, they do not generate results in the report's "Metrics" tab.

Why is my strategy repainting?

Pine scripts *repaint* if they behave differently on historical and realtime bars. If strategies repaint, their backtesting results are not reliable because they do not accurately represent the strategy's behavior in realtime.

Some strategy properties cause repainting:

- The `calc_on_every_tick` setting causes a strategy to recalculate with every price update, which may cause orders and alerts to trigger during the formation of a bar in realtime. By contrast, on historical bars, calculations are performed at the close of the bar.
- The `calc_on_order_fills` setting causes a strategy to recalculate immediately after simulating an order fill. For example, this feature is particularly useful for strategies that rely on entry prices to set exit prices on the entry bar, rather than waiting for the bar to close, such as the first example script in the FAQ entry How can I set stop-loss and take-profit levels as a percentage from my entry point using `calc_on_order_fills`? However, using this setting can introduce *lookahead bias* into the strategy, leading to potentially unrealistic outcomes. For instance, if a strategy's entry conditions are met within a bar that also triggers an exit, the strategy would execute an entry order within the same bar on the next tick. On historical bars, such entries could occur at any of the bar's open, high, low, or close (OHLC) prices, resulting in entry prices that are unrealistically favorable.
- Since strategies and their alerts execute at the close of a historical bar, the next possible moment for an entry order to be filled is the beginning of the next bar. However, the `process_orders_on_close` setting causes the strategy to use the close price of the bar where the condition is met for its order prices instead. See the FAQ entry Why are my orders executed on the bar following my triggers? for more information.

To avoid repainting, set the `calc_on_every_tick`, `calc_on_order_fills`, and `process_orders_on_close` parameters to `false` in the strategy() declaration statement.

Additionally, using unfixed data from a higher timeframe can cause repainting. If the data from the higher timeframe changes during the higher timeframe bar, this can change the script's oputput for historical bars. Ensure that strategies use only fixed values from a higher timeframe, as described in the Avoiding repainting section of the Other timeframes and data page.

Although these are the most common causes of repainting in strategies, they are not the only causes. For additional information, refer to the Repainting page in the User Manual.

How do I turn off alerts for stop loss and take profit orders?

In automated trading strategies, it is common practice to set stop-loss and take-profit orders at the same time as an entry order, using the alert from the entry order as a trigger. In this case, sending alerts for the stop-loss and take-profit order fills can be unnecessary or even problematic. To disable alerts for a specific order placement command, set the `disable_alert` parameter to `true`. The broker emulator still simulates the fills for these orders, but sends no alerts for them.

Here is an example of an order fill command with this parameter set:

```
strategy.exit("Exit", stop = stopLevel, limit = limitLevel, disable_alert = true)
```

Strings and formatting

How can I place text on the chart?

Scripts can display text using the following methods:

- The plotchar() or plotshape() functions for static text, which doesn't change.
- Labels and boxes for dynamic text, which can vary bar to bar.
- Tables for more complex text (static or dynamic) that stays in the same region of the chart.

Plotting text

The plotchar() and plotshape() functions can display fixed text on bars:

- A single plotchar() function call can print a string using the `text` parameter, but only one character using the `char` parameter. To plot only the text and not the character, set `char` to "".
- A plotshape() function call can print a string using the `text` parameter. To plot only the text and not the shape, set the `color` (for the shape) to na and the `textcolor` to something visible.

Plots appears on the bar where the script calls the function, by default, but scripts can offset a plot by a dynamic number of bars to the left or right. On the Y axis, the plots appear above/below the bar, at the top/bottom of the chart, or at an arbitrary price level. Scripts can call a plotchar() or plotshape() function on any number of bars and it counts as a single plot towards the plot limit.

When using these functions, the text cannot change during the execution of the script. The `text` parameter accepts an argument of type "const string", which means it cannot cannot change from bar to bar and cannot be supplied by an input.

This script, for example, does not compile, because the argument to the `text` parameter is a "series string":

//@version=6 indicator("Plotting text demo: incorrect", overlay = true) float rsi = ta.rsi(close, 14) bool rsiUp = ta.crossover(rsi, 50) bool rsiDn = ta.crossunder(rsi, 50) string txt = rsiUp ? "RSI\nUp" : rsiDn ? "RSI\nDown" : "" plotchar(series = rsiUp or rsiDn, title = "Up/Down", char = "R", text = txt, location = location.top, size = size.tiny)

To print different text depending on a logical condition, use two function calls and control them using the `series` parameter. Note that even if the `series` for one or both of the function calls is never true during a script's execution, and so no shape, character or text is ever plotted, *both* functions still count towards the plot limit.

The following script corrects the earlier example, and shows the use of both plotchar() and plotshape() to display text:

//@version=6 indicator("Plotting text demo", overlay = true) float rsi = ta.rsi(close, 14) bool rsiUp = ta.crossover(rsi, 50) bool rsiDn = ta.crossunder(rsi, 50) plotchar(series = rsiUp, title = "Up", char = "▲", location = location.belowbar, color = color.lime, text = "RSI\nUp", size = size.tiny) plotshape(series = rsiDn, title = "Down", style = shape.triangledown, location = location.abovebar, color = color.fuchsia, text = "RSI\nDown", size = size.tiny, textcolor = color.fuchsia)

Labels

Labels are particularly useful for displaying text that can change from one bar to another. The `text` parameter of the label.new() function takes a “series string”, so it can change whenever necessary.

Labels do not count towards the plot limit, but there is a separate limit of how many labels can display on the chart. By default, up to approximately 50 of the *most recent* labels appear on the chart. Programmers can adjust this limit up to 500 by setting the `max_labels_count` parameter in the indicator() or strategy() functions.

The parameters to the label.new() function for text, color, etc., take “series” arguments. This makes labels much more flexible than plots.

The following example script displays the same information as the previous script, but using labels. The background to the labels is transparent (set to na) in this example, to more closely match the style of the previous scripts.

//@version=6 indicator("Drawing labels demo", "", true) float rsi = ta.rsi(close, 14) bool rsiUp = ta.crossover(rsi, 50) bool rsiDn = ta.crossunder(rsi, 50) if rsiUp or rsiDn string labelText = rsiUp ? "▲\nRSI Up" : "RSI Down\n▼" color textColor = rsiUp ? color.lime : color.fuchsia string labelPos = rsiUp ? yloc.belowbar : yloc.abovebar label.new(bar_index, na, labelText, yloc = labelPos, color = color(na), textcolor = textColor)

As well as showing historical information, labels can also be used to show only the latest information on the current bar. The following example script displays the value of RSI in a different color depending on whether it is above or below 50, for the most recent bar only. This is not possible using plotchar() or plotshape(), because the text is fixed, and too many plots would be required to plot every value separately.

//@version=6 indicator("Single label demo", "", true) float rsi = ta.rsi(close, 14) bool rsiAbove50 = rsi >= 50 bool rsiBelow50 = rsi < 50 var label rsiLabel = label.new(na, na, style = label.style_label_left, yloc = yloc.price, color = color.new(color.gray,70)) if barstate.islast color textColor = rsiAbove50 ? color.lime : rsiBelow50 ? color.fuchsia : color(na) rsiLabel.set_x(bar_index + 1) rsiLabel.set_y(open) rsiLabel.set_text(str.format("RSI: {0, number, #.##}", rsi)) rsiLabel.set_textcolor(textColor)

Note that:

- We create the label once, on the first bar, with all its unchanging properties such as style and background color already set.
- We do nothing with the label for all historical bars.
- We update the changing properties of the label such as text and position on the most recent bar and on every realtime bar. This method is more performant than updating the label on all bars or creating and deleting it each bar.

Boxes

Boxes can also display text on the chart, by providing the text to the `text` parameter of the box.new() function. Boxes work with text in a similar way to labels, but with some additional features.

Labels exist specifically to display text – and so the label adjusts to the size of the text. Labels always resize so that all of the text is visible inside of the label.

The main use of boxes is to display the drawing itself. A box attaches to specific points on the chart, and its text might or might not fit into it. To ensure that the text displays in the best possible way, boxes provide some additional features that can not be used in labels: text wrapping and text alignment.

Text contained in the box can automatically wrap if it reaches the border of the box, if the `text_wrap` parameter is set to text.wrap_auto. Additionally, scripts can align the text inside the box along the vertical and horizontal axes. Using the `text_halign` and `text_valign` parameters of box.new(), text can display at one of the nine possible positions inside of the box.

In the example below, we draw a box that spans the last 50 historical bars on the chart, and a label. We add long text to both. With `text_wrap = text.wrap_auto`, the text inside the box automatically wraps to fit the box itself, while the text inside of the label stays unchanged:

//@version=6 indicator("Box and label text", overlay = true) if barstate.islastconfirmedhistory bt = "This long text is inside of a box, which means it is automatically wrapped and scaled to be visible in the constraints of the box." lt = "This long text is inside of a label, which means that it is displayed as is, and the label is simply drawn around it. It doesn't change when the chart is scaled." box.new(bar_index[50], close * 1.1, bar_index, close, text = bt, text_wrap = text.wrap_auto, text_size = 36) label.new(bar_index[25], close * 1.1, lt, size = 36)

Tables

Tables are useful to display information in a fixed position on the chart. Whereas plots and labels can easily show historical information because they are, or can be, linked to specific bars, table contents do not change as users move the cursor over past chart bars. This makes tables best suited for showing *current* information.

The following example script displays the value of RSI in a different color depending on whether it is above or below 50, for the most recent bar only.

//@version=6 indicator("RSI table", "", true) var table rsiDisplay = table.new(position.top_right, 1, 1, bgcolor = color.gray, frame_width = 2, frame_color = color.black) float rsi = ta.rsi(close, 14) bool rsiAbove50 = rsi >= 50 bool rsiBelow50 = rsi < 50 color textColor = rsiAbove50 ? color.lime : rsiBelow50 ? color.fuchsia : color(na) if barstate.isfirst table.cell(rsiDisplay, 0, 0, "") else if barstate.islast table.cell_set_text(rsiDisplay, 0, 0, str.format("RSI: {0, number, #.##}", rsi)) table.cell_set_text_color(rsiDisplay, 0, 0, textColor)

Note that:

- We create the table and its single cell only once, and update the text and color on the most recent bar and on every realtime bar, for performance.
- This script displays the same information as the preceding example did using a label.
- Although this is a simple example, for more complex information, tables are easier to organise and read than labels.

How can I position text on either side of a single bar?

Scripts can position a label to the *right* of a bar by using `style = label.style_label_left`. This style *points* the label to the **right** and *places* it to the **left**. Likewise, a label with `style = label.style_label_right` displays to the right of the bar, pointing left.

To manage the alignment of the text within the label, use the `textalign` parameter.

The following example script draws three labels on the chart's last bar, with different `style` and `textalign` values. User inputs control whether individual labels appear, and the central label is off by default for

readability. If the input to hide the background is enabled, the color is set to na so that it does not appear. Note that the proper way to do this is to cast it to a color by using `color(na)`.

//@version=6 indicator("Text position demo", "", true) hideBackgroundInput = input.bool(false, "Hide Background") color backgroundColor = hideBackgroundInput ? color(na) : color.new(color.gray,70) // @function Prints a label with the specified text at a specific position and alignment. // @param txt (string) The text to be displayed in the label. // @param pos (string) The label style. // @param align (string) The horizontal alignment of text within the label. // @returns (void) Function has no explicit return. print(string txt, string pos, string align) => var label lbl = label.new(na, na, na, xloc.bar_index, yloc.price, backgroundColor, pos, chart.fg_color, size.huge, align, text_font_family = font.family_monospace) label.set_xy(lbl, bar_index, high) label.set_text(lbl, txt) if input.bool(true, "Show Left Label") print("label_left\ntext.align_left", label.style_label_left, text.align_left) if input.bool(true, "Show Right Label") print("label_right\ntext.align_right", label.style_label_right, text.align_right) if input.bool(false, "Show Center Label") print("label_center\nalign_center", label.style_label_center, text.align_center)

How can I stack plotshape() text?

To make multiple text plots visible on the same bar, the text on one plot must be raised or lowered so that it does not overlap with another plot.

To add a blank line in a plotchar() or plotshape() call, add the newline character `\n`. Text above the bar or at the top of the chart can only be *raised*, by adding a newline *after* the text. Newlines added before the text are ignored. Likewise, text below the bar or at the bottom of the chart can only be *lowered*, by adding a newline *before* the text.

The following example script shows how to correctly stack text by inserting a blank line over or under other text:

//@version=6 indicator("Stack text demo", "", true) plotshape(true, "", shape.arrowup, location.abovebar, color.green, textcolor = color.green, text = "A") plotshape(true, "", shape.arrowup, location.abovebar, color.lime, textcolor = color.lime, text = "B\n") plotshape(true, "", shape.arrowdown, location.belowbar, color.red, textcolor = color.red, text = "C") plotshape(true, "", shape.arrowdown, location.belowbar, color.maroon, textcolor = color.maroon, text = "\nD")

How can I print a value at the top right of the chart?

Refer to the Placing a single value in a fixed position section of the Tables page. The example in that section uses a single-cell table to display a string representation of a value in the top-right corner of the chart.

How can I split a string into characters?

The str.split() function splits a string into parts and stores the parts in an array. To split a string into individual characters, use an empty string "" as the `separator` argument. Here is a code example:

//@version=6 indicator("Split a string into characters") string sourceStringInput = input.string("123456789", "String to Split") var array<string> charactersArray = str.split(sourceStringInput, "") if barstate.islast string txt = sourceStringInput + "\n" + str.tostring(charactersArray) var label = label.new(na, na, txt, xloc.bar_index, yloc.price, color(na), label.style_label_left, chart.fg_color, size.large, text.align_left) label.set_xy(label, bar_index, open)

Techniques

How can I prevent the “Bar index value of the `x` argument is too far from the current bar index. Try using `time` instead” and “Objects positioned using xloc.bar_index cannot be drawn further than X bars into the future” errors?

Both these errors occur when creating objects too distant from the current bar. An x point on a line, label, or box can not be more than 9999 bars in the past or more than 500 bars in the future relative to the bar on which the script draws it.

Scripts *can* draw objects beyond these limits, however, using xloc.bar_time instead of the `xloc` parameter, and time as an alternative to bar_index for the `x` arguments.

Note that, by default, all drawings use `xloc.bar_index`, which means that the values passed to their `x`-coordinates are treated as if they are bar indices. If drawings use a `time`-based value without specifying `xloc = xloc.bar_time`, the timestamp – which is usually an `int` value of trillions of milliseconds – is treated as an index of a bar in the future, and inevitably exceeds the 500 future bars limit. To use `time`-based values for drawings, always specify `xloc.bar_time`.

How can I update the right side of all lines or boxes?

Scripts can update the `x2` value of all lines or boxes by storing them in an array and using a for…in loop to iterate over each object. Update the `x2` value using the line.set_x2() or box.set_right() functions.

In the example below, we create a custom array and go over it to extend lines with each new bar:

```
//@version=6 indicator("Update x2 demo", "", true) int activeLevelsInput = input.int(10, "Number of levels") int pivotLegsInput = input.int(5, "Pivot length") // Save pivot prices. float pHi = ta.pivothigh(pivotLegsInput, pivotLegsInput) // Initialize an array for lines on the first bar, sized to match the number of levels to track. var array<line> pivotLines = array.new<line>(activeLevelsInput) // Check for a pivot. Add a new line to the array. Remove and delete the oldest line. if not na(pHi) line newPivotLine = line.new(bar_index[pivotLegsInput], pHi, bar_index, pHi) pivotLines.push(newPivotLine) pivotLines.shift().delete() // Update all line x2 values. if barstate.islast for eachLine in pivotLines eachLine.set_x2(bar_index)
```

As an alternative to adding new drawings to a custom array, scripts can use the appropriate built-in variable that collects all instances of a drawing type. These arrays use the `<drawingNamespace>.all` naming scheme: for example, scritps can access all drawn labels by referring to label.all, all polylines with polyline.all, etc. Scripts can iterate over these arrays in the same way as with custom arrays.

This example implements gets the same result using the line.all built-in array instead:

```
//@version=6 indicator("Update x2 demo", "", true) int activeLevelsInput = input.int(10, "Number of levels") int pivotLegsInput = input.int(5, "Pivot length") // Save pivot prices. float pHi = ta.pivothigh(pivotLegsInput, pivotLegsInput) // Check for a pivot. Delete the oldest line if the array is over the "Number of levels" limit. if not na(pHi) line newPivotLine = line.new(bar_index[pivotLegsInput], pHi, bar_index, pHi) if line.all.size() > activeLevelsInput line.all.first().delete() // Update all line x2 values. if barstate.islast for eachLine in line.all eachLine.set_x2(bar_index)
```

How to avoid repainting when *not* using the `request.security()` function?

Scripts can give deceptive output if they repaint by behaving differently on historical and elapsed realtime bars. This type of repainting is most commonly caused by requesting data from another context using the request.security() function.

Scripts can also change their output during a realtime bar, as the volume, close, high, and low values change. This form of repainting is not normally deceptive or detrimental.

To avoid this kind of repainting and ensure that outputs do not change during a bar, consider the following options:

- Use confirmed values, or the values from the previous bar.
- Set alerts to fire on bar close. Read more about repainting alerts in the FAQ entry Why is my alert firing at the wrong time?
- Use the open in calculations instead of the close.

For further exploration of these methods, see the PineCoders publication "How to avoid repainting when NOT using security()".

How can I trigger a condition n bars after it last occurred?

Using the ta.barssince() function, scripts can implement a condition when a certain number of bars have elapsed since the last occurrence of that condition.

The following example script uses the `cond` condition to plot a blue star when the close value is greater than the open value for two consecutive bars. Then, the `trigger` variable is true only if the `cond` condition is already true *and* the number of bars elapsed since `cond` was last true is greater than `lengthInput`. The script plots a red "O" on the chart, overlaying the blue star, each time these conditions are met. The Data Window displays the count since `cond` was last true.

//@version=6 indicator("`barssince` demo", overlay = true) int lengthInput = input.int(3, "Length") bool cond = close > open and close[1] > open[1] int count = ta.barssince(cond[1]) + 1 bool trigger = cond and count > lengthInput plot(cond ? 0 : count, "Count", display = display.data_window) plotchar(cond) plotchar(trigger, "", "O", color = color.red)

How can my script identify what chart type is active?

Various boolean built-in variables within the `chart.*` namespace enable a script to detect the type of chart it is running on.

The following example script defines a function, `chartTypeToString()`, which uses the `chart.*` built-ins to identify the chart type and convert this information into a string. It then displays the detected chart type in a table on the chart.

//@version=6 indicator("Chart type", "", true) chartTypeToString() => string result = switch chart.is_standard => "Standard" chart.is_heikinashi => "Heikin-Ashi" chart.is_kagi => "Kagi" chart.is_linebreak => "Line Break" chart.is_pnf => "Point and Figure" chart.is_range => "Range" chart.is_renko => "Renko" if barstate.islastconfirmedhistory var table display = table.new(position.bottom_right, 1, 1, bgcolor =

chart.fg_color) table.cell(display, 0, 0, str.format("Chart type: {0}", chartTypeToString()), text_color = chart.bg_color)

How can I plot the highest and lowest visible candle values?

To plot the highest high and lowest low within the range of visible bars, a script can use the chart.left_visible_bar_time and chart.right_visible_bar_time built-ins. These variables allow the script to identify the times of the earliest and latest visible bars on the chart and calculate the maximum or minimum values within that range.

The VisibleChart library by PineCoders offers such functionality with its `high()` and `low()` functions, which dynamically calculate the highest and lowest values of the currently visible bars.

The following example script uses functions from this library to create two horizontal lines on the chart, signifying the highest and lowest price points within the range of visible bars. The script draws labels for these lines, displaying both the price and the corresponding timestamp for each high and low point. As the chart is manipulated through scrolling or zooming, these lines and labels dynamically update to reflect the highest and lowest values of the newly visible bars:

//@version=6 indicator("Chart's visible high/low", "", true) import PineCoders/VisibleChart/5 as PCvc // Calculate the chart's visible high and low prices and their corresponding times. int x1 = PCvc.highBarTime() int x2 = PCvc.lowBarTime() float chartHi = PCvc.high() float chartLo = PCvc.low() // Draw lines and labels on the last bar. if barstate.islast line.new(x1, chartHi, x2, chartHi, xloc.bar_time, extend.both, color.lime) line.new(x1, chartLo, x2, chartLo, xloc.bar_time, extend.both, color.fuchsia) string hiTxt = str.format(" {0}\n{1}", str.tostring(chartHi, format.mintick), str.format_time(x1, format = "dd/MM/yy @ HH:mm")) string loTxt = str.format("{0}\n{1}", str.tostring(chartLo, format.mintick), str.format_time(x2, format = "dd/MM/yy @ HH:mm")) label.new(x1, chartHi, hiTxt, xloc.bar_time, yloc.price, color.new(color.lime, 80), label.style_label_down, color.lime) label.new(x2, chartLo, loTxt, xloc.bar_time, yloc.price, color.new(color.fuchsia, 80), label.style_label_up, color.fuchsia)

Note that:

- Values derived from visible chart variables can change throughout the script's runtime. To accurately reflect the entire visible range, the script defers drawing the lines until the last bar (using barstate.islast).
- Because the visible chart values are defined in the global scope, *outside* the local block defined by barstate.islast, the functions process the entire dataset before determining the final high and low values.

> NoteScripts that use chart.left_visible_bar_time or chart.right_visible_bar_time recalculate their results on *every bar* each time the user scrolls or zooms the chart.

For more information, refer to the VisibleChart library's documentation.

How to remember the last time a condition occurred?

Scripts can store the number of bars between the current bar and a bar on which a condition occurred in various ways:

- Using ta.barssince(). This built-in function is the simplest way to track the distance from the condition.
- Manually replicating the functionality of ta.barssince() by initializing the distance to zero when the condition occurs, then incrementing it by one on each bar, resetting it if the condition occurs again.
- Saving the bar_index when the condition occurs, and calculating the difference from the current bar_index.

Programmers can then use the number of bars with the history-referencing operator [] to retrieve the value of a variable, such as the close, on that bar.

Alternatively, if the script needs *only* the value itself and not the number of bars, simply save the value each time the condition occurs. This method is more efficient because it avoids referencing the series multiple times throughout its history. This method also reduces the risk of runtime errors in scripts if the size of the historical reference is too large.

Here's a script that demonstrates these methods:

//@version=6 indicator("Track distance from condition", "", true) // Plot the high/low from the bar where a condition occurred the last time. // Conditions bool upBar = close > open bool dnBar = close < open bool up3Bars = dnBar and upBar[1] and upBar[2] and upBar[3] bool dn3Bars = upBar and dnBar[1] and dnBar[2] and dnBar[3] display = display.data_window // Method 1: Using "ta.barssince()". plot(high[ta.barssince(up3Bars)], color = color.new(color.blue, 80), linewidth = 16) plot(low[ta.barssince(dn3Bars)], color = color.new(color.red, 80), linewidth = 16) plot(ta.barssince(up3Bars), "1. ta.barssince(up3Bars)", display = display) plot(ta.barssince(dn3Bars), "1. ta.barssince(dn3Bars)", display = display) // Method 2: Manually replicating the functionality of the "ta.barssince()" function. var int barsFromUp = na var int barsFromDn = na barsFromUp := up3Bars ? 0 : barsFromUp + 1 barsFromDn := dn3Bars ? 0 : barsFromDn + 1 plot(high[barsFromUp], color = color.blue, linewidth = 3) plot(low[barsFromDn], color = color.red, linewidth = 3) plot(barsFromUp, "3. barsFromUp", display = display) plot(barsFromDn, "3. barsFromDn", display = display) // Method 3: Storing the `bar_index` value when a condition is met. var int barWhenUp = na var int barWhenDn = na if up3Bars barWhenUp := bar_index if dn3Bars barWhenDn := bar_index plot(high[bar_index - barWhenUp], color = color.new(color.blue, 70), linewidth = 8) plot(low[bar_index - barWhenDn], color = color.new(color.red, 70), linewidth = 8) plot(bar_index - barWhenUp, "2. bar_index - barWhenUp", display = display) plot(bar_index - barWhenDn, "2. bar_index - barWhenDn", display = display) // Method 4: Storing the value when a condition is met. var float highWhenUp = na var float lowWhenDn = na if up3Bars highWhenUp := high if dn3Bars lowWhenDn := low plot(highWhenUp, color = color.new(color.white, 70), linewidth = 1) plot(lowWhenDn, color = color.new(color.white, 70), linewidth = 1)

How can I plot the previous and current day's open?

There are several methods for plotting prices from a higher timeframe (we assume that these scripts are to be run on intraday timeframes).

Using `timeframe.change()`

The timeframe.change() function identifies when a bar in a specified timeframe opens. When a new daily bar opens, the following example script first copies the existing daily opening value to the variable for the previous day, and then updates the opening price for the current day.

```
//@version=6 indicator("Previous and current day open using `timeframe.change()`", "", true) bool newDay = timeframe.change("1D") var float yesterdayOpen = na var float todayOpen = na if newDay yesterdayOpen := todayOpen // We reassign this value first todayOpen := open // and then store today's open plot(yesterdayOpen, "Yesterday's Open", newDay ? na : color.red, 2, plot.style_line) plot(todayOpen, "Today's Open", newDay ? na : color.green, 2, plot.style_line) bgcolor(newDay ? color.new(color.gray, 80) : na)
```

Note that:

- This method uses the chart's timeframe transitions to establish open prices and does not make adjustments for session times.
- For some markets and instrument types, the intraday data and the daily data is expected to differ. For example, the US exchanges like NASDAQ and NYSE include more trades in daily bars than in intraday ones, which results in different OHLC values between intraday and daily data, and in daily volume being far greater than intraday one. As a result, the first open of a trading session on an intraday chart can differ from the open of its respective 1D candle.

Using `request.security()`

To match the values on the chart with the values on higher timeframe charts, it's necessary to access the higher timeframe data feeds. Scripts can achieve this by using the request.security() function.

The following example script requests two data feeds from a higher timeframe. To reduce the risk of repainting, we use only confirmed values for historical bars. The script plots confirmed values retroactively on each preceding day when a new day begins. For the real-time bar of the higher timeframe, which represents the current day, we draw a separate set of lines. The realtime lines can change during the day. While this type of repainting is not apparent here when using the opening price, which does not change after the bar opens, it is more obvious for scripts that use the closing price, which takes the current price until the bar closes.

```
//@version=6 indicator("Previous and current day open using `request.security()`", "", true, max_lines_count = 500) string periodInput = input.timeframe("1D", "Higher timeframe") [htfOpen1, htfOpen2, htfTime, htfTimeClose] = request.security(syminfo.tickerid, periodInput, [open[1], open[2], time[1], time_close[1]], lookahead = barmerge.lookahead_on) [htfRtOpen, htfRtOpen1] = request.security(syminfo.tickerid, periodInput, [open, open[1]]) var line rtOpen = line.new(na, na, na, na, xloc.bar_time, color = color.lime) var line rtOpen1 = line.new(na, na, na, na, xloc.bar_time, color = color.gray) var int rtStart = time var int rtEnd = time_close(periodInput) if ta.change(htfTime) != 0 line.new(htfTime, htfOpen1, htfTimeClose, htfOpen1, xloc.bar_time, color = color.lime) line.new(htfTime, htfOpen2, htfTimeClose, htfOpen2, xloc.bar_time, color = color.gray) rtStart := time rtEnd := time_close(periodInput) line.set_xy1(rtOpen1, rtStart, htfRtOpen1), line.set_xy2(rtOpen1, rtEnd, htfRtOpen1) line.set_xy1(rtOpen, rtStart, htfRtOpen), line.set_xy2(rtOpen, rtEnd, htfRtOpen) bgcolor(timeframe.change(periodInput) ? color.new(color.gray, 80) : na)
```

Using `timeframe`

Instead of writing custom logic to retrieve or calculate prices for a particular timeframe, programmers can run the entire script in that timeframe.

If scripts include the `timeframe` parameter in the indicator() declaration, the user can choose the timeframe in which the script runs. The script can set a default timeframe.

By default, the following script plots the current and previous day's opening prices, similar to the previous examples. It is much simpler, but behaves quite differently. For historical bars, the script returns values when the day closes, effectively one day "late". For realtime and elapsed realtime bars, the script returns live values, if the option "Wait for timeframe closes" is not selected in the script settings.

//@version=6 indicator("Previous and current day open using `timeframe`", "", true, timeframe = "1D", timeframe_gaps = true) plot(open[1], "Yesterday's Open", color.red, 2, plot.style_line) plot(open, "Today's Open", color.green, 2, plot.style_line)

Note that:

- Only simple scripts that do not use drawings can use the `timeframe` parameter.
- Scripts that use the `timeframe` parameter can plot values quite differently depending on which settings are chosen. For an explanation, see the Leveraging multi-timeframe analysis article in our Help Center.

How can I count the occurrences of a condition in the last x bars?

One obvious method is to use a for loop to retrospectively review each of the last x bars and check for the condition. However, this method is inefficient, because it examines all bars in range *again* on every bar, even though it already examined all but the last bar.

In general, using unnecessary, large, or nested for loops can result in slower processing and longer chart loading times.

The simplest and most efficient method is to use the built-in math.sum() function, and pass it a conditional series to count. This function maintains a running total of the count as each bar is processed, and can take a simple or series length.

The following example script uses both of these calculation methods. It also uses a "series" length that adjusts for the first part of the chart, where the number of bars available is less than the length. This way, the functions do not return `na` values.

//@version=6 indicator("Number of occurrences demo", overlay = false) int lengthInput = input.int(100, "Length", minval = 1) // Condition to count. bool isUpBar = close > open // Count using a loop (inefficient). countWithLoop(bool condition, int length) => int count = 0 for i = 0 to length - 1 if condition[i] count += 1 count // Count using Pine's built-in function. Can be "simple" or "series" length. countWithSum(bool condition, int length) => float result = math.sum(condition ? 1 : 0, length) float v1 = countWithSum(isUpBar, math.min(lengthInput, bar_index + 1)) int v2 = countWithLoop(isUpBar, math.min(lengthInput, bar_index + 1)) plot(v1, "Efficient count", color.red, 4) plot(v2, "Inefficient count", color.black, 1)

How can I implement an on/off switch?

An on/off switch is a persistent state that can be turned on once, and persists across bars until it is turned off. Scripts can use the var keyword to initialize a variable only once, and maintain its most recent value across subsequent bars unless it is reassigned. Such persistent states can be boolean values, or integers, or any other type.

The following example script show how to implement this. Each instance of the on and off triggers displays with an arrow and the word "On" or "Off". A green background highlights the bars where the switch is in the

“On” state.

//@version=6 indicator("On/Off condition example", overlay = true) bool upBar = close > open // On/off conditions. bool triggerOn = upBar and upBar[1] and upBar[2] bool triggerOff = not upBar and not upBar[1] // Switch state is saved across bars. var bool onOffSwitch = false // Turn the switch on or off, otherwise persist its state. onOffSwitch := triggerOn ? true : triggerOff ? false : onOffSwitch bgcolor(onOffSwitch ? color.new(color.green, 90) : na) plotchar(triggerOn, "triggerOn", "▲", location.belowbar, color.lime, size = size.tiny, text = "On") plotchar(triggerOff, "triggerOff", "▼", location.abovebar, color.red, size = size.tiny, text = "Off")

How can I alternate conditions?

Scripts can alternate from one state to another strictly, even when the triggers to change state do not occur in strict order. This can be useful to mark only the first trigger and not any subsequent triggers, or to prevent multiple alerts.

The following example script plots all pivots, defined by Williams fractals. These pivots can occur in any order. The script stores the type of the most recent pivot, and confirms the next pivot *only* if it is of the opposite type, such that confirmed pivots appear strictly high-low-high or low-high-low, etc. Confirmed pivots are plotted in a larger size and different color. The chart background color is colored according to the type of the most recent confirmed pivot.

//@version=6 indicator("Alternating states", "", true) lookback = input.int(2, title="Lookback & Lookahead") // Define an enum of allowed pivot types. enum PivotType high low undefined const color red80 = color.new(color.red, 80) const color green80 = color.new(color.green, 80) const color yellow80 = color.new(color.yellow, 80) // Define a variable of type PivotType to track the pivot direction. var PivotType lastPivot = PivotType.undefined // Define pivots. float pivotLowPrice = ta.pivotlow(lookback, lookback) float pivotHighPrice = ta.pivothigh(lookback, lookback) bool isPivotLow = not na(pivotLowPrice) bool isPivotHigh = not na(pivotHighPrice) // Plot triangles for pivot points. plotshape(isPivotLow ? pivotLowPrice : na, "Low", shape.triangleup, location.belowbar, color.yellow, offset = -lookback, size = size.tiny) plotshape(isPivotHigh ? pivotHighPrice : na, "High", shape.triangledown, location.abovebar, color.yellow, offset = -lookback, size = size.tiny) // Confirm highs and lows strictly in order. `PivotType.undefined` handles the case where no pivot has yet occurrred. bool confirmedLow = isPivotLow and (lastPivot == PivotType.high or lastPivot == PivotType.undefined) bool confirmedHigh = isPivotHigh and (lastPivot == PivotType.low or lastPivot == PivotType.undefined) // Plot larger triangles for confirmed pivots. plotshape(confirmedLow ? pivotLowPrice : na, "Low Confirmed", shape.triangleup, location.belowbar, color.green, offset = -lookback, size = size.normal) plotshape(confirmedHigh ? pivotHighPrice : na, "High Confirmed", shape.triangledown, location.abovebar, color.red, offset = -lookback, size = size.normal) // Update last pivot direction. lastPivot := confirmedLow ? PivotType.low : confirmedHigh ? PivotType.high : lastPivot // Color the background of the chart based on the direction of the most recent confirmed pivot. bgcolor(lastPivot == PivotType.low ? green80 : lastPivot == PivotType.high ? red80 : lastPivot == PivotType.undefined ? yellow80 : na)

Note that:

- The script uses an enum variable with three possible values to store the type of the last pivot and to decide whether to confirm subsequent pivots.
- A single boolean value cannot reliably do this, because boolean values can only be `true` or `false` and not `na`. Using a boolean value can cause unexpected behavior, for example, at the beginning of the chart history where no trigger condition has occurred.

- A pair of boolean variables can replicate this behavior, with careful handling. See the FAQ entry "How can I accumulate a value for two exclusive states?" for an example of using two boolean values in this way.
- A string variable can also do the same thing. The advantage of an enum over a string is that all possible allowed values are known, thus avoiding the case where a condition tests for a value that is misspelled, outdated or otherwise not relevant. Such a test silently fails in every possible case, and the corresponding logic never runs. Such tests can therefore cause bugs that are difficult to find.

Can I merge two or more indicators into one?

It is possible to combine indicators, paying attention to the following points:

- Ensure that the scales that the indicators use are compatible, or re-scale them to be compatible. For example, combining a moving average indicator, designed to overlay the bar chart, with a volume bar indicator that's meant for a separate indicator pane is unlikely to display as expected.
- Check that variable names do not overlap.
- Convert each script to the most recent version of Pine Script®, or at least the same version, before combining them.
- Ensure that there is only one version declaration and script declaration in the resulting script.

> NoticeIf the individual indicators are large or computationally complex, programmers might encounter issues with one or more of Pine's limitations when combining them into a single script.

How can I rescale an indicator from one scale to another?

Rescaling an indicator from one scale to another means trying to ensure that the values display within a similar range to other values, from the same indicator or from the chart.

For example, consider a script that displays volume typically measuring in the millions of units, and also RSI, which ranges from zero to one hundred. If the script displays these values in the same pane, the volume is visible but the RSI will be so small as to be unreadable.

Where values are dissimilar like this, they must be *rescaled* or *normalized*:

- If the minimum and maximum possible values are known, or *bounded*, the values can be *rescaled*, that is, adjusted to a new range bounded by different maximum and minimum values. Each value differs in absolute terms, but retains the same *relative* proportion to other rescaled values.
- If the values are *unbounded*, meaning that either the maximum or minimum values, or both, are not known, they must instead be *normalized*. Normalizing means scaling the values relative to historical maximum and minimum values. Because the maximum and minimum historical values can change over time as the script runs on more historical and realtime bars, the new scale is *dynamic* and therefore the new values are *not* exactly proportional to each other.

The example script below uses a `rescale()` function to rescale RSI values, and a `normalize()` function to normalize Commodity Channel Index (CCI) and volume values. Although normalizing is an imperfect solution, it is more complete than using ta.lowest() and ta.highest(), because it uses the minimum and maximum values for the complete set of elapsed bars instead of a subset of fixed length.

```
//@version=6 indicator("Rescaling and normalizing values", "", overlay = false) // @function Rescales a signal with a known scale (bounded) to a new scale. // @param src (series float) The series to rescale. // @param oldMin (simple float) The minimum value of the original signal's scale. // @param oldMax (simple float) The maximum value of the original signal's scale. // @param newMin (simple float) The minimum value of the new scale. // @param newMax (simple float) The maximum value of the new scale. // @returns (float) The rescaled value of the signal. rescale(series float src, simple float oldMin, simple float oldMax, simple float newMin, simple float newMax) => float result = newMin + (newMax - newMin) * (src - oldMin) / math.max(oldMax - oldMin, 10e-10) // @function Rescales a signal with an unknown scale (unbounded) using its historical low and high values. // @param src (series float) The series to rescale. // @param min (simple float) The minimum value of the rescaled series. // @param max (simple float) The maximum value of the rescaled series. // @returns (float) The rescaled value of the signal. normalize(series float src, simple float min, simple float max) => var float historicMin = 10e10 var float historicMax = -10e10 historicMin := math.min(nz(src, historicMin), historicMin) historicMax := math.max(nz(src, historicMax), historicMax) float result = min + (max - min) * (src - historicMin) / math.max(historicMax - historicMin, 10e-10) // ————— Plot normalized CCI cci = ta.cci(close, 20) plot(normalize(cci, 100, 300), "Normalized CCI", #2962FF) // Arbitrary and inexact equivalent of 100 and -100 levels rescaled to the 100/300 scale. band00 = hline(150, "Lower Band", color.new(#C0C0C0, 90), hline.style_solid) band01 = hline(250, "Upper Band", color.new(#C0C0C0, 90), hline.style_solid) fill(band01, band00, color.new(#21328F, 80), "Background") // ————— Plot normalized volume in the same region as the rescaled RSI color volColor = close > open ? #26a69a : #ef5350 plot(normalize(volume, -100, 100), "Normalized volume", volColor, style = plot.style_columns, histbase = -100) hline(100, "", color.new(color.gray, 50), hline.style_dashed) hline(-100, "", color.new(color.gray, 50), hline.style_solid) // ————— Plot rescaled RSI plot(rescale(ta.rsi(close, 14), 0, 100, -100, 100), "Rescaled RSI", #8E1599) hline(0, "RSI 50 level", color.new(color.gray, 70), hline.style_solid) // Precise equivalent of 70 and 30 levels rescaled to the -100/100 scale. band10 = hline(-40, "Lower Band", color.new(#9915FF, 80), hline.style_solid) band11 = hline(40, "Upper Band", color.new(#9915FF, 80), hline.style_solid) fill(band11, band10, color.new(#9915FF, 90), "Background") // ————— Plot original values in Data Window plot(na, "═══════════════════", display = display.data_window) plot(cci, "Original CCI", display = display.data_window) plot(volume, "Original volume", display = display.data_window) plot(ta.rsi(close, 14), "Original RSI", display = display.data_window)
```

How can I calculate my script's run time?

Programmers can measure the time that a script takes to run and see detailed information about which parts of the code take longest in the Pine Profiler. See the section of the User Manual on Profiling and optimization for more information.

How can I save a value when an event occurs?

To save a value when an event occurs, use a *persistent variable*. Scripts declare persistent variables by using the var keyword. Such variables are initialized only once, at bar_index zero, instead of on each bar, and maintain the same value after that unless changed.

In the following example script, the var keyword allows the `priceAtCross` variable to maintain its value between bars until a crossover event occurs, when the script updates the variable with the current close price. The := reassignment operator ensures that the global variable `priceAtCross` is modified. Using the = assignment operator instead would create a new local variable that is inaccessible outside the if block. The new local variable would have the same name as the global variable, which is called *shadowing*. The compiler warns about shadow variables.

//@version=6 indicator("Save a value when an event occurs", "", true) float hiHi = ta.highest(high, 5)[1] var float priceAtCross = na if ta.crossover(close, hiHi) // When a crossover occurs, assign the current close price to `priceAtCross`. priceAtCross := close plot(hiHi) plot(priceAtCross, "Price At Cross", color.orange, 3, plot.style_circles)

How can I count touches of a specific level?

The most efficient way to count touches of a specific level is by tracking the series on each bar. A robust approach requires maintaining separate tallies for up and down bar touches and taking into account any gaps across the level. Using loops instead would be inefficient and impractical in this case.

The following example script records a value of 1 in a series whenever a touch occurs, and uses the math.sum() function to count these instances within the last `touchesLengthInput` bars. This script displays the median and touches on the chart using the `force_overlay` parameter of the `plot*()` functions, and displays the count in a separate pane.

//@version=6 indicator("Median Touches", "", overlay = false) int medianLengthInput = input.int(100, "Median calculation: Number of previous closes") int touchesLengthInput = input.int(50, "Number of previous bars to check for price touches") float median = ta.percentile_nearest_rank(close, medianLengthInput, 50) // Don"t count neutral touches when price doesn't move. bool barUp = close > open bool barDn = close < open // Bar touches median. bool medianTouch = high > median and low < median bool gapOverMedian = high[1] < median and low > median bool gapUnderMedian = low[1] > median and high < median // Record touches. int medianTouchUp = medianTouch and barUp or gapOverMedian ? 1 : 0 int medianTouchDn = medianTouch and barDn or gapUnderMedian ? 1 : 0 // Count touches over the last n bars. float touchesUp = math.sum(medianTouchUp, touchesLengthInput) float touchesDn = math.sum(medianTouchDn, touchesLengthInput) // ———————————— Plots // Markers plotchar(medianTouchUp, "medianTouchUp", "▲", location.belowbar, color.lime, force_overlay = true) plotchar(medianTouchDn, "medianTouchDn", "▼", location.abovebar, color.red, force_overlay = true) // Median plot(median, "Median", color.orange, force_overlay = true) // Base areas. plot(touchesUp, "Touches Up", color.green, style = plot.style_columns) plot(-touchesDn, "Touches Dn", color.maroon, style = plot.style_columns) // Exceeding area. float minTouches = math.min(touchesUp, touchesDn) bool minTouchesIsUp = touchesUp < touchesDn basePlus = plot(minTouches, "Base Plus", display = display.none) hiPlus = plot(not minTouchesIsUp ? touchesUp : na, "High Plus", display = display.none) baseMinus = plot(-minTouches, "Base Plus", display = display.none) loMinus = plot(minTouchesIsUp ? -touchesDn : na, "Low Minus", display = display.none) fill(basePlus, hiPlus, color.lime) fill(baseMinus, loMinus, color.red)

How can I know if something is happening for the first time since the beginning of the day?

One way is to use the ta.barssince() function to check if the number of bars since the last occurrence of a condition, plus one, is greater than the number of bars since the beginning of the new day.

Another method is to use a *persistent state* to decide whether an *event* can happen. When the timeframe changes to a new day, the state is reset to allow the event. If the condition occurs while the state allows it, an event triggers. When the event triggers, the state is set so so as not to allow the event.

The following example script shows both methods.

```
//@version=6 indicator("First time today example", "", true) bool isUpCandle = close > open // ————— Method 1. int barsSincePreviousUpCandle = ta.barssince(isUpCandle[1]) int barsSinceStartOfDay = ta.barssince(timeframe.change("1D")) - 1 bool previousUpCandleWasNotToday = barsSincePreviousUpCandle > barsSinceStartOfDay bool isFirstToday1 = isUpCandle and previousUpCandleWasNotToday plotchar(isFirstToday1, "isFirstToday1", "•", location.top, color = color.silver, size = size.normal) plot(barsSinceStartOfDay, "barsSinceStartOfDay", display=display.data_window) // ————— Method 2. var bool hadUpCandleToday = false // This is a persistent state. bool isFirstToday2 = false // This is a one-off event. if timeframe.change("1D") // When the day begins.. hadUpCandleToday := false // we have not yet had an up candle today, so reset the state. if isUpCandle and not hadUpCandleToday // If this is the first up candle today.. hadUpCandleToday := true // set the persistent state isFirstToday2 := true // and update the event. plotchar(isFirstToday2, "isFirstToday2", "•", location.top, color = color.yellow, size = size.small)
```

How can I optimize Pine Script code?

Optimizing Pine Script code can make scripts run faster and use less memory. For large or complex scripts, optimization can avoid scripts reaching the computational limits.

The Pine Profiler analyzes all significant code in a script and displays how long each line or block takes to run. Before optimizing code, run the Pine Profiler to identify which parts of the code to optimize first. The Pine Profiler section of the User Guide contains an extensive discussion of how to optimize code. In addition, consider the following tips:

- Use strategy scripts only to model trades. Otherwise, use indicator scripts, which are faster.
- Become familiar with the Pine execution model and time series to structure code effectively.
- Declare variables with the var keyword when initialization involves time-consuming operations like complex functions, arrays, objects, or string manipulations.
- Keep operations on strings to a necessary minimum, because they can be more resource-intensive than operations on other types.
- Using built-in functions is usually faster than writing custom functions that do the same thing. Sometimes, alternative logic can be more efficient than using standard functions. For example, use a persistent variable when an event occurs, to avoid using ta.valuewhen(), as described in the FAQ entry How can I save a value when an event occurs?. Or save the bar_index when a condition occurs to avoid using ta.barssince(), as described in the FAQ entry How to remember the last time a condition occurred?.

How can I access a stock's financial information?

In Pine, the request.financial() function can directly request financial data.

On the chart, users can open financial indicators in the "Financials" section of the "Indicators, Metrics & Strategies" window.

How can I find the maximum value in a set of events?

Finding the maximum value of a variable that has a meaningful value *on every bar*, such as the high or low in price, is simple, using the ta.highest() function.

However, if the values do not occur on every bar, we must instead store each value when it occurs and then find the maximum. The most flexible way to do this is by using an array.

The following example script stores pivot highs in a fixed-length array. The array is managed as a queue: the script adds new pivots to the end, and removes the oldest element from the array. To identify the highest value among the stored pivots, we use the array.max() function and plot this maximum value on the chart. Additionally, we place markers on the chart to indicate when the pivots are detected, and the bars where the pivots occurred. By definition, these points are not the same, because a pivot is only confirmed after a certain number of bars have elapsed.

//@version=6 indicator("Max pivot demo", "", true) // Create inputs to specify the pivot legs and the number of last pivots to keep to compare. int pivotLengthInput = input.int(5, "Pivot length", minval = 1) int numPivotsInput = input.int(3, "Number of pivots to check") // Initialize an array with a size based on the number of recent pivots to evaluate. var array<float> pivotsArray = array.new<float>(numPivotsInput) // Find the pivot value and set up a condition to verify if a value has been found. float ph = ta.pivothigh(pivotLengthInput, pivotLengthInput) bool newPH = not na(ph) // When a new pivot is found, add it to the array and discard the oldest value. if newPH pivotsArray.push(ph) pivotsArray.shift() // Display the max value from the array on the chart, along with markers indicating the positions and detection times of the pivot highs. plot(pivotsArray.max()) plotchar(newPH, "newPH", "•", location.abovebar, offset = -pivotLengthInput) plotchar(newPH, "newPH", "▲", location.top)

How can I display plot values in the chart's scale?

To display the names and values of plots from an indicator in the chart's scale, right-click on the chart to open the chart "Settings" menu. In the "Scales and lines" tab, select "Name" and "Value" from the "Indicators and financials" drop-down menu.

How can I reset a sum on a condition?

To sum a series of values, initialize a persistent variable by using the var keyword to the track the sum. Then use a logical test to reset the values when a condition occurs.

In the following example script, we initialize a persistent variable called `cumulativeVolume` to track the sum of the volume. Then we reset it to zero on a Moving Average Convergence/Divergence (MACD) cross up or down.

We plot the cumulative volume on the chart, as well as arrows to show the MACD crosses.

//@version=6 indicator("Reset sum on condition example", overlay = false) const color TEAL = color.new(color.teal, 50) const color RED = color.new(color.red, 50) [macdLine, signalLine, _] = ta.macd(close, 12, 26, 9) bool crossUp = ta.crossover(macdLine, signalLine) bool crossDn = ta.crossunder(macdLine, signalLine) bool doReset = crossUp or crossDn var float cumulativeVolume = na cumulativeVolume += volume // On every bar, we sum the volume. cumulativeVolume := doReset ? 0. : cumulativeVolume // But when we get a cross, we reset it to zero. plot(cumulativeVolume, "Cumulative volume", close >= open ? TEAL : RED, 1, plot.style_columns) plotshape(crossUp, "crossDn", shape.arrowup, location.top, color.lime) plotshape(crossDn, "crossUp", shape.arrowdown, location.top, color.fuchsia)

Note that:

- In the ta.macd() function call, we only require two of the three values returned in the tuple. To avoid unnecessary variable declarations, we assign the third tuple value to an underscore. Here, the underscore acts like a dummy variable.

How can I accumulate a value for two exclusive states?

Consider a simple indicator defined by two exclusive states: *buy* and *sell*. The indicator cannot be in both *buy* and *sell* states simultaneously. In the *buy* state, the script accumulates the volume of shares being traded. In the *sell* state, the accumulation of volume begins again from zero.

There are different ways to code this kind of logic. See the FAQ entry "How can I alternate conditions" for an example of using an enum to manage two exclusive states. The following example script uses two boolean variables to do the same thing.

Additionally, this script demonstrates the concept of *events* and *states*. An event is a condition that occurs on one or more arbitrary bars. A state is a condition that persists over time. Typically, programmers use events to turn states on and off. In turn, states can allow or prevent other processing.

The script plots arrows for events, which are based on rising or falling values of the close price. These events determine which of the two exclusive states is active; the script colors the background according to the current state. The script accumulates bullish and bearish volume only in the corresponding bullish or bearish state, displaying it in a Weis Wave fashion.

//@version=6 indicator("Cumulative volume", "") bool upEvent = ta.rising(close, 2) bool dnEvent = ta.falling(close, 2) var bool upState = false, var bool dnState = false // When the right event occurs, turn the state on; when a counter-event occurs, turn it off; otherwise, persist it. upState := upEvent ? true : dnEvent ? false : upState dnState := upEvent ? false : dnEvent ? true : dnState var float volUp = na, var float volDn = na if upState // For every bar that we are in the up state, volUp += volume // sum the up volume. if dnState volDn += volume if upEvent // If we change state to up, volDn := 0 // reset the down volume. if dnEvent volUp := 0 plot(+volUp, "Up Volume", color.green, 4, plot.style_columns) plot(-volDn, "Dn Volume", color.maroon, 4, plot.style_columns) plotchar(upEvent, "Up Event", "▲", location.bottom, color.green, size = size.tiny) plotchar(dnEvent, "Dn Event", "▼", location.top, color.maroon, size = size.tiny) bgcolor(upState ? color.new(color.green, 90) : dnState ? color.new(color.red, 90) : na)

Note that:

- Equivalent logic using ternary conditions is smaller and potentially more efficient, but not as easy to read, extend, or debug. This more verbose logic illustrates the concepts of events and states, which can apply to many types of scripting problems. This logic is an extension of the on-off switch in the FAQ entry "How can I implement an on/off switch?".
- When using states, it is important to make the conditions for resetting states explicit, to avoid unforeseen problems.
- Displaying all events and states during script development, either on the chart or in the Data Window, helps debugging.

How can I organize my script's inputs in the Settings/Inputs tab?

A script's plots and inputs constitute its user interface. The following example script uses the following techniques to organize inputs for greater clarity:

- **Grouping inputs:** Create a section header for a group of inputs by using the `group` parameter in the input() functions. Use constants for group names to simplify any potential name changes.
- **Visual boundaries:** Use ASCII characters to create separators, establishing visual boundaries for distinct group sections. For continuous separator lines, reference group headers 1 and 2 in our script below, which use ASCII characters 205 or 196. Conversely, the dash (ASCII 45) and Em dash (ASCII 151), shown in group headers 3 and 4, do not join continuously, resulting in a less visually appealing distinction. Note that Unicode characters might display differently across different machines and browsers, potentially altering their appearance or spacing for various users.
- **Indentation of sub-sections:** For a hierarchical representation, use Unicode whitespace characters to indent input sub-sections. Group 3 in our script uses the Em space ( ) 8195 (0x2003) to give a tab-like spacing.
- **Vertical alignment of inlined inputs:** In our script, Group 1 shows how vertical alignment is difficult when inline inputs have varied `title` lengths. To counteract this misalignment, Group 2 uses the Unicode EN space ( ): 8194 (0x2002) for padding, since regular spaces are stripped from the label. For precise alignment, use different quantities and types of Unicode spaces. See here for a list of Unicode spaces of different widths. Note that, much like the separator characters, the rendering of these spaces might differ across browsers and machines.
- **Placing inputs on one line:** Add multiple related inputs into a single line using the `inline` parameter. Group 4 in our script adds the title argument to just the first input and skips it for the others.

```
//@version=6 indicator("Inputs", overlay = true) // Defining options strings improves script readability. // It also enables the creation of boolean variables by comparing these constants with user input strings in a single line of code. string EQ1 = "On" string EQ2 = "Off" // The `GRP*` strings used for group headers demonstrate using ASCII characters to create a visual boundary, // making it easier for users to differentiate between different sections in the menu. // Group 1 demonstrates inline inputs that do not align vertically in the menu. string GRP1 = "═══════════ Settings ═══════════" // ASCII 205 float ao1SrcInput = input.source(close, "AO source", inline = "11", group = GRP1) int ao1LenInput = input.int(14, "Length", inline = "11", group = GRP1) float long1SrcInput = input.source(close, "Signal source", inline = "12", group = GRP1) int long1LenInput = input.int(3, "Length", inline = "12", group = GRP1) // In Group 2, the title of `ao2SrcInput` is padded with three Unicode EN spaces (U+2002) to compensate for the misalignment. string GRP2 = "─────────── Settings ───────────" // ASCII 196 float ao2SrcInput = input.source(close, "AO source   ", inline = "21", group = GRP2) int ao2LenInput = input.int(14, "Length", inline = "21", group = GRP2) float long2SrcInput = input.source(close, "Signal source", inline = "22", group = GRP2) int long2LenInput = input.int(3, "Length", inline = "22", group = GRP2) // This configuration uses Unicode white space characters to indent input sub-sections. We use Em space ( ): 8195 (0x2003). string GRP3 = "——————————— Settings ———————————" // ASCII 151 (Em dash) float level1Input = input.float(65., "First level", group = GRP3) float level2Input = input.float(65., "  Second Level", group = GRP3) bool level3Input = input.string(EQ1, "    Checkbox equivalent", group = GRP3, options = [EQ1, EQ2]) == EQ1 float level4Input = input.float(65., "Widest Legend          ", group = GRP3) // These options demonstrate the use of the `inline` parameter to create structured blocks of inputs that are relevant to one another. string GRP4 = "----------- Settings -----------" // ASCII 45 (dash) bool showMa1Input = input(true, "MA №1", inline = "1", group = GRP4) string ma1TypeInput = input.string("SMA", "", inline = "1", group = GRP4, options = ["SMA", "EMA", "SMMA (RMA)", "WMA", "VWMA"]) float ma1SourceInput = input(close, "", inline = "1", group = GRP4) int ma1LengthInput = input.int(20, "", inline = "1", group = GRP4, minval = 1) color ma1ColorInput = input(#f6c309, "", inline = "1", group = GRP4) bool showMa2Input = input(true, "MA №2", inline = "2", group = GRP4) string ma2TypeInput = input.string("SMA", "", inline = "2", group = GRP4, options = ["SMA", "EMA", "SMMA (RMA)", "WMA", "VWMA"]) float ma2SourceInput = input(close, "", inline = "2", group = GRP4) int ma2LengthInput =
```

input.int(50, "", inline = "2", group = GRP4, minval = 1) color ma2ColorInput = input(#fb9800, "", inline = "2", group = GRP4) bool showMa3Input = input(true, "MA №3", inline = "3", group = GRP4) string ma3TypeInput = input.string("SMA", "", inline = "3", group = GRP4, options = ["SMA", "EMA", "SMMA (RMA)", "WMA", "VWMA"]) float ma3SourceInput = input(close, "", inline = "3", group = GRP4) int ma3LengthInput = input.int(100, "", inline = "3", group = GRP4, minval = 1) color ma3ColorInput = input(#fb6500, "", inline = "3", group = GRP4) // @function Calculates various types of moving averages for the `source` based on the specified `maType`. // @param series (series float) Series of values to process. // @param length (simple int) Number of bars (length). // @param maType (simple string) The type of moving average to calculate. // Options are "SMA", "EMA", "SMMA (RMA)", "WMA", and "VWMA". // @returns (float) The moving average of the `source` for `length` bars back. ma(series float source, simple int length, simple string maType) => switch maType "SMA" => ta.sma(source, length) "EMA" => ta.ema(source, length) "SMMA (RMA)" => ta.rma(source, length) "WMA" => ta.wma(source, length) "VWMA" => ta.vwma(source, length) => na // Calculate the moving averages with the user-defined settings. float ma1 = ma(ma1SourceInput, ma1LengthInput, ma1TypeInput) float ma2 = ma(ma2SourceInput, ma2LengthInput, ma2TypeInput) float ma3 = ma(ma3SourceInput, ma3LengthInput, ma3TypeInput) // Plot the moving averages, if each checkbox is enabled. plot(showMa1Input ? ma1 : na, "MA №1", ma1ColorInput) plot(showMa2Input ? ma2 : na, "MA №2", ma2ColorInput) plot(showMa3Input ? ma3 : na, "MA №3", ma3ColorInput)

Tips:

- Order the inputs to prioritize user convenience rather than to reflect the order used in the script's calculations.
- Never use two checkboxes for mutually exclusive selections. Use dropdown menus instead.
- Remember that dropdown menus can accommodate long strings.
- Provide adequate minimum and maximum values for numeric values, selecting the proper float or int type.
- Customize step values based on the specific needs of each input.
- Because checkboxes cannot be indented, use the input() function's `options` parameter to create dropdown selections so that the sections appear more organized compared to using checkboxes.
- Observe how the `level3Input` is calculated as a boolean variable by comparing the input with the `EQ1` "ON" constant. This method provides a visually appealing indented on-off switch in the menu without adding complexity to the code.
- For a consistent visual appearance, vertically center the separator titles across all inputs. Due to the proportional spacing of the font, achieving this might require some trial and error.
- To ensure that separators align just slightly to the left of the furthest edge of dropdowns, begin with the longest input title, because it sets the width of the window.
- To avoid adjusting separators if the longest input title is shorter than initially anticipated, extend its length using Unicode white space. Refer to the code example for input `level4Input` for a demonstration.

Can I plot values from a local scope?

A script can use `plot*()` functions and other plot visuals only in the global scope – they cannot be included in the local scopes of conditional structures, loops, or user-defined functions and methods. Therefore, plots can only use variables and literals that are declared globally.

However, programmers can extract data from local scopes to the global scope to make the data accessible to `plot*()` functions. Assign the local scope values to globally declared variables, using return expressions

or reference types, then use these global variables in `plot*()` calls to visualize the local data.

Alternatively, use Pine Logs or drawings to display values from within local scopes directly.

Times, dates, and sessions

How can I get the time of the first bar in the dataset?

The following example script initializes a variable using the var keyword on the first bar and then never updates it again. The variable stores the value of the time built-in variable, which represents the time of the bar open in UNIX format (milliseconds since 00:00:00 UTC on 1 January 1970).

//@version=6 indicator('Time at first bar') // Capture the time of the first bar in the dataset. var int t = time plot(t)

How can I convert a time to a date-time string?

The built-in function str.format_time() translates a UNIX timestamp into a string representation of date and time. In the following example script, we provide four distinct methods to obtain the date-time string, two of which offset the date and time into the future. To improve readability, we use a custom function, `timeToString()`, which calls str.format_time() and applies a consistent format, instead of specifying the format for every function call.

The format `"YYYY.MM.dd @ HH:mm:ss"` is similar to the ISO 8601 date and time representation, but with periods to separate parts of the date, and an `@` symbol between the date and time. For more formatting customization options, consult the str.format_time() documentation.

//@version=6 indicator("Time to string example") //@function Converts a timestamp into a formatted date-time string. timeToString(int t) => str.format_time(t, format = "YYYY.MM.dd @ HH:mm:ss") // Capture the time of the first bar in the dataset. var int firstBarTime = time // Create a new table on the first bar only. var table tbl = table.new(position.middle_right, 1, 1) // On the first bar, build the table cell. if barstate.isfirst table.cell(tbl, 0, 0, "", bgcolor = color.yellow, text_color = color.black, text_halign = text.align_left, text_font_family = font.family_monospace) // On the last bar, build display text and populate the table. else if barstate.islast string txt = str.format("Date/time at bar_index = 0 {0}" + "\nCurrent Date/time {1}" + "\nDate/time 4 days from current time {2}" + "\nDate/time at beginning of last bar {3}" + "\nDate/time 4 days after last bar''s start: {4}", timeToString(firstBarTime), timeToString(timenow), timeToString(timestamp(year(timenow), month(timenow), dayofmonth(timenow) + 4, hour(timenow), minute(timenow), second(timenow))), timeToString(time), timeToString(timestamp(year, month, dayofmonth + 4))) table.cell_set_text(tbl, 0, 0, txt)

How can I find out how many days are in the current month?

The following example script calculates the number of days in the current month, including adjustments for leap years. By default, the `daysPerMonth()` function uses the current year and month, but any year or month number can be passed as arguments:

//@version=6 indicator("Days in month") // @function Calculates the number of days in a specified month, accounting for leap years. // @param yearNumber (int) The year of the `monthNumber` month. Optional. Default is the current year. // @param monthNumber (int, optional) The month for which to find the number of days. Optional. Default is the current month. // @returns (int) The number of days in the `monthNumber` month of the `yearNumber` year. daysPerMonth(int yearNumber = year, int monthNumber = month) => bool

leapYear = (yearNumber % 4 == 0 and yearNumber % 100 != 0) or (yearNumber % 400 == 0) int result = switch monthNumber == 2 => leapYear ? 29 : 28 => 31 - (monthNumber - 1) % 7 % 2 plot(daysPerMonth())

How can I detect the chart's last day?

Scripts can detect the last day on a chart in different ways. However, because the last_bar_time and timenow variables constantly update to mirror the time of the newest bar, each subsequent realtime bar is *also* interpreted as if it is the chart's last day.

Using `timeframe.change`

This example script checks for all of these conditions:

- The daily timeframe begins a new day.
- The difference between the time of the last chart bar (retrieved using last_bar_time) and the current bar's time (from time) is less than the time in one day.

The script colors the background red on the last day.

//@version=6 indicator("Last day example", overlay = true) var bool isLastDay = false if timeframe.change("1D") and last_bar_time - time < timeframe.in_seconds("1D") * 1000 isLastDay := true bgcolor(isLastDay ? color.new(color.red, 90) : na)

Using `timenow`

This script uses the timenow variable to retrieve the current time – not the time of the bar on which the script is executing, but the time of the latest chart update. We pass this time as input to the built-in functions year(), month(), and dayofmonth() to calculate the year, month, and day of the month for that time. We compare these calculated values with the built-in variables year, month, and dayofmonth, which store the year, month, and day of the month of the current chart bar. When the year, day and month of the latest chart update match the values of the current bar, then the script is executing on bars from the most recent day.

//@version=6 indicator("Detect today", overlay = true) isToday() => int currentYear = year(timenow) int currentMonth = month(timenow) int currentDay = dayofmonth(timenow) bool result = year == currentYear and month == currentMonth and dayofmonth == currentDay bgcolor(isToday() ? color.new(color.red, 90) : na)

How can I get bar's trading date on symbols with an overnight session?

Some symbols trade overnight, which means that today's trading day actually started yesterday, or even several days ago. Many futures and Forex pairs trade overnight. Trading hours are defined by the exchange. On timeframes of one day and above, the chart visually adjusts the dates of these bars to ensure the date corresponds to the bar's trading day; on intraday timeframes the date and time are shown as they are.

In Pine Script, the opening time of a bar is represented by the variable time, which also returns the date and time without adjustment. Variables that return date-related values, like dayofmonth, weekofyear, etc., are calculated based on this opening time, which can lead to unexpected behavior on overnight symbols.

The example script below outputs the value of the built-in dayofmonth variable onto a chart as a label. The following screenshot shows the output of this script on two symbols: one stock and one Forex. We added a

vertical line on the chart using the drawing tools to highlight the bar that represents May 22nd.

The stock symbol “NASDAQ:AAPL” displays on the upper chart. Its trading hours open and close within a single day with no overnight session. Therefore, the label displays the same day as the vertical line.

In contrast, on the lower chart, which displays the “FX:EURUSD” symbol, the label shows `21` on the bar marked May 22nd. This is because the trading for May 22 actually starts on May 21, at 17:00 in the exchange’s time zone, and the dayofmonth variable uses this time to determine the day of the month for this bar.

//@version=6 indicator("Day number", overlay = true) currTradingDay = dayofmonth labelStr = "`time`: " + str.format_time(time, "dd MMM HH:mm 'ET'") labelStr += "\n`dayofmonth`: " + str.tostring(currTradingDay) label.new(bar_index, high, labelStr)

There is a way to avoid this issue. Each date-related variable like dayofmonth has a corresponding function with the same name, e.g., dayofmonth(). The function takes two parameters, `time` and `timezone`, which together allow you to specify the exact timestamp to convert to a date.

In addition, the time_tradingday variable returns the timestamp of 00:00 UTC of the *trading day* the bar belongs to, regardless of the bar’s actual opening time. You can pass this timestamp to the dayofmonth() function along with the `"UTC"` time zone to extract the date from the trading day of the bar, instead of its opening time. Below, we update our example script to use this method:

//@version=6 indicator("Day number", overlay = true) currTradingDay = dayofmonth(time_tradingday, "UTC") labelStr = "`time_tradingday`: " + str.format_time(time_tradingday, "dd MMM HH:mm 'UTC'", "UTC") labelStr += "\n`dayofmonth(time_tradingday)`: " + str.tostring(currTradingDay) label.new(bar_index, high, labelStr)

Note that:

- Here, we add `"UTC"` as the time zone for the dayofmonth() calculation, because time_tradingday always returns the trading day as 00:00 in the UTC time zone. The variable returns 00:00 because its only purpose is to specify a year, month, and day, and its hour and minute components carry no specific meaning.

How can I plot a value starting n months or years back?

The timestamp() function can accept negative argument values and convert them to accurate dates. For instance, a negative month value deducts the corresponding number of months from the outcome.

Our example script calculates the date three months prior to the current time, using the timestamp() function to create a timestamp that is the configured amount of time prior to the time now, as measured using timenow. As the script executes on each bar, it compares the bar time with the previously calculated timestamp. When the time of a bar matches our specified target date, we assign the high of that bar as our value to plot and color the background red for demonstration.

//@version=6 indicator("Plot value starting n months/years back", overlay = true) int monthsBackInput = input.int(3, minval = 0) int yearsBackInput = input.int(0, minval = 0) bool calcFromNowInput = input(false, "Calculate from current date/time instead of first of the month") bool isTargetDate = time >= timestamp(year(timenow) - yearsBackInput, month(timenow) - monthsBackInput, calcFromNowInput ? dayofmonth(timenow) : 1, calcFromNowInput ? hour(timenow) : 0, calcFromNowInput ? minute(timenow) : 0, calcFromNowInput ? second(timenow) : 0) bool isBeginMonth = not isTargetDate[1] and isTargetDate var

float valueToPlot = na if isBeginMonth valueToPlot := high plot(valueToPlot) bgcolor(isBeginMonth ? color.new(color.red, 80) : na)

How can I track highs and lows for a specific timeframe?

The following example script tracks the high and low values of a timeframe that the user selects. Our script avoids using request.security() calls, which are resource-intensive.

//@version=6 indicator("Periodic high/low example", overlay = true) // Inputs bool showHiInput = input.bool(true, "Show highs") bool showLoInput = input.bool(true, "Show lows") string periodInput = input.timeframe("1D", "Period after which high/low is reset") // Declare with `var` to retain values bar to bar. var float hi = na var float lo = na // When a new period begins, reset hi/lo; otherwise, trail them. bool isNewPeriod = timeframe.change(periodInput) hi := isNewPeriod ? high : math.max(high, hi) lo := isNewPeriod ? low : math.min(low, lo) // Plot the hi, lo, and an invisible mid value for area fill. p1 = plot(hi, "Highs", isNewPeriod ? na : color.new(color.lime, 60), display = showHiInput ? display.all : display.none) p2 = plot(lo, "Lows", isNewPeriod ? na : color.new(color.fuchsia, 60), display = showLoInput ? display.all : display.none) p3 = plot(hl2, editable = false, display = display.none) // Create fills between the current mid price and the highest and lowest price. fill(p1, p3, color = isNewPeriod ? na : color.new(color.lime, 90)) fill(p2, p3, color = isNewPeriod ? na : color.new(color.fuchsia, 90)) bgcolor(isNewPeriod ? color.new(color.gray, 90) : na) // Highlight the background when a new period begins.

Note that:

- We detect the beginning of a new timeframe by passing the period input to the timeframe.change() function.
- The script resets the `hi` and `lo` variables at the beginning of a new timeframe and then trails `hi` up and `lo` down.
- We change the color of plots to na every time a new timeframe begins, to visually distinguish between periods.

How can I track the highs and lows within a specific session or time of day?

To find if a time is within a specific session, pass a time-based session string to the time() function, which retrieves the UNIX time of the current bar based on a given timeframe and session. If the time point falls outside the session, the function returns na.

In the following example script, on the first bar that the time is in session, we set the `hi` and `lo` variables to the source inputs provided by the user (high and low by default). During the session we trail the `hi` value up and the `lo` value down. The script also highlights the background red when a bar's time falls outside the session time.

//@version=6 indicator("Session high/low", overlay = true) const string DEFAULT = "Default" const string EQ1 = "On" const string EQ2 = "Off" // Inputs bool showHiInput = input.string(EQ1, "Show highs", options = [EQ1, EQ2]) == EQ1 bool showLoInput = input.string(EQ1, "Show lows", options = [EQ1, EQ2]) == EQ1 float srcHiInput = input.source(high, "Source for Highs") float srcLoInput = input.source(low, "Source for Lows") string sessionInput = input.session("1200-1500", "Allowed hours") string timezoneInput = input.string(DEFAULT, "Time zone", options = [DEFAULT, "GMT-12", "GMT-11", "GMT-10", "GMT-9", "GMT-8", "GMT-7", "GMT-6", "GMT-5", "GMT-4", "GMT-3", "GMT-2", "GMT-1", "GMT-0", "GMT+1", "GMT+2",

"GMT+3", "GMT+4", "GMT+5", "GMT+6", "GMT+7", "GMT+8", "GMT+9", "GMT+10", "GMT+11", "GMT+12", "GMT+13", "GMT+14"]) // Check to see if we are in allowed hours using session info. int timeInSession = time(timeframe.period, sessionInput, timezoneInput == DEFAULT ? syminfo.timezone : timezoneInput) bool timeIsAllowed = not na(timeInSession) var float hi = na var float lo = na if timeIsAllowed // We are entering allowed hours; reset hi/lo. if not timeIsAllowed[1] hi := srcHiInput lo := srcLoInput else // We are in allowed hours; track hi/lo. hi := math.max(srcHiInput, hi) lo := math.min(srcLoInput, lo) // Plot hi/lo within allowed hours. plot(showHiInput and timeIsAllowed ? hi : na, "Highs", color.lime, 3, plot.style_circles) plot(showLoInput and timeIsAllowed ? lo : na, "Lows", color.fuchsia, 3, plot.style_circles) bgcolor(not timeIsAllowed ? color.new(color.red, 90) : na)

Note that:

- We use the 3-parameter version of the time() function to ascertain whether we are within user-specified hours and time zone for tracking highs and lows.
- We pass a session input and an optional time zone to the time() function.

How can I make an entire custom session visible?

Scripts can use custom time-based sessions by defining a session string, as explained in the Time-based sessions section of the Sessions page.

The following example script takes a user-defined session and marks it on the chart using lines and a linefill. The script highlights the session that the current bar is in, or if the current bar is outside a session, it highlights the next future session:

//@version=6 indicator("Highlight trading session", overlay=true) string sessionInput = input.session("0900-1700:1234567", "Session") if timeframe.in_seconds(timeframe.period) < 300 runtime.error("This script only supports chart timeframes of 5 minutes or more.") // Keep track of when the session started. int sessionStart = na, int sessionEnd = na int barsSince = ta.barssince(na(time(timeframe.period, sessionInput))) if barstate.islast if na(time(timeframe.period, sessionInput)) // If this bar is not in a session.. for i = 1 to 499 // ..search forward for the next session. int barStartTime = time(timeframe.period, sessionInput, bars_back = -i) if not na(barStartTime) and na(sessionStart) sessionStart := barStartTime if not na(sessionStart) and na(barStartTime) sessionEnd := time_close(timeframe.period, sessionInput, bars_back = -(i - 1)) break else // If this bar is in a session.. if na(barsSince) sessionStart := na else int startBack = barsSince - 1 // ..find the start of the session.. sessionStart := time(timeframe.period, sessionInput, bars_back = startBack) for i = 1 to 499 // ..and search forwards for the end of the session. int barStartTime = time(timeframe.period, sessionInput, bars_back = -i) if na(barStartTime) and na(sessionEnd) sessionEnd := time_close(timeframe.period, sessionInput, bars_back = -(i - 1)) break if not na(sessionStart) and not na(sessionEnd) // If we found the start and end of the session.. // ..set up the drawings var line left_line = na, line.delete(left_line) var line right_line = na, line.delete(right_line) var linefill session_fill = na, linefill.delete(session_fill) var label sessionLabel = na, label.delete(sessionLabel) // ..draw the lines left_line := line.new(x1 = sessionStart, y1 = low - syminfo.mintick, x2 = sessionStart, y2 = high + syminfo.mintick, xloc = xloc.bar_time, extend = extend.both, color = color.green, width = 2) right_line := line.new(x1 = sessionEnd, y1 = low - syminfo.mintick, x2 = sessionEnd, y2 = high + syminfo.mintick, xloc = xloc.bar_time, extend = extend.both, color = color.green, width = 2) // ..draw the fill session_fill := linefill.new(left_line, right_line, color.new(color.green, 90)) // .. and draw a label to show the session times. sessionLabel := label.new(x = sessionEnd, y = open, xloc = xloc.bar_time, yloc = yloc.price, text = "Start: " + str.format_time(sessionStart, "MM-dd HH:mm") + "\n" + "End: " + str.format_time(sessionEnd, "MM-dd

HH:mm"), color = color.new(color.green, 50), style = label.style_label_left, textcolor = chart.fg_color, size = size.normal)

Note that:

- The lowest timeframe to use, assuming that the session allows 24-hour trading 7 days a week, to search forward one whole day, is 5 minutes. Therefore, if the chart timeframe is lower than “5m”, the script throws a runtime error.
- The script uses time() and time_close() functions in loops, with positive and negative values for the `bars_back` parameter to search forwards and backwards in time.

How can I track highs and lows for periods within a bar?

Historical bars contain one set of OHLC data for each bar. To identify values and times that start or end *within* a chart bar, intrabar inspection is necessary, using data from lower timeframes.

For instance, on an hourly chart, the market open might occur mid-bar, making it hard to determine pre-market highs and lows. Pulling data from a lower timeframe, such as a 15-minute chart, provides a clearer view of that hour’s events.

The following example script calculates pre-market highs and lows from 15-minute chart data.

- First, we define a function called `hiLoBetweenTime()` to track the highs and lows within a session.
- Next, we modify the symbol’s ticker using ticker.modify() to include extended session data, ensuring that the script identifies premarket highs and lows even when the chart uses regular trading hours.
- Finally, we pass the `hiLoBetweenTime()` function as the `expression` of the request.security() function, with the modified ticker as the `symbol`. The request.security() function evaluates the `hiLoBetweenTime()` function *within the context of the lower timeframe* to calculate the session’s highs and lows.

//@version=6 indicator("Pre-market high/low", overlay = true) // Inputs string timeAllowedInput = input.session("0700-0930", "Allowed Hours") string lowerTfInput = input.timeframe("15", "Intrabar Resolution") string timezoneInput = input.string("Default", "Time zone", options = ["Default", "GMT-12", "GMT-11", "GMT-10", "GMT-9", "GMT-8", "GMT-7", "GMT-6", "GMT-5", "GMT-4", "GMT-3", "GMT-2", "GMT-1", "GMT-0", "GMT+1", "GMT+2", "GMT+3", "GMT+4", "GMT+5", "GMT+6", "GMT+7", "GMT+8", "GMT+9", "GMT+10", "GMT+11", "GMT+12", "GMT+13", "GMT+14"]) // @function Tracks the highest high and lowest low between specified session times. // @param sess (simple string) Session duration in the format "start time - end time". Example: "0930-1600" // @param timeZone (simple string) Time zone of the session in "GMT-0" format. Optional. Default is the symbol's time zone. // @returns ([float, float]) A tuple of the highest high and lowest low between the specified session times. hiLoBetweenTime(simple string sess, simple string timeZone = "Default") => var float hi = na, var float lo = na // Check to see if we are in allowed hours using session and time zone information. bool inSession = not na(time("", sess, timeZone == "Default" ? syminfo.timezone : timeZone)) if inSession if not inSession[1] // We are entering allowed hours; reset hi/lo. hi := high, lo := low else // We are in allowed hours; track high and low. hi := math.max(hi, high), lo := math.min(lo, low) [hi, lo] // Request data from lower timeframe using the `hiLoBetweenTime()` function. [highAtTime, lowAtTime] = request.security(ticker.modify(syminfo.tickerid, session.extended), lowerTfInput, hiLoBetweenTime(timeAllowedInput, timezoneInput)) // Plot the most recent value. plot(highAtTime, "High", color.green) plot(lowAtTime, "Low", color.red) // Raise error if lower tf is the same or greater than chart's tf. if

timeframe.in_seconds() <= timeframe.in_seconds(lowerTfInput) runtime.error("The lower timeframe for intrabar inspection must be lower than the chart's timeframe.")

Note that:

- The script raises an error using runtime.error() if the chosen lower timeframe for intrabar inspection is not shorter than the main chart's timeframe. Including error-checking such as this when working with timeframes makes the script more robust.

How can I detect a specific date and time?

The simplest way to detect a specific date and time is to check the time variable, which returns the timestamp of the current bar's open. If this matches a certain timestamp, then the script has detected that specific date and time.

The following example script colors the background green when the chart time exactly matches a date and time that the user inputs.

//@version=6 indicator("Exact date/time detector", overlay = true) bgcolor(time == input.time(timestamp("2025-03-24 10:00"), "Target Date/Time") ? color.new(color.green, 90) : na)

However, matching an exact date and time is not very useful. If the target time does not coincide with the opening of a candle on the chart timeframe, then that exact time never matches. For example, if the user specifies "2025-03-24 10:01" as the target time for the script above, then it never matches on any timeframe above one minute.

The following script behaves more intuitively, by coloring the background if a target time input by the user falls anywhere within a particular chart bar. The script colors one and only one bar even if the time does not match the open of a bar. The days, hours, and minutes are all tested separately to see if they match the target time, and these boolean conditions must all be true in order for the background to be colored. If the chart timeframe is too high, we set the corresponding lower time conditions to true so that they do not prevent the overall condition from firing. For example, if the timeframe is one day or above, we bypass the `isTargetHour` condition by setting it to true without evaluating the number of hours.

//@version=6 indicator("Date/Time detector", overlay = true) int targetInput = input.time(defval = timestamp("2025-03-24 10:00"), title = "Target Date/Time", confirm = true) bool isTargetYear = year == year(targetInput) bool isTargetMonth = month == month(targetInput) int targetDay = dayofmonth(targetInput) int dayOpen = dayofmonth // The built-in `dayofmonth` variable holds the day of the month of the bar open. int dayClose = dayofmonth(time_close) bool day_isBetweenOpenAndClose = (targetDay >= dayOpen) and (targetDay < dayClose) bool isTargetDay = timeframe.ismonthly or targetDay == dayofmonth or day_isBetweenOpenAndClose bool TF_isOneHourOrLess = timeframe.in_seconds() <= (60 * 60) bool TF_isAboveOneHour = timeframe.in_seconds() > (60 * 60) bool TF_isAboveOrEqualToOneHour = timeframe.in_seconds() >= (60 * 60) bool TF_isDayOrAbove = timeframe.isdwm int targetHour = hour(targetInput) int hourOpen = hour // The built-in `hour` variable holds the hour of the bar open. int hourClose = hour(time_close) bool hour_isBetweenOpenAndClose = (targetHour >= hourOpen) and (targetHour < hourClose) bool isTargetHour = TF_isDayOrAbove or (TF_isOneHourOrLess and (targetHour == hourOpen)) or (TF_isAboveOneHour and hour_isBetweenOpenAndClose) int minuteOpen = minute // The built-in `minute` variable holds the minute of the bar open. int minuteClose = minute(time_close) int targetMinute = minute(targetInput) bool minute_isBetweenOpenAndClose = (targetMinute >= minuteOpen) and (targetMinute < minuteClose) bool isTargetMinute = TF_isAboveOrEqualToOneHour or

minute_isBetweenOpenAndClose bool isTargetTime = isTargetYear and isTargetMonth and isTargetDay and isTargetHour and isTargetMinute bgcolor(isTargetTime ? color.new(color.green, 70) : na)

Note that:

- The script does not work on timeframes less than one minute, because the time input does not use seconds. The script could be extended to work on timeframes of one month and higher.
- We use `confirm = true` for the time input, which prompts the script user to select a date and time when the script first loads on the chart. The time selected in this way is always equal to the time of the selected bar's open, but the user can change it in the "Settings" window.
- We use the built-in variables year, month, dayofmonth, hour, and minute, which store the time of the current bar's open.
- We use the built-in functions year(), month(), dayofmonth(), hour(), and minute() to retrieve the time of the current bar's close, and to evaluate the time that is input by the user.

How can I know the date when the highest value was found?

To determine the date on which the highest value occurred, the following example script uses the ta.highest() function to find the highest value within a certain lookback period, and the ta.highestbars() function to find how many bars back this value occurred. If the highest point occurs on the current bar, the number of bars back, or *offset*, is zero. In our example script, when this condition is met, we update labels showing the time at the respective highest or lowest price.

//@version=6 indicator("Date of High/Low", overlay = true) int lengthInput = input.int(20) // Create labels. They do not display until the na values for the time and price are updated with new values. var label hiLabel = label.new(na, na, "", color = na, textcolor = color.lime) var label loLabel = label.new(na, na, "", color = na, textcolor = color.fuchsia) // Find the highest and lowest values over the input lookback and the bars on which they occurred. float hiValue = ta.highest(lengthInput), float loValue = ta.lowest(lengthInput) float hiOffset = ta.highestbars(lengthInput), float loOffset = ta.lowestbars(lengthInput) // If the high or low occur on the current bar, update the label and time variables. if hiOffset == 0 label.set_xy(hiLabel, bar_index, hiValue) label.set_text(hiLabel, str.format_time(time, "YYYY.MM.dd @ HH:mm:ss")) if loOffset == 0 label.set_xy(loLabel, bar_index, loValue) label.set_text(loLabel, str.format_time(time, "YYYY.MM.dd @ HH:mm:ss")) // Plot the highest and lowest values. plot(hiValue, "High", color.lime) plot(loValue, "Low", color.fuchsia)

Note that:

- We declare the high and low labels with var but supply na values for the x and y coordinates, so that the labels do not appear until these parameters are defined.
- When the bar offset functions return zero, we modify the respective label to display the bar time.
- The script plots the highest and lowest level for visual reference.

How can I detect bars opening at a specific hour?

The following example script demonstrates three methods for detecting bars that open at 18:00 hours:

Check the current bar's time

We check whether the built-in hour variable is equal to eighteen and the minute variable is equal to zero.

Detect the start of the session

Using the time() function with a time-based session string returns the time of the current bar if it is within the session time. If it's outside the session time, the function returns na. We determine the first bar that falls within the session by verifying that the current returned time is not na but was na one bar earlier.

Check current time against a timestamp

We generate a timestamp using the timestamp() function by specifying 18 hours and 0 minutes, as well as the current day, month, and year. We then verify whether the bar's time matches this timestamp.

//@version=6 indicator("18:00 hours", overlay = true) int timeInSession = time(timeframe = timeframe.period, session = "1800-1900") int timestamp1800 = timestamp(year, month, dayofmonth, 18, 00, 00) bool method1 = hour == 18 and minute == 00 bool method2 = not na(timeInSession) and na(timeInSession[1]) bool method3 = timestamp1800 == time plotchar(method1 ? 1 : na, "method1", "", location.abovebar, color.red, text="1", size=size.tiny) plotchar(method2 ? 2 : na, "method2", "", location.abovebar, color.orange, text="2\n", size=size.tiny) plotchar(method3 ? 3 : na, "method3", "", location.abovebar, color.yellow, text="3\n\n", size=size.tiny)

Note that:

- To ensure that the numbers do not plot on top of one another, we use the technique described in the FAQ entry How can I stack plotshape text?

Can I time the duration of a condition?

Normally, variables reset every time the script performs a new iteration. This process is called *rollback* and is described in the Execution model page of the User Manual.

By contrast, variables declared with varip *preserve* their values between realtime updates. This behavior allows scripts to track conditions during bars.

The following example script uses the `secondsSince()` function from the PineCoders' time library. This function uses variables declared with varip to track the number of seconds that elapse within a realtime bar while a condition remains true. Our script waits for price to move a user-defined number of ticks from the open, and then starts a timer. When the appropriate number of seconds elapses, it triggers an alert. A table displays these conditions in real time.

//@version=6 indicator("Condition timer", overlay = true) import PineCoders/Time/4 as PCtime string TICKS_TT = "The number of ticks price must move above the open to trigger the alert." string SEC_TT = "Seconds for which the condition must be continuously true before the alert triggers." string RESET_TT = "When checked, the duration resets every time a new realtime bar begins." int ticksInput = input.int(2, minval = 1, title = "Number Of Ticks From Open", tooltip = TICKS_TT) float secondsInput = input.int(20, minval = 10, title = "Seconds condition must last", tooltip = SEC_TT) bool resetInput = input.bool(true, title = "Reset timing on new bar", tooltip = RESET_TT) float targetTicks = open + (syminfo.mintick * ticksInput) bool targetTicksReached = close >= targetTicks // Calculate seconds elapsed since price reached the target. int secondsSinceTarget = PCtime.secondsSince(targetTicksReached, resetInput and barstate.isnew) bool timeAlert = secondsSinceTarget > secondsInput // Has the timer expired? string alertTime = str.format_time(secondsSinceTarget * 1000, "mm:ss") // Format a time string for the timer label. // Set the contents for the label depending on the stage of the alert timer. string alertString = "Waiting for price to reach

" + str.tostring(targetTicks) + "(" + str.tostring(ticksInput) + " ticks from " + str.tostring(open, format.mintick) + ")\nCurrent price: " + str.tostring(close, format.mintick) alertString := timeAlert ? "Timed Alert Triggered\n\n" + alertTime : targetTicksReached ? "Condition Detected...\n\nTimer count\n" + alertTime : alertString if barstate.islast var table statusTable = table.new(position.top_right, 1, 2, bgcolor = color.new(color.black, 70)) // Row 1: Combined tick conditions and current status string row1Text = "Target price: " + str.tostring(targetTicks, format.mintick) + " (+" + str.tostring(ticksInput) + " ticks from " + str.tostring(open, format.mintick) + ")\n" + (targetTicksReached ? "Price condition reached" : "Current price: " + str.tostring(close, format.mintick)) table.cell(statusTable, 0, 0, row1Text, text_color = chart.fg_color, bgcolor = targetTicksReached ? color.new(color.blue, 50) : color.new(color.black, 70)) if targetTicksReached // Only show the timer on row 2 when the price condition is reached. string row2Text = "Time required: " + str.tostring(secondsInput) + " seconds\n" + (timeAlert ? "TIMED ALERT TRIGGERED" : "Timer: " + alertTime) table.cell(statusTable, 0, 1, row2Text, text_color = chart.fg_color, bgcolor = timeAlert ? color.new(color.green, 50) : color.new(color.blue, 50)) if timeAlert // Fire alert if timer is triggered. alert("Timed Alert Triggered")

Note that:

- The `secondsSince()` function from the PineCoders' time library resets the timer if the supplied condition (in this case, the price condition) becomes false at any time.
- The `secondsSince()` function only works in real time, because it relies on intrabar updates, which are not present on historical bars. Therefore, it is possible for the timer to exceed a given specified time, especially on symbols with low liquidity, if a bar update does not occur around the time that the timer should finish.
- We include the option to reset the timer on every bar or not, for demonstration purposes. The price condition in this example script resets on each bar, but other implementations might use different conditions.

For more information on the `secondsSince()` function and the use of varip variables, consult the PineCoders' Using `varip` variables publication.

How can I identify the nth occurrence of a weekday in the month?

The following example script colors the background of the nth occurence of a user-configurable weekday. This identification can be useful for scheduled events that occur on specific weekdays, such as certain options expiry days.

// @version=6 indicator("N-th weekday of the month", overlay = true) int occurrenceInput = input.int(3, "Occurrence", 1, 5) // The occurrence of the weekday to check for in the current month. enum Wday Mon = "Monday" Tue = "Tuesday" Wed = "Wednesday" Thu = "Thursday" Fri = "Friday" wdInput = input.enum(Wday.Mon, "Weekday") int weekdayInt = wdInput == Wday.Mon ? 1 : wdInput == Wday.Tue ? 2 : wdInput == Wday.Wed ? 3 : wdInput == Wday.Thu ? 4 : wdInput == Wday.Fri ? 5 : na // @function Calculates the number of days in a specified month, accounting for leap years. // @param yearNumber (int) The year of the `monthNumber` month. Optional. Default is the current year. // @param monthNumber (int, optional) The month for which to find the number of days. Optional. Default is the current month. // @returns (int) The number of days in the `monthNumber` month of the `yearNumber` year. daysPerMonth(int yearNumber = year, int monthNumber = month) => bool isLeapYear = (yearNumber % 4 == 0 and yearNumber % 100 != 0) or (yearNumber % 400 == 0) int result = switch monthNumber == 2 => isLeapYear ? 29 : 28 => 31 - (monthNumber - 1) % 7 % 2 //@function Creates a timestamp representing the N-th occurrence of a specified weekday within a given month. //@param yearNumber (int) The year of the

timestamp. //@param monthNumber (int) The month of the timestamp. //@param weekdayNumber (int) The weekday of the timestamp. Can be a value between 1 and 7, where 1 is a Monday. //@param occurrence (int) The occurrence of the `weekdayNumber` to check for. //@returns (int) The timestamp at the N-th `occurrence` of the `weekdayNumber` in the month. weekdayOfMonth(int yearNumber, int monthNumber, int weekdayNumber, int occurrence) => int startTime = timestamp(yearNumber, monthNumber, 1) int daysInTheMonth = daysPerMonth(yearNumber, monthNumber) int endTime = timestamp(yearNumber, monthNumber, daysInTheMonth) int weekday = dayofweek(startTime) - 1 if syminfo.timezone == "Etc/UTC" and not timeframe.isintraday weekday -= 1 if weekday == 0 weekday := 7 int offset = weekdayNumber - weekday if offset < 0 offset := 7 + offset int result = startTime + (offset + 7 * (occurrence - 1)) * 86400000 if result > endTime result := na result int occurrenceTime = weekdayOfMonth(year(time_close), month(time_close), weekdayInt, occurrenceInput) bool isAtOccurrence = time_close[1] < occurrenceTime and time_close >= occurrenceTime plot(occurrenceTime, "Time of the N-th weekday occurrence", color.orange, display = display.data_window) plot(time_close, "Bar close time", display = display.data_window) bgcolor(isAtOccurrence ? color.purple : na, title = "Time condition highlight")

Note that:

- We use an enum input to specify the target weekday, and convert it to an integer using a switch structure.
- The custom function `daysPerMonth()` calculates the number of days in any given month, considering leap years.
- The `weekdayOfMonth()` function determines the timestamp for the nth occurrence of a selected weekday within a specific month. This function first calculates the start time of the month and assesses the total number of days it contains. The function accounts for the unique characteristic of futures symbols, which begin trading on Sunday night, potentially causing a one-day discrepancy in the calculation.
- We plot the time of the bar close and of the occurence to the Data Window.

How can I count down the remaining time in a bar?

Users can display a countdown on the price scale of the chart that shows the time remaining in each bar, by enabling the “Countdown to bar close” option in the “Scales and lines” section of the chart “Settings” menu. Our example script below displays a similar countdown timer. This script functions on intraday and “1D” timeframes only. For timeframes longer than “1D”, more complex logic is necessary. The script throws a runtime error if the chart timefrmae is greater than one day.

We subtract timenow from time_close to calculate the time remaining in the current bar, and then display the result in a table.

Because Pine scripts run only when there is a chart update, countdown timers do not usually update every second. The script refreshes more often on symbols with higher liquidity.

//@version=6 indicator("Countdown timer", overlay = true) if not ((timeframe.isdaily and timeframe.multiplier == 1) or timeframe.isintraday) runtime.error("This script functions only on daily or intraday timeframes.") // Inputs for bullish and bearish candle colors color bullishCandleColor = input.color(color.green, "Bullish Candle Color") color bearishCandleColor = input.color(color.red, "Bearish Candle Color") // Analyse the candle colors to see if black or white text has better contrast f_contrastColor(bgColor) => // Calculate luminance (relative brightness) using standard formula luminance = 0.2126 * color.r(bgColor) + 0.7152 * color.g(bgColor) + 0.0722 * color.b(bgColor) contrastColor = luminance > 127.5 ? color.black : color.white

color bullishContrastColor = f_contrastColor(bullishCandleColor) color bearishContrastColor = f_contrastColor(bearishCandleColor) int timeLeftInBar = time_close - math.min(timenow, time_close) var table timer = table.new(position = position.middle_right, columns = 1, rows = 1) if barstate.isfirst table.cell(timer, 0, 0, text_color = chart.fg_color, text_size = 11) else if barstate.islast string timeFormat = timeLeftInBar >= 60 * 60 * 1000 ? "HH:mm:ss" : "mm:ss" string countDown = str.format_time(timeLeftInBar, timeFormat, "UTC-0") table.cell_set_text(timer, 0, 0, countDown) bool isUpCandle = close >= open table.cell_set_text_color(timer, 0, 0, isUpCandle ? bullishContrastColor : bearishContrastColor) color bgcolor = isUpCandle ? bullishCandleColor : bearishCandleColor table.cell_set_bgcolor(timer, 0, 0, bgcolor)

Note that:

- We use the str.format_time() function to present the timestamp in a format similar to the price scale countdown timer. The function uses a conditional format so that we display hours only if the time remaining is one hour or greater.
- The built-in bar countdown uses a background the same color as the current candle. Pine scripts do not have access to the chart candle color settings, but if the user sets the correct colors in the script settings, our countdown displays in the same way.
- We use a custom function to analyze the luminance of the background colors and decide whether black or white text has the better contrast.

How can I get the week of the month?

Counting the week of the month in which the current bar occurs is not always straightforward. Some symbols trade overnight, so the time of the bar open is technically the previous day to the bar close, and sometimes the previous week. Because weekofyear, month, and similar variables use time (the time of the bar open) in their calculations, they can give unexpected results. The following example script, which uses these variables, gives unexpected results on some symbols such as US futures charts. For example, it increments the number of weeks on a Tuesday if the first trading day of the month is a Monday.

// @version=6 indicator("Week of month demo 1") bool isNewWeek = ta.change(weekofyear) != 0 bool isNewMonth = ta.change(month) != 0 var int weekCount = na weekCount := timeframe.ismonthly ? na : isNewMonth ? 1 : isNewWeek ? weekCount + 1 : weekCount plot(weekCount, "Week of month", chart.fg_color) bgcolor(isNewWeek ? color.new(chart.fg_color, 90) : na) bgcolor(isNewMonth ? color.new(color.lime, 80) : na)

The following amended script version uses the functions weekofyear() and month() instead of the equivalent variables, and evaluates the time_tradingday variable, which, unlike time, returns the beginning time *of the trading day* that the bar belongs to. Using this variable, and fixing the time zone to “UTC”, avoids problems with sessions that span days.

// @version=6 indicator("Week of month demo 2") bool isNewWeek = ta.change(weekofyear(time_tradingday, "UTC")) != 0 bool isNewMonth = ta.change(month(time_tradingday, "UTC")) != 0 var int weekCount = na weekCount := timeframe.ismonthly ? na : isNewMonth ? 1 : isNewWeek ? weekCount + 1 : weekCount plot(weekCount, "Week of month", chart.fg_color) bgcolor(isNewWeek ? color.new(chart.fg_color, 90) : na) bgcolor(isNewMonth ? color.new(color.lime, 80) : na)

While the preceding examples are useful for understanding the difference between time and time_tradingday, the simplest solution is to use the timeframe.change() function, which detects changes in a given timeframe. We can rewrite the boolean conditions in our example script as follows:

bool isNewWeek = timeframe.change("1W") bool isNewMonth = timeframe.change("1M")

Note that:

- All versions of the script reset the number of weeks at the beginning of the month, then increment the count each new week.
- These scripts do not plot anything on timeframes of one month or greater.
- In Pine V5 and below, scripts can assign values of type “float” and “int” to a boolean variable. A value of zero is equivalent to false, and any other value to true. In later versions of Pine, scripts can assign only boolean values to boolean variables. So in V5, this code works: `bool isNewMonth = ta.change(month)` but in later versions this equivalent code is needed: `bool isNewMonth = ta.change(month) != 0`.

Variables and operators

What is the variable name for the current price?

In Pine Script®, the close variable represents the current price. It provides the *closing price* of each historical bar, and, for indicator scripts, the *current price* of the most recent realtime bar. The close value of an open bar can change on each tick to reflect the latest price.

Strategy scripts usually execute only once on each historical *and* realtime bar, at the bar close. Consequently, during a realtime bar, the close variable holds the *previous* bar's closing price. However, if a script sets the `calc_on_every_tick` parameter of the strategy() declaration statement to `true`, the strategy executes with each price change of the realtime bar, like indicators do. As a result, close holds the latest realtime price update.

To reference the closing price of the previous bar, use `close[1]`. Learn more about using square brackets to reference previous values in the history-referencing operator section.

Why declare variables with the `var` keyword?

The var keyword is useful for storing data across multiple bars. By default, the value assigned to a variable is *reset* and calculated again on each new bar. This process is called rollback.

If a script declares a variable with the var keyword, this *persistent* variable is initialized only *once*. Variables declared in the global scope are initialized on the first bar. Variables declared in a local block are initialized the first time that the local block executes. After its initial assignment, a persistent variable maintains its last value on subsequent bars until it is reassigned a new value.

In the example below, we demonstrate how to accumulate volume across bars, comparing an ordinary and a persistent "float" variable.

//@version=6 indicator("Var keyword example") // Declare and initialize a persistent variable by using `var`. var float a = 0 // Declare and initialize a normal float variable. float b = 0 // Reset the values of a and b whenever a new day begins. if timeframe.change("D") a := 0 b := 0 // Add the current volume to both a and b. a += volume b += volume // Plot the values of `a` and `b`. The value of `a` accumulates over time; `b` is reinitialized at every bar. plot(a, "a", close > open ? #089981 : #f23645, style = plot.style_columns) plot(b, "b", color.yellow)

What is `varip` used for?

The varip keyword declares variables whose values persist *within the same realtime bar*. This contrasts with the typical mode of Pine's execution model, where variables are reset to their last committed value with *each realtime script execution*, potentially many times in each bar.

Recall that the var keyword allows a variable to retain its value from bar to bar – however, the value still resets on each script execution *within* a bar. The varip keyword takes this persistence a step further and escapes the rollback process, or re-initialization, on *each price update* within the same realtime bar.

As a result, varip (which stands for "variable intrabar persist") variables can perform calculations that span *across executions* in the same bar. For example, they can track the number of realtime updates that occur

within a realtime bar.

It's important to note that varip only affects the behavior of code on realtime bars, not historical ones. Therefore, backtest results on strategies based on varip variables might not accurately reflect the behavior of those historical bars. Similarly, calculations on historical bars won't reproduce the script's realtime behavior.

To distinguish between var and varip, add the following script to a live market symbol. With realtime updates, the varip plot increments within a bar on each price update, whereas the var plot stays constant within a bar:

//@version=6 indicator("varip vs var demo") // `var` : Retains value across bars, resets on intrabar price updates. // 'varip': Retains value across bars and across intrabar price updates within a realtime bar. var int varCount = -1 varip int varipCount = -1 // Increment `varCount` on each bar and `varipCount` on each intrabar price update. varCount += 1 varipCount += 1 // Plot values for comparison. plot(varCount, "var counter", color.fuchsia, 4) plot(varipCount, "varip counter", color.lime)

Note that:

- Both plots in the above script are the *same* for historical bars, because there are no intrabar updates on historical bars.

What's the difference between `==`, `=`, and `:=`?

The = operator declares and initializes variables, assigning a specific value to a named variable. For example, `a = 0` sets the variable `a` to hold the value 0.

The := property reassigns values to existing variables. For instance, if a script declared `int a = 1`, a subsequent line `a := 2` updates the value of `a` to `2`, which is possible because integer variables are *mutable*, or changeable.

Finally, the == operator is a comparison operator. It checks the equality between two values, returning a Boolean (true/false) result. For instance, `a == b` is true if `a` and `b` hold the same value. The opposite operator is !=, which is true if the two variables are *not* equal.

The following script initializes two variables `a` and `b`, reassigs `a`, and then performs and plots equality comparisons.

//@version=6 indicator("Variable operators demo", overlay = true) // Define two variables `a` and `b` using `=`, representing the high and low of each bar. float a = high float b = low // Define the initial line color as lime color lineColor = color.lime // When there are fewer than 10 bars left on the chart, use `:=` to update `a` to `b` and change the line color. if last_bar_index - bar_index < 10 a := b lineColor := color.fuchsia // Plot the variable 'a' to visualize its change in value. // Initially, 'a' represents the 'high' of each bar. // If there are fewer than 10 bars remaining in the chart, 'a' is updated to represent the 'low' of each bar. plot(a, "Our line", lineColor, 2) // Plot a checkmark character whenever `a` is equal to `b`. plotchar(a == b, "a equals b", "✅", location.bottom) // Plot a cross character whenever `a` is not equal to `b`. plotchar(a != b, "a does not equal b", "❌", location.bottom)

Can I use the `:=` operator to assign values to past values of a series?

Historical values are fixed and cannot be changed. Just as we can't alter the past in real life, scripts are unable to modify historical values in a series, because they are read-only. For example, the following script generates an error:

//@version=6 indicator("Changing historical values demo", overlay = true) // Initialize a variable to hold the value of the current bar's high. series float a = high // Reassign the *previous* value of the series `a` to hold the low of the current bar. a[1] := low // This line causes a compilation error. plot(a, color = chart.fg_color, linewidth = 3)

However, scripts *can* assign or reassign a value to the current instance of a series, and assign a historical value of a series to a variable. The following version of our script works without error.

//@version=6 indicator("Changing historical values demo", overlay = true) // Initialize a variable to hold the value of the current bar's high. series float a = high // Reassign the *current* value of the series `a` to hold the high of the *previous* bar. a := high[1] plot(a, color = chart.fg_color, linewidth = 3)

Why do the OHLC built-ins sometimes return different values than the ones shown on the chart?

The OHLC (Open, High, Low, Close) values displayed on the chart and the values returned by the built-in OHLC variables open, high, low, close can differ. This is because data feeds can contain price points that exceed a symbol's defined *tick precision*. While visually, chart prices are always rounded to tick precision, the built-in variables maintain their original, unrounded values.

For instance, if an exchange feed provides a closing price of 30181.07, which is more precise than the symbol's 0.1 tick size, the chart displays a rounded value of 30181.1, whereas the built-in variable holds the unrounded value of 30181.07.

Subtle differences, while not immediately obvious, can lead to significant outcomes, especially in scripts requiring precise calculations or when diagnosing unexpected behaviors in scripts. An example of this is in detecting crossover events. Discrepancies between unrounded and rounded values can cause scripts to identify crossover events in one scenario but not in the other.

One way to mitigate this issue is to round the OHLC built-in variables to the nearest tick size before using them in calculations. The script below highlights discrepancies between actual OHLC values and their rounded counterparts, visually indicating any differences by coloring the background red:

//@version=6 indicator("Different tick values example", overlay = true, precision = 10) // @function Rounds each OHLC value to the nearest minimum tick size. // @returns A tuple containing the rounded values. OHLCToMinTick() => [math.round_to_mintick(open), math.round_to_mintick(high), math.round_to_mintick(low), math.round_to_mintick(close)] //@function Checks whether two float values are equal or not. //@param v1 (series float) The first value to compare. //@param v2 (series float) The second value to compare. //@returns The color blue if the values are equal or red otherwise. getTickColor(series float v1, series float v2) => color result = v1 != v2 ? color.red : color.blue // Round each OHLC value to the nearest mintick size. [o, h, l, c] = OHLCToMinTick() // Plot the original and rounded values of each OHLC component in the data window. // If a value and its rounded counterpart are not equal, color the plot red. Otherwise, color it blue. plot(o, "o", getTickColor(o, open), display = display.data_window) plot(open, "open", getTickColor(o, open), display = display.data_window) plot(h, "h", getTickColor(h, high), display = display.data_window) plot(high, "high", getTickColor(h, high), display = display.data_window) plot(l, "l",

getTickColor(l, low), display = display.data_window) plot(low, "low", getTickColor(l, low), display = display.data_window) plot(c, "c", getTickColor(c, close), display = display.data_window) plot(close, "close", getTickColor(c, close), display = display.data_window) // If any of the original and rounded values of OHLC components are not equal, set the background color to red. bgcolor(o != open or h != high or l != low or c != close ? color.new(color.red, 90) : na)

Why do some logical expressions not evaluate as expected when `na` values are involved?

In Pine Script, every type of variable can take an na value – *except* Boolean variables, which can only be true or false. Here, na stands for “not available”, and signifies the absence of a value, similar to NULL in other programming languages.

Although Boolean values themselves cannot be na, logical expressions that evaluate to true or false can depend on variables of other types that *can* be na.

This behavior can cause unexpected outcomes because any valid logical comparison that includes na values *always returns false*.

The following example script evaluates a single comparison where one value is always na. The user can choose which comparison to evaluate from a set. A label displays the chosen comparison and its result.

//@version=6 indicator("`na` example") string conditionInput = input.string("a != b", "Condition", options=["a != b","a == b", "a > b", "a < b"]) int a = 1 int b = na bool condition = switch conditionInput "a != b" => a != b "a == b" => a == b "a > b" => a > b "a < b" => a < b if barstate.islastconfirmedhistory string conditionText = condition ? "true" : "false" label.new(x = bar_index, y = high, text = "a = 1\nb = na\n" + conditionInput + ": " + conditionText, color = condition ? color.green : color.red, textcolor = color.new(chart.fg_color, 0), style = label.style_label_down, size = size.large)

To avoid unwanted false negatives, write code that checks for na values and, if necessary, replaces them. For a discussion of na values and how to manage them, see the na value section of the User Manual.

Visuals

Why can't I use a plot in an `if` or `for` statement?

In Pine Script®, scripts cannot place plot() calls directly within if or for statements – or in any other local scopes. The compiler needs to know about all plots during script compilation.

However, scripts *can* plot values conditionally, by changing the series or color of the plot.

Our example script plots two ALMA moving averages only when the shorter average is below the longer one. It fills between the two averages on every bar, but the fill color is na unless the shorter average is above or equal to the longer one.

//@version=6 indicator("Conditional plot example", "", true) // Calculate two ALMAs. float ma1 = ta.alma(close, 21, 0.85, 6) float ma2 = ta.alma(close, 50, 0.85, 6) // Calculate whether the MAs are in bullish or bearish order. var bool areBullCrossed = false if ma1 > ma2 areBullCrossed := true else areBullCrossed := false // Plot the MAs to the Data Window for use in the fill. p1 = plot(ma1, "MA 1: Conditional fill", display = display.data_window, editable = false) p2 = plot(ma2, "MA 2: Conditional fill", display = display.data_window, editable = false) // Fill the MAs with color only when the MAs are in bullish order. fill(p1, p2, areBullCrossed ? color.aqua : na) // Plot the MAs only when they are in bearish order. plot(not areBullCrossed ? ma1 : na, "MA 1: Conditional plot", color.fuchsia, 3, plot.style_linebr) plot(not areBullCrossed ? ma2 : na, "MA 2: Conditional plot", color.new(color.fuchsia, 70), 5, plot.style_linebr)

Note that:

- The script updates the Boolean variable that describes whether the moving averages are in bullish order – and thereby controls the plots and fill – in a *local scope*. However, the script declares it in the *global scope* so that the fill() and plot() calls can use it.
- The moving averages are each plotted twice. The plots that display in the Data Window must be assigned to variables so that the fill() function call can reference them. The fill() function cannot use the later plots, which display on the chart, because they have na values when the fill color is not na.
- We use `plot.style_linebr` as the argument for the `style` parameter of the plot() function so that the plot does not span bars with an na plot value.

Can I plot diagonals between two points on the chart?

Scripts can plot diagonal lines between two points on a chart by using plots or line objects.

Using plots

The plot() function connects consecutive data points with straight lines.

The default value of the `style` argument for the plot() function is `plot.style_line`. This style of plot connects the plotted points on either side of bars that have na series values to each other with a line. If the points are at different heights, the lines are diagonal.

The functions ta.pivotlow() and ta.pivothigh() return na for all bars except those with identified pivots. The following example script draws diagonal lines joining pivot highs and lows using plot().

//@version=6 indicator("Diagonal plots", overlay = true) // User input to define the number of bars to look back and forwards to define pivots. int pivotBarsInput = input.int(5, "Pivot Bars") // Find the value of pivot lows and highs. The value is `na` if the current bar is not a pivot. float pivotLow = ta.pivotlow(pivotBarsInput, pivotBarsInput) float pivotHigh = ta.pivothigh(pivotBarsInput, pivotBarsInput) // Define the offset for the plot. Negative offset values plot on bars earlier than the current bar. pivotOffset = -1 * pivotBarsInput // Join the pivot values with plotted lines. plot(pivotLow, "Pivot Low", color.fuchsia, offset = pivotOffset) plot(pivotHigh, "Pivot High", color.teal, offset = pivotOffset) // Place a circle on each pivot to emphasize the points being connected, offsetting it back to the pivot bar. plot(pivotLow, "Pivot Low", color.fuchsia, 3, plot.style_circles, offset = pivotOffset) plot(pivotHigh, "Pivot High", color.teal, 3, plot.style_circles, offset = pivotOffset)

Note that:

- The ta.pivotlow() and ta.pivothigh() functions confirm pivot highs and lows only after a specified number of bars, so the script must offset the plotted lines by the same number of bars. Plot offsets cannot change during script execution.

Using lines

Line objects provide more flexibility than plots. The plot() function can plot a line, symbol, or area only at the bar on which the script is executing (or at a *fixed* offset from it). In contrast, line objects can be created on *any* bar. Similarly, whereas plots are fixed once the bar closes, line properties can be updated at any time.

The following example script demonstrates these advantages. Like the example script in the previous section, Using plots, this script draws lines between pivot highs and lows. In this case, however, the pivot highs and lows are confirmed an *unpredictable* number of bars afterwards, so plots are completely unsuitable for drawing lines between them. Line drawings are suitable because scripts can offset lines an arbitrary and dynamic number of bars into the past. The script below also retroactively *changes* the color of drawn lines, which are initially white, depending on whether the *next* line of the same type slopes up or down. Such updates are not possible with plots.

//@version=6 indicator("Diagonal lines", overlay = true, max_lines_count = 500) int lookbackInput = input.int(defval=10, title="Lookback Bars", minval=1) enum PivotState // Stores the state of the pivot detection. none // Start from zero. hadPivot // Have we found a pivot? hadCounter // Have we had a down candle for a high or a up candle for a low? var PivotState highState = PivotState.none, var PivotState lowState = PivotState.none // Create state objects. updatePivotState(PivotState state, bool foundPivot, bool counterCandle, bool newExtreme, bool isConfirmedPivot) => if newExtreme // Start again if we make a new high/low. PivotState.none else switch state // Progress the state when we get the right events. PivotState.none => foundPivot ? PivotState.hadPivot : state PivotState.hadPivot => counterCandle ? PivotState.hadCounter : state PivotState.hadCounter => isConfirmedPivot ? PivotState.none : state => state // Default: persist. // Calculate the events. float highestHigh = ta.highest(high, lookbackInput) float lowestLow = ta.lowest(low, lookbackInput) bool foundHigh = high[1] == highestHigh[1] and high < highestHigh[1] bool foundLow = low[1] == lowestLow[1] and low > lowestLow[1] bool newHigh = high >= highestHigh bool newLow = low <= lowestLow // Update the state. bool isConfirmedHigh = highState == PivotState.hadCounter and close > open bool isConfirmedLow = lowState == PivotState.hadCounter and close < open highState := updatePivotState(highState, foundHigh, close < open, newHigh, isConfirmedHigh) lowState := updatePivotState(lowState, foundLow, close > open, newLow, isConfirmedLow) // Store the price and bar index of the pivots. var float pivotHighPrice = na, var float pivotLowPrice = na var int pivotHighBarIndex = na, var int pivotLowBarIndex = na if foundHigh pivotHighPrice := high[1] pivotHighBarIndex := bar_index[1] if foundLow pivotLowPrice := low[1] pivotLowBarIndex := bar_index[1] //

Store the current and previous high and low points. var chart.point prevHighPivot = na, var chart.point highPivot = na var chart.point prevLowPivot = na, var chart.point lowPivot = na var line pivotHighLine = na, var line pivotLowLine = na // Draw the lines drawPivotLine(chart.point prevPivot, line prevLine, int barIndex, float price) => if not na(prevPivot) line.set_color(prevLine, price < prevPivot.price ? color.red : color.green) newLine = line.new(x1=prevPivot.index, y1=prevPivot.price, x2=barIndex, y2=price, color=color.white, width=2) newLine if isConfirmedHigh prevHighPivot := highPivot // Shift current to previous highPivot := chart.point.from_index(pivotHighBarIndex, pivotHighPrice) pivotHighLine := drawPivotLine(prevHighPivot, pivotHighLine, pivotHighBarIndex, pivotHighPrice) if isConfirmedLow prevLowPivot := lowPivot // Shift current to previous lowPivot := chart.point.from_index(pivotLowBarIndex, pivotLowPrice) pivotLowLine := drawPivotLine(prevLowPivot, pivotLowLine, pivotLowBarIndex, pivotLowPrice)

Note that:

- This script creates custom pivots using a combination of a lookback similar to that used in Williams Fractals and price confirmation. Potential pivots require a candle in the counter-trend direction *followed by* a candle in the trend direction in order to confirm. This means that the candle on which pivots confirm cannot be predicted, unlike Williams pivots.
- Because the pivot is an arbitrary number of bars back from the confirming candle, we cannot plot circles to highlight the pivots, as we did for the script example in the previous section.
- We use an enum to store the state of the pivot confirmation process. This, together with a set of explicit rules for changing the state, forms a simple state machine. Such constructs can be easier to debug than maintaining and resetting multiple persistent global variables that depend on each other.
- Scripts can create only a certain number of lines. The limit is set by the `max_lines_count` parameter of the indicator() or strategy() declaration. The default is 50, and the maximum is 500 per script.

How can I plot a line with gaps?

Scripts can plot lines with gaps on specific bars by setting the argument of *either* the `series` parameter or `color` parameter of the plot() function to na for some bars.

Note that this is different to plotting dashed lines using `linestyle = plot.linestyle_dashed` in the plot() call, which plots a line with gaps at regular intervals that do not correspond to particular bars.

Using na values for the color or series enables scripts to omit a plot on an *arbitrary* selection of bars. These two methods have different effects:

- An na *color* value omits the plotted line from the most recent bar with a non-na value to the first bar with an na value. The size of the gap is equal to the number of consecutive bars with na values.
- An na *series* value omits the plot point for each bar with an na value, effectively **increasing the size of the gap by one bar**.

The following example script shows the effect of these two methods. It plots two straight lines above the chart bars, one using each method. At a configurable interval, the `series` or `color` argument is na, and the background color changes for that bar.

//@version=6 indicator("Lines with gaps demo", "", true) var float lowerPlot = na, var float upperPlot = na bool change = bar_index % input.int(5, minval = 2, title = "Bar interval") == 0 float atr = ta.atr(14) if change lowerPlot := high + atr upperPlot := high + (1.5 * atr) plot(lowerPlot, "Plot with Varying Color", color = change ? na : color.red, linewidth = 2, style = plot.style_line) plot(change ? na : upperPlot, "Plot with Varying Series", color = color.blue, linewidth = 2, style = plot.style_linebr) bgcolor(change ? color.new(color.gray, 80) : na)

Note that:

- The style for the plot with na `series` values must be `plot.style_linebr` in order to show gaps. The default value of `plot.style_line` fills in the gaps.

How do I plot a line using start/stop criteria?

To plot a line based on start and stop criteria, consider the following structured approach:

1. Define start and stop conditions.
2. Control variables for *when* and *where* to plot.
3. Choose a plotting style. Use na for either the `series` or `color` parameter of the plot() function. See the section How can I plot a line with gaps? above for examples.
4. Optionally, use debug. For example, plot logical states using the plotchar() function.

In the following example script, the start condition is the detection of a new pivot. The start condition turns on the `doPlot` Boolean flag that controls when to plot, and captures the value to plot in the `savedValue` "float" variable. The stop condition is price closing above the pivot level. The script uses a bar timer as an extra stop condition. Either stop condition turns off the flag and *resets* the plot value to na. The script plots debug characters for all logical conditions.

//@version=6 indicator("Starting and stopping a plot", overlay = true) // Set the maximum number of bars for a line. int expiryBarsInput = input.int(50, "Maximum bars to plot line", minval = 0) var bool doPlot = false // Whether to plot the line. var float savedValue = na // The value to plot. var int timerStart = na // The bar on which the line started. // Define conditions for the start and end of the line plot. bool startCondition = not na(ta.pivothigh(close, 5, 2)) bool closeAboveLevel = ta.crossover(close, savedValue) bool hasExpired = not (bar_index < timerStart + expiryBarsInput) bool stopCondition = closeAboveLevel or hasExpired float atr = ta.atr(14) if startCondition and not doPlot // If this is the start of a *new* plot, savedValue := high + (atr /2) // set the value to plot, timerStart := bar_index // and start the timer. // Start, stop, or persist the state that controls the plot. doPlot := startCondition ? true : stopCondition ? false : doPlot // Plot the line if `doPlot` is true. plot(doPlot ? savedValue : na, "Saved Value", color.white, style = plot.style_linebr) // Debug: Plot the logical events. plotchar(startCondition, "startCondition", "►", location.abovebar, color.green, size = size.tiny) plotchar(closeAboveLevel, "closeAboveLevel", "◄", location.belowbar, color.red, size = size.tiny) plotchar(hasExpired, "hasExpired", "✕", location.abovebar, color.gray, size = size.tiny)

Note that:

- The order in which scripts turn conditions on and off is important when dealing with persistent states, both in terms of the order in the script and within ternary conditions.

How can I plot a support or trend line?

Support is a horizontal zone on a chart where analysts consider that a declining price is likely to turn upwards. Conversely, *resistance* is a horizontal area from which a rising price is likely to turn downwards. Trend lines are usually diagonal lines that function as support or resistance.

Different analysts – and different Pine scripts – understand and implement support, resistance, and trend lines differently. In the sections below, we provide some simple examples.

Plotting support and resistance

The following example script tracks levels of support and resistance until price action breaks them. The script uses the `ta.pivot*()` built-in functions to detect pivot highs and lows, then draws horizontal lines from these points. This script visualizes lows as support (green lines) and highs as resistance (red lines) for simplicity. If the close of a bar crosses a line, the script stops extending that line:

//@version=6 indicator("Support and resistance demo", "", true, max_lines_count = 500) color hiPivotColorInput = input.color(color.fuchsia, "High pivot color") color loPivotColorInput = input.color(color.lime, "Low pivot color") int pivotSizeInput = input.int(5, "Pivot lookback/look forward") int maxLineLengthInput = input.int(100, "Maximum line length", minval = 2) // @function Extends every line in `lineArray` until price crosses it or it exceeds `maxLength` in length. // @param lineArray (array<line>) An array storing the lines to check. // @param maxLength (int) The maximum length a line can have. // @returns (void) The function has no explicit return. checkLinesForBreaches(array<line> lineArray, int maxLength) => // If there are no lines, the `from` value is `na` and we do not loop. int fromValue = lineArray.size() > 0 ? lineArray.size() - 1 : na // We loop the array in reverse to avoid errors even when we remove the first element in the array. for i = fromValue to 0 // Check each line crosses and length. line eachLine = lineArray.get(i) float linePrice = eachLine.get_price(bar_index) bool lineWasCrossed = math.sign(close[1] - linePrice) != math.sign(close - linePrice) bool lineIsTooLong = bar_index - eachLine.get_x1() > maxLength // Set the rightmost point of each line to the current bar. eachLine.set_x2(bar_index) // Set lines inactive if they have been crossed or are too long. if lineWasCrossed or lineIsTooLong // Stop the line from extending to the right. eachLine.set_extend(extend.none) // Remove the line from the array. The line stays on the chart but does not extend on further bars. lineArray.remove(i) // Arrays of active lines. var array<line> hiPivotLines = array.new<line>() var array<line> loPivotLines = array.new<line>() // Detect new pivots and record their values. float hiPivot = ta.pivothigh(pivotSizeInput, pivotSizeInput) float loPivot = ta.pivotlow(pivotSizeInput, pivotSizeInput) // Create new lines on new pivots and add to arrays. x1 = bar_index - pivotSizeInput if not na(hiPivot) hiPivotLines.push(line.new(x1, hiPivot, bar_index, hiPivot, extend = extend.right, color = hiPivotColorInput)) if not na(loPivot) loPivotLines.push(line.new(x1, loPivot, bar_index, loPivot, extend = extend.right, color = loPivotColorInput)) // Extend lines if they are still active. checkLinesForBreaches(hiPivotLines, maxLineLengthInput) checkLinesForBreaches(loPivotLines, maxLineLengthInput)

Note that:

- When price crosses a line, it becomes inactive. The script no longer updates inactive lines and does not extend them to the right.
- We store lines in one of two arrays, and remove lines from their array if they become inactive. Removing an object from an array does *not* delete the object.

Plotting trend lines

The following example script uses the `ta.pivot*()` built-in functions to detect pivot highs and lows, and then draws lines that connect the two most recent pivots of the same type. The lines extend indefinitely to the right. If the script draws a new line that causes the total number of lines to exceed a specified maximum number, it deletes the oldest line:

//@version=6 indicator("Simple trend lines demo", overlay = true, max_lines_count = 500) int pivotSizeInput = input.int(20, "Pivot lookback/look forward", minval = 1, maxval = 50) int maxLinesInput = input.int(2, "Quantity of lines to track per pivot", minval = 0, maxval = 50) color loTrendColorInput = input.color(color.lime, "Low pivot") color hiTrendColorInput = input.color(color.fuchsia, "High pivot") // @function Draws a line from the previous pivot when a new pivot of the same type occurs. // Keeps the

number of lines under a specified number by deleting excess lines. // @param lineArray (array<line>) An array to store and manage the line objects. // @param pivotValue (float) The pivot price when a pivot is found. // @param pivotSize (simple int) The size of the pivot lookback/look forward. // @param maxLinesCount (simple int) The maximum number of lines to keep. // @param lineColor (color) The color for the line. // @returns (void) The function has no explicit return. queueLine(array<line> lineArray, float pivotValue, simple int pivotSize, simple int maxLinesCount, color lineColor) => var array<chart.point> pointArray = array.new<chart.point>(2, chart.point.new(na, na, na)) if not na(pivotValue) pointArray.push(chart.point.from_index(bar_index - pivotSize, pivotValue)) if pointArray.size() > 2 pointArray.shift() chart.point firstPoint = pointArray.first() chart.point secondPoint = pointArray.last() line ln = line.new(firstPoint, secondPoint, extend = extend.right, color = lineColor, style = line.style_dotted) lineArray.push(ln) if lineArray.size() > maxLinesCount line.delete(lineArray.shift()) // Initialize two empty arrays for the high and low trend lines on the first bar. var array<line> hiLinesArray = array.new<line>() var array<line> loLinesArray = array.new<line>() // Detect new pivots. float hiPivot = ta.pivothigh(pivotSizeInput, pivotSizeInput) float loPivot = ta.pivotlow(pivotSizeInput, pivotSizeInput) // Draw new lines between the two most recent pivots when a pivot occurs, and add them to the line array. // The number of lines is limited to the specified number, after which lines are removed from the array *and* deleted. queueLine(hiLinesArray, hiPivot, pivotSizeInput, maxLinesInput, hiTrendColorInput) queueLine(loLinesArray, loPivot, pivotSizeInput, maxLinesInput, loTrendColorInput) // Visually highlight the pivot points with a dot. plot(hiPivot, "Pivot High", hiTrendColorInput, 3, plot.style_circles, offset = -pivotSizeInput) plot(loPivot, "Pivot Low", loTrendColorInput, 3, plot.style_circles, offset = -pivotSizeInput)

Note that:

- For simplicity, we do not deactivate lines when price crosses them. For an example of how to do this, see the example script from the previous section, Plotting support and resistance.
- We store the lines in arrays, which makes it easier to manage them. In this script, unlike in the example script in the previous section, we delete the line object at the same time as we remove it from the array.

How can I use colors in my indicator plots?

The strategic use of color in indicator plots helps comprehension, pattern and trend recognition, and differentiation of categories or values. For example, different hues can represent different data thresholds, and gradients can indicate increases or decreases in values. When a script transforms numbers into colors in an intelligent way, it enhances the user's ability to spot anomalies, trends, and significant data points. Here are some Pine features that script authors can use to work with colors:

Predefined colors

Without needing to specify hexadecimal or RGB (Red Green Blue) values, programmers can use predefined color constants, such as color.red. In the Pine Editor, clicking the automatically generated color swatch next to a color constant opens up a color picker; choosing a different color from the picker updates the code that defines the color.

Custom colors

For a more personalized appearance, programmers can create custom colors, either by specifying a hexadecimal code, or by using the color.rgb() function to specify the RGB values. For the hexadecimal and RGB equivalents of the built-in color constants, see the table in the Constant colors section.

Transparency settings

The color.new() function can create colors with a specific transparency. The `transp` parameter can even take series values, meaning that a single color declaration can provide a color with dynamic transparency.

Conditional coloring

Scripts can assign different colors to variables based on logical conditions. Programmers can use conditional colors to color plots, fills, shapes, drawings, or the chart background differently on different bars.

Gradient transitions

The color.from_gradient() function creates color gradients that can highlight shifts in data values while ensuring a smooth transition between colors. For detailed guidance and innovative examples on implementing gradients, consult the Color Gradient Framework by PineCoders.

Fills

Scripts can create shaded areas between lines, plots, or hlines, as well as within boxes and polylines. Fills can be especially useful for highlighting ranges, zones, or contrasts.

In our example script below, we use various color display techniques such as conditional colors, fills, and gradients to depict areas of rising or falling values, as well as overbought and oversold levels. The script automatically recognizes whether the user's chart background is light or dark, and adjusts the color scheme accordingly. For a light theme, it captures the ambiance of a day at the beach with coastal teal and coral hues, while for a dark theme, it reflects the vibrant neon hues of city nightlife:

//@version=6 indicator("Using colors in Pine", explicit_plot_zorder = true) float tsi = ta.tsi(close, 13, 23) * 100, float tsl = ta.ema(tsi, 13) // Calculate TSI and its EMA. // @function Determines if a given background color corresponds to a light theme based on its brightness level. // @param bgColor (color) The background color to check. Optional. Default value is the chart's background color. // @returns (bool) True if the background color is "light" (has a brightness greater than 0.5), otherwise false. isLightTheme(color bgColor = chart.bg_color) => float r = color.r(bgColor) float g = color.g(bgColor) float b = color.b(bgColor) float brightness = (r + g + b) / (3 * 255) bool isLight = brightness > 0.5 var bool isLightTheme = isLightTheme() // Define color schemes based on whether the theme is light or dark. color tsiDnUpColor = isLightTheme ? #1E90FF : #BA33FF, color tsiDnDnColor = isLightTheme ? #FF6B6B : #8100FF color tsiUpUpColor = isLightTheme ? #00CED1 : #1FE0F3, color tslBullColor = isLightTheme ? #00CED1 : #1FE0F3 color tslBearColor = isLightTheme ? #1E90FF : #0088A3, color tsiBullColor = isLightTheme ? #FFD700 : #33FF57 color tsiBearColor = isLightTheme ? #FF6B6B : #00940D, color tsiUpDnColor = isLightTheme ? #FFD700 : #FF21D4 color bullBgColor = isLightTheme ? #FFD700 : #80FFFF, color bearBgColor = isLightTheme ? #FF6B6B : #FF80FF color obFillColor = isLightTheme ? #FFD700 : #33FF57, color osFillColor = isLightTheme ? #FF6B6B : #33FF57 // Find the direction of the TSI, signal line, and trend. Calculate the difference for histogram values. bool tsiIsBull = tsi >= tsi[1], bool tslIsBull = tsl >= tsl[1], bool trendIsBull = tsi >= tsl float diff = tsi - tsl // Get line and fill colors based on trend and plot direction. color tsiColor = tsiIsBull ? tsiBullColor : tsiBearColor, color tslColor = tslIsBull ? tslBullColor : tslBearColor color fillColor = trendIsBull ? tsiIsBull ? tsiUpUpColor : tsiUpDnColor : tsiIsBull ? tsiDnUpColor: tsiDnDnColor // Create invisible horizontal lines at +30 and -30 and fill areas between them with a gradient. h1 = hline(30, color = color(na)), h2 = hline(-30, color = color(na)) fill(h2, h1, 30, 5, color.new(obFillColor, 80), color(na)) fill(h2, h1, -5, -30, color(na), color.new(osFillColor, 80)) // Define colors and transparency for the histogram bar based on trend direction and theme and plot the bar. color barBgColor = isLightTheme ? trendIsBull ?

bullBgColor : bearBgColor : chart.bg_color color barBdColor = trendIsBull ? bullBgColor : bearBgColor int barBdTransp = isLightTheme ? 50 : 80 int barBgTransp = isLightTheme ? 90 : 0 float barHi = math.max(0, 0 + diff) float barLo = math.min(0, 0 + diff) plotcandle(barLo, barLo, barHi, barHi, color = color.new(barBgColor, barBgTransp), bordercolor = color.new(barBdColor, barBdTransp), wickcolor = color(na), display = display.pane) // Plot the TSI and its EMA p1 = plot(tsi, "TSI", color.new(tsiColor, 30)) p2 = plot(tsl, "Signal", color.new(tslColor, 30)) p3 = plot(tsl > 30 ? 30 : na, display = display.none) p4 = plot(tsl < -30 ? -30 : na, display = display.none) // Fill between the signal line and overbought or oversold levels with a gradient. fill(p2, p3, 80, 30, bearBgColor, color.new(bearBgColor, 90)) fill(p2, p4, -30, -80, color.new(bullBgColor, 90), bullBgColor) // Fill the cloud between the TSI and signal line with a gradient. fill(p1, p2, tsi, tsl, fillColor, color.new(fillColor, 70))

Note that:

- TSI and signal plots use distinct colors for each plot, with different colors for rising and falling values.
- We implement horizontal lines to demarcate overbought and oversold zones and fill between them with a color gradient. This gives the areas an appearance of scaled severity as the plots travel through them.
- We fill between the signal line and the overbought and oversold levels with another gradient if the plot exceeds either, to further emphasize potential irregularities.
- The script uses the plotcandle() function to create bordered column bars painted in bullish or bearish hues based on the trend's direction, serving as a visually distinct histogram that depicts the distance between the TSI and signal line.
- We use a four-color fill between the TSI and signal line to clarify trend direction and changes in trend. For instance, an upward trend with a declining TSI features a distinct color from the general rising trend hue.

How do I make my indicator plot in the main chart pane?

By default, new scripts display in a separate pane. To make a script display in the main chart pane instead, use `overlay = true` in the strategy() or indicator() declaration statement:

indicator("My Script", overlay = true)

The default value of the `overlay` parameter is `false`. If a programmer changes the value of the `overlay` parameter *after* an indicator was already added to the chart, they need to remove the indicator and add it to the chart again for the change to take effect.

Note that users of an indicator can move scripts from a separate pane to the chart pane and vice-versa using the "Move to" option from the "More" script menu, regardless of the value of the `overlay` parameter.

How can I plot vertical lines on a chart?

Scripts can plot vertical lines in three main ways:

- Drawing lines
- Plotting histograms
- Coloring the background

By drawing lines

Advatages of drawing vertical lines include that lines can be drawn on past or future bars. Each script can draw up to a maximum of 500 lines. If a script draws a line that has both its x coordinates at the same bar, different y coordinates, and extends in both directions, the line is vertical. The following example script draws vertical lines every 10 bars:

//@version=6 indicator("Vertical line demo", overlay = true, max_lines_count = 500) float atr = ta.atr(5) if bar_index % 10 == 0 line.new(bar_index, open, bar_index, open + atr, extend = extend.both, color = color.silver)

By plotting histograms

Histogram plots that join a very large number and a very low number appear as vertical lines. Advantages of using histogram plots include that there is no limit on the number of lines. With this method, unlike using line drawings, the user must right-click the price scale and select “Scale price chart only” to avoid distorting the chart vertically. Scripts can plot histograms, like all types of plot, only on the bar where the script is executing or at a *fixed* offset.

The following example script plots a very large number, 10e20, every 10 bars, and sets the base of the histogram to its negative value:

//@version=6 indicator("Histogram demo", overlay = true) plot(bar_index % 10 == 0 ? 10e20 : na, "vLine", color.silver, 1, plot.style_histogram, histbase = -10e20)

By coloring the background

Coloring the background for a single bar displays as a vertical line. Advantages of this method include simplicity, no limit on the number of lines, and no need to adjust the price scale. Disadvantages include no control of the width of the line – it is always exactly one bar wide, and the width scales with the number of bars that display on the chart. Scripts can change background color only on the bar on which the script is currently executing; offsetting the change is not possible.

//@version=6 indicator("Background color demo", overlay = true) bgcolor(bar_index % 10 == 0 ? chart.fg_color : na)

How can I toggle hline() levels on and off?

Scripts can toggle the display of horizontal levels plotted using hline() in several ways. Conditionally setting the argument of the `price` or `color` parameter to na shows or hides the level in a similar way that setting the `series` or `color` to na does for plotted lines, as described in the entry How do I plot a line using start/stop criteria? above.

> NoteUnlike the `series` and `color` parameters of the `plot.*()` functions, the `price` and `color` parameters of the hline() function require “input” or “const” values. Therefore, the function cannot dynamically show or hide its displayed value on each bar. Users can toggle horizontal levels only by using inputs.

Additionally, the `display` parameter of the hline() function can take an input value as its argument. This parameter controls where the line displays.

The following example script demonstrates all of these methods:

//@version=6 indicator("Toggle `hline()` using `na` values", overlay = false) bool showHlineInput1 = input.bool(true, "Show line 1") h1 = hline(price = showHlineInput1 ? 70 : na, color = chart.fg_color) bool showHlineInput2 = input.bool(true, "Show line 2") h2 = hline(price = 50, color = showHlineInput2 ? chart.fg_color : color(na)) bool showHlineInput3 = input.bool(true, "Show line 3") h3 = hline(price = 30, color = chart.fg_color, display = showHlineInput3 ? display.all : display.none)

How can I draw lines or labels into the future?

Individual *plotted* lines and shapes cannot be drawn into the future – only the entire series can be offset. By contrast, scripts can extend any *drawn* lines or boxes, or position drawn labels, at an arbitrary distance beyond the last data point. There are two ways to achieve this: using bar_index and using xloc.bar_time.

Using bar_index

The bar_index built-in variable represents the sequential number of the current bar, starting from zero for the first bar in the chart history and incrementing by 1 for each subsequent bar. Drawing objects with their `xloc` parameter set to xloc.bar_index can use a `bar_index` as their x coordinates. If the `xloc` parameter is not specified, it defaults to xloc.bar_index.

To project a certain number of bars into the future or past, simply add or subtract that number from the current bar_index. For instance, `bar_index + 20` positions the object 20 bars into the future.

> NoticeScripts can position drawings a maximum of 500 bars into the future or 10,000 bars into the past using this method.

The following example script draws lines and labels on the most recent pivot high and low, and extends the lines into the future beyond the last bar:

//@version=6 indicator("Draw into the future", overlay = true) int offsetInput = input.int(10, "Offset bars", maxval = 500) int pivotLengthInput = input.int(20, "Pivot lookback/look forward", minval = 2) // Initialize the drwaing objects. var line hiLine = line.new(na, na, na, na, color = chart.fg_color, style = line.style_dotted) var line loLine = line.new(na, na, na, na, color = chart.fg_color, style = line.style_dotted) var label hiLabel = label.new(na, na, na, color = color(na), textcolor = chart.fg_color, style = label.style_label_left) var label loLabel = label.new(na, na, na, color = color(na), textcolor = chart.fg_color, style = label.style_label_left) // Calculate the pivots on every bar. float pivotHighPrice = ta.pivothigh(pivotLengthInput, pivotLengthInput) float pivotLowPrice = ta.pivotlow(pivotLengthInput, pivotLengthInput) // Update the drawings when we get a pivot. if not na(pivotHighPrice) line.set_xy1(hiLine, bar_index - pivotLengthInput, pivotHighPrice), line.set_y2(hiLine, pivotHighPrice) label.set_y(hiLabel, pivotHighPrice) label.set_text(hiLabel, str.tostring(pivotHighPrice, format.mintick)) if not na(pivotLowPrice) line.set_xy1(loLine, bar_index - pivotLengthInput, pivotLowPrice), line.set_y2(loLine, pivotLowPrice) label.set_y(loLabel, pivotLowPrice) label.set_text(loLabel, str.tostring(pivotLowPrice, format.mintick)) // Update the position of the labels and the right end of the lines on the last bar. if barstate.islast hiLine.set_x2(bar_index + offsetInput) loLine.set_x2(bar_index + offsetInput) hiLabel.set_x(bar_index + offsetInput) loLabel.set_x(bar_index + offsetInput)

Note that:

- We update the properties of the drawings each time we get a pivot. In this section, we call the setting functions for the lines and labels in the normal way.
- On the last bar of the dataset, we adjust the `x2` point of both lines and the `x` point of the labels to extend into the future by the user-defined offset amount. In this section, for demonstration purposes, we call the same setting functions as methods by using dot notation syntax.

Using time

Drawing objects can also be positioned based on UNIX time values, by using xloc.bar_time for an object's `xloc` parameter. The time values can be timestamps, or a bar's open time, or any other calculated time.

By using time to position objects, there is no limitation as to how far into the future objects can display. Any valid timestamp positions the object accordingly.

There are some challenges to this method, however:

- Project a specific number of bars into the future using time can be complex. Failing to account for weekends and market holidays can lead to discrepancies in non-24/7 markets. For instance, trying to project 10 hourly bars ahead on a Friday evening might inadvertently position the object during a non-trading weekend slot.
- Bar time is based on UNIX time. When using raw date values, it's important to consider both the exchange's time zone and the user's local settings. Discrepancies can arise due to time zone differences, leading to objects displaying in unexpected positions on the chart. This can be particularly confusing if users of the script are located in various time zones or if the script is used on symbols from exchanges in different regions.

For help in displaying objects a specific number of time units into the future, refer to the Time Offset Calculation Framework by Pinecoders.

How can I keep only the last *n* number of drawings?

The two most robust and scalable ways to keep only the last *n* number of drawing objects are:

- Using the appropriate built-in `*.all` array.
- Adding drawings to an array and using the array as a queue.

> NoteThe arguments of `max_labels_count` and the other `max_*_count` parameters in the indicator() and strategy() declaration statements represent *approximate* values. To maintain a *precise* number of active drawings, use one of the two methods explained below.

Using a `*.all` array

The quickest and easiest method to limit the number of drawings displayed is to use the built-in `*.all` array for the drawing type. These arrays automatically contain all drawings of that type that currently display on the chart.

The `*.all` arrays are *read-only*. Scripts cannot change the arrays directly by pushing or shifting elements, but they can *update* or *delete* elements in the arrays. The following example script keeps a maximum of 10

labels on the chart. It gets the first label's ID via the array.first() function and then deletes that label with the label.delete() function.

//@version=6 indicator("Limit labels using *.all", overlay = true) const int LABEL_LIMIT = 10 label.new(bar_index, high, str.tostring(high)) if label.all.size() > LABEL_LIMIT label.all.first().delete()

Note that:

- The items in the array are in the order that they were drawn, so if a script displays drawings in a different order than it creates them, using a `*.all` array gives unpredictable results.
- Only built-in drawing types have `*.all` arrays available, so if a script uses objects of user-defined types as containers for several drawings, using an array as a queue instead is the only option.
- When the script deletes a label, the label is *also* automatically removed from the label.all array. By contrast, if a script adds drawings to an array manually, it must both remove the object from the array *and* delete it. In this case, array.shift() is usually the best choice, because it removes the first value from an array and returns it for further manipulation.

Using an array as a queue

A more flexible method to keep only the last *n* number of drawings is to use an array as a queue. Each time the script adds a new element to the array, it removes the oldest element. This is possible because each element in an array has a unique *index*, and array indexes *always* begin at zero. If we remove element `0` from an array with several elements, the element that was at index `1` is now at index `0`, the element at index `2` moves down to index `1`, and so on.

The following example script uses this behavior to keep a queue of labels and a queue of vertical lines to a defined user-configurable length. We demonstrate two different methods of managing the size of arrays:

- For labels, we create an array of the correct size, and always remove the same number of objects as we add. This method is simpler, but it requires programmers to ensure that objects are always added and removed in the same quantities (for example, by using a function to add and remove elements).
- For lines, we create an *empty* array, append lines to it, and only delete the oldest element if the array exceeds the maximum size. This more robust method is especially suitable for larger arrays, because the size of the array becomes meaningful: it corresponds to the number of – for example – lines drawn on the chart.

//@version=6 indicator("Limit drawings using arrays", overlay = true) int drawingQtyInput = input.int(5, "Quantity of drawings to show", minval = 0, maxval = 50) int drawingIntervalInput = input.int(4, "Draw every n bars", minval = 1, maxval = 10) // Create a label array sized according to the user-selected quantity. var array<label> labelsArray = array.new<label>(drawingQtyInput) if bar_index % drawingIntervalInput == 0 // Every n bars // Draw a label and add it to the *end* of the array labelsArray.push(label.new(bar_index, high, str.tostring(high), style = label.style_label_down, color = color.new(chart.fg_color, 30), textcolor = chart.bg_color)) // Remove the oldest label and delete it. label.delete(labelsArray.shift()) // Create an *empty* line array. var array<line> linesArray = array.new<line>() // Every n bars (offset -1 from the labels), draw a vertical line and add it to the *end* of the array. if (bar_index + 1) % drawingIntervalInput == 0 linesArray.push(line.new(bar_index, high, bar_index, low, color = chart.fg_color, extend = extend.both, width = 3)) // If the array is larger than the number of drawings to display, remove the oldest line and delete it. if linesArray.size() > drawingQtyInput line.delete(linesArray.shift())

Is it possible to draw geometric shapes?

Geometric shapes can illustrate patterns, mark zones of interest, or create other visual aids for technical analysis. Scripts can use two main methods for drawing shapes: polylines and lines.

Drawing with polylines

Polylines offer an efficient method for constructing complex shapes on a chart. To draw a shape with polylines, follow these steps:

1. Create an array containing objects of type chart.point that define the vertices of your desired shape.
2. Pass this array to the polyline.new() function. This function draws sequential lines between the points in the order they appear in the array.
3. Choose between straight or curved lines, enabling the creation of a diverse range of shapes.

This method is particularly useful for joining many points with a single object. A single polyline object can join up to 10,000 points, and one script can contain up to 100 polyline objects.

The following example script creates five sets of *interactive points* by placing price and time inputs inline. When the user adds the script to the chart, the script prompts them to place five points on the chart. The script adds the points to an array and passes the array to the polyline.new() function, which connects the points with lines, constructing a shape.

//@version=6 indicator("Polylines example", overlay = true) //@variable If `true`, the connections in the polyline are curved lines. bool curvedInput = input.bool(false, "Curve Polyline") //@variable If `true`, connects the first point in the polyline to the last point. bool closedInput = input.bool(true, "Close Polyline") //@variable The color of the space filled by the polyline. color fillcolor = input.color(color.new(color.blue, 90), "Fill Color") // Time and price input pairs for the polyline's points. int p1x = input.time(0, "p1", confirm = true, inline = "p1") float p1y = input.price(0, " ", confirm = true, inline = "p1") int p2x = input.time(0, "p2", confirm = true, inline = "p2") float p2y = input.price(0, " ", confirm = true, inline = "p2") int p3x = input.time(0, "p3", confirm = true, inline = "p3") float p3y = input.price(0, " ", confirm = true, inline = "p3") int p4x = input.time(0, "p4", confirm = true, inline = "p4") float p4y = input.price(0, " ", confirm = true, inline = "p4") int p5x = input.time(0, "p5", confirm = true, inline = "p5") float p5y = input.price(0, " ", confirm = true, inline = "p5") if barstate.islastconfirmedhistory //@variable An array of `chart.point` objects for the new polyline. var array<chart.point> points = array.new<chart.point>() // Push new `chart.point` instances into the `points` array. points.push(chart.point.from_time(p1x, p1y)) points.push(chart.point.from_time(p2x, p2y)) points.push(chart.point.from_time(p3x, p3y)) points.push(chart.point.from_time(p4x, p4y)) points.push(chart.point.from_time(p5x, p5y)) // Add labels for each `chart.point` in `points`. label l1p1 = label.new(points.get(0), "p1", xloc.bar_time, color = na) label l1p2 = label.new(points.get(1), "p2", xloc.bar_time, color = na) label l2p1 = label.new(points.get(2), "p3", xloc.bar_time, color = na) label l2p2 = label.new(points.get(3), "p4", xloc.bar_time, color = na) label l3p1 = label.new(points.get(4), "p5", xloc.bar_time, color = na) // Create a new polyline that connects each `chart.point` in the `points` array, starting from the first. polyline.new(points, curvedInput, closedInput, xloc.bar_time, fill_color = fillcolor)

Note that:

- The five sets of time and price inputs are *paired* to form interactive points because they share matching `inline` arguments.

Drawing with lines

Lines are simpler than polylines. A line in Pine Script is straight connection between two points. Here's how to use lines for drawing shapes:

- Determine the start and end points for each line, recognizing that every edge of your shape must be delineated separately.
- For each side of the shape, input the precise starting and ending coordinates to create individual line segments.
- Ensure that the end of one line meets the start of the next, to form a closed shape.

An advantage of using separate lines is that scripts can customize the style of each line separately. Additionally, arrays are optional for line management – as opposed to mandatory for polylines – which can result in simpler code in some cases. For large, complex shapes, however, polylines are more convenient.

The following example script creates six sets of *interactive points* by pairing price and time inputs, using the same `inline` arguments. When the script is added to the chart, it prompts the user to click six points on the chart (because the inputs use `confirm = true`). The script joins the points and creates two triangles.

//@version=6 indicator("Triangles", "", true) // Create interactive point inputs using pairs of inputs with the same `inline` argument, and the `confirm` parameter. string GRP1 = "Triangle A" int x1AInput = input.time(0, "Point 1", inline = "A1", group = GRP1, confirm = true) float y1AInput = input.price(0, "", inline = "A1", group = GRP1, confirm = true) int x2AInput = input.time(0, "Point 2", inline = "A2", group = GRP1, confirm = true) float y2AInput = input.price(0, "", inline = "A2", group = GRP1, confirm = true) int x3AInput = input.time(0, "Point 3", inline = "A3", group = GRP1, confirm = true) float y3AInput = input.price(0, "", inline = "A3", group = GRP1, confirm = true) string GRP2 = "Triangle B" int x1BInput = input.time(0, "Point 1", inline = "B1", group = GRP2, confirm = true) float y1BInput = input.price(0, "", inline = "B1", group = GRP2, confirm = true) int x2BInput = input.time(0, "Point 2", inline = "B2", group = GRP2, confirm = true) float y2BInput = input.price(0, "", inline = "B2", group = GRP2, confirm = true) int x3BInput = input.time(0, "Point 3", inline = "B3", group = GRP2, confirm = true) float y3BInput = input.price(0, "", inline = "B3", group = GRP2, confirm = true) // @function Constructs a triangle on the chart using three vertices and a specified line color. // @param vx1 (int) The bar time of the first vertex. // @param vy1 (float) The price of the first vertex. // @param vx2 (int) The bar time of the second vertex. // @param vy2 (float) The price of the second vertex. // @param vx3 (int) The bar time of the third vertex. // @param vy3 (float) The price of the third vertex. // @param lineColor (color) The color of the triangle's edges. // @param labelName (string) The text to display in the label at the triangle's peak. // @returns (label) The label to plot the text for the triangle. drawTriangle(int vx1, float vy1, int vx2, float vy2, int vx3, float vy3, color lineColor, string name) => line.new(vx1, vy1, vx2, vy2, xloc = xloc.bar_time, color = lineColor) line.new(vx2, vy2, vx3, vy3, xloc = xloc.bar_time, color = lineColor) line.new(vx1, vy1, vx3, vy3, xloc = xloc.bar_time, color = lineColor) array<int> xValues = array.from(vx1, vx2, vx3) array<float> yValues = array.from(vy1, vy2, vy3) float yMax = array.max(yValues) int xMax = array.get(xValues, array.indexof(yValues, yMax)) label.new(xMax, yMax, name, xloc = xloc.bar_time, color = color(na), textcolor = color.gray) // We draw the shapes only once for efficiency. if barstate.isfirst drawTriangle(x1AInput, y1AInput, x2AInput, y2AInput, x3AInput, y3AInput, color.lime, "A") drawTriangle(x1BInput, y1BInput, x2BInput, y2BInput, x3BInput, y3BInput, color.fuchsia, "B")

How can I color the chart's background on a condition detected on the last bar?

The usual way to color the chart background is by calling the bgcolor() function, which colors the chart background *for the bar on which it is called*. The background for a particular bar cannot be changed on later

bars, and bars cannot be colored retroactively, using this method.

To color the entire chart background based on a condition detected on the last bar, one workaround is to use a table sized to the visible area of the chart, and color the background of the cell. See this example in the page on tables.

ADVANCED

Arrays

> TipThis page contains *advanced* material. If you're new to Pine Script®, start by learning about core language components – such as the type system and the basics of the execution model – and explore other, more accessible features before venturing further.

Introduction

Pine Script *arrays* are one-dimensional collections that can store multiple values or references in a single location. Arrays are a more robust alternative to declaring a set of similar variables (e.g., `price00`, `price01`, `price02`, ...).

All elements in an array must be of the same built-in type, user-defined type, or enum type.

Similar to lines, labels, and other reference types, arrays and their data are accessed using *references*, which we often refer to as *IDs*. Pine Script does not use an indexing operator to access individual array elements. Instead, functions including array.get() and array.set() read and write the elements of the array associated with a specific ID.

Scripts access specific elements in an array by specifying an *index* in calls to these functions. The index starts at 0 and extends to one less than the number of elements in the array. Arrays in Pine Script can have dynamic sizes that vary across bars, as scripts can change the number of elements in an array on any execution. A single script can create multiple array instances. The total number of elements in any array cannot exceed 100,000.

> Note
>
> We often refer to index 0 as the *beginning* of an array, and the highest index value as the *end* of the array.
>
> Additionally, for the sake of brevity, we sometimes use the term "array" to mean "array ID".

Declaring arrays

Pine Script uses the following syntax for array declarations:

```
[var/varip ][array<type> ]<identifier> = <expression>
```

Where `<type>` is a *type template* that defines the type of elements that the array can contain, and `<expression>` is an expression that returns either the ID of an array or na. See the Collections section of the Type system page to learn about type templates.

When declaring an array variable, programmers can use the array keyword followed by a type template to explicitly define the variable's *type identifier* (e.g., `array<int>` for a variable that can reference an array of "int" values).

> NoticeIt is also possible to specify an array variable's type by prefixing its declaration with the *element* type keyword, followed by empty *square brackets* (`[]`). For example, a variable whose declaration includes `int[]` as the type keyword accepts the type `array<int>`. However, this *legacy* format is *deprecated*; future versions of Pine Script might not support it. Therefore, we recommend using the `array<type>` format to define type identifiers for consistency.

Specifying a type identifier for a variable or function parameter that holds array references is usually optional. The only exceptions are when initializing an identifier with an `na` value, defining exported library functions whose parameters accept array IDs, or declaring user-defined types with fields for storing array IDs. Even when not required, note that specifying an array variable's type helps promote readability, and it helps the Pine Editor provide relevant code suggestions.

The following line of code declares an array variable named `prices` that has an initial reference of na. This variable declaration *requires* a type identifier, because the compiler cannot automatically determine the type that na represents:

array<float> prices = na

Scripts can use the following functions to create new arrays: array.new<type>(), array.from(), or array.copy(). Each of these functions creates a new array and returns a non-na ID for use in other parts of the code. Note that these functions accept "series" arguments for all parameters, meaning the constructed arrays can have dynamic sizes and elements on each call.

The following example creates an empty "float" array and assigns its ID to a `prices` variable. Specifying a type identifier for the `prices` variable is *not* required in this case, because the variable automatically *inherits* the function's returned type (`array<float>`):

prices = array.new<float>(0)

> Note
>
> The `array` namespace also includes *legacy functions* for creating arrays of specific *built-in types*. These functions include array.new_int(), array.new_float(), array.new_bool(), array.new_color(), array.new_string(), array.new_line(), array.new_linefill(), array.new_label(), array.new_box() and array.new_table().
>
> However, we recommend using the general-purpose array.new<type>() function, because it can create an array of *any* supported type, including user-defined types.

The `initial_value` parameter of the `array.new*()` functions enables users to set *all* initial elements in the array to a specified value or reference. If a call to these functions does not include an `initial_value` argument, it creates an array filled with na elements.

The following line declares an array variable named `prices` and assigns it the ID of an array containing two elements. Both elements in the array hold the current bar's close value:

prices = array.new<float>(2, close)

To create an array without initializing all elements to the same value or reference, use array.from(). This function determines the array's size and the type of elements it stores based on the arguments in the function call. All arguments supplied to the call must be of the *same type*.

For example, both lines of code in the following example show two ways to create a "bool" array using array.from() and declare a variable to store its ID:

statesArray = array.from(close > open, high != close) array<bool> statesArray = array.from(close > open, high != close)

Using `var` and `varip` keywords

Programmers can use the var and varip keywords to instruct a script to declare an array variable on only one bar instead of on each execution of the variable's scope. Array variables declared using these keywords point to the same array instances until explicitly reassigned, allowing an array and its elements to persist across bars.

When declaring an array variable using these keywords and pushing a new value to the end of the referenced array on each bar, the array will grow by one on each bar and be of size `bar_index + 1` (bar_index starts at zero) by the time the script executes on the last bar, as this code demonstrates:

//@version=6 indicator("Using `var`") //@variable An array that expands its size by 1 on each bar. var a = array.new<float>(0) array.push(a, close) if barstate.islast //@variable A string containing the size of `a` and the current `bar_index` value. string labelText = "Array size: " + str.tostring(a.size()) + "\nbar_index: " + str.tostring(bar_index) // Display the `labelText`. label.new(bar_index, 0, labelText, size = size.large)

The same code without the var keyword would *reinitialize* the `a` variable with the ID of a new, empty array on every execution. In that case, after execution of the array.push() call, the array.size() *method* call (`a.size()`) would return a value of 1.

> Notice
>
> Array variables declared using varip behave similarly to those declared using var, with two key differences. Firstly, the arrays that they reference can finalize updates to their elements on *any* available tick – not only on a bar's closing tick. Secondly, arrays referenced by varip variables can contain only the following data:
>
> - Values of any fundamental type.
> - IDs of the chart.point, footprint, or volume_row type.
> - References to objects of a user-defined type that have fields for storing only data of either of the above types or the IDs of other collections containing only these types.

Reading and writing array elements

Scripts can write values to existing individual array elements using array.set(), and read using array.get(). When using these functions, it is imperative that the `index` in the function call is always less than or equal to the array's size (because array indices start at zero). To get the size of an array, use the array.size() function.

The following example uses the set() method to populate a `fillColors` array with instances of one base color using different transparency levels. It then uses array.get() to retrieve one of the colors from the array based on the location of the bar with the highest price within the last `lookbackInput` bars:

//@version=6 indicator("Distance from high", "", true) lookbackInput = input.int(100) FILL_COLOR = color.green // Declare array and set its values on the first bar only. var fillColors = array.new<color>(5) if barstate.isfirst // Initialize the array elements with progressively lighter shades of the fill color. fillColors.set(0, color.new(FILL_COLOR, 70)) fillColors.set(1, color.new(FILL_COLOR, 75)) fillColors.set(2, color.new(FILL_COLOR, 80)) fillColors.set(3, color.new(FILL_COLOR, 85)) fillColors.set(4, color.new(FILL_COLOR, 90)) // Find the offset to highest high. Change its sign because the function returns a negative value. lastHiBar = - ta.highestbars(high, lookbackInput) // Convert the offset to an array index, capping it to 4 to avoid a runtime error. // The index used by `array.get()` will be the equivalent of `floor(fillNo)`. fillNo = math.min(lastHiBar / (lookbackInput / 5), 4) // Set background to a progressively lighter fill with increasing distance from location of highest high. bgcolor(array.get(fillColors, fillNo)) // Plot key values to the Data Window for debugging. plotchar(lastHiBar, "lastHiBar", "", location.top, size = size.tiny) plotchar(fillNo, "fillNo", "", location.top, size = size.tiny)

Another technique for initializing the elements in an array is to create an *empty array* (an array with no elements), then use array.push() to append **new** elements to the end of the array, increasing the size of the array by one on each call. The following code is functionally identical to the initialization section from the preceding script:

// Declare array and set its values on the first bar only. var fillColors = array.new<color>(0) if barstate.isfirst // Initialize the array elements with progressively lighter shades of the fill color. array.push(fillColors, color.new(FILL_COLOR, 70)) array.push(fillColors, color.new(FILL_COLOR, 75)) array.push(fillColors, color.new(FILL_COLOR, 80)) array.push(fillColors, color.new(FILL_COLOR, 85)) array.push(fillColors, color.new(FILL_COLOR, 90))

This code is equivalent to the one above, but it uses array.unshift() to insert new elements at the *beginning* of the `fillColors` array:

// Declare array and set its values on the first bar only. var fillColors = array.new<color>(0) if barstate.isfirst // Initialize the array elements with progressively lighter shades of the fill color. array.unshift(fillColors, color.new(FILL_COLOR, 90)) array.unshift(fillColors, color.new(FILL_COLOR, 85)) array.unshift(fillColors, color.new(FILL_COLOR, 80)) array.unshift(fillColors, color.new(FILL_COLOR, 75)) array.unshift(fillColors, color.new(FILL_COLOR, 70))

We can also use array.from() to create the same `fillColors` array with a single function call:

//@version=6 indicator("Using `var`") FILL_COLOR = color.green var array<color> fillColors = array.from(color.new(FILL_COLOR, 70), color.new(FILL_COLOR, 75), color.new(FILL_COLOR, 80), color.new(FILL_COLOR, 85), color.new(FILL_COLOR, 90)) // Cycle background through the array's colors. bgcolor(array.get(fillColors, bar_index % (fillColors.size())))

The array.fill() function points all array elements, or the elements within the `index_from` to `index_to` range, to a specified `value`. Without the last two optional parameters, the function fills the whole array, so:

a = array.new<float>(10, close)

and:

a = array.new<float>(10) a.fill(close)

are equivalent, but:

a = array.new<float>(10) a.fill(close, 1, 3)

only fills the second and third elements (at index 1 and 2) of the array with `close`. Note how the array.fill() function's last parameter, `index_to`, must have a value one greater than the last index the function will fill. The remaining elements will hold `na` values, as the array.new<type>() function call does not contain an `initial_value` argument.

Looping through array elements

When looping through an array's element indices and the array's size is unknown, one can use the array.size() function to get the maximum index value. For example:

//@version=6 indicator("Protected `for` loop", overlay = true) //@variable An array of `close` prices from the 1-minute timeframe. array<float> a = request.security_lower_tf(syminfo.tickerid, "1", close) //@variable A string representation of the elements in `a`. string labelText = "" for i = 0 to (array.size(a) == 0 ? na : array.size(a) - 1) labelText += str.tostring(array.get(a, i)) + "\n" label.new(bar_index, high, text = labelText)

Note that:

- We use the request.security_lower_tf() function which returns an array of close prices at the `1 minute` timeframe.
- This code example will throw an error if you use it on a chart timeframe smaller than `1 minute`.
- for loops do not execute if the `to` expression is na. Note that the `to` value is only evaluated once upon entry.

An alternative method to loop through an array is to use a for…in loop. This approach is a variation of the standard for loop that can iterate over the value references and indices in an array. Here is an example of how we can write the code example from above using a `for...in` loop:

//@version=6 indicator("`for...in` loop", overlay = true) //@variable An array of `close` prices from the 1-minute timeframe. array<float> a = request.security_lower_tf(syminfo.tickerid, "1", close) //@variable A string representation of the elements in `a`. string labelText = "" for price in a labelText += str.tostring(price) + "\n" label.new(bar_index, high, text = labelText)

Note that:

- for…in loops can return a tuple containing each index and corresponding element. For example, `for [i, price] in a` returns the `i` index and `price` value for each element in `a`.

A while loop statement can also be used:

//@version=6 indicator("`while` loop", overlay = true) array<float> a = request.security_lower_tf(syminfo.tickerid, "1", close) string labelText = "" int i = 0 while i < array.size(a) labelText += str.tostring(array.get(a, i)) + "\n" i += 1 label.new(bar_index, high, text = labelText)

Scope

Users can declare arrays within the global scope of a script, as well as the local scopes of functions, methods, and conditional structures. Unlike some of the other built-in types, namely *fundamental* types, scripts can modify globally-assigned arrays from within local scopes, allowing users to implement global variables that any function in the script can directly interact with. We use the functionality here to calculate progressively lower or higher price levels:

//@version=6 indicator("Bands", "", true) //@variable The distance ratio between plotted price levels. factorInput = 1 + (input.float(-2., "Step %") / 100) //@variable A single-value array holding the lowest `ohlc4` value within a 50 bar window from 10 bars back. level = array.new<float>(1, ta.lowest(ohlc4, 50)[10]) nextLevel(val) => newLevel = level.get(0) * val // Write new level to the global `level` array so we can use it as the base in the next function call. level.set(0, newLevel) newLevel plot(nextLevel(1)) plot(nextLevel(factorInput)) plot(nextLevel(factorInput)) plot(nextLevel(factorInput))

History referencing

The history-referencing operator [] can access the history of array variables, allowing scripts to interact with past array instances previously assigned to a variable.

To illustrate this, let's create a simple example to show how one can fetch the previous bar's `close` value in two equivalent ways. This script uses the [] operator to get the array instance assigned to `a` on the previous bar, then uses an array.get() method call to retrieve the value of the first element (`previousClose1`). For `previousClose2`, we use the history-referencing operator on the `close` variable directly to retrieve the value. As we see from the plots, `previousClose1` and `previousClose2` both return the same value:

//@version=6 indicator("History referencing") //@variable A single-value array declared on each bar. a = array.new<float>(1) // Set the value of the only element in `a` to `close`. array.set(a, 0, close) //@variable The array instance assigned to `a` on the previous bar. previous = a[1] previousClose1 = na(previous) ? na : previous.get(0) previousClose2 = close[1] plot(previousClose1, "previousClose1", color.gray, 6) plot(previousClose2, "previousClose2", color.white, 2)

Inserting and removing array elements

Inserting

The following three functions can insert new elements into an array.

array.unshift() inserts a new element at the beginning of an array (index 0) and increases the index values of any existing elements by one.

array.insert() inserts a new element at the specified `index` and increases the index of existing elements at or after the `index` by one.

//@version=6 indicator("`array.insert()`") a = array.new<float>(5, 0) for i = 0 to 4 array.set(a, i, i + 1) if barstate.islast label.new(bar_index, 0, "BEFORE\na: " + str.tostring(a), size = size.large) array.insert(a, 2, 999) label.new(bar_index, 0, "AFTER\na: " + str.tostring(a), style = label.style_label_up, size = size.large)

array.push() adds a new element at the end of an array.

Removing

These four functions remove elements from an array. The first three also return the value of the removed element.

array.remove() removes the element at the specified `index` and returns that element's value.

array.shift() removes the first element from an array and returns its value.

array.pop() removes the last element of an array and returns its value.

array.clear() removes all elements from an array. Note that clearing an array won't delete any objects its elements referenced. See the example below that illustrates how this works:

//@version=6 indicator("`array.clear()` example", overlay = true) // Create a label array and add a label to the array on each new bar. var a = array.new<label>() label lbl = label.new(bar_index, high, "Text", color = color.red) array.push(a, lbl) var table t = table.new(position.top_right, 1, 1) // Clear the array on the last bar. This doesn't remove the labels from the chart. if barstate.islast array.clear(a) table.cell(t, 0, 0, "Array elements count: " + str.tostring(array.size(a)), bgcolor = color.yellow)

Using an array as a stack

Stacks are LIFO (last in, first out) constructions. They behave somewhat like a vertical pile of books to which books can only be added or removed one at a time, always from the top. Pine Script arrays can be used as a stack, in which case we use the array.push() and array.pop() functions to add and remove elements at the end of the array.

`array.push(prices, close)` will add a new element to the end of the `prices` array, increasing the array's size by one.

`array.pop(prices)` will remove the end element from the `prices` array, return its value and decrease the array's size by one.

See how the functions are used here to track successive lows in rallies:

//@version=6 indicator("Lows from new highs", "", true) var lows = array.new<float>(0) flushLows = false //@function Removes the last element from the `id` stack when `cond` is `true`. array_pop(id, cond) => cond and array.size(id) > 0 ? array.pop(id) : float(na) if ta.rising(high, 1) // Rising highs; push a new low on the stack. lows.push(low) // Force the return type of this `if` block to be the same as that of the next block. bool(na) else if lows.size() >= 4 or low < array.min(lows) // We have at least 4 lows or price has breached the lowest low; // sort lows and set flag indicating we will plot and flush the levels. array.sort(lows, order.ascending) flushLows := true // If needed, plot and flush lows. lowLevel = array_pop(lows, flushLows) plot(lowLevel, "Low 1", low > lowLevel ? color.silver : color.purple, 2, plot.style_linebr) lowLevel := array_pop(lows, flushLows) plot(lowLevel, "Low 2", low > lowLevel ? color.silver : color.purple, 3, plot.style_linebr) lowLevel := array_pop(lows, flushLows) plot(lowLevel, "Low 3", low > lowLevel ? color.silver : color.purple, 4, plot.style_linebr) lowLevel := array_pop(lows, flushLows) plot(lowLevel, "Low 4", low > lowLevel ? color.silver : color.purple, 5, plot.style_linebr) if flushLows // Clear remaining levels after the last 4 have been plotted. lows.clear()

Using an array as a queue

Queues are FIFO (first in, first out) constructions. They behave somewhat like cars arriving at a red light. New cars are queued at the end of the line, and the first car to leave will be the first one that arrived to the red light.

In the following code example, we let users decide through the script's inputs how many labels they want to have on their chart. We use that quantity to determine the size of the array of labels we then create, initializing the array's elements to `na`.

When a new pivot is detected, we create a label for it, saving the label's ID in the `pLabel` variable. We then queue the ID of that label by using array.push() to append the new label's ID to the end of the array, making our array size one greater than the maximum number of labels to keep on the chart.

Lastly, we de-queue the oldest label by removing the array's first element using array.shift() and deleting the label referenced by that array element's value. As we have now de-queued an element from our queue, the array contains `pivotCountInput` elements once again. Note that on the dataset's first bars we will be deleting `na` label IDs until the maximum number of labels has been created, but this does not cause runtime errors. Let's look at our code:

//@version=6 MAX_LABELS = 100 indicator("Show Last n High Pivots", "", true, max_labels_count = MAX_LABELS) pivotCountInput = input.int(5, "How many pivots to show", minval = 0, maxval = MAX_LABELS) pivotLegsInput = input.int(3, "Pivot legs", minval = 1, maxval = 5) // Create an array containing the user-selected max count of label IDs. var labelIds = array.new<label>(pivotCountInput) pHi = ta.pivothigh(pivotLegsInput, pivotLegsInput) if not na(pHi) // New pivot found; plot its label `pivotLegsInput` bars behind the current `bar_index`. pLabel = label.new(bar_index - pivotLegsInput, pHi, str.tostring(pHi, format.mintick), textcolor = color.white) // Queue the new label's ID by appending it to the end of the array. array.push(labelIds, pLabel) // De-queue the oldest label ID from the queue and delete the corresponding label. label.delete(array.shift(labelIds))

Negative indexing

The array.get(), array.set(), array.insert(), and array.remove() functions support *negative indexing*, which references elements starting from the end of the array. An index of `-1` refers to the last element in the array, an index of `-2` refers to the second to last element, and so on.

When using a *positive* index, functions traverse the array *forwards* from the beginning of the array (*first to last* element). The first element's index is `0`, and the last element's index is `array.size() - 1`. When using a *negative* index, functions traverse the array *backwards* from the end of the array (*last to first* element). The last element's index is `-1`, and the first element's index is `-array.size()`:

array<string> myArray = array.from("first", "second", "third", "fourth", "last") // Positive indexing: Indexes forwards from the beginning of the array. myArray.get(0) // Returns "first" element myArray.get(myArray.size() - 1) // Returns "last" element myArray.get(4) // Returns "last" element // Negative indexing: Indexes backwards from the end of the array. myArray.get(-1) // Returns "last" element myArray.get(-myArray.size()) // Returns "first" element myArray.get(-5) // Returns "first" element

Like positive indexing, negative indexing is bound by the size of the array. For example, functions operating on an array of 5 elements only accept indices of 0 to 4 (first to last element) or -1 to -5 (last to first element). Any other indices are out of bounds and will raise a runtime error.

We can use negative indices to retrieve, update, add, and remove array elements. This simple script creates an “int” `countingArray` and calls the array.get(), array.set(), array.insert(), and array.remove() functions to perform various array operations using negative indices. It displays each array operation and its corresponding result using a table:

//@version=6 indicator("Negative indexing demo", overlay = false) //@variable A table that displays various array operations and their results. var table displayTable = table.new(position.middle_center, 2, 15, bgcolor = color.white, frame_color = color.black, frame_width = 1, border_width = 1) //@function Initializes a `displayTable` row to output a "string" of an `arrayOperation` and the `operationResult`. displayRow(int rowID, string arrayOperation, operationResult) => //@variable Is white if the `rowID` is even, light blue otherwise. Used to set alternating table row colors. color rowColor = rowID % 2 == 0 ? color.white : color.rgb(33, 149, 243, 75) // Display the `arrayOperation` in the row's first cell. displayTable.cell(0, rowID, arrayOperation, text_color = color.black, text_halign = text.align_left, bgcolor = rowColor, text_font_family = font.family_monospace) // Display the `operationResult` in the row's second cell. displayTable.cell(1, rowID, str.tostring(operationResult), text_color = color.black, text_halign = text.align_right, bgcolor = rowColor) if barstate.islastconfirmedhistory //@variable Array of "int" numbers. Holds six multiples of 10, counting from 10 to 60. array<int> countingArray = array.from(10, 20, 30, 40, 50, 60) // Initialize the table's header cells. displayTable.cell(0, 0, "ARRAY OPERATION") displayTable.cell(1, 0, "RESULT") // Display the initial `countingArray` values. displayTable.cell(0, 1, "Initial `countingArray`", text_color = color.black, text_halign = text.align_center, bgcolor = color.yellow) displayTable.cell(1, 1, str.tostring(countingArray), text_color = color.black, text_halign = text.align_right, bgcolor = color.yellow) // Retrieve array elements using negative indices in `array.get()`. displayRow(2, "`countingArray.get(0)`", countingArray.get(0)) displayRow(3, "`countingArray.get(-1)`", countingArray.get(-1)) displayRow(4, "`countingArray.get(-countingArray.size())`", countingArray.get(-countingArray.size())) // Update array elements using negative indices in `array.set()` and `array.insert()`. countingArray.set(-2, 99) displayRow(5, "`countingArray.set(-2, 99)`", countingArray) countingArray.insert(-5, 878) displayRow(6, "`countingArray.insert(-5, 878)`", countingArray) // Remove array elements using negative indices in `array.remove()`. countingArray.remove(-3) displayRow(7, "`countingArray.remove(-3)`", countingArray)

Note that not all array operations can use negative indices. For example, search functions like array.indexof() and array.binary_search() return the *positive* index of an element if it’s found in the array. If the value is not found, the functions return `-1`. However, this returned value is **not** a negative index, and using it as one would incorrectly reference the last array element. If a script needs to use a search function’s returned index in subsequent array operations, it must appropriately differentiate between this `-1` result and other valid indices.

Calculations on arrays

While series variables can be viewed as a horizontal set of values stretching back in time, Pine Script's one-dimensional arrays can be viewed as vertical structures residing on each bar. As an array's set of elements is not a time series, Pine Script's usual mathematical functions are not allowed on them. Special-purpose functions must be used to operate on all of an array's values. The available functions are: array.abs(), array.avg(), array.covariance(), array.min(), array.max(), array.median(), array.mode(), array.percentile_linear_interpolation(), array.percentile_nearest_rank(), array.percentrank(), array.range(), array.standardize(), array.stdev(), array.sum(), array.variance().

Note that contrary to the usual mathematical functions in Pine Script, those used on arrays do not return `na` when some of the values they calculate on have `na` values. There are a few exceptions to this rule:

- When all array elements have `na` value or the array contains no elements, `na` is returned. `array.standardize()` however, will return an empty array.
- `array.mode()` will return `na` when no mode is found.

Manipulating arrays

Concatenation

Two arrays can be merged — or concatenated — using array.concat(). When arrays are concatenated, the second array is appended to the end of the first, so the first array is modified while the second one remains intact. The function returns the array ID of the first array:

//@version=6 indicator("`array.concat()`") a = array.new<float>(0) b = array.new<float>(0) array.push(a, 0) array.push(a, 1) array.push(b, 2) array.push(b, 3) if barstate.islast label.new(bar_index, 0, "BEFORE\na: " + str.tostring(a) + "\nb: " + str.tostring(b), size = size.large) c = array.concat(a, b) array.push(c, 4) label.new(bar_index, 0, "AFTER\na: " + str.tostring(a) + "\nb: " + str.tostring(b) + "\nc: " + str.tostring(c), style = label.style_label_up, size = size.large)

Joining

The array.join() function converts an “int”, “float”, or “string” array’s elements into strings, then *joins* each one to form a single “string” value with a specified `separator` inserted between each combined value. It provides a convenient alternative to converting values to strings with str.tostring() and performing repeated string concatenation operations.

The following script demonstrates the array.join() function’s behaviors. It requests tuples of “string”, “int”, and “float” values from three different contexts with request.security() calls, creates separate arrays for each type with array.from(), then creates joined strings with the array.join() function. Lastly, it creates another array from those strings with array.from() and joins them with another array.join() call, using a newline as the separator, and displays the final string in the table:

//@version=6 indicator("Joining demo") //@function Returns a tuple containing the ticker ID ("string"), bar index ("int"), and closing price ("float"). dataRequest() => [syminfo.tickerid, bar_index, close] if barstate.islast //@variable A single-cell table displaying the results of `array.join()` calls. var table displayTable = table.new(position.middle_center, 1, 1, color.blue) // Request data for three symbols. [ticker1, index1, price1] = request.security("SPY", "", dataRequest()) [ticker2, index2, price2] = request.security("GLD", "", dataRequest()) [ticker3, index3, price3] = request.security("TLT", "", dataRequest()) // Create separate "string", "int", and "float" arrays to hold the requested data. array<string> tickerArray = array.from(ticker1, ticker2, ticker3) array<int> indexArray = array.from(index1, index2, index3) array<float> priceArray = array.from(price1, price2, price3) // Convert each array's data to strings and join them with different separators. string joined1 = array.join(tickerArray, ", ") string joined2 = indexArray.join("|") string joined3 = priceArray.join("\n") //@variable A joined "string" containing the `joined1`, `joined2`, and `joined3` values. string displayText = array.from(joined1, joined2, joined3).join("\n---\n") // Initialize a cell to show the `displayText`. displayTable.cell(0, 0, displayText, text_color = color.white, text_size = 36)

Note that:

- Each array.join() call inserts the specified separator only between each element string. It does *not* include the separator at the start or end of the returned value.

- The array.join() function uses the same numeric format as the default for str.tostring(). See the String conversion and formatting section of the Strings page to learn more.
- Calls to array.join() cannot directly convert elements of “bool”, “color”, or other types to strings. Scripts must convert data of these types separately.

Sorting

Scripts can *sort* arrays containing values of the “int”, “float”, or “string” type by using the array.sort() function. The function’s `order` parameter accepts one of the two `order.*` constants to specify the sorting order. If the argument is order.ascending (the default), a call to the function rearranges the specified array’s elements in ascending order by value. If the argument is order.descending, the call rearranges the elements in descending order instead.

If an array contains “int” or “float” elements, the array.sort() function sorts the array using each element’s numeric value. If it uses ascending order, the element with the *lowest* value becomes the array’s *first* element (at index 0), and the one with the *highest* value becomes the *last* element. If it sorts in descending order, the element with the *highest* value becomes the first element, and the one with the *lowest* value becomes the last.

The following example script uses the array.from() function to create an array of arbitrary “float” values on the last historical bar. It then sorts the array in ascending order, and then in descending order, using two array.sort() calls. The script creates a string representation of the array after each step, then formats those representations into a single string and displays the result in a label:

//@version=6 indicator("Sorting numeric arrays demo") if barstate.islastconfirmedhistory //@variable References an array of arbitrary "float" values. array<float> numbers = array.from(2.1, 0.5, 1.2, 0.1, 1.4, 0.6) //@variable A string representing the array's unsorted, ascending, and descending order. string displayStr = "Unsorted: " + str.tostring(numbers) + "\n" // Sort the array in ascending order. // The `order` argument is optional; `order.ascending` is the default. array.sort(numbers, order = order.ascending) // Concatenate a string representation of the sorted array with the `displayStr` value. displayStr += "Ascending: " + str.tostring(numbers) + "\n" // Sort the `numbers` array again, this time in descending order. numbers.sort(order = order.descending) // Concatenate another string representation of the sorted result. displayStr += "Descending: " + str.tostring(numbers) // Display the final string's text in a label. label.new(bar_index, 0, displayStr, style = label.style_label_center, size = 30, textalign = text.align_left, text_font_family = font.family_monospace)

If an array contains “string” elements, the array.sort() function sorts the elements based on the Unicode values of the strings’ *individual characters*. The sorting algorithm initially compares the *first* character in each string, then compares subsequent characters as necessary if multiple strings have matching characters at the same position. The strings that have leading characters with the lowest Unicode values move to the beginning of the array if the order is ascending, or to the end of the array if the order is descending.

The example script below defines an arbitrary literal string, then uses the str.split() function to split the string and construct an array of substrings. Afterward, the script calls the array.sort() function to sort the array’s elements in ascending order. The script displays formatted text representing the original string, and the array’s structure before and after sorting, in a label on the last historical bar:

//@version=6 indicator("Sorting string arrays demo") if barstate.islastconfirmedhistory //@variable A literal string to split at each `,` character. string originalStr = "abc,abC,Abc,ABC,{ABC},!,123,12.3, " //@variable References an array of substrings formed by splitting the original string at each comma. array<string>

splitStrArray = str.split(originalStr, ",") //@variable A string to represent the original string and the array of substrings. string displayStr = str.format("Original string: "{0}"\n\nSubstring array: {1}\n", originalStr, splitStrArray) // Sort the array in ascending order, based on the Unicode values of characters in each string. splitStrArray.sort() // Concatenate a string representing the sorted result. displayStr += str.format("Sorted array: {0}\n", splitStrArray) // Display the final `displayStr` value's text in a label. label.new(bar_index, 0, displayStr, style = label.style_label_center, size = 30, textalign = text.align_left, text_font_family = font.family_monospace)

Note that:

- The `" "` string appears first in the sorted array because standard whitespace and control characters have the *lowest* Unicode values (U+0000 - U+0020). The space character's Unicode value is U+0020.
- ASCII *digits* (U+0030 - U+0039) have *lower* Unicode values than all *letter* characters. Therefore, the sorted array lists all strings that start with digits before those that start with letters.
- *Uppercase* ASCII letters (U+0041 - U+005A) have lower Unicode values than *lowercase* ASCII letters (U+0061 - U+007A). Therefore, strings that start with `A` appear *before* those that start with `a` in the sorted array.
- Some ASCII punctuation marks and symbols have lower Unicode values than ASCII letters or digits, and some others have Unicode values that are between or higher than those of such characters. For instance, the sorted array lists the `"!"` string before other strings except for `" "` because the Unicode value of `!` is U+0021. By contrast, it lists the `"{ABC}"` string at the end because the `{` character's Unicode value is U+007B.

Every array.sort() call directly *changes* the positions of elements in the original array, as demonstrated above. However, in some cases, a programmer might need to access an array's elements in a sorted order *without* rearranging the array itself.

To access an array's sorted elements without modifying the array, programmers can use the array.sort_indices() function. This function creates a *separate* "int" array containing the *indices* of the original array's elements, organized in the *sorted order* for those elements. Scripts can use the indices in the resulting array to read the original array's elements in the specified order (order.ascending by default) while also preserving the original array's unsorted order for other calculations.

The following example script queues close values into a persistent array across the chart. It calls the array.sort_indices() function on the last historical bar to get the ID of an array containing sorted indices, and constructs a string representation of both arrays. Then, it loops through the array of indices using a for…in loop. On each iteration, the script concatenates the string with another string representing a value from the `prices` array, that value's index in the array, and the value's sorted position. It then displays the final string in a label:

//@version=6 indicator("Getting sorted indices demo") //@variable References a persistent array that stores the last 10 `close` values. var array<float> prices = array.new<float>(10) // Push a new value to the end of the array, and remove the oldest (first) element. prices.push(close) prices.shift() if barstate.islastconfirmedhistory //@variable References an "int" array containing the `prices` array indices in ascending order by element value. // The `array.sort_indices()` call maps sorted positions in the array without modifying it. array<int> indices = prices.sort_indices() //@variable A formatted string to display in a label. string displayStr = str.format("Prices: {0}\n\nSorted indices: {1}\n\nSort results:", prices, indices) // Loop through the `indices` array. // The `i` variable stores the current index of the `indices` array's element. // The `index` variable stores that element's value (the index for one of the `prices` array's elements). // Using

`index` to retrieve `prices` array's elements accesses those elements in ascending order. for [i, index] in indices // Concatenate the `displayStr` value with a string representing the sorted `prices` array element, // the original position (index) of the element, and the sorted position of that element. displayStr += str.format("\nPrice: {0,number,0.000}, Original position: {1} -> Sorted position: {2}", prices.get(index), index, i) // Display the final string's text in a label. label.new(bar_index, 0, displayStr, style = label.style_label_center, size = 20, textalign = text.align_left, text_font_family = font.family_monospace)

> NoteIf an “int”, “float”, or “string” array contains elements with `na` values or empty strings (e.g., ""), an array.sort() call moves those elements to the *end* of the array if the `order` argument is order.ascending, or to the *beginning* of the array if the argument is order.descending. Likewise, the array constructed by an array.sort_indices() call stores the indices for na values or empty strings as its *first* or *last* elements, depending on the `order` argument.

Sorting arrays of user-defined types

The array.sort() and array.sort_indices() functions can also sort arrays whose elements refer to objects of user-defined types (UDTs). For such arrays, the functions compare values from one of the “int”, “float”, or “string” *fields* of each object referenced by the array’s elements, using the sorting rules described in the Sorting section above.

The `sort_field` parameter of these functions specifies *which* object field they analyze to sort a UDT array’s elements. The parameter can specify a field using either a *“const int”* or *“const string”* argument:

- A “const int” argument specifies a field by its *field index*, where a value of 0 refers to the *first* field listed in the type declaration, 1 refers to the *second* field, and so on. The value can be any non-negative, non-na number up to one less than the total number of fields.
- A “const string” argument specifies a field by its *identifier (name)*. The string must literally match one of the field names listed in the type declaration.

The default `sort_field` value is 0, meaning that an array.sort() or array.sort_indices() call attempts to compare values from the first field of each object referenced by the specified array if no argument is specified.

The following example script demonstrates the sorting behavior for arrays of UDT elements. The script declares a custom type named `myType` with three fields: `field0`, `field1`, and `field2`. On the last historical bar, it creates five `myType` objects, stores their IDs in an array, then executes an array.sort() call to sort the array in ascending order using each object’s first, second, or third field, depending on the selected inputs. The script loops through the sorted array using a for…in loop to create a custom string representation of its structure, then displays the resulting string’s text in a label:

//@version=6 indicator("Sorting UDT arrays demo") //@type A custom type for creating objects that store "float", "string", and "int" values. type myType float field0 // This field's index is 0. string field1 // This field's index is 1. int field2 // This field's index is 2. //@variable A string to indicate whether the script specifies sorting fields by index or name. string specifyInput = input.string("Index", "Specify a field using its", ["Index", "Name"]) //@variable The index of the field to use for sorting if the `specifyInput` value is `"Index"`. int indexInput = input.int(0, "Field index", 0, 2, active = specifyInput == "Index") //@variable The name of the field to use for sorting if the `specifyInput` value is `"Name"`. string nameInput = input.string("field0", "Field name", ["field0", "field1", "field2"], active = specifyInput == "Name") if barstate.islastconfirmedhistory

//@variable References an array that stores the IDs of `myType` objects. array<myType> udtArray = array.from(myType.new(field0 = 2.0, field1 = "D", field2 = 1), myType.new(field0 = 1.0, field1 = "E", field2 = 2), myType.new(field0 = 3.0, field1 = "C", field2 = 3), myType.new(field0 = 5.0, field1 = "A", field2 = 4), myType.new(field0 = 4.0, field1 = "B", field2 = 5)) // Sort the array in ascending order. Use the field at the specified index if the `specifyInput` value is `"Index"`. if specifyInput == "Index" switch indexInput 0 => udtArray.sort(sort_field = 0) 1 => udtArray.sort(sort_field = 1) 2 => udtArray.sort(sort_field = 2) // Otherwise, sort using the field with the specified name. else switch nameInput "field0" => udtArray.sort(sort_field = "field0") "field1" => udtArray.sort(sort_field = "field1") "field2" => udtArray.sort(sort_field = "field2") //@variable A string representing the structure of the sorted array. string displayStr = switch specifyInput "Index" => str.format("Sorted using field at index {0}\n\n[", indexInput) => str.format("Sorted using field named "{0}"\n\n[", nameInput) // Concatenate formatted strings to represent the array's structure. for [i, id] in udtArray displayStr += str.format(" (field0: {0,number,0.0}, field1: {1}, field2: {2}),\n", id.field0, id.field1, id.field2) // Adjust the final result to align enclosing brackets. displayStr := str.replace(str.substring(displayStr, 0, str.length(displayStr) - 2), "[", "[") + "]" // Display the final string's text in a label. label.new(bar_index, 0, displayStr, style = label.style_label_center, size = 30, textalign = text.align_left, text_font_family = font.family_monospace)

Note that:

- The `sort_field` parameter accepts only values that have the *"const"* qualifier; it cannot accept values qualified as "input", "simple", or "series". Therefore, to sort the array using an input-specified field, this script uses a *separate* array.sort() call for each input combination.

It's important to emphasize that the array.sort() and array.sort_indices() functions can sort UDT arrays only by referencing object fields of the type "int", "float", or "string". They cannot sort elements using fields of any other type.

For example, the following script declares a custom `myColor` type whose first field is of the type "color". It creates an array of `myColor` IDs, then attempts to sort the array using an array.sort() call. The call does not include a `sort_field` argument, so it references each object's *first* field, which is *incompatible* with the sorting algorithm. Consequently, a *compilation error* occurs:

//@version=6 indicator("Incompatible sorting field demo", overlay = true) //@type A custom type for creating objects that contain color information. type myColor color c // Index 0. float r // Index 1. float g // Index 2. float b // Index 3. //@function Creates a new `myColor` instance with pseudorandom field values. randColor() => color c = color.rgb(math.random(0, 128), math.random(128, 255), math.random(128, 255)) myColor.new(c = c, r = color.r(c), g = color.g(c), b = color.b(c)) //@variable References an array of `myColor` IDs. var array<myColor> arr = array.new<myColor>() if barstate.isfirst // Populate the array with 10 `myColor` IDs. for i = 1 to 10 arr.push(randColor()) // Call `array.sort()` using the default `sort_field` argument (0). // This call causes a *compilation error*, because the `array.sort()` function cannot sort "color" values. arr.sort() //@variable The index of the array element to retrieve. int ind = nz(int(math.round(9 * (close - low) / (high - low)))) //@variable The `myColor` ID stored at index `ind`. myColor id = arr.get(ind) // Color the bar using the `id.c` value. barcolor(id.c)

To resolve the error, we can either rearrange the type declaration to list one of the type's "float" fields as the *first* one, or include a `sort_field` argument in the array.sort() call to specify one of those fields. For example:

//@version=6 indicator("Changing first field demo", overlay = true) //@type A custom type for creating objects that contain color information. type myColor float g // Moved to index 0. color c // Moved to index 1. float r // Moved to index 2. float b // Moved to index 3. //@function Creates a new `myColor` instance with

pseudorandom field values. randColor() => color c = color.rgb(math.random(0, 128), math.random(128, 255), math.random(128, 255)) myColor.new(c = c, r = color.r(c), g = color.g(c), b = color.b(c)) //@variable References an array of `myColor` IDs. var array<myColor> arr = array.new<myColor>() if barstate.isfirst // Populate the array with 10 `myColor` IDs. for i = 1 to 10 arr.push(randColor()) // This call does *not* cause an error, because the default `sort_field` argument now refers // to the type's `g` field ("float"). arr.sort() //@variable The index of the array element to retrieve. int ind = nz(int(math.round(9 * (close - low) / (high - low)))) //@variable The `myColor` ID stored at index `ind`. myColor id = arr.get(ind) // Color the bar using the `id.c` value. barcolor(id.c)

The array.sort() and array.sort_indices() functions can sort UDT arrays whose referenced objects have “int”, “float”, or “string” fields that contain `na` values. However, these functions **cannot** sort UDT arrays that contain na *elements*. In a UDT array, an na element represents a *nonexistent ID*, meaning that there is *no associated object* that contains the field required for sorting. Consequently, attempting to sort a UDT array with one or more na elements causes a *runtime error*.

For example, the script below declares a type named `Number` with a single “float” field named `value`. On the last historical bar, it creates an array containing multiple `Number` IDs, two of which are na. Calling array.sort() to rearrange that array causes an error, because the na elements in the array do not refer to valid `Number` objects:

//@version=6 indicator("Cannot sort `na` IDs demo") //@variable A custom type for creating objects that store a single "float" value. type Number float value if barstate.islastconfirmedhistory //@variable References an array of `Number` IDs, two of which are `na`. array<Number> numbers = array.from(Number.new(1.2), na, Number.new(5.4), na, Number.new(3.14)) // This call causes a runtime error. The `na` elements do not refer to valid `Number` objects, so the function // cannot access `value` fields for sorting. numbers.sort() //@variable A string representing the array's structure. string displayStr = "[" // Concatenate a string representing the `value` field from each object referenced by the sorted array. for number in numbers displayStr += str.tostring(number.value) + ", " // Remove the final `", "` sequence and add a closing bracket. displayStr := str.substring(displayStr, 0, str.length(displayStr) - 2) + "]" // Display the resulting string's text in a label. label.new(bar_index, 0, displayStr, style = label.style_label_center, size = 40, textalign = text.align_left)

To prevent such errors, *remove* all na IDs from a UDT array before using the array.sort() or array.sort_indices() function on it, or *replace* them with the IDs of *new* objects that contain na *fields* instead.

For example, the script version below includes a user-defined function named `replaceNa()`, which replaces na `Number` IDs in an array with the IDs of new objects that contain na `value` fields. Using this function before sorting the array with the array.sort() call prevents the runtime error:

//@version=6 indicator("Replacing `na` IDs for sorting demo") //@variable A custom type for creating objects that store a single "float" value. type Number float value //@function Replaces `na` instances in a specified `Number` array with the IDs of new `Number` objects with `na` fields. replaceNa(array<Number> arrID) => if array.includes(arrID, na) // Loop through the array referenced by `arrID`. for [i, objID] in arrID // If the `objID` variable stores `na`, replace the element at index `i` with a new `Number` ID. if na(objID) arrID.set(i, Number.new()) if barstate.islastconfirmedhistory //@variable References an array of `Number` IDs, two of which are `na`. array<Number> numbers = array.from(Number.new(1.2), na, Number.new(5.4), na, Number.new(3.14)) // If we call `replaceNa()` before sorting the array, no error occurs, because all elements now refer // to a valid `Number` object. replaceNa(numbers) numbers.sort() //@variable A string representing the array's structure. string displayStr = "[" // Concatenate a string representing the `value` field from each

object referenced by the sorted array. for number in numbers displayStr += str.tostring(number.value) + ", " // Remove the final `", "` sequence and add a closing bracket. displayStr := str.substring(displayStr, 0, str.length(displayStr) - 2) + "]" // Display the resulting string's text in a label. label.new(bar_index, 0, displayStr, style = label.style_label_center, size = 40, textalign = text.align_left)

Note that:

- This example moves the IDs of all objects with an na `value` field to the *end* of the array because the script's array.sort() call sorts the array's elements in ascending order. If we use order.descending as the `order` argument, those elements move to the *beginning* of the array instead.

Reversing

Use array.reverse() to reverse an array:

//@version=6 indicator("`array.reverse()`") a = array.new<float>(0) array.push(a, 0) array.push(a, 1) array.push(a, 2) if barstate.islast array.reverse(a) label.new(bar_index, 0, "a: " + str.tostring(a))

Copying arrays

A *copy* of an array is a *separate* array object initialized to have the same structure as the original. Any changes to the elements stored by the copy do not affect the contents of the original array. Scripts can create shallow copies for arrays of any type by using the array.copy() function. For reference-type arrays, scripts can also use custom logic to create deep copies whose elements reference separate objects.

> NoteAssigning one variable's stored array ID to another variable *does not* create a copy of the array. Any changes to the array referenced by one of the variables directly affect the one referenced by the other, because both refer to the same array object. See the Modifying variables vs. objects section of the Type system page to learn more about this behavior.

Shallow copies

The array.copy() function creates a *shallow copy* of an array of any type. A shallow copy is a new array object that initially has the same size as the original and stores the *same* values or references (IDs) as elements.

Although a shallow copy initially shares the same structure as the original array, it is an *independent object*. The script can insert new elements and overwrite, remove, reverse, or sort existing elements in the copied array *without* affecting the original array's contents, regardless of the type of elements it contains.

For example, the following script creates an array of "float" values and assigns its ID to a variable named `original` on the last historical bar. It then uses the array.copy() function to create a shallow copy of the array and assigns its ID to a variable named `copy`. The script uses the array.push(), array.shift(), array.set(), and array.sort() functions to modify the copied array, then displays a string representation of the original array and the copy in a label. As shown by the label's text, the copied array's contents change, while the original array remains *unchanged*:

//@version=6 indicator("Modifying shallow copies demo") if barstate.islastconfirmedhistory //@variable References an array of arbitrary "float" values. array<float> original = array.from(-1.0, 4.0, 3.0, 2.0)

```
//@variable References a shallow copy of the `original` array. array<float> copy = array.copy(original)
//#region // The calls in this region modify only the *copied* array. They do not affect the original array, // because the `copy` and `original` variables reference separate objects. // If we change `copy` to `original` in these calls, they affect only the original array, not the copy. // Add a new element to the end of the copy. copy.push(1.0) // Remove the copy's first element. copy.shift() // Assign a new value to the copy's second element. copy.set(1, 3.5) // Sort the elements of the copy in ascending order. array.sort(copy) //#endregion
//@variable A string representation of the original array and the copied array after modification. string displayStr = str.format("Original: {0}\nModified copy: {1}", str.tostring(original), str.tostring(copy)) // Display the text from the formatted string in a label. label.new( bar_index, 0, displayStr, style = label.style_label_center, size = 36, textalign = text.align_left, text_font_family = font.family_monospace )
```

For an array of value types, all changes to the data associated with a shallow copy are completely *separate* from the original array, because the only way to change the data accessed by an element in either array is by replacing the element.

By contrast, it *is* possible to modify the data associated with arrays of reference types through a shallow copy. When the array.copy() function copies an array of object IDs, it does **not** create new copies of the *objects* referenced by those elements; it creates a new array whose elements store the **same IDs** as the original. Consequently, if the script modifies the *object* referenced by one of the elements in the shallow copy, it also modifies the one referenced by the corresponding element in the original array, and vice versa, because *both* elements refer to the *same object* that exists elsewhere in memory.

The following example demonstrates this behavior. The script below creates an array containing the ID of a chart point on the last historical bar, then uses the array.copy() function to create a shallow copy of that array. The script uses the -= and := operators to modify the `index` field of the chart point referenced by the original array and the `price` field of the one referenced by the copy. Then, it displays formatted text containing the chart coordinates retrieved through both arrays inside labels positioned at those coordinates. Both labels anchor to the same location on the chart and display the same values, because the elements in the original array and the shallow copy access the same chart.point object:

```
//@version=6 indicator("Shallow copy of reference-type array demo") //@function Creates a string representing an array of `chart.point` IDs. Each listed item enclosed in parentheses // represents the `index` and `price` coordinates from the referenced chart point. repr(array<chart.point> source) => string result = "[" for item in source result += str.format("(index: {0,number,#}, price: {1,number,#.#####}), ", item.index, item.price) result := str.substring(result, 0, str.length(result) - 2) + "]" if barstate.islastconfirmedhistory
//@variable References an array containing a single `chart.point` ID. array<chart.point> original = array.from(chart.point.from_index(index = bar_index, price = 0)) // The `array.copy()` call below copies the *original* chart point's *ID* into the new array. // It **does not** create a copy of that object for the new array to reference. array<chart.point> shallowCopy = array.copy(original) // Equivalent to: array.from(original.first()) // These variables reference the same chart point as the arrays, not copies of it. chart.point originalPoint = original.first() chart.point pointFromCopy = shallowCopy.first() // These two operations modify the `index` and `price` fields of the **same** chart point, regardless of whether // they access that object using the `originalPoint` or `pointFromCopy` variable: originalPoint.index -= 10 // Equivalent to: pointFromCopy.index -= 10 pointFromCopy.price := 1 // Equivalent to: originalPoint.price := 1 // Create string representations of the two arrays. // Because both arrays reference the *same* object, both strings contain the *same data*. string originalStr = "Original array:\n" + repr(original) string shallowCopyStr = "Shallow copy:\n" + repr(shallowCopy) // Draw labels using the chart point referenced by `originalPoint` and `pointFromCopy`. // The two labels anchor to the same coordinates (x = bar_index - 10, y = 1). label.new(originalPoint, originalStr, size = 36) label.new(pointFromCopy, shallowCopyStr, size = 36, style = label.style_label_up)
```

Note that:

- Assigning the IDs retrieved from an array to separate variables *does not* create copies of the associated objects. For instance, the `originalPoint` and `pointFromCopy` variables in this script reference the *same* chart point as their respective arrays.

Deep copies

A *deep copy* of a reference-type array is a new array that has the same structure as the original, but stores separate elements. Unlike a shallow copy, which stores the *same* object IDs as the original array, a deep copy contains the IDs of *copies* of the original objects. In other words, the deep copy references *different objects* initialized with the same properties as the original objects. Therefore, modifying the objects referenced by the original array or the deep copy does *not* affect the objects referenced by the other array.

Creating a deep copy of an array is necessary only if the array stores elements of a *reference type*, and the script must modify the objects referenced by the copy *without* modifying those referenced by the original, or vice versa. If the original and copied arrays must reference the same objects, or if the script does not modify the array's associated objects, creating a *shallow copy* with the array.copy() function is sufficient.

> NoteFor a value-type array, a shallow copy is semantically the *same* as a deep copy, because values are not referenced entities that the script can modify *outside* the array. The only way to change the data associated with the elements of a value-type array, or its copies, is to *overwrite* those elements.

To create a deep copy of an array, programmers can do either of the following:

- Create a new array of the same type with the array.new<type>() function. For each element in the original array, create a *copy* of the referenced object, then insert the copied object's ID into the new array at the same index.
- Create a shallow copy of the original array with the array.copy() function, then convert the shallow copy to a deep copy by *replacing* each element with the ID of a copied object.

In the example below, we modified the *second* script from the Shallow copies section by adding a user-defined function named `deepCopy()`. The function uses an array.copy() call to create an initial shallow copy of the array, then loops through the copy's elements to convert it to a deep copy. Each iteration creates a shallow copy of the object referenced by the current element, then calls the array.set() function to replace that element with the copied object's ID.

This version of the script calls `deepCopy(original)` to create a deep copy of the original array rather than creating a shallow copy. With this change, modifying the object referenced by the copied array no longer affects the one referenced by the original, because both arrays now reference different objects:

//@version=6 indicator("Deep copies demo") //@function Creates a deep copy of a reference-type array. Each element in the copy refers to a shallow copy of // the object referenced by the corresponding element in the original array. //@param source The ID of an array with elements of the `chart.point` type, a drawing type, or a user-defined type. //@returns The ID of the deep copy. deepCopy(source) => // Create an initial shallow copy of the `source` array. result = source.copy() // Loop through the array elements to convert the new array to a deep copy. for [i, item] in result // Copy each object referenced by `item`, and use the copy's ID as the new element at index `i`. copiedItem = item.copy() result.set(i, copiedItem) // Return the new

array's ID. result //@function Creates a string representing an array of `chart.point` IDs. Each listed item enclosed in parentheses // represents the `index` and `price` coordinates from the referenced chart point. repr(array<chart.point> source) => string result = "[" for item in source result += str.format("(index: {0,number,#}, price: {1,number,#.#####}), ", item.index, item.price) result := str.substring(result, 0, str.length(result) - 2) + "]" if barstate.islastconfirmedhistory //@variable References an array containing a single `chart.point` ID. array<chart.point> original = array.from(chart.point.from_index(index = bar_index, price = 0)) //@variable References a deep copy of the original array. The deep copy stores a *separate* `chart.point` ID. array<chart.point> deepCopy = deepCopy(original) // These variables reference the same chart points as their respective arrays, not separate copies. chart.point originalPoint = original.first() chart.point pointFromCopy = deepCopy.first() // Because the deep copy of the array stores the ID of a separate chart point, these two operations no longer // modify the same object. The first modifies the original chart point, and the second modifies a copy. originalPoint.index -= 10 pointFromCopy.price := 1 // Create string representations of the two arrays. // Because both arrays now reference different objects, the two strings contain different data. string originalStr = "Original array:\n" + repr(original) string deepCopyStr = "Deep copy:\n" + repr(deepCopy) // Draw labels using the chart point referenced by `originalPoint` and `pointFromCopy`. // The two labels now anchor to different coordinates: (x = bar_index - 10, y = 0) and (x = bar_index, y = 1). label.new(originalPoint, originalStr, size = 36) label.new(pointFromCopy, deepCopyStr, size = 36, style = label.style_label_up)

Note that:

- This script's `deepCopy()` function is compatible with arrays of user-defined types, drawing types, and chart points. However, it cannot copy arrays of footprint or volume_row elements, because Pine Script does not include built-in `*.copy()` methods for those types.

Slicing arrays

A *slice* of an array is a separate array that references the original array's elements across a specific index range. Unlike a copy, which *duplicates* an original array's structure and stores the data in a *different* location, a slice provides *direct access* to a subset of the original array's contents. Changing the elements in a slice directly changes the elements in the original array, and vice versa.

Scripts can create slices of arrays of any type by using the array.slice() function. The function's `index_from` and `index_to` parameters specify the index range of the elements that the resulting slice references from the original array. The `index_from` parameter accepts an "int" value from 0 to *one less* than the array's size. The `index_to` parameter requires an "int" value that is greater than the `index_from` value and less than or equal to the array's size. The slice references all the original array's elements from the index specified by the `index_from` argument to the index that is *one less* than the `index_to` argument.

As with the element indices in a typical array, the indices in a slice range from 0 to one less than the slice's size. However, these indices **do not** directly represent the *same* element indices in the original array. Each element in the slice corresponds to the original array's element at the index `index_from + i`, where `i` is the index of the slice's element. For instance, if the `index_from` value is *2*, the slice's element at index 1 refers to the original array's element at index *3*.

Programmers often use slices to directly overwrite, remove, reverse, or sort elements in an array over a specific index range, or to perform calculations that require only a specific subset of an array's elements.

The following example demonstrates the unique behaviors of array slices. On the last historical bar, the script creates an array of “int” values. Then, it creates a slice of that array to reference the elements from index 1 to index 3. The script modifies the original array to change the slice’s contents, then modifies the slice to change the contents of the original array. The script displays string representations of both arrays after each change in a table in the center of the chart pane:

```
//@version=6 indicator("Slicing demo") //@function Creates a table row to display information about a main array and its slice. method makeRow(table this, int row, string note, array<int> main, array<int> slice) => this.cell(0, row, note, text_color = chart.fg_color) this.cell( 1, row, str.format("Main: {0}\n\nSlice: {1}", main, slice), text_color = chart.fg_color, text_halign = text.align_left ) if barstate.islastconfirmedhistory //@variable References an array of integers. array<int> mainArray = array.from(1, 2, 3, 4, 5, 6, 7) //@variable References a slice of the original array from index 1 to index 3 (index 4 is not included). array<int> slice = array.slice(mainArray, 1, 4) // Initialize the display table and populate its first row. table displayTable = table.new(position.middle_center, 2, 10, border_color = chart.fg_color, border_width = 1) displayTable.makeRow(0, "Initial arrays", mainArray, slice) //#region // Any changes that affect the original array's elements at indices 1 through 3 directly affect the // contents of the slice. // Adding or removing elements in the main array changes the elements in the slice's range. mainArray.unshift(0) displayTable.makeRow(1, "Insert a new element at the start of the main array", mainArray, slice) mainArray.remove(2) displayTable.makeRow(2, "Remove the main array element at index 2", mainArray, slice) // If the original array no longer contains elements in the slice's range, the size of the slice reduces. for i = 1 to 6 mainArray.pop() displayTable.makeRow(3, "Reduce the main array to one element", mainArray, slice) // The slice's size increases again for each element added to the original array in the required index range. for i = 1 to 7 mainArray.push(i) displayTable.makeRow(4, "Push 7 new elements into the main array", mainArray, slice) //#endregion //#region // Modifying the contents of the slice directly modifies the original array's elements in the index range // covered by the slice. // Updating the first element in the slice changes the element at index 2 in the original array. slice.set(0, -1) displayTable.makeRow(5, "Update the first element in the slice", mainArray, slice) // Inserting an element into the slice increases its index range and adds a new element to the original array. slice.push(-3) displayTable.makeRow(6, "Add a new element to the end of the slice", mainArray, slice) // Removing an element from the slice removes the same element from the original array. slice.shift() displayTable.makeRow(7, "Remove the slice's first element", mainArray, slice) // Rearranging the slice also rearranges the original array's elements in the covered range. slice.sort(order.descending) displayTable.makeRow(8, "Sort the slice in descending order", mainArray, slice) //#endregion // Reassigning the `mainArray` variable does not affect the slice. The slice continues referencing the original // array's elements, while the `mainArray` variable now references a completely separate array. mainArray := array.new<int>(5, 1) displayTable.makeRow(9, "Assign a new array ID to `mainArray`", mainArray, slice)
```

Note that:

- If the original array no longer contains elements at the indices required by the slice, the slice’s size is automatically *reduced*, as shown by row 3 in the table. However, the script can re-add elements to restore the slice to its *maximum* size, as shown by row 4.
- The maximum index range covered by a slice changes only if the script explicitly *inserts* new elements into the slice or *removes* existing ones, as shown by rows 6 and 7.
- As shown by the final table row, *reassigning* an array variable that a script uses to create a slice does *not* affect that slice. A slice always references a subset of the elements from a specific array; it is not associated with any variables or fields that store that array’s ID.

> TipIf a script must access a portion of an array's data and *modify* that data without affecting the contents of the original array, create a shallow copy of the slice by using the array.copy() function, or use custom logic to create a deep copy. See the Copying arrays section above to learn more about copies and how they differ from slices.

Searching arrays

We can test if a value is part of an array with the array.includes() function, which returns true if the element is found. We can find the first occurrence of a value in an array by using the array.indexof() function. The first occurrence is the one with the lowest index. We can also find the last occurrence of a value with array.lastindexof():

//@version=6 indicator("Searching in arrays") valueInput = input.int(1) a = array.new<float>(0) array.push(a, 0) array.push(a, 1) array.push(a, 2) array.push(a, 1) if barstate.islast valueFound = array.includes(a, valueInput) firstIndexFound = array.indexof(a, valueInput) lastIndexFound = array.lastindexof(a, valueInput) label.new(bar_index, 0, "a: " + str.tostring(a) + "\nFirst " + str.tostring(valueInput) + (firstIndexFound != -1 ? " value was found at index: " + str.tostring(firstIndexFound) : " value was not found.") + "\nLast " + str.tostring(valueInput) + (lastIndexFound != -1 ? " value was found at index: " + str.tostring(lastIndexFound) : " value was not found."))

We can also perform a binary search on an array but note that performing a binary search on an array means that the array will first need to be sorted in ascending order only. The array.binary_search() function will return the value's index if it was found or -1 if it wasn't. If we want to always return an existing index from the array even if our chosen value wasn't found, then we can use one of the other binary search functions available. The array.binary_search_leftmost() function, which returns an index if the value was found or the first index to the left where the value would be found. The array.binary_search_rightmost() function is almost identical and returns an index if the value was found or the first index to the right where the value would be found.

> NoticeSearch functions like array.indexof() and array.binary_search() return an array index if the requested element is found, or `-1` if it's not present. Note that these functions only return *positive indices*, while other functions like array.get() accept *both* positive and negative indices. Ensure that scripts do **not** misconstrue a search function's returned `-1` result as a negative index in their subsequent logic.

Error handling

Malformed `array.*()` call syntax in Pine scripts will cause the usual **compiler** error messages to appear in Pine Editor's console, at the bottom of the window, when you save a script. Refer to the Pine Script v6 Reference Manual when in doubt regarding the exact syntax of function calls.

Scripts using arrays can also throw **runtime** errors, which appear as an exclamation mark next to the indicator's name on the chart. We discuss some of the most common runtime errors in this section.

Index xx is out of bounds. Array size is yy

This error is the most frequent one programmers encounter when using arrays. The error occurs when the script references a *nonexistent* array index. The “xx” value represents the out-of-bounds index the function tried to use, and “yy” represents the array’s size. Recall that array indices start at zero — not one — and end at the array’s size, minus one. For instance, the last valid index in a three-element array is `2`.

To avoid this error, you must make provisions in your code logic to prevent using an index value outside the array’s boundaries. This code example generates the error because the last `i` value in the loop’s iterations is beyond the valid index range for the `a` array:

```
//@version=6 indicator("Out of bounds index") a = array.new<float>(3) for i = 1 to 3 array.set(a, i, i) plot(array.pop(a))
```

To resolve the error, last `i` value in the loop statement should be less than or equal to 2:

```
for i = 0 to 2
```

To iterate over all elements in an array of *unknown* size with a for loop, set the loop counter’s final value to one less than the array.size() value:

```
//@version=6 indicator("Protected `for` loop") sizeInput = input.int(0, "Array size", minval = 0, maxval = 100000) a = array.new<float>(sizeInput) for i = 0 to (array.size(a) == 0 ? na : array.size(a) - 1) array.set(a, i, i) plot(array.pop(a))
```

When sizing arrays dynamically using a field in the script's *Settings/Inputs* tab, protect the boundaries of that value using input.int()'s `minval` and `maxval` parameters:

```
//@version=6 indicator("Protected array size") sizeInput = input.int(10, "Array size", minval = 1, maxval = 100000) a = array.new<float>(sizeInput) for i = 0 to sizeInput - 1 array.set(a, i, i) plot(array.size(a))
```

See the Looping through array elements section of this page for more information.

Cannot call array methods when ID of array is 'na'

If an array variable is initialized with na, using `array.*()` functions on that variable is *not allowed*, because the variable does not store the ID of an existing array. Note that an empty array containing no elements still has a valid ID. A variable that references an empty array still holds a valid ID, whereas a variable that stores na does not. The code below demonstrates this error:

```
//@version=6 indicator("Array methods on `na` array") array<int> a = na array.push(a, 111) label.new(bar_index, 0, "a: " + str.tostring(a))
```

To avoid the error, create an empty array and assign its reference to the variable instead. For example:

```
array<int> a = array.new<int>(0)
```

Note that the `array<int>` type identifier in the above declaration is optional. We can define the variable without it. For example:

```
a = array.new<int>(0)
```

Array is too large. Maximum size is 100000

This error appears if your code attempts to declare an array with a size greater than 100,000. It will also occur if, while dynamically appending elements to an array, a new element would increase the array's size past the maximum.

Cannot create an array with a negative size

We haven't found any use for arrays of negative size yet, but if you ever do, we may allow them :)

Cannot use shift() if array is empty.

This error occurs if array.shift() is called to remove the first element of an empty array.

Cannot use pop() if array is empty.

This error occurs if array.pop() is called to remove the last element of an empty array.

Index 'from' should be less than index 'to'

When two indices are used in functions such as array.slice(), the first index must always be smaller than the second one.

Slice is out of bounds of the parent array

This message occurs whenever the parent array’s size is modified in such a way that it makes the shallow copy created by a slice point outside the boundaries of the parent array. This code will reproduce it because after creating a slice from index 3 to 4 (the last two elements of our five-element parent array), we remove the parent’s first element, making its size four and its last index 3. From that moment on, the shallow copy which is still pointing to the “window” at the parent array’s indices 3 to 4, is pointing out of the parent array’s boundaries:

//@version=6 indicator("Slice out of bounds") a = array.new<float>(5, 0) b = array.slice(a, 3, 5) array.remove(a, 0) c = array.indexof(b, 2) plot(c)

Built-ins

Introduction

Pine Script® has hundreds of *built-in* variables and functions. They provide your scripts with valuable information and make calculations for you, dispensing you from coding them. The better you know the built-ins, the more you will be able to do with your Pine scripts.

On this page, we present an overview of some of Pine's built-in variables and functions. They will be covered in more detail in the pages of this manual covering specific themes.

All built-in variables and functions are defined in the Pine Script v6 Reference Manual. It is called a "Reference Manual" because it is the definitive reference on the Pine Script language. It is an essential tool that will accompany you anytime you code in Pine, whether you are a beginner or an expert. If you are learning your first programming language, make the Reference Manual your friend. Ignoring it will make your programming experience with Pine Script difficult and frustrating — as it would with any other programming language.

Variables and functions in the same family share the same *namespace*, which is a prefix to the function's name. The ta.sma() function, for example, is in the `ta` namespace, which stands for "technical analysis". A namespace can contain both variables and functions.

Some variables have function versions as well, e.g.:

- The ta.tr variable returns the "True Range" of the current bar. The ta.tr(true) function call also returns the "True Range", but when the previous close value which is normally needed to calculate it is na, it calculates using `high - low` instead.
- The time variable gives the time at the open of the current bar. The time(timeframe) function returns the time of the bar's open from the `timeframe` specified, even if the chart's timeframe is different. The time(timeframe, session) function returns the time of the bar's open from the `timeframe` specified, but only if it is within the `session` time. The time(timeframe, session, timezone) function returns the time of the bar's open from the `timeframe` specified, but only if it is within the `session` time in the specified `timezone`.

Built-in variables

Built-in variables exist for different purposes. These are a few examples:

- Price- and volume-related variables: open, high, low, close, hl2, hlc3, ohlc4, and volume.
- Symbol-related information in the `syminfo` namespace: syminfo.basecurrency, syminfo.currency, syminfo.description, syminfo.main_tickerid, syminfo.mincontract, syminfo.mintick, syminfo.pointvalue, syminfo.prefix, syminfo.root, syminfo.session, syminfo.ticker, syminfo.tickerid, syminfo.timezone, and syminfo.type.
- Timeframe (a.k.a. "interval" or "resolution", e.g., 15sec, 30min, 60min, 1D, 3M) variables in the `timeframe` namespace: timeframe.isseconds, timeframe.isminutes, timeframe.isintraday, timeframe.isdaily, timeframe.isweekly, timeframe.ismonthly, timeframe.isdwm, timeframe.multiplier, timeframe.main_period, and timeframe.period.

- Bar states in the `barstate` namespace (see the Bar states page): barstate.isconfirmed, barstate.isfirst, barstate.ishistory, barstate.islast, barstate.islastconfirmedhistory, barstate.isnew, and barstate.isrealtime.
- Strategy-related information in the `strategy` namespace: strategy.equity, strategy.initial_capital, strategy.grossloss, strategy.grossprofit, strategy.wintrades, strategy.losstrades, strategy.position_size, strategy.position_avg_price, strategy.wintrades, etc.

Built-in functions

Many functions are used for the result(s) they return. These are a few examples:

- Math-related functions in the `math` namespace: math.abs(), math.log(), math.max(), math.random(), math.round_to_mintick(), etc.
- Technical indicators in the `ta` namespace: ta.sma(), ta.ema(), ta.macd(), ta.rsi(), ta.supertrend(), etc.
- Support functions often used to calculate technical indicators in the `ta` namespace: ta.barssince(), ta.crossover(), ta.highest(), etc.
- Functions to request data from other symbols or timeframes in the `request` namespace: request.dividends(), request.earnings(), request.financial(), request.quandl(), request.security(), request.splits().
- Functions to manipulate strings in the `str` namespace: str.format(), str.length(), str.tonumber(), str.tostring(), etc.
- Functions used to define the input values that script users can modify in the script's “Settings/Inputs” tab, in the `input` namespace: input(), input.color(), input.int(), input.session(), input.symbol(), etc.
- Functions used to manipulate colors in the `color` namespace: color.from_gradient(), color.rgb(), color.new(), etc.

Some functions do not return a result but are used for their side effects, which means they do something, even if they don't return a result:

- Functions used as a declaration statement defining one of three types of Pine scripts, and its properties. Each script must begin with a call to one of these functions: indicator(), strategy() or library().
- Plotting or coloring functions: bgcolor(), plotbar(), plotcandle(), plotchar(), plotshape(), fill().
- Strategy functions placing orders, in the `strategy` namespace: strategy.cancel(), strategy.close(), strategy.entry(), strategy.exit(), strategy.order(), etc.
- Strategy functions returning information on indivdual past trades, in the `strategy` namespace: strategy.closedtrades.entry_bar_index(), strategy.closedtrades.entry_price(), strategy.closedtrades.entry_time(), strategy.closedtrades.exit_bar_index(), strategy.closedtrades.max_drawdown(), strategy.closedtrades.max_runup(), strategy.closedtrades.profit(), etc.
- Functions to generate alert events: alert() and alertcondition().

Other functions return a result, but we don't always use it, e.g.: hline(), plot(), array.pop(), label.new(), etc.

All built-in functions are defined in the Pine Script v6 Reference Manual. You can click on any of the function names listed here to go to its entry in the Reference Manual, which documents the function's signature, i.e., the list of *parameters* it accepts and the qualified type of the value(s) it returns (a function can return more than one result). The Reference Manual entry will also list, for each parameter:

- Its name.
- The qualified type of the value it requires (we use *argument* to name the values passed to a function when calling it).
- If the parameter is required or not.

All built-in functions have one or more parameters defined in their signature. Not all parameters are required for every function.

Let's look at the ta.vwma() function, which returns the volume-weighted moving average of a source value. This is its entry in the Reference Manual:

The entry gives us the information we need to use it:

- What the function does.
- Its signature (or definition):

```
ta.vwma(source, length) → series float
```

- The parameters it includes: `source` and `length`
- The qualified type of the result it returns: “series float”.
- An example showing it in use: `plot(ta.vwma(close, 15))`.
- An example showing what it does, but in long form, so you can better understand its calculations. Note that this is meant to explain — not as usable code, because it is more complicated and takes longer to execute. There are only disadvantages to using the long form.
- The “RETURNS” section explains exacty what value the function returns.
- The “ARGUMENTS” section lists each parameter and gives the critical information concerning what qualified type is required for arguments used when calling the function.
- The “SEE ALSO” section refers you to related Reference Manual entries.

This is a call to the function in a line of code that declares a `myVwma` variable and assigns the result of `ta.vwma(close, 20)` to it:

myVwma = ta.vwma(close, 20)

Note that:

- We use the built-in variable close as the argument for the `source` parameter.
- We use `20` as the argument for the `length` parameter.
- If placed in the global scope (i.e., starting in a line's first position), it will be executed by the Pine Script runtime on each bar of the chart.

We can also use the parameter names when calling the function. Parameter names are called *keyword arguments* when used in a function call:

myVwma = ta.vwma(source = close, length = 20)

You can change the position of arguments when using keyword arguments, but only if you use them for all your arguments. When calling functions with many parameters such as indicator(), you can also forego keyword arguments for the first arguments, as long as you don't skip any. If you skip some, you must then use keyword arguments so the Pine Script compiler can figure out which parameter they correspond to, e.g.:

indicator("Example", "Ex", true, max_bars_back = 100)

Mixing things up this way is not allowed:

indicator(precision = 3, "Example") // Compilation error!

When calling built-ins, it is critical to ensure that the arguments you use are of the required qualified type, which will vary for each parameter.

To learn how to do this, one needs to understand Pine Script's type system. The Reference Manual entry for each built-in function includes an “ARGUMENTS” section which lists the qualified type required for the argument supplied to each of the function's parameters.

Conditional structures

Introduction

The conditional structures in Pine Script® are if, switch, and once.

Scripts can use all of these structures for their side effects, i.e., the actions they perform, like reassigning values to variables, or calling functions. The if and switch structures can also return a value or a tuple which can then be assigned to a variable (or multiple variables in the case of tuples). The once structure cannot return a value.

Scripts can use an if, switch, or once structure inside the local scope of a loop, function, conditional structure, or other structure.

Some Pine Script built-in functions are **not** callable from within the local blocks of conditional structures, including barcolor(), bgcolor(), plot(), plotshape(), plotchar(), plotarrow(), plotcandle(), plotbar(), hline(), fill(), alertcondition(), indicator(), strategy(), and library().

This restriction does not entail their functionality cannot be controlled by conditions evaluated by your script — only that it cannot be done by including them in conditional structures. Note that while `input*.()` function calls are allowed in local blocks, their functionality is the same as if they were in the script's *global scope*.

The local blocks in conditional structures must be indented by four spaces or a tab.

`if` structure

`if` used for its side effects

An if structure used for its side effects has the following syntax:

```
if <expression>
    <local_block>
{else if <expression>
    <local_block>}
[else
    <local_block>]
```

where:

- Parts enclosed in square brackets (`[]`) can appear zero or one time, and those enclosed in curly braces (`{}`) can appear zero or more times.
- <expression> must be of "bool" type or be auto-castable to that type, which is only possible for "int" or "float" values (see the Type system page).
- <local_block> consists of zero or more statements followed by a return value, which can be a tuple of values. It must be indented by four spaces or a tab.
- There can be zero or more `else if` clauses.
- There can be zero or one `else` clause.

When the <expression> following the if evaluates to true, the first local block is executed, the if structure's execution ends, and the value(s) evaluated at the end of the local block are returned.

When the <expression> following the if evaluates to false, the successive `else if` clauses are evaluated, if there are any. When the <expression> of one evaluates to true, its local block is executed, the if structure's execution ends, and the value(s) evaluated at the end of the local block are returned.

When no <expression> has evaluated to true and an `else` clause exists, its local block is executed, the if structure's execution ends, and the value(s) evaluated at the end of the local block are returned.

When no <expression> has evaluated to true and no `else` clause exists, na is returned. The only exception to this is if the structure returns “bool” values – in that case, false is returned instead.

Using if structures for their side effects can be useful to manage the order flow in strategies, for example. While the same functionality can often be achieved using the `when` parameter in `strategy.*()` calls, code using if structures is easier to read:

if (ta.crossover(source, lower)) strategy.entry("BBandLE", strategy.long, stop=lower, oca_name="BollingerBands", oca_type=strategy.oca.cancel, comment="BBandLE") else strategy.cancel(id="BBandLE")

Restricting the execution of your code to specific bars can be done using if structures, as we do here to restrict updates to our label to the chart's last bar:

//@version=6 indicator("", "", true) var ourLabel = label.new(bar_index, na, na, color = color(na), textcolor = color.orange) if barstate.islast label.set_xy(ourLabel, bar_index + 2, hl2[1]) label.set_text(ourLabel, str.tostring(bar_index + 1, "# bars in chart"))

Note that:

- We initialize the `ourLabel` variable on the script's first bar only, as we use the var declaration mode. The value used to initialize the variable is provided by the label.new() function call, which returns a label ID pointing to the label it creates. We use that call to set the label's properties because once set, they will persist until we change them.
- What happens next is that on each successive bar the Pine Script runtime will skip the initialization of `ourLabel`, and the if structure's condition (barstate.islast) is evaluated. It returns `false` on all bars until the last one, so the script does nothing on most historical bars after bar zero.
- On the last bar, barstate.islast becomes true and the structure's local block executes, modifying on each chart update the properties of our label, which displays the number of bars in the dataset.
- We want to display the label's text without a background, so we make the label's background na in the label.new() function call, and we use `hl2[1]` for the label's *y* position because we don't want it to move all the time. By using the average of the **previous** bar's high and low values, the label doesn't move until the moment when the next realtime bar opens.
- We use `bar_index + 2` in our label.set_xy() call to offset the label to the right by two bars.

`if` used to return a value

An if structure used to return one or more values has the following syntax:

```
[<declaration_mode>] [<type>] <identifier> = if <expression>
```

```
    <local_block>
{else if <expression>
    <local_block>}
[else
    <local_block>]
```

where:

- Parts enclosed in square brackets (`[]`) can appear zero or one time, and those enclosed in curly braces (`{}`) can appear zero or more times.
- <declaration_mode> is the variable's declaration mode
- <type> is optional, as in almost all Pine Script variable declarations (see types)
- <identifier> is the variable's name
- <expression> can be a literal, a variable, an expression or a function call.
- <local_block> consists of zero or more statements followed by a return value, which can be a tuple of values. It must be indented by four spaces or a tab.
- The value assigned to the variable is the return value of the <local_block>, or na if no local block is executed. If other local blocks return “bool” values, false will be returned instead.

This is an example:

//@version=6 indicator("", "", true) string barState = if barstate.islastconfirmedhistory "islastconfirmedhistory" else if barstate.isnew "isnew" else if barstate.isrealtime "isrealtime" else "other" f_print(_text) => var table _t = table.new(position.middle_right, 1, 1) table.cell(_t, 0, 0, _text, bgcolor = color.yellow) f_print(barState)

It is possible to omit the *else* block. In this case, if the `condition` is false, an *empty* value (`na`, `false`, or `""`) will be assigned to the `var_declarationX` variable.

This is an example showing how na is returned when no local block is executed. If `close > open` is `false` in here, na is returned:

x = if close > open close

Scripts can contain `if` structures with nested `if` and other conditional structures. For example:

if condition1 if condition2 if condition3 expression

However, nesting these structures is not recommended from a performance perspective. When possible, it is typically more optimal to compose a single `if` statement with multiple logical operators rather than several nested `if` blocks:

if condition1 and condition2 and condition3 expression

`switch` structure

The switch structure exists in two forms. One switches on the different values of a key expression:

```
[[<declaration_mode>] [<type>] <identifier> = ]switch <expression>
    {<expression> => <local_block>}
```

```
    => <local_block>
```

The other form does not use an expression as a key; it switches on the evaluation of different expressions:

```
[[<declaration_mode>] [<type>] <identifier> = ]switch
    {<expression> => <local_block>}
    => <local_block>
```

where:

- Parts enclosed in square brackets (`[]`) can appear zero or one time, and those enclosed in curly braces (`{}`) can appear zero or more times.
- <declaration_mode> is the variable's declaration mode
- <type> is optional, as in almost all Pine Script variable declarations (see types)
- <identifier> is the variable's name
- <expression> can be a literal, a variable, an expression or a function call.
- <local_block> consists of zero or more statements followed by a return value, which can be a tuple of values. It must be indented by four spaces or a tab.
- The value assigned to the variable is the return value of the <local_block>, or na if no local block is executed.
- The `=> <local_block>` at the end allows you to specify a return value which acts as a default to be used when no other case in the structure is executed.

Only one local block of a switch structure is executed. It is thus a *structured switch* that doesn't *fall through* cases. Consequently, `break` statements are unnecessary.

Both forms are allowed as the value used to initialize a variable.

As with the if structure, if no local block is executed, the expression returns either false (when other local blocks return a “bool” value) or na (in all other cases).

`switch` with an expression

Let's look at an example of a switch using an expression:

//@version=6 indicator("Switch using an expression", "", true) string maType = input.string("EMA", "MA type", options = ["EMA", "SMA", "RMA", "WMA"]) int maLength = input.int(10, "MA length", minval = 2) float ma = switch maType "EMA" => ta.ema(close, maLength) "SMA" => ta.sma(close, maLength) "RMA" => ta.rma(close, maLength) "WMA" => ta.wma(close, maLength) => runtime.error("No matching MA type found.") float(na) plot(ma)

Note that:

- The expression we are switching on is the variable `maType`, which is of “input int” type (see here for an explanation of what the “input” qualifier is). Since it cannot change during the execution of the script, this guarantees that whichever MA type the user selects will be executing on each bar, which is a requirement for functions like ta.ema() which require a “simple int” argument for their `length` parameter.

- If no matching value is found for `maType`, the switch executes the last local block introduced by `=>`, which acts as a catch-all. We generate a runtime error in that block. We also end it with `float(na)` so the local block returns a value whose type is compatible with that of the other local blocks in the structure, to avoid a compilation error.

`switch` without an expression

This is an example of a switch structure which does not use an expression:

//@version=6 strategy("Switch without an expression", "", true) bool longCondition = ta.crossover(ta.sma(close, 14), ta.sma(close, 28)) bool shortCondition = ta.crossunder(ta.sma(close, 14), ta.sma(close, 28)) switch longCondition => strategy.entry("Long ID", strategy.long) shortCondition => strategy.entry("Short ID", strategy.short)

Note that:

- We are using the switch to select the appropriate strategy order to emit, depending on whether the `longCondition` or `shortCondition` "bool" variables are `true`.
- The building conditions of `longCondition` and `shortCondition` are exclusive. While they can both be `false` simultaneously, they cannot be `true` at the same time. The fact that only **one** local block of the switch structure is ever executed is thus not an issue for us.
- We evaluate the calls to ta.crossover() and ta.crossunder() **prior** to entry in the switch structure. Not doing so, as in the following example, would prevent the functions being executed on each bar, which would result in a compiler warning and erratic behavior:

//@version=6 strategy("Switch without an expression", "", true) switch // Compiler warning! Will not calculate correctly! ta.crossover(ta.sma(close, 14), ta.sma(close, 28)) => strategy.entry("Long ID", strategy.long) ta.crossunder(ta.sma(close, 14), ta.sma(close, 28)) => strategy.entry("Short ID", strategy.short)

`once` structure

Unlike the if and switch structures, which are evaluated on every execution of their containing scope, the once structure is evaluated only once, when its condition is true for the first time, and then never again after that. See the `once` on the realtime bar section below for details of the behavior of the once structure on the realtime bar, where it *can* fire more than once.

The once structure has the following syntax:

```
once [<condition>]
    <statements>
```

where:

- `<condition>` is an optional parameter, `true` by default, that takes an argument of type "series bool". This conditional expression controls the structure's execution. The structure executes its local block when the expression evaluates to `true`. It prevents additional executions of the block after the condition is `true` for the first time on a closed bar.
- `<statements>` is the block of statements and expressions that execute when the condition evaluates to `true`. The block must be indented by four spaces or a single tab.

The following example uses the once structure to draw a label the first time that price closes above a moving average. It also draws another label on the same condition, with equivalent code using an if structure and a Boolean flag:

//@version=6 indicator("`once` simple demo", overlay = true) //@variable The simple moving average of `close` prices, using the input length. mySMA = ta.sma(close, 20) //@function Prints a label above or below the current bar at a given price with the supplied text. printLabel(float y, string txt, bool isAbove = true) => label.new(bar_index, y, txt, xloc.bar_index, yloc.price, #2195f382, isAbove ? label.style_label_down : label.style_label_up, chart.fg_color) // The first time price closes above the SMA, print a label. once close > mySMA printLabel(mySMA, "First close above\nthe moving average\ndetected using `once`.") //@variable This persistent flag is `true` if we have printed a label; `false` otherwise. var bool printedLabel = false // The first time price closes above the SMA, print a label. if close > mySMA and printedLabel == false printLabel(mySMA, "First close above \nthe moving average\ndetected using `if` plus a flag.", false) printedLabel := true // Set the flag so that this `if` structure does not fire again. // Plot the SMA line. plot(mySMA)

Note that:

- The equivalent logic using the if structure and a Boolean flag achieves the same result, but uses more code and is less performant.

Unlike the other conditional structures, the if and switch structures, the once structure does *not* return usable values. Scripts cannot assign a `once` statement or a call to a function that ends with a `once` statement to a variable or a tuple of variables.

The following example script demonstrates this limitation. The first structure correctly reassigns a “string” variable *inside* a once block. The second structure uses an if structure and a Boolean flag to emulate a once block; it *returns* a value that is additively assigned to the same “string” variable. The third structure is commented out. It attempts to *return* a string from a once block and additively assign it the same “string” variable as the previous structures. Uncommenting this third structure causes a *compilation error*, because a once structure *cannot* return usable values. When the script is in a form that compiles, it displays the final string in a label:

//@version=6 indicator("`once` compilation error demo", overlay = true) //@variable The text to display in the label. var string txt = "" // Structure 1: Assign a success message to the `txt` variable once, *inside* the `once` structure. once close > open txt := "Structure 1 works." //@variable A flag indicating whether Structure 2 fired. var bool condition2Fired = false // Structure 2: If this condition has not fired before, set the `condition2Fired` flag to `true` and return a // success message. txt += if close > open and not condition2Fired condition2Fired := true // Set the flag. "\nStructure 2 works." // Return a string. It is appended to the `txt` variable's value. // // Uncommenting this section causes a compilation error, because it attempts to use the returned value from // // the `once` structure, and `once` structures cannot return values. // txt += once close > open // txt := "\nStructure 3 works." // Draw a label on the last bar showing the `txt` string. if barstate.islast label.new(bar_index + 1, open, txt, style = label.style_label_left)

`once` on the realtime bar

On the live realtime bar, a once structure that has not yet fired is evaluated on *each tick*. If the structure’s condition is `true`, its statements are executed. However, unless that tick is the *closing* tick of the bar, the structure’s state is *reset* by Pine’s rollback process on the bar’s next tick. If the structure’s condition is `true` on a subsequent tick of the same bar, its statements are executed again.

In this way, a once structure can execute its statements multiple times on one realtime bar.

The block must execute on a bar's *closing tick* in order to make the once structure inactive. An inactive once structure never executes its statements again.

This behavior can affect code that is *not* reset by rollback, including varip variables, function calls that create Pine Logs, strategy commands, and `alert()` calls. Refer to the Executions on realtime bars section of our User Manual for advanced details about rollback and its exceptions.

If a script uses code that is not reset by rollback, one way to prevent multiple executions of a once structure is to append `and barstate.isconfirmed` to its condition. This addition means that the condition can be `true` only on the closing tick of a realtime bar, or on a historical bar.

The following example script demonstrates this behavior. It declares two different counter variables using the varip keyword, so that their values persist across intrabar rollbacks. It increments these counters inside two once blocks. Both structures are inside an `if barstate.isrealtime` block so that they are active when the script starts executing on its first realtime bar. The first once structure uses an empty condition, so it increases its counter and logs its message on every tick of the first realtime bar. The second once block uses `barstate.isconfirmed` as its condition, so it increases its counter and logs its message only on the *closing tick* of the first realtime bar. We can see that both counters are zero on historical bars. On the first realtime bar on which the script executes, the on-close counter increments to 1 because the block that increments it runs only once, while the per-tick counter increases beyond 1 because its block runs several times during the bar. On subsequent bars, the values of both counters persist without incrementing, because the once blocks cannot run again. The log messages show the same behavior as the plots:

//@version=6 indicator("`once` upon a realtime bar") //@variable counts how many times the first `count` block executes per bar. varip int realTimeCount = 0 //@variable counts how many times the second `count` block, which runs only on the closing tick, executes per bar. varip int closingTickCount = 0 // Evaluate our `once` blocks only on realtime bars, so that a closed bar does not fire them once and forever. if barstate.isrealtime // Fires on every tick of the first realtime bar on which the script executes. once realTimeCount += 1 // Increase the realtime count by one. log.info("Close > open on this tick.") // Log a message. // Fires on the *closing tick* of the first realtime bar on which the script executes. once barstate.isconfirmed closingTickCount += 1 // Increase the closing count by one. log.info("🕛 Close > open on the closing tick.") // Log a message. // Plot both counts. plot(realTimeCount, "Once per realtime tick") plot(closingTickCount, "Once per realtime bar", color.red) // Plot a zero line. hline(0, linestyle = hline.style_dotted)

Note that:

- If the counter variables are instead declared using the var keyword, they both increment to 1 and remain there, because their values are *rolled back* after each realtime tick.

Matching local block type requirement

When multiple local blocks are used in structures, the type of the return value of all its local blocks must match. This applies only if the structure is used to assign a value to a variable in a declaration, because a variable can only have one type, and if the statement returns two incompatible types in its branches, the variable type cannot be properly determined. If the structure is not assigned anywhere, its branches can return different values.

This code compiles fine because close and open are both of the `float` type:

```
x = if close > open close else open
```

This code does not compile because the first local block returns a `float` value, while the second one returns a `string`, and the result of the `if`-statement is assigned to the `x` variable:

```
// Compilation error! x = if close > open close else "open"
```

Declaration statements

Introduction

In Pine Script®, a *declaration statement* is a mandatory function call that declares the script's *type* and its *properties* at *compile time*. The available declaration functions are indicator(), strategy(), and library(). Each type of script has different capabilities and behaviors, the compiler uses different rules to compile them, and Pine's runtime system also executes them differently.

Every script must include exactly **one** declaration statement, and that statement must be in the script's global scope. Our style guide recommends placing the statement directly below the `//@version=` compiler annotation at the top of the source code. For example:

//@version=6 indicator("My script") // Declares that the script is an indicator named "My script" with default properties. // Plot the `close` series across the chart. plot(close)

The parameters of a declaration statement define various script-wide properties and default behaviors. Only the `title` parameter, which sets the script's *main title*, requires an argument. Supplying arguments to any other parameters is *optional*. Note that all parameters in each declaration statement require arguments qualified as "const". They *cannot* accept values with the "input", "simple", or "series" type qualifier.

The `indicator()`, `strategy()`, and `library()` sections below explain the parameters available for each declaration statement and how they affect a script, as well as various unique characteristics of each script type.

> TipThe Pine Editor displays the Reference Manual documentation for a declaration function and its parameters in a pop-up window as the user types the statement. To view the complete Reference Manual entry from inside the editor, press the CTRL or CMD key and click the indicator(), strategy(), or library() identifier.

`indicator()`

The indicator() function declares that the script is an *indicator*. Indicators perform calculations across a dataset to generate visuals, alerts, or Pine Logs. They are the most common type of scripts in Pine.

The built-in Relative Strength Index (RSI) script is an example of an indicator. It calculates the RSI of a specified source series and plots the result in a separate pane. It can also plot a smoothed RSI, display divergence signals, and generate divergence alerts.

Indicators have several distinct characteristics, including the following:

- Indicators are the *only* scripts that can use alert triggers from calls to both the alert() and alertcondition() functions.
- Unlike strategies, indicators *cannot* use any `strategy.*` built-ins or simulate trades.
- Unlike libraries, indicators cannot *export* code for use in other scripts. However, other scripts that include source inputs can retrieve values from an indicator's *plots* created by plot() calls.

- The Pine Screener is compatible with indicators only. The screener can display plotted values from an indicator's plot() function calls, and *filter* the results using data from other plot() or alertcondition() calls.
- Indicators always execute *once per bar* on historical bars, and *once per data feed update (tick)* on realtime bars.
- Indicators must include *at least one* call to a function that creates one of the following outputs: plot visuals, drawing visuals, alert triggers, or Pine Logs.

The signature for the indicator() function is as follows:

```
indicator(title, shorttitle, overlay, format, precision, scale, max_bars_back,
```

The following sections explain the parameters of the indicator() declaration statement and how they work.

> NoteAll the parameters described below, excluding `timeframe` and `timeframe_gaps`, also apply to the strategy() function.

`title` and `shorttitle`

The required `title` parameter defines the script's *main title*. The script displays the specified "string" title in all possible chart locations by default. Additionally, the "Publish script" window automatically suggests using that title for a script publication.

> NoteIf the `title` argument is an empty string, the script uses "Study" as its main title on the chart, and the "Publish script" window does not suggest a publication title.

The optional `shorttitle` parameter defines a *short display title* for the script. If the declaration statement includes a `shorttitle` argument that is not an empty string, the string's text appears instead of the main title in multiple chart locations, including:

- The script's status line on the chart.
- The chart's object tree and data window.
- The script's "Settings" window.
- The "Condition" section of the "Create alert" dialog box.
- The listed alerts and logs for the script in the "Alerts" pane.
- The Pine Logs pane.

> NoteThese parameters are **not related** to the name that the user assigns when saving or renaming a script project using the menu options in the Pine Editor. The text assigned from those options defines the name that the user searches in their *personal library* to open the script in the editor, add the script to the chart, or run the script in the Pine Screener.

The example script below plots an Exponential Moving Average (EMA) for a selected source series and length. The declaration statement sets the script's main title to `"Exponential Moving Average indicator"`. However, because the declaration statement also includes the argument `shorttitle = "EMA"`, the script's status line and the data window display "EMA" instead of the main title. Hovering over the short title in the status line reveals a tooltip containing the script's main title:

//@version=6 indicator("Exponential Moving Average indicator", shorttitle = "EMA") //@variable The source series for which to calculate the EMA. float sourceInput = input.source(ohlc4, "Source") //@variable The length value for the EMA's smoothing factor. int lengthInput = input.int(20, "Length", minval = 1) // Calculate the EMA for the specified series and length, and plot the result as a color-coded line. float ema = ta.ema(sourceInput, lengthInput) plot(ema, "EMA", ema > ema[1] ? color.green : color.red, 3)

> TipThe `shorttitle` parameter is intended for creating an *abbreviated* display title for a script. As such, the compiler raises a warning if the specified string contains more characters than the recommended limit. If you encounter the warning, we recommend reducing the string's length. If you want the script to display a longer title, specify that title in the `title` argument and *remove* the `shorttitle` argument.

`overlay`, `scale`, and `behind_chart`

The `overlay`, `scale`, and `behind_chart` parameters of the declaration statement configure where the script displays its chart outputs. They control the *global default* display location and scaling behavior of the script's visuals, separate to the individual properties of plot visuals or drawing visuals.

> NoteThese parameters affect a script's visuals only **once**, when the user first adds the script to their chart. If an instance of the script is already on the chart, changing the `overlay`, `scale`, or `behind_chart` arguments in the declaration statement **does not** affect that instance's display location or scale. Programmers must *re-add* the script to the chart to view the results of such changes.

The `overlay` parameter specifies which *default* chart pane the script uses to display its visuals when the user adds the script to their chart. If the argument is `true`, the script's visuals appear in the *main chart pane* by default, or in another script's pane if the user adds it to the chart via the "Add indicator/strategy on" option in the other script's "More" menu. If the `overlay` argument is `false` (default), the script's visuals occupy a *separate chart pane* by default.

The `scale` parameter defines the location of the script's *price scale* and the scaling behavior of the script's plots and drawings. The possible arguments are scale.left, scale.right, and scale.none. The behaviors associated with this parameter are as follows:

- If the declaration statement includes *any* `scale` argument, the script scales its visuals *independently* to fit the vertical range of the pane that it occupies.
- If the argument is scale.left or scale.right, and the script overlays on an existing pane, the script adds a *separate* scale for its visuals on the specified side of that pane.
- If the script occupies a separate pane, an argument of scale.left or scale.right moves that pane's scale to the specified side without creating a new scale.
- If the argument is scale.none, which is valid only if the `overlay` argument is `true`, the script displays plotted numbers directly on the scale of an existing pane without creating a new scale. If the user

moves the script to a separate pane, the script displays values on a new price scale in that pane.
- If the statement does not include a `scale` argument, the script uses the main price scale for the pane it occupies, and it does *not* scale its visuals separately if it overlays on an existing pane.

The following example indicator plots an RSI as translucent, color-coded columns. The script displays the columns on the main chart pane because its declaration statement includes `overlay = true`. Additionally, the script adds a separate scale to the left side of the pane and scales its plotted values independently because the statement uses scale.left as the `scale` argument:

//@version=6 indicator("`scale` demo", overlay = true, scale = scale.left, format = format.percent) //@variable The RSI of the `close` series with a length of 14. float rsi = ta.rsi(close, 14) // Plot the RSI as translucent columns. Use an orange color for values of 50 or greater, and blue for others. plot(rsi, "RSI", rsi <= 50 ? color.new(color.orange, 70) : color.new(color.blue, 75), 2, plot.style_columns)

Note that:

- The script formats the plotted numbers and the values in the left-side scale as *percentages* because the indicator() statement includes the argument `format = format.percent`. See the `format` and `precision` section below to learn more.

The `behind_chart` parameter determines the *visual order* of the script's plots and drawings relative to the main chart series. Specifying an argument for this parameter affects the script's visuals only if the `overlay` argument is `true`, because the behavior does not apply to non-overlay scripts. If the `behind_chart` value is `true` (default), the script's visuals appear *behind* the main series. If the value is `false`, they appear *in front* of the main series and can cover the chart's bars.

> NoteThe functions that create plots, background colors, and drawings include a `force_overlay` parameter. If a call to these functions specifies `true` as the `force_overlay` argument, its resulting visuals always appear over the *main* chart pane and use that pane's scale, regardless of the `overlay`, `behind_chart`, and `scale` arguments in the script's declaration statement. See the `overlay` section of the Visuals overview page to learn more about this feature.

`format` and `precision`

The `format` and `precision` parameters of the declaration statement control the default appearance of *plotted numbers* in the script's status line, the price scales, and the data window.

The `precision` parameter determines the default number of *fractional digits* that the script shows for plotted values and the numbers in the price scale. It accepts a value from 0 to 16. This parameter affects the appearance of all plotted numbers except for those formatted using format.volume, because the decimal precision rules of the built-in volume format supersede other precision settings. If the declaration statement does not include a `precision` argument, the script inherits its default precision settings from the main chart series, or from another script if it accesses one of that script's plots using a source input.

The `format` parameter determines whether the script displays plotted numbers and the numbers in the price scale using a price, percentage, or volume format, or if it inherits formatting settings from the chart or another script. The possible arguments are format.price, format.percent, format.volume and format.inherit. The default is format.inherit.

Below, we list how a script formats plotted values when using each of these arguments:

`format.price`

The script formats plotted values as whole numbers with two fractional digits by default. For example, a script that uses this argument and default precision settings formats a plot value of 122 as 122.00, and a value of 122.355 as 122.36. If a rounded value is greater than or equal to 1000, the script uses a comma as the thousands separator. For instance, it formats a value of 14489245 as 14,489,245.00. If a number is extremely large, the script formats it as a rounded value in E notation (e.g., `1e+21`).

`format.percent`

The script applies similar formatting rules to those defined by format.price, and it appends a percent sign (`%`) to express plotted values as percentages. By default, the format rounds plotted numbers to two fractional digits. For example, it formats the value 39.787 as 39.79%. This format does *not* recalculate values to express them as percentages. To represent a *ratio* as a percentage when using this format, multiply the value by 100 before plotting it.

`format.volume`

The script formats plotted numbers as *abbreviated* values that follow special precision rules. If a rounded value is greater than or equal to 1000, the script includes a letter representing a multiplied quantity: “K” for thousand, “M” for million, “B” for billion, or “T” for trillion. For example, it formats a plot value of 2474 as 2.47K, and a value of 14489245 as 14.49M. If a value is extremely large, the script displays a number with commas or E notation followed by “T”. For values less than 1000, the script displays those values rounded to the nearest whole number by default. Note that these formatting rules can apply to any plotted numbers; they are not limited to only volume values.

`format.inherit`

The script inherits the same formatting settings as those defined for the main chart series, or the global formatting settings for another script if it accesses one of the script’s plots using a source input. For example, the script uses price formatting when applied to a stock chart series, and percentage formatting when applied to a bond chart series.

> NoteIf the declaration statement uses format.inherit as the `format` argument, changing the script’s *precision* settings via the `precision` parameter or the “Precision” field in the script’s “Settings/Style” tab causes it to *ignore* the inherited format and instead use format.price settings with the specified precision, even if the inherited format uses format.volume rules.

The example indicator below plots volume values as color-coded columns, and it plots the average value over a specified number of bars as a line. The indicator() declaration statement includes format.volume as the `format` argument to apply the volume formatting rules described above to the script’s plots and scale. On our daily “NASDAQ:NFLX” chart, the current plotted values are in *millions*, so the script displays the numbers in an abbreviated format with “M” as the suffix:

//@version=6 indicator("`format.volume` demo", format = format.volume) //@variable The number of bars over which to calculate the average volume. int lengthInput = input.int(14, "Average length", minval = 1) //@variable A red color if `close < open`, and green otherwise. The color is 50% transparent if `volume <=

volume[1]`. color volColor = switch volume > volume[1] => close < open ? color.red : color.green => close < open ? color.new(color.red, 50) : color.new(color.green, 50) // Plot volume as columns, and the average volume as a line. // Both of these plots automatically format numbers using `format.volume` rules. plot(volume, "Volume", volColor, style = plot.style_columns) plot(ta.sma(volume, lengthInput), "Avg volume", color.blue, linewidth = 2)

Note that the `plot*()` functions also include `format` and `precision` parameters, which enable scripts to define specific formatting behaviors for each separate plot. By default, a plot automatically inherits the default format and precision settings defined by the declaration statement, as demonstrated by the previous example. However, if a `plot*()` call includes `format` or `precision` arguments, those arguments *take precedence* over the script's default settings.

For example, in the script version below, we added the argument `format = format.price` to the plot() call for the average volume display. With this change, the script formats the average volume values using the rules defined by format.price, while the volume plot and the price scale both continue to use the default format.volume rules specified by the declaration statement:

//@version=6 indicator("`format.volume` demo", format = format.volume) //@variable The number of bars over which to calculate the average volume. int lengthInput = input.int(14, "Average length", minval = 1) //@variable A red color if `close < open`, and green otherwise. The color is 50% transparent if `volume <= volume[1]`. color volColor = switch volume > volume[1] => close < open ? color.red : color.green => close < open ? color.new(color.red, 50) : color.new(color.green, 50) // Because this call does not include a `format` argument, it inherits `format.volume` rules. plot(volume, "Volume", volColor, style = plot.style_columns) // By contrast, this call uses `format.price` rules, because the `format` argument here // *overrides* the script's default plot format. plot(ta.sma(volume, lengthInput), "Avg volume", color.blue, linewidth = 2, format = format.price)

`max_bars_back`

The `max_bars_back` parameter of the declaration statement, if it has a specified argument, sets the initial maximum *history-referencing length* for each series in a script. It accepts an "int" value from 0 to 5000, representing the number of past data points maintained in memory for *all* variables and expressions.

As a script executes, Pine's runtime system stores data for each variable and expression across bars in fixed-length historical buffers. The script can access *past bar data* from these buffers by using the `[]` history-referencing operator or the built-in functions that reference history internally. For example, the expression `close[10]` retrieves the last saved value of the close variable from *10 bars back*.

By default, the system automatically sizes each historical buffer by analyzing the historical references that the script executes as it loads across historical bars. For resource efficiency, each buffer typically contains only enough past data to accommodate the script's historical references, but *not more*. For instance, if a script requests the value of a variable from up to 500 bars back as it loads across a chart's history, the buffer for that variable typically includes data for only the latest 500 past bars.

In most cases, this automatic sizing process accommodates a script's historical references without issues. Therefore, manually setting the sizes of historical buffers is often **unnecessary**. However, in some cases, the system might fail to determine appropriate buffer sizes on its own, resulting in a runtime error. One possible way to resolve that error is to set the default size of the script's historical buffers in advance by including a `max_bars_back` argument in the declaration statement.

Before using this parameter, ensure that *all* or *most* of the series in the script actually require historical buffers with the *same* specific size. Manually setting buffers in a script to use a specific size when unnecessary can **negatively** impact the script's performance. If only *specific* series require manually sized buffers, either of the following approaches is far more efficient:

- Use the max_bars_back() *function* to set the sizes of only the problematic historical buffers.
- Structure the script's history-referencing operations to request the *maximum* required amount of history on the *first bar*.

See the historical buffer limit error page to learn more. For *advanced* details about the workings of historical buffers, refer to the Historical buffers section of the Execution model page.

`timeframe` and `timeframe_gaps`

The `timeframe` parameter of the indicator() declaration statement sets the script's *main timeframe*. It enables the script to perform calculations on the data for a different timeframe than that of the chart without requiring `request.*()` function calls. The parameter accepts a valid timeframe string, such as `"1D"` for the daily timeframe or `"30"` for the 30-minute timeframe. If an argument is not specified, or if the value is an empty string (`""`), the script executes on the data for the current chart's timeframe.

> TipScripts can retrieve a string representing the main timeframe specified in the declaration statement, even while executing other data requests with `request.*()` calls, by using the timeframe.main_period variable.

The `timeframe_gaps` parameter determines how the script handles *time gaps* when plotting data from a *higher timeframe*. It allows an argument only if the declaration statement also includes a `timeframe` argument. The parameter works similarly to the `gaps` parameter of request.security() and other `request.*()` functions. If the value is `true` (default), the script plots values only on the chart bars where new, *confirmed* data is available from the specified timeframe, and displays na results on other bars. If `false`, the script plots the *last retrieved values* from the higher timeframe on the chart bars where new data is not available.

> NoteThe indicator() declaration statement can include arguments for these parameters only if the script *does not* use drawing objects or alert() function calls, because scripts cannot evaluate outputs from such code on other datasets.

If the declaration statement specifies a `timeframe` argument, the script automatically adds a "Calculation" group with a "Timeframe" input to the "Settings/Inputs" tab. If the statement includes a `timeframe_gaps` argument, the script also adds a "Wait for timeframe closes" input below the "Timeframe" input. These inputs enable users to customize the script's main timeframe and its gap-handling behavior without modifying the source code. To learn more about them, see the Leveraging multi-timeframe analysis article in our Help Center.

The following example demonstrates the behavior of both parameters. The indicator below calculates and plots the 14-bar average of close values on a specified timeframe. The indicator() declaration statement includes `"1D"` as the `timeframe` argument, so it performs calculations using daily data for the current chart's symbol by default, regardless of the chart's timeframe. On an intraday chart, the script plots an "x-cross"

shape only on the *last* chart bar for each trading day by default, and na on other bars, because the declaration statement also includes the argument `timeframe_gaps = true`:

//@version=6 indicator("`timeframe` and `timeframe_gaps` demo", overlay = true, behind_chart = false, timeframe = "1D", timeframe_gaps = true // <- These arguments automatically add inputs to the "Settings/Inputs" tab.) //@variable The 14-bar average `close` value on the script's main timeframe ("1D" by default). float avgClose = ta.sma(close, 14) // Plot the `avgClose` series as "x-cross" shapes. // By default, a new shape appears only on the last chart bar for each "1D" period. On other bars, the plot shows `na`. plotshape(avgClose, "Avg close", shape.xcross, location.absolute, size = size.small)

Note that:

- If the specified timeframe is *lower* than or *equal* to the chart's timeframe, the script plots an "x-cross" shape on *every* chart bar.
- An alternative way to achieve this script's default result without using these parameters is to plot the value returned by the call `request.security(syminfo.tickerid, "1D", ta.sma(close, 14), gaps = barmerge.gaps_on)`. Refer to the Other timeframes and data page to learn more about `request.*()` functions.

`explicit_plot_zorder`

The `explicit_plot_zorder` parameter of the declaration statement determines the *visual order* in which the script's plots, horizontal levels, and fills *stack* on the chart.

If the value is `true`, the script visually stacks plots, levels, and fills based on the order of the `plot*()`, hline(), and fill() function calls in the source code, where each written call's output appears *on top* of the outputs from the calls that *precede* it. For example, if the code lists a fill() call after a plot() call, the resulting fill appears on top of the plot. Likewise, if the code lists a plot() call after an hline() call, the plot appears on top of the horizontal line.

If the value is `false` (default), the script visually stacks its plots, levels, and fills based on the order of those visuals in the z-index, regardless of the order in which the function calls for each type of output occur in the code. Horizontal levels always appear on top of plots, and plots always appear on top of fills. However, visual outputs of the *same* type or group still stack on top of each other based on the order of their function calls. For example, if a script includes two calls to the plot() function, the *second* plot appears on top of the first.

> NoteThis parameter **does not** affect visuals created by the bgcolor() function or drawing objects. Background colors and drawings *always* stack on the chart in the order of the *z-index*, regardless of the `explicit_plot_zorder` argument in the script's declaration statement.

`max_lines_count`, `max_labels_count`, `max_boxes_count`, **and** `max_polylines_count`

The `max_lines_count`, `max_labels_count`, `max_boxes_count`, and `max_polylines_count` parameters of the declaration statement limit the number of line, label, box, and polyline drawing objects that the script can maintain in memory. As the script creates new objects of a given drawing type, the runtime system *deletes* the *oldest* drawings of that type as necessary if the number of active objects exceeds the script's limit.

The `max_lines_count`, `max_labels_count`, and `max_boxes_count` parameters accept an “int” value from 1 to 500, and the `max_polylines_count` parameter accepts an “int” value from 1 to 100. The default for each parameter is 50.

> NoteThe limits defined by these parameters are approximate. The maximum number of active drawings can vary slightly across bars. Programmers can precisely limit the number of active drawings by using the `*.delete()` functions (e.g., line.delete()) on the elements of the built-in `*.all` array for each drawing type (e.g., line.all).

See the Line, box, polyline, and label limits section of the Limitations page and the Total number of objects section of the Lines and boxes page to learn more about drawing limits.

`calc_bars_count`

The `calc_bars_count` parameter of the declaration statement sets the default *maximum* number of most recent *historical bars* that the script can access for its calculations. It accepts an “int” value that is greater than or equal to 0.

If the value is 0 (default), the script executes on *all* the available bars in the dataset, starting from the first available bar. If the value is greater than 0, the script instead starts executions on the bar that is N bars before the *latest* available bar at loading time, or on the dataset’s first bar if the value exceeds the number of available bars. Additionally, a positive `calc_bars_count` argument adds a “Calculation” group with a *“Calculated bars”* input to the script’s “Settings/Inputs” tab, where users can adjust the number of historical bars available to the script without editing the source code.

The following example script plots the close series across a limited number of historical bars and all realtime bars. The indicator() declaration statement includes the argument `calc_bars_count = 40`, which forces the script to treat the last 40 historical bars as the *only* ones available in the dataset by default:

//@version=6 // The `calc_bars_count` argument in this declaration statement specifies that the script can use // only the last 40 historical bars for its calculations by default. It also adds a "Calculated bars" // input to the script's "Settings/Inputs" tab. indicator("`calc_bars_count` demo", calc_bars_count = 40) // Plot the `close` series on the specified number of recent historical bars and all realtime bars. plot(close, "Close", linewidth = 2)

> NoteLimiting the bars on which a script can execute with the `calc_bars_count` parameter also limits the data points available for history-referencing operations and bar indexing. A script treats the first bar on which it *executes* as the *earliest* bar in the dataset, with a bar_index value of 0, even if earlier bars are otherwise available from the data feed. Therefore, [] operations *cannot* retrieve data from bars before the first bar determined by the `calc_bars_count` argument.

`dynamic_requests`

The `dynamic_requests` parameter of the declaration statement specifies whether the script can use `request.*()` function calls to execute dynamic requests. If the value is `true` (default), the script can:

- Include calls to `request.*()` functions inside the local scopes of conditional structures and loops, and in the operands of conditional expressions.
- Use "series" arguments that vary across bars to specify the ticker identifier, timeframe, and other settings of a `request.*()` call.
- Execute nested requests, where one `request.*()` call evaluates another inside its context.

If the value is `false`, the script is more limited in how it can use `request.*()` functions:

- All `request.*()` calls must execute in the script's global scope, and outside the conditional operands of ternary or and/or operations.
- All `request.*()` parameters except for `expression` require arguments with the "simple" type qualifier or a weaker qualifier, meaning their values *cannot change* across bars.
- A `request.*()` call whose `expression` argument depends on another `request.*()` call *cannot* evaluate the other call within its context.

The following example script calculates a weighted moving average (WMA) of hl2 values over a specified number of chart bars. It also uses a request.security() call within an if structure to optionally calculate the latest confirmed WMA on a specified higher timeframe. The script can use the request inside the if structure because the indicator() declaration statement's `dynamic_requests` argument is `true`:

//@version=6 indicator("Conditional dynamic requests demo", overlay = true, behind_chart = false, dynamic_requests = true) //@variable The number of bars to use in the WMA calculation. int lengthInput = input.int(5, "WMA length", minval = 1) //@variable Specifies whether to retrieve a higher-timeframe WMA. bool htfRequestInput = input.bool(true, "Show higher-timeframe WMA") //@variable A higher-timeframe string for the data request. string timeframeInput = input.timeframe("1W", "Higher timeframe", active = htfRequestInput) //@variable The weighted moving average of `hl2` values over the specified length. float chartWMA = ta.wma(hl2, lengthInput) //@variable The WMA calculated on the higher timeframe if the `htfRequestInput` value is `true`, and `na` otherwise. float requestedWMA = na if htfRequestInput // Raise an error if the specified timeframe is *not* higher than the chart's timeframe. if timeframe.in_seconds(timeframeInput) <= timeframe.in_seconds(timeframe.period) runtime.error("The requested timeframe must be higher than the chart's timeframe.") // Execute the `request.security()` call for the higher-timeframe request. // The call works in this `if` structure because the declaration statement enables dynamic requests. // If we change the `dynamic_requests` argument to `false`, this call causes a *compilation error*. requestedWMA := request.security(syminfo.tickerid, timeframeInput, ta.wma(hl2, lengthInput)[1], lookahead = barmerge.lookahead_on) // Plot the chart WMA and the optional HTF WMA. plot(chartWMA, "Chart WMA", color.teal, 4) plot(requestedWMA, "HTF WMA", color.purple, 4)

Note that:

- The script behaves the same if we remove the `dynamic_requests` argument from the declaration statement, because the default argument is `true`.
- If we change the `dynamic_requests` argument to `false`, the request.security() call causes a *compilation error* because the script cannot use it inside the if statement. To resolve the error without enabling dynamic requests, programmers must move the call to the *global scope*.
- Users can change the "Higher timeframe" input in the "Settings/Inputs" tab only if they select the "Show higher-timeframe WMA" checkbox, because the input.timeframe() call includes the argument `active = htfRequestInput` to control when the input is *active*. See the Input function parameters section of the Inputs page to learn more about `active` and other input parameters.

To learn more about the `request.*()` functions and the differences between dynamic and non-dynamic requests, refer to the Other timeframes and data page.

`strategy()`

The strategy() function declares that the script is a *strategy*. Strategies can simulate orders and trades across a dataset, enabling users to backtest and forward test their trading systems. They have many similar capabilities to indicators, while also providing the ability to analyze hypothetical trading performance in a dedicated tab.

The built-in RSI Strategy script is an example of a simple strategy. The script simulates entering and exiting positions based on the RSI crossing the defined overbought and oversold levels. It displays trade markers directly on the chart and shows a detailed strategy report in a separate panel below the chart.

Scripts declared as strategies have several unique characteristics, including the following:

- Strategies are the only scripts that can send orders to the broker emulator and display simulated performance results using the strategy report feature.
- The “Settings” window for strategy scripts features a unique “Properties” tab, where users can customize the properties of the strategy simulation. Programmers can specify *default* properties for this tab via the unique parameters in the strategy() statement.
- Unlike indicators, strategies cannot run on data for other timeframes. They always use the same timeframe as the chart.
- Strategies *cannot* create alert triggers using the alertcondition() function, but they can create them by using calls to the alert() function. Additionally, unlike indicators, they can generate special alerts from order fill events.
- Unlike the plots created by indicators or libraries, strategy plots are *not* accessible to source inputs in other scripts.
- Strategies execute differently from indicators or libraries. By default, they execute strictly *once per closed bar* and do *not* execute on open bars. However, users can customize a strategy’s calculation behavior to enable additional executions on open bars or after the broker emulator fills an order.
- Strategies must include at least one call to an order placement command, or to a function that creates plot visuals, drawing visuals, alert triggers, or Pine Logs.

The strategy() function has the following signature:

```
strategy(title, shorttitle, overlay, format, precision, scale, pyramiding, calc
```

Because strategies have many of the same features as indicators, the strategy() function includes most of the indicator() function’s parameters. The only exceptions are the `timeframe` and `timeframe_gaps` parameters, because strategies cannot execute on other timeframes.

> Tip
> Programmers can convert compatible indicator scripts into strategies by replacing the indicator() declaration statement with the strategy() declaration statement, using the same arguments, then adding

> calls to commands such as strategy.entry() and strategy.exit() to create orders. See an example in the How can I turn my indicator into a strategy section of the Strategies FAQ page.

The unique parameters in the strategy() declaration statement define the *default properties* of the strategy simulation, including the initial simulated capital, default order sizes, hypothetical trading costs, and calculation behaviors. The sections below explain these unique parameters. To learn about the other parameters that are common to both indicators and strategies, see the `indicator()` section above.

For detailed information about how to use the unique strategy() function parameters and the built-ins in the `strategy` namespace, refer to the Strategies page.

`pyramiding`

The `pyramiding` parameter of the strategy() declaration statement accepts an “int” value specifying the default maximum number of *open trades*, from the orders created by strategy.entry() calls, that a strategy allows for a single position. The default argument is 1, meaning that the strategy can open only *one* long or short trade at a time using orders from strategy.entry() calls and *cannot* execute another entry order in the *same direction* until after the existing trade closes. Users can adjust the script’s pyramiding limit without editing the code by using the “Pyramiding” input in the script’s “Settings/Properties” tab.

> NotePyramiding affects only entry orders from calls to the strategy.entry() command; it does **not** affect the behavior of orders from strategy.order() calls.

The following example strategy uses two calls to the strategy.entry() command to create market orders for entering long and short trades. The call for long orders executes once every five bars, excluding multiples of 30, and the one for short orders executes once every 30 bars. The strategy() declaration statement includes the argument `pyramiding = 3`, meaning that the strategy can enter up to *three trades* for the same position using strategy.entry() calls by default.

As shown below, although the strategy’s long condition (highlighted by the purple background) occurs *five* times before the short condition (highlighted by the orange background), the strategy executes only **three** entry orders for each long position instead of five. Once the number of open trades reaches three, it does not execute new long entry orders until after the short order *closes* the existing long position:

//@version=6 strategy("Strategy `pyramiding` demo", overlay = true, pyramiding = 3, default_qty_value = 10) // The `pyramiding = 3` argument above specifies that, by default, the strategy cannot use `strategy.entry()` calls to // maintain an open position consisting of more than three trades. //@variable The value for the short condition: `true` on every 30th bar, and `false` otherwise. bool sellCondition = bar_index % 30 == 0 //@variable The value for the long condition: `true` on every 5th bar, excluding multiples of 30, and `false` otherwise. bool buyCondition = bar_index % 5 == 0 and not sellCondition if buyCondition // Place a market order named "buy" to close any short position and enter or add to a long position. strategy.entry("buy", strategy.long) if sellCondition // Place a market order named "sell" to close any long position and enter or add to a short position. strategy.entry("sell", strategy.short) // Highlight the background when the `buyCondition` or `sellCondition` value is `true`. bgcolor(sellCondition ? color.new(color.orange, 80) : buyCondition ? color.new(color.purple, 85) : na, title = "Order conditions highlight")

Note that:

- By default, the orders from the strategy.entry() command automatically close an existing position in the opposite direction and enter a new trade with the specified quantity. See the Reversing positions section of the Strategies page for more information about this behavior.
- The `default_qty_value` argument in the declaration statement specifies the initial default size of the strategy's orders. See the `default_qty_type` and `default_qty_value` section to learn more.
- The strategy enters a new trade after every occurrence of the long condition only if the `pyramiding` value is at least 5.

`calc_on_every_tick`, `calc_on_order_fills`, `calc_on_every_history_tick`, and `process_orders_on_close`

The `calc_on_every_tick`, `calc_on_order_fills`, `calc_on_every_history_tick`, and `process_orders_on_close` parameters of the strategy() declaration statement specify the strategy's default calculation behaviors. If the argument for each of these parameters is `false` (default), the strategy executes strictly *once per bar*, on each bar's *closing tick*, and the broker emulator fills each order from the strategy on the *open* of the next available bar. Specifying a value of `true` for any of these parameters changes the strategy's default execution and order-fill behaviors. Script users can override the specified defaults by adjusting the Script execution settings and the "Order execution delay" input in the script's "Settings/Properties" tab.

The `calc_on_every_tick` parameter specifies whether the strategy performs a *new execution* on *each new tick* of a realtime bar by default. If the value is `true`, the strategy executes once after *every update* from the realtime data feed, similar to how an indicator executes, instead of waiting for each realtime bar to close. This parameter does *not* affect the strategy's executions on *historical bars*, because realtime tick information is not available on those bars.

The `calc_on_order_fills` parameter specifies whether the strategy can immediately recalculate and place additional orders on any bar where an *order fills* by default. If the value is `true`, the strategy *re-executes* on the next available tick following any tick where the broker emulator fills an order, even if that tick occurs during an open bar. This behavior enables the script to execute *more than once* on any bar where an order fill occurs – up to four times per historical bar by default (at the open, high, low, and close), and up to once for each new tick on a realtime bar. As the script executes on historical ticks, the variables that store price and volume information for the current bar consistently store the bar's *final values*. This behavior can cause *lookahead bias* when executing on ticks before a bar's close. See the `calc_on_order_fills` section of the Strategies page to learn more.

The `calc_on_every_history_tick` parameter specifies whether the strategy executes on *every* available tick on *historical bars* by default. An argument for this parameter must include the parameter's name (e.g., `calc_on_every_history_tick = true`). If the value is `true`, the strategy executes on all historical ticks, even the ticks where the broker emulator does not fill a new order. As the script executes across the historical data, the values of the built-in variables that hold price and volume information for the current bar update on *each tick* to represent the bar's *developing* values rather than its *final* values. With this setting, the script executes four times per historical bar, or up to four times per *lower-timeframe* bar if the script uses high historical bar detail. This strategy behavior is available only to accounts with Premium and Ultimate plans, and it is usable only on *standard* chart types.

> NoticeModifying a strategy's execution settings can cause it to behave *differently* on realtime bars and historical bars, and therefore repaint after it reloads. Additionally, with recalculation after order fills enabled, the broker emulator can fill some historical orders at prices that are not typically possible in real-

> world trading. Therefore, when modifying these settings, exercise caution and examine the script's behaviors carefully to avoid misleading results.

The `process_orders_on_close` parameter specifies whether the broker emulator can fill an order on the *same closing tick* where the strategy creates the order by default. If the value is `false` (default), the earliest point at which the broker emulator can fill an order that occurs on a bar's close is at the *open* of the *following bar*, because that point is the next possible tick. If the value is `true`, the emulator fills the order *immediately* on the bar's close instead of waiting for the next bar's opening tick.

For example, the following strategy simulates opening a position after one exponential moving average (EMA) crosses over another. On each bar where the EMAs cross, the script highlights the chart's background, then creates a long or short market order on that bar's closing tick. With the default behavior defined by `process_orders_on_close = false`, the broker emulator does not fill each order on the same bar where the strategy creates it. Instead, it fills the order at the open of the following bar, because that point is the next available tick:

//@version=6 strategy("`process_orders_on_close` demo", overlay = true, process_orders_on_close = false) // Calculate fast and slow moving averages. float fastMA = ta.ema(close, 13) float slowMA = ta.ema(close, 26) // Set long and short order conditions based on crosses of the moving averages. //@variable Is `true` if `fastMA` crosses above `slowMA`. bool longCondition = ta.crossover(fastMA, slowMA) //@variable Is `true` if `fastMA` crosses under `slowMA`. bool shortCondition = ta.crossunder(fastMA, slowMA) if longCondition strategy.entry("buy", strategy.long) if shortCondition strategy.entry("sell", strategy.short) // Plot the moving averages, and highlight the bars where order conditions occur. plot(fastMA, "Fast MA", color.blue, linewidth = 2) plot(slowMA, "Slow MA", color.orange, linewidth = 2) // Highlights background blue if long entry condition occurs, or orange if short entry condition occurs. bgcolor(longCondition ? color.new(color.blue, 85) : shortCondition ? color.new(color.orange, 80) : na)

If we include `process_orders_on_close = true` in the strategy() declaration statement, the broker emulator is no longer limited to filling our strategy's orders on the next available tick by default. Instead, it fills the orders immediately on each bar's close:

//@version=6 strategy("`process_orders_on_close` demo", overlay = true, process_orders_on_close = true) // Calculate fast and slow moving averages. float fastMA = ta.ema(close, 13) float slowMA = ta.ema(close, 26) // Set long and short order conditions based on crosses of the moving averages. //@variable Is `true` if `fastMA` crosses above `slowMA`. bool longCondition = ta.crossover(fastMA, slowMA) //@variable Is `true` if `fastMA` crosses under `slowMA`. bool shortCondition = ta.crossunder(fastMA, slowMA) if longCondition strategy.entry("buy", strategy.long) if shortCondition strategy.entry("sell", strategy.short) // Plot the moving averages, and highlight the bars where order conditions occur. plot(fastMA, "Fast MA", color.blue, linewidth = 2) plot(slowMA, "Slow MA", color.orange, linewidth = 2) // Highlights background blue if long entry condition occurs, or orange if short entry condition occurs. bgcolor(longCondition ? color.new(color.blue, 85) : shortCondition ? color.new(color.orange, 80) : na)

> NoticeForcing orders to fill on a bar's close can be helpful in some scenarios, such as when backtesting manual strategies where traders enter or exit positions immediately before the market closes. However, it's crucial to understand that it can also cause *misleading* results in some cases, because creating and filling orders on the same tick is *not* typically possible in real-world trading.

See the Altering calculation behavior section of the Strategies page to learn more about the `calc_on_every_tick`, `calc_on_order_fills`, `calc_on_every_history_tick`, and `process_orders_on_close` parameters. For detailed information about how scripts execute on historical and realtime bars, and how these parameters affect executions, refer to the Execution model page.

`slippage` and `backtest_fill_limits_assumption`

The `slippage` parameter of the strategy() declaration statement specifies the default number of ticks that the strategy applies to the fill prices of *all* market orders and stop orders to simulate slippage. If the argument is a positive "int" value, the strategy adds the specified number of ticks to the fill prices of long orders and subtracts it from the fill prices of short orders. This behavior helps simulate the disparity between expected and actual fill prices that might occur in real-world trading. If the `slippage` argument is 0 (default), the strategy fills orders at their expected prices without simulating any slippage. Users can change the specified slippage amount via the "Slippage" input in the strategy's "Settings/Properties" tab.

The `backtest_fill_limits_assumption` parameter specifies the default number of ticks by which the market price must *exceed* the prices of limit orders before the broker emulator can fill the orders. If the argument is a positive "int" value, the broker emulator fills a limit order at the defined price only if the market price moves *past* it by the specified number of ticks in the favorable direction. This behavior helps simulate the possibility of unfilled limit orders, as filling limit orders in the real world requires sufficient liquidity and price action around the limit level. If the argument is 0 (default), the emulator fills orders as soon as the market price reaches the limit price or a more favorable value.

> NoticeLimit verification can cause order fills to occur at *different times*, depending on how long it takes for the market price to exceed limit levels by the specified amount. This tradeoff is necessary to enable filling limit orders at their verified prices without introducing lookahead bias in the simulation. However, in some cases, it can also cause some limit orders to fill at times that are not possible in the real world. We therefore recommend users understand this price-time tradeoff and analyze their strategies carefully when adding verification to limit orders.

`default_qty_type` and `default_qty_value`

The `default_qty_type` and `default_qty_value` parameters of the strategy() declaration statement specify the initial *default order size* for the strategy.entry() and strategy.order() commands. If a call to either command does not specify an order size, the resulting order uses the default order size defined by these parameters. Users can adjust these properties via the "Default order size" inputs in the script's "Settings/Properties" tab.

The `default_qty_type` parameter specifies the default *quantity type* for each order from strategy.entry() and strategy.order() calls. The possible arguments and their effects are as follows:

- strategy.fixed – The default order size is a fixed number of contracts, shares, lots, or units, depending on the instrument.
- strategy.cash – The default size is a fixed number of units of the account currency specified by the `currency` argument.
- strategy.percent_of_equity – The default size is a fixed percentage of the strategy's available equity.

The default argument is strategy.fixed.

The `default_qty_value` parameter accepts a “float” value that specifies the amount of the defined quantity type to use as the default order size. The default argument is 1, meaning that the strategy uses the default order size of one contract/share/lot/unit, one unit of the account currency, or one percent of the available equity, depending on the `default_qty_type` argument.

The specified default order size applies only to the orders from strategy.entry() and strategy.order() calls that do *not* include a `qty` argument. If a call to either command does include a `qty` argument, that call creates an order for the number of contracts/shares/lots/units specified by the argument instead of using the default quantity type and value. See the Position sizing section of the Strategies page for an example.

> NoteThe `default_qty_type` and `default_qty_value` parameters do not affect orders from the strategy.exit() or strategy.close() commands, because those commands create orders specifically for *closing* trades. The default order size for those commands is the size of the trades to which they apply.

The following example demonstrates how different default order sizes can affect a strategy’s entry orders. The script below uses a strategy.entry() call, without a `qty` argument, to place a long market order when the close and volume values are rising over a specified number of bars, then uses a strategy.close_all() call to close the open position when the close value is falling while the volume value is rising. It also plots the value of the strategy.position_size variable in a separate pane to visualize the size of each open position.

The strategy() statement in this example includes the arguments `default_qty_type = strategy.fixed` and `default_qty_value = 20`, which set the strategy’s default order size to 20 contracts/shares/lots/units. As shown by the trade markers and the plot on our “NYSE:UBER” chart below, each order from the strategy.entry() command consistently opens a 20-share trade:

//@version=6 // The `default_qty_*` arguments in this declaration statement specify that, by default, `strategy.entry()` and // `strategy.order()` calls create orders for 20 contracts/shares/lots/units if they do not specify a `qty` argument. strategy("`default_qty_type` and `default_qty_value` demo", default_qty_type = strategy.fixed, default_qty_value = 20) //@variable The number of bars for the `ta.rising()` and `ta.falling()` calculations. int lengthInput = input.int(2, "Length", minval = 1, display = display.none) // Determine if the `close` series is rising or falling over `lengthInput` bars, and if the `volume` series is rising. bool risingClose = ta.rising(close, lengthInput) bool fallingClose = ta.falling(close, lengthInput) bool risingVolume = ta.rising(volume, lengthInput) if risingVolume switch // Place a long market order if the `close` and `volume` values are both rising. risingClose => strategy.entry("Long entry", strategy.long) // Place an order to close the position if the `close` value is falling while the `volume` value is rising. fallingClose => strategy.close_all() // Plot the size of the current position. The plotted value is 0 if a position is not open. plot(strategy.position_size, "Position size", style = plot.style_area)

If we edit the declaration statement to use the argument `default_qty_type = strategy.percent_of_equity`, the strategy sets the default size of each entry order to allocate 20% of its current available equity instead of the amount required to purchase 20 shares. Now, the trade markers and plot show *varying sizes*, because the number of shares that corresponds to the default order size varies with both the strategy’s available equity and the current market price:

//@version=6 // The `default_qty_*` arguments in this declaration statement specify that, by default, `strategy.entry()` and // `strategy.order()` calls create orders for 20% of the available equity if they do not specify a `qty` argument. strategy("`default_qty_type` and `default_qty_value` demo", default_qty_type =

strategy.percent_of_equity, default_qty_value = 20) //@variable The number of bars for the `ta.rising()` and `ta.falling()` calculations. int lengthInput = input.int(2, "Length", minval = 1, display = display.none) // Determine if the `close` series is rising or falling over `lengthInput` bars, and if the `volume` series is rising. bool risingClose = ta.rising(close, lengthInput) bool fallingClose = ta.falling(close, lengthInput) bool risingVolume = ta.rising(volume, lengthInput) if risingVolume switch // Place a long market order if the `close` and `volume` values are both rising. risingClose => strategy.entry("Long entry", strategy.long) // Place an order to close the position if the `close` value is falling while the `volume` value is rising. fallingClose => strategy.close_all() // Plot the size of the current position. The plotted value is 0 if a position is not open. plot(strategy.position_size, "Position size", style = plot.style_area)

`initial_capital` and `currency`

The `initial_capital` parameter of the strategy() declaration statement specifies the default *initial account balance* for the strategy's simulation, as a quantity of the account currency. It accepts a positive "int" or "float" argument. The default is 1000000. Users can change the strategy's initial account balance by adjusting the "Initial capital" input in the script's "Settings/Properties" tab.

The `currency` parameter specifies the strategy's default *account currency*. It is the currency unit for the strategy's initial capital and for the internal calculations in the simulation that express values as currency amounts (equity, profit and loss, commission, etc.). The parameter accepts a `currency.*` constant (e.g., currency.USD) or a string representing a valid *currency code*, (e.g., `"USD"`). The default is currency.NONE, which specifies that the strategy uses the *same currency* as that of the quoted prices on the chart. Users can change the strategy's account currency via the input next to the "Initial capital" input in the "Settings/Properties" tab.

If the specified account currency differs from the chart's currency, the strategy *converts* monetary values in its calculations to express them in the account currency. However, the prices of the strategy's orders remain expressed in the chart's currency. To convert necessary monetary values to the account currency, the strategy typically uses the previous *daily* value of a corresponding *currency pair* as the conversion rate, or the value from a spread if no direct currency pair is available. See the Currency section of the Strategies page for more information.

`commission_type` and `commission_value`

The `commission_type` and `commission_value` parameters of the strategy() declaration statement specify the default commission fees that the broker emulator applies to the strategy's simulated transactions. Users can customize the strategy's commission settings via the "Commission" inputs in the "Settings/Properties" tab.

The `commission_type` parameter determines the default *commission type* for each executed order. The possible arguments and their effects are as follows:

- strategy.commission.cash_per_order – The default commission for each transaction is a fixed number of units in the strategy's account currency.
- strategy.commission.cash_per_contract – The commission is a fixed account currency amount for each traded contract/lot/share/unit.
- strategy.commission.percent – The commission is a fixed percentage of each transaction's value.

The default argument is strategy.commission.percent.

The `commission_value` parameter accepts a positive “int” or “float” value specifying the default fee amount for the commission type. For example, if the value is 1, the strategy simulates a fee of one unit of the account currency per transaction, one unit of the account currency per contract/share/lot/unit, or one percent of each transaction’s size by default, depending on the `commission_type` value. The default argument is 0, meaning that the strategy does not simulate commission unless the user specifies a nonzero value for the first “Commission” input in the “Properties” tab.

See the Commission section of the Strategies page for an example of how changing the `commission_*` arguments can affect a strategy’s simulated performance results.

`close_entries_rule`

The `close_entries_rule` parameter of the strategy() declaration statement determines the order in which the strategy simulation closes the trades in an open market position. It accepts one of two “string” arguments: `"FIFO"` or `"ANY"`. If the value is `"FIFO"`, the broker emulator follows *First In, First Out (FIFO)* rules when closing market positions. Under these rules, the *earliest* open trade is always the *first* to close, regardless of the entry IDs specified by the script’s strategy.exit() or strategy.close() calls. If the value is `"ANY"`, the broker emulator *ignores* FIFO rules and closes the trades specified by the exit commands, even if an earlier trade with a different entry ID is open. The default is `"FIFO"`.

> NoteUsers cannot customize a strategy’s exit order rules from the script’s “Settings/Properties” tab, unlike other strategy properties. The only way to change this property is by specifying a `close_entries_rule` argument in the strategy() statement.

Refer to the Closing a market position section of the Strategies page for an example of how changing the `close_entries_rule` argument can affect a strategy’s exit behavior.

`margin_long` and `margin_short`

The `margin_long` and `margin_short` parameters of the strategy() declaration statement specify the default *margin requirements* for the strategy’s long and short positions, respectively. The strategy automatically converts the values of these parameters to *leverage* values, then uses the results to set the default values for the “Long leverage” and “Short leverage” inputs in the “Settings/Properties” tab.

Margin is the percentage of a position’s value that the simulated account must retain in its balance as *collateral* for the broker emulator to cover the rest of the position. It is the *inverse* of *leverage*. For example, if the margin requirement for a long position is 50%, the strategy must maintain sufficient funds to cover *half* of the open position. This level of margin means that the strategy’s leverage is 2:1. In other words, the strategy can risk up to *twice* its available balance on a simulated trade.

The default `margin_long` and `margin_short` arguments are 100, meaning that the strategy must cover *100%* of each long and short position using its simulated account balance. When using these arguments, the “Long leverage” and “Short leverage” inputs show a default value of 1, because 100% margin is equivalent to 1:1 leverage.

If a strategy’s available funds drop below the required margin percentage, the broker emulator triggers a *margin call event*, which forcibly *liquidates* part or all of the simulated position to cover the loss. See the Margin and leverage section of the Strategies page to learn more about this behavior. For detailed

information about how margin and leverage work in Pine, refer to the How to simulate trading with leverage in Pine Script article in our Help Center.

> NoticeIf a strategy's long or short margin percentage is *zero*, it effectively has *infinite* leverage. It can open and maintain positions of *any size*, regardless of its simulated account balance. This behavior can cause extremely **misleading** results, because real-world brokers require traders to fund at least part of their positions. Therefore, we do not recommend using a value of 0 as the `margin_long` or `margin_short` argument.

`risk_free_rate`

The `risk_free_rate` parameter of the strategy() declaration statement specifies the annual percentage return of a hypothetical *risk-free* investment. The strategy uses the specified risk-free rate to calculate the Sharpe ratio and Sortino ratio metrics displayed in the strategy report. The default value is 2, meaning that these metrics assess the strategy's *risk-adjusted returns* relative to a hypothetical 2% risk-free rate.

> NoteUsers cannot adjust the risk-free rate from the "Settings/Properties" tab. The only way to change the value is by specifying a `risk_free_rate` argument in the strategy() statement.

`use_bar_magnifier`

The `use_bar_magnifier` parameter of the strategy() declaration statement specifies whether the strategy enables high historical bar detail by default. Users can override the specified default via the Bar detalization settings in the strategy's "Settings/Properties" tab and at the top of the strategy report. If the value is `true`, the broker emulator retrieves available prices from a *lower timeframe* by default for more precise intrabar order fills on historical bars. If the argument is `false` (default), the broker emulator relies on default *assumptions* about intrabar price movement rather than using prices from a lower timeframe. See the Broker emulator section of the Strategies page to learn more about this feature.

> NoteHigh historical bar detail is available only to accounts with Premium and Ultimate plans. Lower-tier plans cannot use it, regardless of whether the declaration statement includes `use_bar_magnifier = true`.

`fill_orders_on_standard_ohlc`

The `fill_orders_on_standard_ohlc` parameter of the strategy() declaration statement specifies whether the broker emulator fills the strategy's orders using actual prices by default when the strategy executes on a Heikin Ashi chart. Users can activate or deactivate the feature via the "Heikin Ashi mode" input in the strategy's "Settings/Properties" tab. If the value is `false`, the emulator fills the strategy's orders using the chart's *synthetic prices* by default. If `true`, it fills the orders using the *actual* open, high, low, and close prices from a *standard chart* dataset for more realistic results. The default argument is `false`.

> NoticeThis feature **does not** affect backtests on other non-standard charts, such as Renko or Kagi. A strategy always uses the chart's synthetic prices when executing on those chart types, and therefore

> produces *unreliable* results, regardless of the specified `fill_orders_on_standard_ohlc` argument. See the Strategy produces unrealistic results on non-standard chart types article in our Help Center to learn more.

`library()`

The library() function declares that the script is a library. Libraries *export* reusable functions, methods, user-defined types (UDTs), enum types, or constant variables. Libraries can also include *non-exported* code to demonstrate how they work and how to use them. Indicators, strategies, and other libraries can use the import keyword to import a published library's exported code components. Importing components from libraries often helps programmers streamline script creation and simplify source code.

The VisibleChart publication from PineCoders is an example of a library. It exports functions that perform calculations on the chart's visible bars. The example script in the FAQ entry Can I create an indicator that plots like the built-in Volume or Volume Profile indicators demonstrates how scripts can import and use functions from this library.

Because the primary purpose of a library is to export components for other scripts, they have multiple unique characteristics, including the following:

- Libraries are the only scripts that can use the export keyword.
- Libraries *cannot* directly create alert triggers, but they can *export* custom functions that contain alert() calls. Indicators and strategies can use the alert triggers from calls to those functions.
- A library's title acts similarly to a *namespace* identifier when another script imports the library. Therefore, unlike indicators and strategies, libraries must follow identifier naming rules in their titles.
- User-defined functions and methods exported by libraries must prefix each declared parameter with a type keyword.
- The example code of a library executes similarly to an indicator. When applied to a chart, the code executes *once per bar* on historical bars and *once per tick* on realtime bars.
- Libraries use default indicator properties when their code executes on a chart. The declaration statement of a library does not include parameters for setting decimal precision, plot formatting, scales, drawing limits, or other script properties.
- Unlike indicators, libraries can use the available `strategy.*` built-ins. However, unlike strategies, a library's example code does *not* display trade markers on the chart or generate a strategy report.
- For a library to compile, it must use the export keyword to export *at least one* function, method, enum, UDT, or "const" variable.

The library() function's signature is as follows:

```
library(title, overlay, dynamic_requests) → void
```

The `overlay` parameter behaves the same as that of the indicator() and strategy() declaration statements. Refer to the `overlay`, `scale`, and `behind_chart` section above for information about this parameter.

The `title` and `dynamic_requests` parameters are also common to the indicator() and strategy() functions. However, they have some *unique* characteristics in libraries, as explained in the sections below.

`title`

The `title` parameter of the library() declaration statement specifies the library's unique name, which other scripts *reference* to import and use the library's code. For example, if a `userName` user publishes a library that uses `"foo"` as the `title` argument, another script imports version 1 of the library using the following import statement:

```
// Imports version 1 of the `foo` library from the `userName` user. import userName/foo/1
```

The script can then use the library's defined title (or a specified *alias*) similarly to a *namespace* to access the imported components. For example, if the library exports a function named `bar()`, the script that imports the library references the library's title using *dot notation* when calling the function:

```
// Calls the `bar()` function from the `foo` library and assigns the result to a variable. result = foo.bar()
```

Because a library's title behaves as a *code identifier* in other scripts, the `title` argument must follow identifier naming rules. The "string" argument can contain ASCII letters (`a–z` and `A–Z`), numeric digits (`0–9`), and underscores (_). The argument cannot be an empty string, cannot contain spaces or special characters, and cannot *start* with a numeric digit. Special characters include any of the following:

- Basic punctuation, including periods (`.`), commas (`,`), quotation marks (`"`), apostrophes (`'`), exclamation points (`!`), etc.
- Symbols that scripts use for syntax, such as parentheses (`( )`), square brackets (`[ ]`), plus signs (`+`), hyphens (`–`), asterisks (`*`), slashes (`/`), and percent signs (`%`).
- Currency symbols, such as `$` or `€`.
- Non-ASCII letters and digits, such as the Unicode character `𝔸` (U+1D5A0).
- Other Unicode characters, such as emoji or special-purpose symbols.

For example, a string such as `"Library_for_14_day_averages"` is a valid `title` argument for the library() declaration statement, but an argument such as `"Library for 14–day averages"` causes a *compilation error*.

> NoteWhen preparing a publication for a library, the `title` argument appears as the *suggested title* in the "Publish script" window. Programmers can specify a custom title for the publication if they wish. However, to import the library in another script, the import statement requires the `title` argument defined in the library's declaration statement, **not** the publication's custom title. Therefore, we recommend using the `title` argument as the title of a library publication for consistency.

If a user applies the library directly to their chart, the `title` argument's text appears as the display name in all relevant chart locations, including the script's status line, the data window, and the Pine Logs pane.

`dynamic_requests`

The `dynamic_requests` parameter of the library() declaration statement specifies whether the library can use dynamic requests. If the argument is `true` (default), the library can use `request.*()` function calls with "series" arguments to define the requested ticker ID and timeframe, include `request.*()` calls in the local scopes of conditional structures or loops, and execute nested requests. Additionally, the library can *export* user-defined functions and methods that use `request.*()` calls within their function scopes.

> NoteAll `request.*()` calls in a library's *exported* functions **cannot** use `expression` arguments that *depend* on any of the functions' *parameters*. However, `request.*()` call arguments that define ticker ID,

> timeframe, and other settings of a request *can* depend on exported function parameters.

If the `dynamic_requests` argument is `false`, the library allows `request.*()` calls only in the *global scope* or within *non-exported* functions, and those calls require arguments with “simple” or a weaker type qualifier for all parameters except for `expression`.

The example library below exports a custom `requestFinancialInsights()` function, which uses multiple `request.*()` calls to retrieve the quarterly Earnings Per Share (EPS), total revenue, total outstanding shares for a stock, and estimates the instrument’s market capitalization. The function returns a tuple containing all four values. The library can export this function because its declaration statement enables dynamic requests.

The library’s example code, listed below the user-defined function, demonstrates one way that programmers who import the library can use the function. The code creates a table and populates its cells with a `requestFinancialInsights()` call’s results on the last available bar:

//@version=6 //@description This library exports a function that uses dynamic `request.*()` calls to retrieve multiple financial // metrics for a stock instrument. library("FinancialInsights", overlay = true, dynamic_requests = true) //#region --- Exported code --- //@function Requests the latest quarterly Earnings Per Share (EPS), total revenue, and total outstanding shares, // and calculates the latest market capitalization value for the specified stock. //@param symbol The symbol or ticker ID for the data requests. Requires an exchange prefix (e.g., `"NASDAQ:AAPL"`). //@returns A tuple containing the EPS, total revenue, outstanding shares, and market cap values, respectively. export requestFinancialInsights(string symbol) => //@variable The latest Earnings Per Share reported for the stock. float eps = request.earnings(symbol, earnings.actual) //@variable The quarterly total revenue reported for the issuing company. float totalRevenue = request.financial(symbol, "TOTAL_REVENUE", "FQ") //@variable The quarterly total number of outstanding shares reported for the stock. float totalSharesOutstanding = request.financial(symbol, "TOTAL_SHARES_OUTSTANDING", "FQ") //@variable The market capitalization, estimated by multiplying outstanding shares by current share price. float marketCap = totalSharesOutstanding * close // Return the four results in a tuple. [eps, totalRevenue, totalSharesOutstanding, marketCap] //#endregion //#region --- Example code --- // The code defined below shows an example of *how to use* the library's exported function. // This code is *not* exported; a script that imports the library cannot access it. //@variable References a `table` object that displays financial insights for the stock represented on the chart. var table tbl = table.new(position.top_right, 2, 5, color.yellow, border_color = color.gray, border_width = 1) // Initialize row and column header cells in the table on the first bar. if barstate.isfirst tbl.cell(0, 1, "Latest EPS"), tbl.cell(0, 2, "Total revenue") tbl.cell(0, 3, "Total outstanding shares"), tbl.cell(0, 4, "Market cap") tbl.cell(1, 0, str.format("{0} ({1})", syminfo.tickerid, syminfo.currency)) if barstate.islast // Call the `requestFinancialInsights()` function and declare a tuple of variables to store the data on the last bar. [currEPS, currRevenue, currShares, currMarketCap] = requestFinancialInsights(syminfo.tickerid) // Populate the remaining table cells with the retrieved results. tbl.cell(1, 1, str.tostring(currEPS, "0.00")) tbl.cell(1, 2, str.tostring(currRevenue, format.volume)) tbl.cell(1, 3, str.tostring(currShares, format.volume)) tbl.cell(1, 4, str.tostring(currMarketCap, format.volume)) //#endregion

Note that:

- We included `dynamic_requests = true` in the library() statement only to emphasize the `dynamic_requests` parameter. Specifying this argument is unnecessary; the value is `true` by default. A

compilation error occurs if we change the value to `false`, because the library cannot export the custom function or call it within the example code's if structure.

- The `//@description` annotation at the top of the script sets a *default description* for the library. Similarly, the `//@function`, `//@param`, and `//@returns` annotations specify documentation for the exported function. Users who import this hypothetical library can hover over its identifiers to view the formatted text from these annotations. Additionally, the "Publish script" window uses these annotations to generate a default publication description.
- The source code includes `//#region` and `//#endregion` annotations to define *collapsible regions* that visually separate the library's exported code from its non-exported code in the Pine Editor.

See the Request publication from the TradingView account for an advanced example of a library that exports custom functions using dynamic requests.

ADVANCED

Enums

> TipThis page contains *advanced* material. If you're new to Pine Script®, start by learning about core language components – such as the type system and the basics of the execution model – and explore other, more accessible features before venturing further.

Introduction

Pine Script Enums, otherwise known as *enumerations*, *enumerated types*, or enum types, are unique data types with all possible values (*members*) explicitly defined by the programmer in advance. They provide a human-readable, expressive way to declare distinct sets of *predefined values* that variables, conditional expressions, and collections can accept, allowing more strict control over the values used in a script's logic.

Declaring an enum

To declare an enum, use the enum keyword with the following syntax:

```
[export ]enum <enumName>
    <field_1>[ = <title_1>]
    <field_2>[ = <title_2>]
    ...
    <field_N>[ = <title_N>]
```

Each **field** in the enum represents a unique, *named member* (value) of the enum type. Users can specify optional "const string" **titles** for enum fields to add extra information about what their values represent. If the programmer does not specify a field's title, its title is the "string" representation of its name. Enum inputs display enum field titles within their dropdown menus in a script's "Settings/Inputs" tab. Scripts can also retrieve enum field titles using the str.tostring() function, allowing their use in additional calculations. See this section below for more information.

While the above syntax may look similar to the syntax for declaring user-defined types (UDTs), it's crucial to understand that enum types and UDTs serve different purposes. Scripts use UDTs to create objects with "series" fields that can hold values of *any* specified type. In contrast, enums are distinct groups of "const" fields representing the specific, *predefined values* of the same *unique* type. Scripts can use these types to define identifiers and collections that allow only a limited set of possible values.

For example, this code block declares a `Signal` enum with three fields: `buy`, `sell`, and `neutral`. Each field represents a distinct member (possible value) of the `Signal` enum type. Any variable declared with this type accepts only these members or na:

//@enum An enumeration of named values representing buy, sell, and neutral signal states. //@field buy Represents a "Buy signal" state. //@field sell Represents a "Sell signal" state. //@field neutral Represents a "neutral" state. enum Signal buy = "Buy signal" sell = "Sell signal" neutral

Note that:

- The `Signal` identifier represents the enum's name, which signifies the *unique type* to which the fields belong.
- We use the `//@enum` and `//@field` annotations to document the meaning of the enum and its members in the code.
- Unlike the `buy` and `sell` fields, the `neutral` field does not include a specified title. Therefore, its title is the "string" representation of its *name* (`"neutral"`).

To retrieve a member of an enum, use *dot notation* syntax on the enum name. For example, the following expression retrieves the `fieldName` member of the `enumName` type:

enumName.fieldName

As with other types, scripts can assign enum members to variables, function parameters, and UDT fields, enabling strict control over their allowed values.

For instance, the code line below declares a `mySignal` variable whose value is the `neutral` member of the `Signal` enum. Any value assigned to this variable later must also be of the same enum type:

mySignal = Signal.neutral

Note that the above line does not require specifying the variable's *type* as `Signal`, because the compiler can automatically determine that information from the assigned value. However, if we use na as the initial value instead, we must include `Signal` as the variable's type keyword to specify that `mySignal` accepts members of the `Signal` type:

Signal mySignal = na

Using enums

Scripts can compare enum members with the == and != operators and use the results of those comparisons in conditional structures, allowing the convenient creation of logical patterns with a reduced risk of unintended values or operations.

The following example declares an `OscType` enum with three fields representing different oscillator choices: `rsi`, `mfi`, and `cci`. The `calcOscillator()` function compares the `OscType` members within a switch structure to determine which `ta.*()` function it uses to calculate an oscillator. The script calls `calcOscillator()` using the value from an enum input as the `selection` argument, and then plots the returned oscillator value on the chart:

//@version=6 indicator("Using enums demo") //@enum An enumeration of oscillator choices. enum OscType rsi = "Relative Strength Index" mfi = "Money Flow Index" cci = "Commodity Channel Index" //@variable An enumerator (member) of the `OscType` enum. Its type is "input OscType". OscType oscInput = input.enum(OscType.rsi, "Oscillator type") //@function Calculates one of three oscillators based on a specified `selection` value. //@param source The series of values to process. //@param length The number of bars in the calculation. //@param selection Determines which oscillator to calculate. // Possible values are: `OscType.rsi`, `OscType.mfi`, `OscType.cci`, or `na`. calcOscillator(float source, simple int length, OscType selection) => result = switch selection OscType.rsi => ta.rsi(source, length) OscType.mfi => ta.mfi(source, length) OscType.cci => ta.cci(source, length) // Plot the value of a `calcOscillator()` call with `oscInput` as the `selection` argument. plot(calcOscillator(close, 20, selection = oscInput))

Note that:

- The `selection` parameter of the `calcOscillator()` function can accept one of only *four* possible values: `OscType.rsi`, `OscType.mfi`, `OscType.cci`, or na.
- The “Oscillator type” input in the script’s “Settings/Inputs” tab displays all `OscType` field titles in its dropdown menu. See the Enum input section of the Inputs page to learn more.

It's crucial to note that each declared enum represents a *unique* type. Scripts **cannot** compare members of different enums or use such members in expressions requiring a specific enum type, even if the fields have identical names and titles.

In this example, we added an `OscType2` enum to the above script and changed the `oscInput` variable to use a member of that enum. The script now causes a *compilation error*, because it cannot use a member of the `OscType2` enum as the `selection` argument in the `calcOscillator()` call:

//@version=6 indicator("Incompatible enums demo") //@enum An enumeration of oscillator choices. enum OscType rsi = "Relative Strength Index" mfi = "Money Flow Index" cci = "Commodity Channel Index" //@enum An enumeration of oscillator choices. Its fields *do not* represent the same values those in the `OscType` enum. enum OscType2 rsi = "Relative Strength Index" mfi = "Money Flow Index" cci = "Commodity Channel Index" //@variable An enumerator (member) of the `OscType2` enum. Its type is "input OscType2". OscType2 oscInput = input.enum(OscType2.rsi, "Oscillator type") //@function Calculates one of three oscillators based on a specified `selection` value. //@param source The series of values to process. //@param length The number of bars in the calculation. //@param selection Determines which oscillator to calculate. // Possible values are: `OscType.rsi`, `OscType.mfi`, `OscType.cci`, or `na`. calcOscillator(float source, simple int length, OscType selection) => result = switch selection OscType.rsi => ta.rsi(source, length) OscType.mfi => ta.mfi(source, length) OscType.cci => ta.cci(source, length) // This line causes an error. The `selection` parameter of `calcOscillator()` requires an `OscType` value, // but the `oscInput` value has a *different* type (`OscType2`). plot(calcOscillator(close, 20, selection = oscInput))

Utilizing field titles

The “string” titles of an enum's fields allow programmers to add extra information to each member. These field titles appear within a dropdown input in the script's “Settings/Inputs” tab when the script uses the input.enum() function.

Scripts can also use enum field titles in their calculations and logic. To access the title of an enum member, use the str.tostring() function on the member.

The following example combines the titles from members of two separate enums to create a ticker identifier for a data request. The script declares two enums, `Exchange` and `Pair`, whose fields represent *exchange* and *currency pair* names. It creates two enum inputs using these enums, and assigns their values to the `exchangeInput` and `pairInput` variables. The script uses str.tostring() on those variables to retrieve the selected titles, and then concatenates the results to form the “Exchange:Symbol” pair for the request.security() call:

//@version=6 indicator("Utilizing field titles demo") //@enum An enumeration of cryptocurrency exchanges. All field titles are the same as the field names. enum Exchange BINANCE BITSTAMP BITFINEX COINBASE KRAKEN //@enum An enumeration of cryptocurrency pairs. All the field titles are the same as the field names. enum Pair BTCUSD ETHUSD SOLUSD XRPUSD //@variable An enumerator (member) of the `Exchange` enum. Exchange exchangeInput = input.enum(Exchange.BINANCE, "Exchange") //@variable An enumerator (member) of the `Pair` enum. Pair pairInput = input.enum(Pair.BTCUSD, "Pair") //@variable The exchange-symbol pair for the data request. simple string symbol = str.tostring(exchangeInput) + ":" +

str.tostring(pairInput) // Plot the `close` value requested from the `symbol` context. plot(request.security(symbol, timeframe.period, close), "Requested close", color.purple, 3)

Note that:

- None of the members of the `Exchange` or `Pair` enums have specified titles. Therefore, each field's title is the “string” representation of its name, as shown by the script's enum inputs.
- Calling the str.tostring() function on an enum field is the **only** way to retrieve its title for additional calculations. The str.format() and `log.*()` functions *cannot* accept enum members. To use a field's title in a string formatting function, call str.tostring() on the field first, then pass the resulting “string” to the function.

Collecting enum members

Pine Script collections (arrays, matrices, and maps) can store enum members, allowing strict control over the values they can contain. To create a collection of enum members, include the enum’s *name* in the *type template* of the `*.new*()` function from the collection’s namespace (e.g., array.new<type>()).

For example, the following code block creates an empty array object to store members of a `FooBar` enum. The only values that the array can contain are `FooBar.foo`, `FooBar.bar`, `FooBar.baz`, and na:

//@variable An enumeration of miscellaneous named members. enum FooBar foo bar baz //@variable An array that can only contain the following values: `FooBar.foo`, `FooBar.bar`, `FooBar.baz`, `na`. array<FooBar> fooBarArray = array.new<FooBar>()

Enums are particularly helpful when working with maps, as unlike other *non-fundamental* types, scripts can declare maps with *keys* of an enum type, enabling strict control over all possible keys allowed in their key-value pairs.

The following example uses a map with enum keys and “int” values to track and count signal states across chart bars. The script's `Signal` enum contains five fields representing specific named states. The `signalCounters` map uses the `Signal` name as the *first keyword* in its type template to specify that it can accept only `Signal` members as keys.

The script uses a switch structure to calculate a `signalState` variable whose value is a member of the `Signal` enum, which it uses to determine the counter value to update in the `signalCounters` map. It constructs a “string” to represent the key-value pairs of the map and displays the result in a single-cell table on the last chart bar:

//@version=6 indicator("Collecting enum members demo", overlay = true) //@enum An enumeration of named signal states. enum Signal strongBuy = "Strong buy" buy = "Buy" neutral = "Neutral" sell = "Sell" strongSell = "Strong sell" //@variable The number of bars in the signal calculation. int lengthInput = input.int(50, "Length", 2) //@variable A map of `Signal.*` keys and "int" values counting the number of bars with each signal state. // Allowed keys: `Signal.strongBuy`, `Signal.buy`, `Signal.neutral`, `Signal.sell`, `Signal.strongSell`, `na`. var map<Signal, float> signalCounters = map.new<Signal, float>() //@variable A single-cell table displaying the key-value pairs of the `signalCounters` map. var table infoTable = table.new(position.top_right, 1, 1, chart.fg_color) if barstate.isfirst // Put `Signal.*`-"int" pairs into the `signalCounters` map to establish insertion order. signalCounters.put(Signal.strongBuy, 0) signalCounters.put(Signal.buy, 0) signalCounters.put(Signal.neutral, 0) signalCounters.put(Signal.sell, 0) signalCounters.put(Signal.strongSell, 0) // Initialize the `infoTable` cell. infoTable.cell(0, 0, text_color =

chart.bg_color, text_halign = text.align_left, text_size = size.large) // Calculate the EMA and Percent rank of `source` data over `length` bars. float ema = ta.ema(close, lengthInput) float rank = ta.percentrank(close, lengthInput) //@variable A `Signal` member representing the current signal state based on `ema` and `rank` values. Signal signalState = switch close > ema => rank > 70 ? Signal.strongBuy : rank > 50 ? Signal.buy : Signal.neutral close < ema => rank < 30 ? Signal.strongSell : rank < 50 ? Signal.sell : Signal.neutral => Signal.neutral // Add 1 to the value in the `signalCounters` map associated with the `signalState` key. signalCounters.put(signalState, signalCounters.get(signalState) + 1) // Update the `infoTable` cell's text using the keys and values from the `signalCounters` map on the last bar. if barstate.islast string tableText = "" for [state, count] in signalCounters tableText += str.tostring(state) + ": " + str.tostring(count) + "\n" infoTable.cell_set_text(0, 0, str.trim(tableText))

Note that:

- The `signalCounters` map can contain up to *six* key-value pairs, as the `Signal` enum has *five* predefined values, plus a possible value of na, and maps cannot contain *repetitive* keys.
- The script declares the `signalCounters` variable using the var keyword, signifying that the assigned map instance persists across executions.
- On the first chart bar, the script uses five map.put() calls to establish the *insertion order* of keys in the `signalCounters` map. See this section of the Maps page for more information.
- To minimize *resource usage*, the script declares the `infoTable` variable and initializes the referenced table's cell only on the *first bar*, and then updates the cell's text on the *latest bar*. See the Reducing drawing updates section of the Profiling and optimization page to learn more.

Shadowing

In contrast to user-defined types (UDTs), which can have names that *shadow* some built-in types or namespaces, enum types require *unique* names that do **not** match any built-in types or namespaces.

For example, this code declares four enums named `Syminfo`, `syminfo`, `polyline`, and `ta`. The last three all cause a compilation error because their names match built-in namespaces:

//@version=6 indicator("Shadowing demo") // Naming an enum "Syminfo" with a capital "S" works without an issue. enum Syminfo abcd // In contrast, the names "syminfo", "polyline", "ta", etc. cause a compilation error because they match // built-in namespaces: enum syminfo abcd enum polyline abcd enum ta abcd

Execution model

Introduction

Pine Script® relies on an event-driven, sequential execution model to control how a script's compiled source code runs in charts, alerts, Deep Backtesting mode, and the Pine Screener.

In contrast to the traditional execution model of most programming languages, Pine's runtime system executes a script *repeatedly* on the sequence of *historical bars* and *realtime ticks* in the dataset on which it runs, performing *separate* calculations for *each bar* as it progresses. After each execution on a closed bar, the necessary data from that execution becomes part of an internal time series, and the script can use that data in its calculations on subsequent bars.

This combination of sequential executions and storage enables programmers to use minimal code to write scripts with dynamic calculations that advance across a dataset bar by bar.

The execution model and time series structure closely connect to the type system – together, they define how a script behaves as it runs on a dataset. Although it's possible to write simple scripts without understanding these foundational topics, learning about them and their nuances is key to becoming proficient in Pine Script.

This page explains the execution model in two parts: The basics and The details. The first part provides quick, actionable information about the model for beginners. The second part offers an *advanced*, in-depth breakdown of the model's workings and unique behaviors. To make the most of the information on this page, we recommend that newcomers to Pine Script start with The basics, learn about other topics in this manual, and then come back to this page for the advanced details.

The basics

The following sections outline core principles of the execution model for beginners. If you are new to Pine Script, start here.

Bar-by-bar execution

The dataset for a symbol on a given timeframe, as shown on a chart, consists of a sequence of bars representing a *time series*. Each bar in the sequence represents the price and volume for a specific time period. The first (leftmost) bar on a chart corresponds to the *earliest* period, and the last (rightmost) bar corresponds to the *most recent* period.

Much of the power of Pine Script stems from its ability to process this time series data efficiently. When a user runs a script, its code does *not* execute just once; it executes from start to end on *each bar* in the symbol's dataset individually, progressing from the first available bar to the most recent bar. Each separate script execution performs calculations or generates outputs (e.g., plots) for a *specific bar* using the data available on that bar.

A script can retrieve price, volume, and other essential data for each bar on which it executes by using the built-in variables that hold bar information, such as open, high, low, close, and volume. These variables automatically *update* before each new execution to store the values for the *current bar*.

For example, the simple script below uses the plot() function to display the series of close values (i.e., the closing price of each bar) on the chart:

//@version=6 indicator("Bar-by-bar execution demo", overlay = true, behind_chart = false) // Plot the `close` series on the chart. // This call defines the plotted point for the current bar on each execution. plot(close, "Close price", chart.fg_color, 5)

When a user first adds this script to their chart, its code executes *once* for *every bar* in the available dataset. As the script runs on the data, two primary steps occur on each bar:

1. The close variable automatically updates to hold the current bar's latest price.
2. The plot() function call plots the updated close value at the current bar's position.

When the script finishes its run from the first bar to the most recent bar, the result is a simple *line plot* showing the progression of closing prices across the chart's history:

Note that the above script evaluates the plot() function call once for every bar on the chart, not just once in total. On each separate execution, the call defines the plotted point for the current bar: the chart's first bar during the first execution, the second bar during the next, and so on.

This pattern illustrates a key principle of Pine's execution model: on each successive execution, a script *re-evaluates* function calls and other expressions within its required *scopes* to perform separate calculations for the current bar.

> NoteThe scope of an expression is the part of the code where the script can access it. A script evaluates its *global* scope *once* per execution, i.e., on *every bar*. In contrast, it evaluates *local* scopes – such as the code inside conditional structures, user-defined functions, and loops – *zero*, *one*, or *several* times per execution, depending on the logic.

Repeated code evaluation also applies to variable declarations. By default, a script does not declare a variable only once throughout its runtime; the script *re-declares* that variable and assigns an initial value based on the current bar's data during *each* new evaluation of its scope.

Let's look at a simple example. The following script declares an `x` variable of the "int" type with an initial value of 0. Then, it increases the variable's value by 10 with the addition assignment operator (+=). The script calls plot() to display the value of `x` on each bar in a separate pane:

//@version=6 indicator("Repeated declarations demo") //@variable A user-defined variable. The script declares this variable and initializes it to 0 on *every* execution. int x = 0 // Increase the value of `x` by 10 on every bar. x += 10 // Plot the value of `x`. // Because `x` begins at 0 on every execution, and the script adds 10 to that value, the plotted value is always 10. plot(x, "`x` value", color.blue, 3)

As shown above, the script plots a value of 10 on every bar, because the `x` variable *does not* carry over from bar to bar; the script declares the variable *repeatedly*. On each bar, the script re-declares `x` with an initial value of 0, then adds 10 to that value, resulting in a final value of 10 for every plotted point.

Programmers can change the behavior of a variable, enabling it to *persist* and preserve updates to its value *across bars*, by including the var keyword in its declaration, as described in the Declaration modes section of the Variable declarations page.

Below, we modify the previous script by adding var to the x declaration. Now, the script declares and initializes x only *once* – on the *first bar* – and that variable persists across *all* bars that follow. The script now plots a line that *increases* by 10 on each bar, because x preserves the result from each addition across the chart's history. The value changes from 0 to 10 on the first bar, then to 20 on the second, and so on:

//@version=6 indicator("Persistent declarations demo") //@variable A *persistent* variable. The script initializes this variable only on the *first execution*. // The variable preserves all changes to its value on each closed bar. var int x = 0 // Increase the value of `x` by 10 on every bar. x += 10 // Plot the `x` series on the chart. // Because the script declares `x` using `var` and then increments its value, the value never resets to 0. // The plotted value is 10 on the first bar, 20 on the next, and so on. plot(x, "`x` series", color.blue, 3)

Storing and using data from previous bars

As a script runs on a dataset, the states of its variables, function calls, and other expressions are automatically *committed (saved)* to an internal *time series* on each bar, creating historical trails of previous bar values that the script can access during its calculations on the current bar. The script can use these previous values by doing either of the following:

- Using the [] history-referencing operator. The number in the square brackets represents how many *bars back* from the current bar the script looks to retrieve a past value. For instance, `close[1]` retrieves the close value from *one bar before* the current bar, and `close[100]` retrieves the value from *100 bars back*.
- Calling the built-in functions that calculate on past values internally, such as `ta.*()` functions. For instance, `ta.change(close, 10)` calculates the difference between the current value of close and its value from 10 bars back.

The example below uses both of the above techniques to perform calculations based on data from previous bars. The script calculates a series of bar-by-bar price returns and plots the result as color-coded columns. It declares two global variables on each bar: `priceReturn` for the calculated returns, and `returnColor` for the plot's color. The `priceReturn` value is the result of dividing the current one-bar change in closing prices (`ta.change(close, 1)`) by the previous bar's closing price (`close[1]`). The `returnColor` value is color.teal if the current value of `priceReturn` is higher than the value from the previous bar (`priceReturn[1]`), and color.maroon otherwise:

//@version=6 indicator("Storing and using data from previous bars demo") //@variable The one-bar price return, based on the current and *previous* bars' `close` values. // This variable's final value on each bar automatically becomes part of the internal time series. float priceReturn = ta.change(close, 1) / close[1] //@variable Is `color.teal` if the `priceReturn` value is above the value on the previous bar; `color.maroon` otherwise. color returnColor = priceReturn > priceReturn[1] ? color.teal : color.maroon // Plot the current `priceReturn` value as a column, colored using the value of `returnColor`. plot(priceReturn, "Price return", returnColor, 1, plot.style_columns)

Note that:

- This script does *not* plot a column on bar 0 (the *first* bar). The `priceReturn` value is na on that bar, because there is *no previous bar* available for the script to reference at that point.

> NoticeFor consistency, use historical references only on variables or expressions that the script evaluates on **every bar**, in the *global* scope. A script that references the history of variables or expressions defined

in *local* scopes – such as the code inside an if statement – can cause *unintended results*. The compiler warns users about this behavior directly inside the Pine Editor. For *advanced* details on this behavior, see the Time series in scopes section.

Realtime bars

When a script first runs on a chart, all *closed* bars in the accessed dataset are *historical bars*. These bars represent data for elapsed time periods where the final price and volume are *confirmed*. All indicators execute **once** per historical bar.

When the rightmost bar on the chart is *open*, it is a *realtime bar*. Unlike a historical bar, whose values are final, a realtime bar *updates* its values as new price or volume data becomes available. After the bar closes, it becomes an *elapsed realtime bar*, which is then no longer subject to change as the script runs.

Because the final values for a realtime bar are *unknown* until the bar closes, an indicator executes differently on that bar than it does on historical bars. The script executes not once, but **repeatedly** on the realtime bar – once for each new *update (tick)* – to *recalculate* its results using the latest data.

Note

On the open realtime bar, variables such as high, low, close, and volume hold the *latest* available values for the bar. These values **do not** represent *confirmed* data for the bar until the script executes on that bar's *closing tick*.

Scripts can identify historical and realtime bars, and whether a realtime bar's data is confirmed, by using the barstate.ishistory, barstate.isrealtime, and barstate.isconfirmed variables. See the Bar states page to learn more.

Before each recalculation on the realtime bar, the data for a script's variables, expressions, and outputs on that bar is *cleared*, or *reset*. We refer to this process as *rollback*. The purpose of rollback is to revert the script to the same confirmed state it had when the realtime bar opened. This process ensures that the script's calculations for the bar operate only on the latest available data, without relying on *temporary data* from the bar's *previous ticks*.

NoticeA variable can *escape* rollback and persist across *all ticks* within a bar if its declaration includes the varip keyword. This behavior is helpful for calculations that require data from before a bar's closing tick. However, it can also cause repainting, because the ticks that occur before a realtime bar's close become **unavailable** after the script reloads. Refer to the Declaration modes section of the Variable declarations page to learn more about this keyword.

Let's look at rollback and recalculation in action. The following script uses ta.stoch() to calculate the Stochastic oscillator based on the close, high, and low values over a specified number of bars, then plots the

result in a separate pane. It also calls bgcolor() to highlight the background on each realtime bar – where barstate.isrealtime is `true` – for visual reference:

//@version=6 indicator("Recalculation on realtime bars demo") //@variable The number of bars in the Stochastic calculation. Users can change this value in the "Settings/Inputs" tab. int lengthInput = input.int(10, "Length", 1) //@variable The Stochastic oscillator, based on the `close`, `high`, and `low` values over `lengthInput` bars. float stochastic = ta.stoch(close, high, low, lengthInput) // Plot the `stochastic` value for each bar. plot(stochastic, "Stochastic %K", color.teal, 3) // Highlight the background of each realtime bar. bgcolor(barstate.isrealtime ? color.new(color.purple, 80) : na, title = "Realtime background highlight")

When we add the script to our chart, it executes once per bar in the chart's history, from the leftmost bar to the rightmost bar. However, the rightmost bar on our chart is still *open*. Therefore, it is a *realtime bar*, not a historical bar. After the script reaches that bar, it begins executing once for *every new update* to the bar's data. Each new script execution calculates on the latest available prices and *replaces* the bar's previous result.

For instance, in the initial image below, the oscillator's value 10 seconds into the open realtime bar (the one with the purple background) is 32.08:

Every time the bar updates, rollback *resets* the script's data for that bar, and the script *recalculates* its result using the latest high, low, and close values. Here, halfway through the realtime bar's period, the oscillator's plot now shows a value of 16.71:

Recalculation continues for each successive update to the bar. Then, the script reaches the bar's closing tick, where the prices become *confirmed*. On that tick, the script calculates the oscillator's final value of 19.35. Afterward, another realtime bar opens, and the pattern of rollback and recalculation continues on that bar:

Note that:

- Only the values for a realtime bar's *final tick* become part of the internal time series. The values from ticks *before* the bar's close are **not** saved.
- The input.int() function returns a value of the "input int" *qualified type*. Values qualified as "input" are established *before* the first script execution, and they remain consistent throughout the script's runtime. If the user changes the "Length" input to a new value, the script *restarts* to perform new calculations across the dataset using that value. See the Inputs page and the Qualifiers section of the Type system page to learn more about script inputs and the "input" qualifier.
- If the script restarts, all the realtime bars from the previous script run become *historical bars* in the new run. Therefore, after restarting, the script executes only **once** on each of those bars and does *not* highlight their background.

> Note*Strategy* scripts do not execute in the same way as indicators by default; they execute only *once* on *every bar*, including all *realtime bars*. Calculations occur on each realtime bar only after the bar **closes**. Users can change this behavior with the `calc_on_every_tick`, `calc_on_order_fills`, and `calc_on_every_history_tick` parameters of the strategy() function. See the Strategies page to learn more about strategy scripts and how they differ from indicators.

The details

The following sections provide in-depth details about Pine's execution model, including the mechanics of executions on historical bars and realtime bars, which events trigger script executions, and how the runtime system maintains data across executions in a time series format.

> TipNew to Pine Script? To make the most of the *advanced* information below, start by learning The basics and understanding other core concepts – such as the type system, variable declarations, operators, conditional structures, and user-defined functions – before venturing further.

Executions on historical bars

When a script loads on the chart or in another location after an execution-triggering event, its compiled source code executes on *every* accessible bar in the current dataset in order, starting with the first bar.

> NoteThe first bar that a script accesses depends on the `calc_bars_count` parameter of its declaration statement and the limits of the user's *plan*. If the `calc_bars_count` argument is a nonzero value that is less than or equal to the plan's bar limit, script executions begin at the specified number of bars *before* the latest bar. Otherwise, executions start at the *earliest* available bar.

While the script loads, the runtime system performs the following steps for *each bar* that it accesses:

1. It updates the built-in variables that hold bar information. For instance, the system sets the open, high, low, and close variables to hold the OHLC price values of the bar *before* each execution.
2. It executes the script's compiled code from start to end using the data available as of the current bar.
3. After the execution ends, the system commits (saves) all necessary data for the current bar to the time series. The script can then access that data from historical buffers during its executions on subsequent bars by using the history-referencing operator or the built-in functions that reference past bars internally.

These steps repeat for every successive bar up to the most recent bar. After the runtime system completes this process across the dataset, the script's committed *outputs* – such as plots, drawings, Pine Logs, and strategy report results – become available to the user.

All the closed bars on which the script executes while loading are *historical*, because they represent data points that were confirmed before the event that triggered the loading process. By default, all scripts execute **once** for each historical bar.

> TipScripts can identify which bars have a historical state with the barstate.ishistory variable. Its value is `true` for every closed bar accessed during the script's loading time and `false` for all bars that close afterward. See the Bar states page to learn more about `barstate.*` variables.

Let's examine a simple indicator to understand how script executions work on historical bars.

The script below calculates the 20-bar moving average of close values and plots the result on the chart. The color of the plot depends on whether the average is above or below the value on the previous bar. The script

also increments an `executionNum` variable to count code executions, then plots the result alongside bar_index for comparison. Additionally, it highlights the background of historical bars in orange for visual reference:

//@version=6 indicator("Executions on historical bars demo") //@variable The average of the latest 20 `close` values. float sma = ta.sma(close, 20) //@variable Is `color.green` if the `sma` value is above the value on the previous bar; `color.red` otherwise. color plotColor = sma > sma[1] ? color.green : color.red //@variable Tracks the current execution number, where 0 represents the first execution. varip int executionNum = -1 // Add 1 to the `executionNum` value. executionNum += 1 // Display the `sma` as a line plot on the main chart pane, colored by the `plotColor`. plot(sma, "SMA", plotColor, 3, force_overlay = true) // Display the `executionNum` and `bar_index` series in a separate pane. plot(executionNum, "Execution number", color.purple, 5) plot(bar_index, "Bar index", color.aqua, 2) // Highlight the chart's background in translucent orange when `barstate.ishistory` is `true`. bgcolor(barstate.ishistory ? color.new(color.orange, 70) : na, title = "Historical highlight", force_overlay = true)

The statements and expressions in this source code might appear static at first glance. However, they have *dynamic* behavior across bars because the system executes the script *repeatedly* – once for each successive data point. Below, we inspect the code step by step to explain how the script works during its historical executions.

The indicator() call at the top of the code is a declaration statement that defines the script's type and properties once, at *compile time*. This statement does not execute as the script runs on the dataset:

indicator("Executions on historical bars demo")

Before each script execution on a bar, the runtime system updates the built-in bar_index and close variables required in the calculations. The bar_index value is the bar's global *time series index*, where 0 represents the first bar, 1 represents the second, and so on. The close variable holds the bar's *latest price*. For historical bars, its value is the *final price* at the bar's closing time.

Each time that the script executes, it declares and initializes a global `sma` variable of the "float" type. This variable declaration happens on *every* execution because the code line does not specify a declaration mode. The variable's assigned value is the result of a ta.sma() function call. The call returns the average of the latest 20 close values as of the current bar, or na if fewer than 20 bars are available. After the execution ends, the system commits the new value of `sma` to the time series:

//@variable The average of the latest 20 `close` values. float sma = ta.sma(close, 20)

Note that:

- The `//@variable` comment above the `sma` declaration is an annotation that *documents* the variable in the code. The Pine Editor displays the comment in a pop-up window when the user hovers the mouse pointer over the variable.

During each execution, the script also initializes a `plotColor` variable of the "color" type. The script uses a ternary operation that compares the current `sma` value to `sma[1]` – the *last committed value* for `sma` as of the *previous bar* – to determine the `plotColor` variable's assigned value. If the current `sma` value is higher than the last committed value, the `plotColor` value is color.green. Otherwise, it is color.red:

//@variable Is `color.green` if the `sma` value is above the value on the previous bar; `color.red` otherwise. color plotColor = sma > sma[1] ? color.green : color.red

In contrast to the variables above, the script *does not* initialize the `executionNum` variable on every execution. Instead, initialization happens only *once* – on the *first* bar – because the variable declaration is in the *global scope* and uses the varip keyword. Once initialized, the variable *persists* across all subsequent bars and the ticks within those bars. Only the reassignment or compound assignment operators can change its value:

//@variable Tracks the current execution number, where 0 represents the first execution. varip int executionNum = -1

The code following the `executionNum` declaration uses the addition assignment operator (+=) to increase the variable's value by one on each new execution. Starting from -1, the value increases to 0 on the first execution after initialization, then 1 on the second, and so on:

// Add 1 to the `executionNum` value. executionNum += 1

The script evaluates the plot() and bgcolor() calls on every execution. Each plot() call creates a new point on a line plot at the bar's location on the time axis. The bgcolor() call creates a background color for the bar based on a ternary expression. The background is translucent orange if barstate.ishistory is `true`. Otherwise, it is na (no color):

// Display the `sma` as a line plot on the main chart pane, colored by the `plotColor`. plot(sma, "SMA", plotColor, 3, force_overlay = true) // Display the `executionNum` and `bar_index` series in a separate pane. plot(executionNum, "Execution number", color.purple, 5) plot(bar_index, "Bar index", color.aqua, 2) // Highlight the chart's background in translucent orange when `barstate.ishistory` is `true`. bgcolor(barstate.ishistory ? color.new(color.orange, 70) : na, title = "Historical highlight", force_overlay = true)

Note that:

- The plot() and bgcolor() calls that include `force_overlay = true` display their visuals on the main chart pane. The other plot() calls output visuals in a separate pane, because the indicator() call does not include `overlay = true`.

After the system executes the script on all available data points and finishes loading, the script's outputs then become visible on the chart:

Note that:

- When the script first loads, *all* bars, including the latest one, have an orange background because they initially represent *historical* data. However, the latest bar on our chart is still open, meaning it is a *realtime bar*. After a new tick arrives from the realtime data feed, the bar's values update, and the script executes *again* on that bar. The orange background for the bar then *disappears* because the system sets the value of barstate.ishistory to `false`.
- The `executionNum` and bar_index values are identical on historical bars because the script executes *once per bar* on that part of the dataset. However, they begin to differ on the realtime bar. On that bar, the script executes after *every new update* to recalculate its results, and the `executionNum` value increases each time. See the Executions on realtime bars section to learn more.
- An alternative, more robust method to track code executions is to use the Pine Profiler. The profiler analyzes the total runtime and execution count of every significant part of the source code. To learn more about this feature, see the Profiling and optimization page.

It's important to note that, unlike indicators, strategies can execute *more than once* per historical bar, depending on the specified calculation behavior. If the strategy() declaration statement includes `calc_on_order_fills = true`, or if the user selects the "On order fill" checkbox in the strategy's Script execution settings, the runtime system executes the script on *each available tick* where the broker emulator fills an order, or once per bar when there is no order to fill. Similarly, if the statement includes `calc_on_every_history_tick = true`, or if the user selects the "On history bar tick" checkbox in the strategy's execution settings, the system executes the script on *every* historical tick – even on the ticks where the broker emulator does not fill an order.

Let's look at a simple example. The following strategy changes the direction of its simulated position on each execution. If there is an open short position or no position, the strategy places a market order to close all short trades and enter a long trade. If a long position is open, the strategy places a market order to close it and open a short trade.

As with the previous example, this script increments an `executionNum` variable declared with varip to count new executions, plots the result alongside the bar_index series for comparison, and highlights the background of historical bars in orange with a bgcolor() call:

//@version=6 strategy("Default strategy behavior on historical bars demo") // Place a market order to close short trades and enter a long trade when there is a short position or no position. // Otherwise, if a long position is open, place a market order to close the long trades and enter a short trade. if strategy.position_size <= 0 strategy.entry("Long", strategy.long) else strategy.entry("Short", strategy.short) //@variable Tracks the current execution number, where 0 represents the first execution. varip int executionNum = -1 // Add 1 to the `executionNum` value. executionNum += 1 // Display the `executionNum` and `bar_index` series in a separate pane. plot(executionNum, "Execution number", color.purple, 5) plot(bar_index, "Bar index", color.aqua, 2) // Highlight the chart's background in translucent orange when `barstate.ishistory` is `true`. bgcolor(barstate.ishistory ? color.new(color.orange, 70) : na, title = "Historical highlight", force_overlay = true)

Note that:

- The strategy.entry() command creates entry orders. By default, a long entry using this command reverses an open short position, and a short entry reverses an open long position. See the Reversing positions section of the Strategies page to learn more.

The script above uses the default calculation behavior: it places a new order only at the close of each bar. The broker emulator fills the order at the next bar's opening price, as the trade markers on the chart above indicate. The `executionNum` and bar_index plots show the same values because the script executes only once per bar.

If we include `calc_on_order_fills = true` in the strategy() declaration statement, then by default, the runtime system *re-executes* the script on a bar after each new order fill to update the calculations. Our script's logic generates a new order on *every* execution, and the broker emulator considers historical bars to have *four ticks* for filling orders by default (the open, high, low, and close). Therefore, with this change, the script executes **four times** per historical bar instead of only once. As shown below, the strategy now shows four trade markers on each historical bar, and the `executionNum` value is four times that of the bar_index variable:

//@version=6 strategy("Calculation after order fill on historical bars demo", calc_on_order_fills = true) // Place a market order to close short trades and enter a long trade when there is a short position or no

position. // Otherwise, if a long position is open, place a market order to close the long trades and enter a short trade. if strategy.position_size <= 0 strategy.entry("Long", strategy.long) else strategy.entry("Short", strategy.short) //@variable Tracks the current execution number, where 0 represents the first execution. varip int executionNum = -1 // Add 1 to the `executionNum` value. executionNum += 1 // Display the `executionNum` and `bar_index` series in a separate pane. plot(executionNum, "Execution number", color.purple, 5) plot(bar_index, "Bar index", color.aqua, 2) // Highlight the chart's background in translucent orange when `barstate.ishistory` is `true`. bgcolor(barstate.ishistory ? color.new(color.orange, 70) : na, title = "Historical highlight", force_overlay = true)

Note that:

- This script can execute *more than four* times per chart bar if it enables high historical bar detail. When using high historical detail, the broker emulator uses OHLC values from a *lower timeframe* to determine the available historical ticks for filling orders. Therefore, the script executes up to four times per *lower-timeframe bar* when using this setting.
- The script can execute numerous times on a *realtime* bar, depending on the updates from the data feed, because *each new update* to the bar is a valid tick for filling the strategy's orders.
- An alternative way to confirm the script's increased execution count is to select and clear the "On order fill" check box in the strategy's Script execution settings while profiling the code.

Executions on realtime bars

After a script running on the chart or in an alert executes across all historical bars in a dataset, the runtime system continues to execute the script on the current bar, if it is open, and on any new bars that form later. We refer to these bars as *realtime bars*, because they represent incoming data from a separate data feed that the script can access only *after* it finishes loading.

As explained in the previous section, historical bars represent confirmed data points. By contrast, a realtime bar represents an initially *unconfirmed* data point that evolves as new updates (ticks) arrive from the realtime data feed. With each new tick, the bar's high, low, close, volume, and other values update to represent the latest data while the bar remains open. After the bar closes, it becomes an *elapsed realtime bar*, whose values no longer change. Then, a new realtime bar opens after another tick arrives, and that bar updates as new data becomes available.

> TipScripts can identify which bars have a realtime state with the barstate.isrealtime variable. Its value is `true` for every bar that closes *after* the script's loading time and `false` for all previous bars. Additionally, scripts can detect *closed bars* with the barstate.isconfirmed variable. To learn more about these and other `barstate.*` variables, refer to the Bar states page.

As an indicator or library script runs on an open realtime bar, its compiled code executes once after **every new update** from the data feed. With each new execution, the script recalculates its results for that bar using the latest data. Consequently, the states of the script's variables, expressions, and objects can *change* with each new execution while the bar remains open. The system *commits* the script's data for the realtime bar only after the bar closes.

After each script execution that occurs *before* a bar's closing tick, the runtime engine executes a *rollback* process. Rollback *resets* applicable script data to the latest committed states in the time series. This process

enables the script to recalculate the bar's results using only the latest available data – without the influence of *temporary* data from executions on the bar's previous ticks.

Below, we explain how recalculation and rollback affect a script's data and outputs, along with some notable exceptions to this process:

Reinitialize variables

The runtime system erases the states of any variables that the script initializes during its executions before a bar's close, excluding those declared using the varip keyword. When the script executes again after rollback, it *reinitializes* the variables with new values or references based on the latest available data.

Likewise, the system does not preserve the *temporary* states of built-in variables that hold values for the current bar. Before the new script execution, it sets the variables to use the bar's most recent data. For instance, the system updates close, high, and low with the latest, highest, and lowest prices reported since the bar's opening time.

Reset changes to `var` variables

Variables that use the var keyword in their declaration remain initialized after the *first* time that their scopes execute on a bar's *closing tick*. Their assigned values or references *persist* across subsequent bars, changing only after reassignment or compound assignment operations.

Although these variables preserve data across successive bars, they **do not** preserve data across executions on the *ticks* of an open bar. Rollback reverts all variables declared with var before the current bar to the last committed states in the time series as of the previous bar.

For instance, if a variable declared with var has a value of 20 on the open bar and 19 on the previous bar, the variable's value reverts to 19 before the script executes on the next tick of the same bar. The temporary value of 20 does not persist.

Replace plotted outputs

The `plot*()`, bgcolor(), barcolor(), and fill() functions create visual outputs on *every bar*. These outputs are *temporary* on the open realtime bar. When the script executes again after rollback, the new outputs for the bar from calls to these functions *replace* the ones from the previous tick.

For example, when the expression `plot(close)` executes on the open bar, it displays the bar's latest close value as of the current execution. However, the plotted result is **temporary** until the bar closes. After rollback, the close variable updates, then the script calls plot() again on the next execution to replace the output from the previous tick and display the new value.

Remove and revert objects

User-defined types (UDTs) and special types, such as collections and drawing types, are reference types. They define structures from which scripts create *objects* – independent entities that store data elsewhere in memory. Variables of these types hold *references (IDs)* that provide access to specific objects and their data; the variables do **not** store objects directly.

If a script creates objects on an open bar and does not assign their references to variables declared with the varip keyword, the rollback process *removes* those objects. During the next execution on the open bar, the script creates *new objects* if the updated logic allows it.

For example, if a script calls label.new() to create a label object on the open bar, the system *deletes* that object during rollback. On the next execution, the script evaluates label.new() again, creating a *new* label that replaces the output. The label created on the previous tick no longer exists.

Similarly, for objects of built-in or user-defined types with references assigned to var variables, the rollback process reverts any changes to those objects that occur on the open bar. The only exception is for UDTs with *fields* that include the varip keyword. See the Objects page for more information.

Exceptions

The runtime system does not revert *all* the data from script executions on an open bar. The following are notable exceptions to the rollback process:

- Variables or fields declared with the varip keyword **do not** revert to a previously committed state. They persist across *all* script executions after initialization, even those on the ticks of an open realtime bar.
- Logged messages in the Pine Logs pane do not disappear after rollback. The messages from any `log.*()` calls during executions on the ticks of realtime bars remain in the pane until the script reloads.
- The data from strategy orders placed or filled on the ticks within a bar is not subject to rollback. If a strategy script creates orders or the broker emulator fills orders on an open bar, the data from those events persists.
- Rollback does not erase logs for alerts from the “Alerts” menu. All messages from a script alert remain visible until the user restarts the alert.
- Runtime errors from the system or the runtime.error() function completely *stop* script executions. If an error occurs at any point while a script executes on an open bar, the system halts the script and does not revert the error after new updates from the data feed.

Let’s inspect the behavior of a simple indicator on realtime bars. The following script calculates an RSI of close values using ta.rsi() and displays the result with a plot() call. To track the number of executions that occur *per bar*, the script increments an `executions` variable declared with varip and calculates its one-bar change using ta.change(). The script converts each bar’s execution count to a string with str.tostring(), then displays the result in a color-coded label at the bar’s high. The label is purple if the bar is open. Otherwise, it is gray. The script also highlights the background of realtime bars in orange using bgcolor():

//@version=6 indicator("Executions on realtime bars demo") //@variable The 14-bar RSI of `close` prices. float rsi = ta.rsi(close, 14) //@variable Tracks the number of script executions, where 1 represents the first execution. varip int executions = 0 // Add 1 to the `executions` value. executions += 1 //@variable Is `color.gray` if the bar is confirmed (closed); `color.purple` otherwise. color labelColor = barstate.isconfirmed ? color.gray : color.purple // Calculate the one-bar change in `executions`, then convert the value to a string and display the result in a label. // Each call to `label.new()` creates a *new* `label` object. label.new(bar_index, high, str.tostring(ta.change(executions)), color = labelColor, textcolor = color.white, size = 20, force_overlay = true) // Plot the `rsi` value with colors based on whether the value is above 50 or not. plot(rsi, "RSI", rsi > 50 ? color.teal : color.maroon, 3) // Highlight the chart's background in translucent orange when `barstate.isrealtime` is `true`. bgcolor(barstate.isrealtime ? color.new(color.orange, 70) : na, title = "Realtime highlight", force_overlay = true)

When we first add the script to the chart, it does *not* add an orange background to any bar because it calculates only on data that exists at the script’s loading time. This data is *historical*. Each bar’s label shows a value of 1 because indicators always execute *once* per historical bar:

Notice the countdown timer and the *purple* label for the latest bar in the chart above. These both indicate that the bar is *open* and subject to changes. A new update from the data feed affects the bar's values, triggering rollback and a new script execution to recalculate the results.

When rollback occurs, the runtime system reverts the internal data of the ta.rsi() call to its last committed state, erases the state of the `rsi` variable, and deletes the latest label object. However, the system does not revert the `executions` variable because it uses the varip keyword.

After rollback, the system updates the built-in close, high, and `barstate.*` variables using the current bar's latest data, and the new execution begins. The script evaluates the ta.rsi() call using the new close price and reinitializes the `rsi` variable with the returned value. Then, it increases the `executions` value by one, evaluates ta.change() again, and creates a new label at the bar's current high price to show the updated result. Lastly, it evaluates the plot() and bgcolor() calls to replace the bar's plotted visuals. The last bar's label remains purple because the bar is still open, but the background color is now *orange* because barstate.isrealtime is `true`:

As subsequent updates become available from the data feed, the pattern of rollback and re-execution continues, and the script's outputs for the bar update with each new execution:

The last time that rollback and another execution occur on this bar is after the *closing tick*, when the bar becomes an *elapsed* realtime bar. After the final execution, the bar's label is *gray* because barstate.isconfirmed is `true`. The runtime system then *commits* necessary data from this execution to the time series for calculations on future bars.

Then, a new realtime bar opens after another update from the data feed, and the execution pattern continues:

Note that:

- Although the previous bar is now confirmed, it still has an orange background corresponding to a *realtime* state because it closed **after** the script's loading time. When the script later reloads after an execution-triggering event, that bar becomes *historical*.

It's important to note that strategies often execute differently than indicators on realtime bars. By default, they execute only **once** per bar at each *closing tick* without undergoing rollback. However, users can modify a strategy's calculation behavior to allow rollback and re-execution on a bar before its closing tick.

If the strategy() statement includes `calc_on_every_tick = true`, or if the user selects the "On realtime bar tick" checkbox in the strategy's Script execution settings, the script executes on a realtime bar after *each new update* from the data feed, similar to an indicator.

Additionally, if the strategy() statement includes `calc_on_order_fills = true` or the user selects the "On order fill" checkbox in the strategy's execution settings, the script executes on *each tick* where the broker emulator fills an order. With this behavior, the system can execute the script multiple times on the open bar, but only on the ticks where an *order fill* occurs.

> NoteRollback typically occurs only after script executions on realtime bars. However, it can also happen on *historical bars* for strategies that use the "On order fill" or "On history bar tick* execution settings, because such scripts can execute *more than once* on each historical bar.

To summarize the general process for script executions on realtime bars:

- An indicator or library script executes on the *first available tick* in an open realtime bar, then *once per update* to recalculate the results for the bar using the latest data. A strategy script executes only on the bar's *closing tick* by default, but users can modify its calculation behavior to allow executions while the bar is open.
- Before each new script execution on an open bar, the runtime system executes a *rollback* process, which *reverts* all applicable variables, expressions, and objects to their *last committed states* as of the previous bar's close.
- After the script executes on an *elapsed* realtime bar's closing tick, the system *commits* necessary data from that execution to the time series for access on later bars. It does **not** commit the data from executions on the bar's *unconfirmed* values from previous ticks.

Events that trigger script executions

Several events cause a script to load and execute across all the available bars in a dataset. The specific events that trigger the loading process depend on where the script runs.

For a script on the chart, the following events always cause the script to load and perform *new executions* on every bar:

- The user adds the script to the chart for the first time from the Pine Editor or the "Indicators, metrics, and strategies" menu.
- The user saves an update to the script while it is active on the chart.
- The chart is refreshed while the script is active.

Other events also trigger the loading process for a script on the chart. However, these events do not *always* cause new script executions on past bars. The results from running a script with a unique combination of settings are often temporarily *cached*. If cached data exists for a selected combination of settings, the system loads the script using that data. See the Caching section for more information.

Below are the additional events that cause a script to load on the chart, either by performing new executions across the dataset or by using available cached data:

- The user selects new values for the inputs or strategy properties in the script's "Settings" menu.
- The script uses the chart.left_visible_bar_time or chart.right_visible_bar_time variable, and the visible chart range changes.
- The script uses the chart.fg_color or chart.bg_color variable, and the user changes the chart's background color.
- The chart loads a new dataset with a different *timeframe* or *ticker identifier*. Several user actions affect a chart's ticker ID, such as selecting a symbol from the "Symbol Search" menu, changing the chart type, toggling data modifications in the chart's settings, and activating Bar Replay mode.
- The user opens or closes the Pine Logs pane.
- The user activates or deactivates the Pine Profiler.

For scripts used in other locations, the following events trigger the loading process:

- The user creates a new script alert from the "Create Alert" dialog box.
- The user pauses and restarts an alert instance from the "Alerts" menu.
- The user selects a new testing range in the strategy report generated by a strategy script.

- The user clicks the “Scan” button in the Pine Screener to run the script on the datasets from a chosen watchlist.

After a script loads, either of the following causes new script executions on an *open bar*:

- One of the events above causes the script to load again and execute across the *entire dataset* up to the bar.
- The script runs on the chart or in an alert, and the bar updates after new data becomes available. The system performs *rollback* and re-executes the script on that bar using the latest data. The only exception is if the script is a strategy that does not allow recalculation on the new tick.

When a script completely reloads on the chart or in an alert after an applicable event, all the elapsed realtime bars from the script’s previous run become historical bars in the new run, because they represent *confirmed* data points that the script accesses from *a different data feed* as it loads.

The bars in a symbol’s dataset come from two distinct data feeds: the *historical* feed and the *realtime* feed. The historical feed reports only the *final* values for each bar, whereas the realtime feed includes the *temporary* values from all available ticks. When a realtime bar becomes historical after a script restarts, the values from the bar’s previous ticks are no longer accessible; only the **final** price, volume, and other values remain. Therefore, if a script relies on temporary data from realtime bars in its calculations, it might behave differently after reloading.

For example, the following script calculates the one-bar arithmetic return of the close series and displays the result as a line plot. On each realtime bar, the script updates three variables declared with varip to track the first, highest, and lowest return values calculated during executions across the bar’s ticks, then calls plotcandle() to plot a candle showing the values. Additionally, it uses bgcolor() to highlight the background of realtime bars in orange:

//@version=6 indicator("Reloading a script demo", precision = 5) //@variable The one-bar arithmetic return of the `close` series. float priceReturn = ta.change(close, 1) / close[1] // Declare persistent variables to track the first, highest, and lowest `priceReturn` values across ticks in // each realtime bar. varip float o = na varip float h = na varip float l = na if barstate.isrealtime // On the first tick in the realtime bar, reassign `o`, `h`, and `l` to hold the value of `priceReturn`. if barstate.isnew o := priceReturn h := priceReturn l := priceReturn // Otherwise, reassign `h` and `l` to the bar's highest and lowest `priceReturn` value as of the current tick. else h := math.max(h, priceReturn) l := math.min(l, priceReturn) // Plot candles to display the `o`, `h`, `l`, and `priceReturn` values for each realtime bar. // The candles do not appear on historical bars, because `o`, `h`, and `l` are `na` on those bars. plotcandle(o, h, l, priceReturn, "Return candles", color.blue, chart.fg_color, bordercolor = chart.fg_color) // Display the `priceReturn` series as a purple line plot. plot(priceReturn, "Return plot", color.purple, 3) // Highlight the background of all realtime bars in orange. bgcolor(barstate.isrealtime ? color.new(color.orange, 70) : na, title = "Realtime highlight")

After the script loads on the chart and executes on several realtime bars, all the elapsed realtime bars, as well as the open realtime bar, include plotted return candles and an orange background color:

After an applicable event, such as a chart refresh, the script *reloads* and executes across the dataset again. All the closed bars with a realtime state in the previous run become *historical* bars in the new run. The results thus change because our script relies on realtime data. As shown below, the script does not display candles or background colors for previous bars after we refresh the chart. Those outputs appear only for the latest bar, after new ticks become available, because that bar is now the **only** one with a realtime state:

Note that:

- The barstate.isnew variable has a value of `true` when a realtime bar opens, and `false` on all subsequent updates to the bar. If the script reloads midway through a realtime bar's progression, only the background color appears on that bar. The script does not show a candle on the first realtime bar in that case, because its `o`, `h`, and `l` variables hold na until the first time that barstate.isnew is `true`.

> Note
>
> A script might also behave differently after reloading due to differences between bars constructed from realtime data and those retrieved from historical data feeds. Occasionally, a data provider must adjust the values of a closed bar originally built from realtime updates. In such cases, the script accesses the adjusted values only after *restarting* and retrieving the bar from the *historical* feed. Although such adjustments are typically minor and infrequent, they can cause slight variations in a script's calculations on former realtime bars.
>
> Refer to the Repainting page to learn more about potential differences between historical and realtime script behaviors and their causes.

Caching

When a script runs on a chart for the *first time* using a *unique configuration*, the data from that run is often temporarily cached for reuse. The cached data is erased after the chart is refreshed or the user updates the script's source code.

In this context, the configuration refers to the combined state of all script, chart, and developer tool settings that can affect the script's executions. This combination includes:

- The values of inputs in the script's "Settings/Inputs" tab.
- The values of the strategy properties in the "Settings/Properties" tab.
- The values of the `chart.*` variables whose qualifiers are "input" (chart.left_visible_bar_time, chart.right_visible_bar_time, chart.fg_color, and chart.bg_color).
- The chart's timeframe.
- The chart's ticker identifier.
- Whether the Pine Logs pane is open or closed.
- Whether the Pine Profiler is active or not.

Each time that a script runs using a unique combination of settings, it executes from start to end on each bar in the dataset to perform new calculations. If possible, the script's data from the run is then cached. If cached data is available on past bars for a selected combination of settings, the runtime system loads the script using that data.

This behavior enables users to change a script's inputs, alter the chart, and toggle developer tools without losing information – including bar states – from previous script runs using different settings. Additionally, caching helps reduce loading times and resource requirements when switching between settings or adding multiple instances of the same script to the chart.

To understand this behavior, let's revisit the script from the previous section. The script has different behaviors on historical and realtime bars. In the version below, we've added a `lengthInput` variable that holds the value from an input.int() call. The script uses this variable to define the length of the ta.change() calculation and the offset of the history-referencing operator:

//@version=6 indicator("Caching demo", precision = 5) //@variable The bar span of the `priceReturn` calculation. int lengthInput = input.int(5, "Length", 1) //@variable The arithmetic return of the `close` series across `lengthInput` bars. float priceReturn = ta.change(close, lengthInput) / close[lengthInput] // Declare persistent variables to track the first, highest, and lowest `priceReturn` values across ticks in // each realtime bar. varip float o = na varip float h = na varip float l = na if barstate.isrealtime // On the first tick in the realtime bar, reassign `o`, `h`, and `l` to hold the value of `priceReturn`. if barstate.isnew o := priceReturn h := priceReturn l := priceReturn // Otherwise, reassign `h` and `l` to the bar's highest and lowest `priceReturn` value as of the current tick. else h := math.max(h, priceReturn) l := math.min(l, priceReturn) // Plot candles to display the `o`, `h`, `l`, and `priceReturn` values for each realtime bar. // The candles do not appear on historical bars, because `o`, `h`, and `l` are `na` on those bars. plotcandle(o, h, l, priceReturn, "Return candles", color.blue, chart.fg_color, bordercolor = chart.fg_color) // Dispaly the `priceReturn` series as a purple line plot. plot(priceReturn, "Return plot", color.purple, 3) // Highlight the background of all realtime bars in orange. bgcolor(barstate.isrealtime ? color.new(color.orange, 70) : na, title = "Realtime highlight")

After we add the script to our 1m chart and let it run for a few minutes with a "Length" input value of 5, the script plots candles and highlights the background for the latest few bars, because barstate.isrealtime is `true` on those bars:

Let's change the "Length" input to a new value, causing the script to reload and execute across the dataset again. Here, we changed the value from 5 to 10 and let the script execute on some new ticks. The script no longer displays candles and background colors for the same bars after restarting, because it now accesses the data for those formerly realtime bars from the *historical* data feed:

As shown above, the realtime bar information from the first run is *not available* when we change the script's input to a new value. However, the data from that previous run still exists in memory. If we revert the "Length" input's value to 5, the candle plot and background colors start on the same bar as the first run:

If we add a second instance of the script to the chart, using the same settings, the runtime system loads the new instance using the cached data instead of executing it entirely from scratch. As such, its outputs are *identical* to those from the first script instance, even though we added it to the chart a few bars later:

Similarly, cached data usually remains available even if we remove the script from our chart and add it again.

> TipYou can clear cached data for a script on the chart at any time by simply reloading the chart.

Time series

A symbol's dataset is a form of *time series* – a sequence of collected values indexed by time. Each bar represents a distinct data point, anchored to a specific time, that contains price and volume data for a particular period. This data format thus shows how a symbol's values progress across time in successive periodic steps.

Pine Script's internal time series structure follows a similar format. After executing a script on a closed bar's confirmed values, the runtime system *commits (saves)* the results of the script's statements and expressions to internal time series for later use. Each bar with committed data has an assigned *index* in the series, where 0 represents the first bar, 1 represents the second, and so on. Scripts can retrieve this index with the bar_index variable.

Scripts can access the data committed to the time series on past bars by using the [] history-referencing operator. The value between the operator's square brackets specifies the position of the referenced bar in the time series as a *relative offset* behind the current bar. For variables and expressions in the global scope, an offset value of 1 refers to the previous bar at `bar_index - 1` (one bar back), a value of 2 refers to the bar at `bar_index - 2` (two bars back), and so on. An offset of 0 always refers to the *current bar*.

For example, consider the open variable, which holds the opening price of the current bar on which the script executes. Before each script execution on a new bar, the runtime system commits the open value from the last execution on the previous bar. Then, it updates the variable to hold the current bar's opening price. To access the committed open value for the previous bar, we can use the expression `open[1]`. To access the committed value from 10 bars back, we use `open[10]`.

The script below performs three history-referencing operations to retrieve the current bar's open value, the value from one bar back, and the value from a user-specified number of bars back. Then, it plots the retrieved values on the chart for comparison:

//@version=6 indicator("History referencing demo", overlay = true, behind_chart = false) //@variable The number of bars back from which to retrieve the `open` price for `pastOpen`. int offsetInput = input.int(10, "Bar offset", 0) //@variable The current bar's opening price. `open[0]` is equivalent to using `open` without the `[]` operator. float currOpen = open[0] //@variable The last committed `open` value. Represents the previous bar's value, or `na` if no previous bar exists. float prevOpen = open[1] //@variable The `open` value committed `offsetInput` bars back, or `na` if no bar exists at that offset. float pastOpen = open[offsetInput] // Plot `currOpen`, `prevOpen`, and `pastOpen` for comparison. plot(currOpen, "Current `open`", color.blue, 2) plot(prevOpen, "Previous bar `open`", color.purple, 3) plot(pastOpen, "Past `open` from custom offset", color.orange, 4)

Note that:

- The expression `open[0]` is equivalent to using open without the history-referencing operator, because an offset of 0 refers to the current bar.
- At the beginning of the chart's dataset, the expressions `open[1]` and `open[offsetInput]` return na because they refer to previous bars that are unavailable.
- Each history-referencing expression also leaves a trail of values in the time series. Therefore, it is possible to retrieve past states of the expression using another history-referencing operation, e.g., `(open[offsetInput])[1]`.
- Internally, the system maintains a *limited amount* of time series data for variables and expressions in fixed-length *historical buffers*. These buffers define the *maximum offsets* allowed for history-referencing operations. See the next section, Historical buffers, to learn more.

Another way that scripts use committed values from a time series is by calling the built-in functions that reference history internally, such as those in the `ta.*` namespace. For example, the expression `ta.highest(high, 20)` calculates the highest value from the high series over a 20-bar window. It compares the series' current value to the committed values from the previous 19 bars to determine the result. The script below executes this call on each bar and plots the resulting series on the chart. Additionally, the script

colors the background of the last 20 bars on the chart to highlight the bars used in the latest execution's ta.highest() call:

//@version=6 indicator("History referencing in functions demo", overlay = true, behind_chart = false) //@variable The highest value from the `high` series across the 20 most recent bars. // The `ta.highest()` call compares the current `high` to the last 19 committed values. float highest = ta.highest(high, 20) // Plot the `highest` series on the chart. plot(highest, "20-bar high", color.purple, 3) // Color the background of the last 20 bars, i.e., the bars used by the latest execution's `ta.highest()` call. bgcolor(color.new(color.blue, 70), show_last = 20, title = "Last 20 bar highlight")

Note that:

- The first 19 bars of the chart have a plotted value of na, because the ta.highest() function call requires the high values from the current bar and 19 previous bars to calculate the result.
- All function calls and expressions that do not return "void" leave historical trails in the time series, just like variables. Therefore, scripts can use an expression such as `ta.highest(high, 20)[10]` to retrieve the 20-bar high from 10 bars back.
- The ta.highest() function and other functions that access past values from a time series must execute in the **global scope** for consistent calculations. Time series storage for variables and expressions in local scopes works *differently* than that for global values. See the Time series in scopes section for more information.

> Note
>
> Programmers should not confuse time series with the "series" qualifier. The time series concept describes how Pine's runtime system stores and retrieves data across successive script executions. In contrast, the "series" qualifier describes variables and expressions whose values can change from bar to bar, such as open.
>
> To understand this distinction, consider the timeframe.period variable, which is of the "simple string" *qualified type*. The variable's value cannot change because its *qualifier* is "simple", but it still leaves a trail of successive values in the time series. It is possible, though not very useful, to retrieve the value from 10 bars back using an expression such as `timeframe.period[10]`. The returned value equals the timeframe.period value for all bars with a bar_index of 10 and above. However, the expression's result is "series string", because the expression returns a *different* value (na) on the first 10 bars.
>
> See the Qualifiers section of the Type system page to learn more about "series" and other qualifiers.

Historical buffers

To promote efficiency and help ensure computing resources remain available for all users, the Pine Script runtime system uses fixed-length *historical buffers* to maintain a *limited amount* of time series data for all variables and expressions. These historical buffers define the *maximum* number of committed data points

that a script can access on any bar via the history-referencing operator or the built-in functions that reference past bars internally.

For most series, the underlying historical buffer can contain data from up to **5000** past bars. The only exception is for some built-in series such as open, close, and time, whose buffers can store data for *more* than 5000 bars.

Although these buffers can contain thousands of data points at their maximum size, a script might not *require* that much past data for its calculations on any bar. Therefore, the runtime system automatically optimizes the size of each series' historical buffer based on the historical references that the script performs as it loads on the dataset. Each resulting buffer contains *only* the amount of past data required by the script's calculations and *not more*.

For instance, if the maximum number of bars back for which a script references the value of a variable on historical bars is 500, the system maintains a historical buffer that includes only the latest 500 committed values of that variable. The buffer does not store 5000 committed values, because the script *does not* require all that extra data. This behavior thus helps to minimize a script's resource requirements while preserving the integrity of its calculations.

To determine the sufficient buffer size for each variable and expression in a script, the runtime system performs the following process during the script's loading time:

1. It analyzes all the historical references that occur while executing the script on the dataset's first **244 bars**, then sets the initial size of each buffer to the minimum size that accommodates those references.
2. While executing the script on subsequent bars, it checks if the script attempts to access data from previous bars that are beyond the limits of the defined buffers. If the script's historical references exceed the buffer limits on any bar, the system restarts the loading process and tries a larger buffer size.
3. In the rare case that a historical buffer's size remains insufficient after several calculation attempts, the system stops the script and raises a runtime error.

It's crucial to emphasize that the runtime system defines the sizes of all historical buffers only while executing a script on *historical bars*. It does **not** adjust any historical buffers during executions on *new bars* from the realtime data feed. If a script references past data from beyond a historical buffer's limits while executing on a realtime bar, it causes a runtime error.

For example, the script below retrieves a past value from the close series using the history-referencing operator with an offset of 100 bars back on historical bars and 150 bars back on realtime bars. Because the script references data from 100 bars back during all historical executions, the system sets the close buffer's size to include only 100 past values. Consequently, an error occurs when the script executes on the open realtime bar, because a historical offset of 150 is *beyond* the buffer's limit:

//@version=6 indicator("Max bars back error demo", overlay = true) // @variable The historical offset for retrieving past values from the `close` series. // If the bar is historical, the offset is 100. Otherwise, the offset is 150. int offset = barstate.ishistory ? 100 : 150 // @variable The value of `close` from `offset` bars back. // This code causes a *runtime error* on a realtime bar. During all code executions on historical bars, // the script requires only the latest 100 past values of `close`, so the system sets the buffer size to // include only the past 100 values. The offset of 150 is thus *out of bounds*. float pastClose = close[offset] // Plot the

`pastClose` series. plot(pastClose, "Past `close`", chart.fg_color, 3) // Highlight the background of all realtime bars in orange. bgcolor(barstate.isrealtime ? color.new(color.orange, 70) : na, title = "Realtime highlight")

For cases like these, programmers can *manually* set the size of a historical buffer to ensure it contains a sufficient amount of data by doing any of the following:

- Modify the script to reference the maximum required number of bars back with the [] operator during its execution on the *first bar*.
- Call the max_bars_back() function to explicitly set the historical buffer size for a *specific* series.
- Include a `max_bars_back` argument in the indicator() or strategy() declaration statement to set the initial size of *all* historical buffers.

> NoticeThe larger the size of a historical buffer, the more memory resources it requires. A script with buffers that are too large can cause the "Memory limits exceeded" error. Therefore, when manually setting the buffer size for a series, use the **smallest** possible size that accommodates *all* required historical references to that series.

Below, we modified the script by including the expression `max_bars_back(close, 150)`, which sets the size of the close buffer to include 150 past values. With the appropriate buffer size manually defined, the script's history-referencing operation no longer causes an error on realtime bars:

//@version=6 indicator("Manual buffer sizing demo", overlay = true) // @variable The historical offset for retrieving past values from the `close` series. // If the bar is historical, the offset is 100. Otherwise, the offset is 150. int offset = barstate.ishistory ? 100 : 150 // Set the size of the `close` historical buffer to include 150 past values, ensuring the script has exactly // the amount of history that it requires on realtime bars. max_bars_back(close, 150) // @variable The value of `close` from `offset` bars back. // This code no longer causes an error when it executes on a realtime bar, because the historical // buffer has an appropriate size defined in advance. float pastClose = close[offset] // Plot the `pastClose` series. plot(pastClose, "Past `close`", chart.fg_color, 3) // Highlight the background of all realtime bars in orange. bgcolor(barstate.isrealtime ? color.new(color.orange, 70) : na, title = "Realtime highlight")

> TipManually setting historical buffer sizes can also improve a script's resource efficiency in some cases. As explained above, the runtime system *restarts* a script to recalculate its buffers if any historical reference exceeds a buffer's limit after the first **244 bars**. This process increases the script's loading time and memory use. Setting the appropriate buffer size in advance with max_bars_back() prevents the script from restarting for buffer calculations. See the Minimizing historical buffer calculations section of the Profiling and optimization page for more information.

Time series in scopes

The *scope* of a variable or expression refers to the part of the script where it is defined and accessible in the code. Every script has one *global* scope and zero or more *local* scopes.

All variables and expressions in a script that are outside user-defined functions or methods, conditional structures, loops, and user-defined type or enum type declarations belong to the *global scope*. The script

evaluates variables and expressions from this scope *once* for *every execution* across bars and ticks in the dataset.

All functions, methods, conditional structures, and loops create their own *local scopes*. The variables and expressions defined within a local scope belong exclusively to that scope. In contrast to the global scope, a script does *not* always evaluate a local scope once per execution; the script might evaluate the scope *zero*, *one*, or *several* times per execution, depending on its logic.

For the runtime system to commit data from a variable or expression and queue that data into a historical buffer on any bar, a script must evaluate the scope of that code *once* when it executes on the bar's *closing tick*. If the script does not evaluate the scope, the runtime system *cannot* add new data to the historical buffer. Similarly, if the script evaluates the scope repeatedly within a loop, the historical buffer cannot store series data for *each* separate iteration, because each entry in the time series corresponds to a single bar.

Therefore, time series behave differently in global and local scopes: the historical buffers for global variables and expressions *always* contain committed data for *consecutive* past bars, whereas the buffers for local variables often contain an **inconsistent** history of committed data.

> NoteFunction and method *parameters* have the same historical buffer behaviors as *local variables*. Each parameter in a function call has a *unique buffer*, and the system can add new data to that buffer only on a bar's closing tick. If the script does not evaluate the function call on every closing tick, the buffers for its parameters contain an inconsistent history. This behavior applies even if the argument supplied to a parameter is a *global variable*.

If a script references the history of a global variable using an expression such as `myVariable[1]`, the historical offset of 1 always refers to the confirmed `myVariable` value from the *previous bar*. Likewise, an expression such as `myVariable[9]` retrieves the variable's value from *nine bars back*. Either expression consistently accesses the value corresponding to a *specific* number of bars back, because the runtime system commits a new value for that global variable on *every bar*.

In contrast, the result of using the [] operator on a *local* variable **does not** represent the value from a specific number of bars back. Instead, it represents the variable's *last committed value* as of the bar at the specified offset. For instance, suppose `myVariable` is a local variable, and the script last evaluated the variable's scope 10 bars before the current bar. A history-referencing operation with any offset from *1 to 9* on that variable – such as `myVariable[1]`, `myVariable[5]`, or `myVariable[9]` – returns the variable's value from **10 bars back**, because there is not a recent committed value after that point for the operator to access. This behavior often leads to *unintended results*. Therefore, to ensure consistency, we recommend using historical references only on variables or expressions that the script evaluates on *every bar*.

The following example demonstrates how a simple history-referencing operation behaves inside a user-defined function's scope when a script does *not* call the function on every bar. The script below defines a custom `upDownColor()` function, which compares the current value of its `source` parameter to the last committed value (`source[1]`) on each call. The function returns color.blue if the current `source` value is higher than the previous value. Otherwise, it returns color.orange.

The script uses this function *conditionally*, inside a ternary operation, to determine the color of a plot that shows the remainder from dividing bar_index by a specified value. If the `remainder` variable's value is nonzero, the operation calls `upDownColor(remainder)` to calculate the color (blue or orange). If the value is 0,

the operation does *not* use the call and instead returns color.gray. The `remainder` value *increases* on each bar, except for when it returns to 0 – causing the gray color. Therefore, a user might expect the plot's color to be only blue or gray on every bar. However, the color changes to *orange* on each bar after the one where the color is gray, even though the `remainder` value on that bar is *higher* than the value on the previous bar:

//@version=6 indicator("Local historical references demo") //@function Returns `color.blue` if `source` is above its last committed value; `color.orange` otherwise. // For consistent results, this function should execute on *every bar*, because it uses the // history-referencing operator on the `source` series. // // Even if the argument supplied to `source` comes from a global variable, the `source` parameter remains // part of the function's *local scope*. The system maintains a *separate historical buffer* for the `source` // series in each function call instance. The buffer contains only the committed `source` values from the bars // where the function call occurs. If the call does not occur on a bar, the buffer for `source` contains // **no new data** for that bar. upDownColor(float source) => source > source[1] ? color.blue : color.orange //@variable The value by which to divide the `bar_index` value. int divisorInput = input.int(5, "Divisor", 1) //@variable The remainder of dividing `bar_index` by `divisorInput`. float remainder = bar_index % divisorInput //@variable Is `color.gray` if `remainder` equals 0, and the result of `upDownColor(remainder)` otherwise. // The `upDownColor()` call does not execute on every bar. Therefore, it does *not* always compare the // `remainder` value from one bar back to calculate the color. Instead, the function compares the current // `remainder` to the value from the last bar where `remainder` is nonzero. color plotColor = remainder == 0 ? color.gray : upDownColor(remainder) // Plot the `remainder` series and color it using `plotColor`. The plot is orange after each bar where `remainder == 0`, // because the `upDownColor()` function call does not have data for that bar to use in its logic. plot(remainder, "Remainder", plotColor, 5)

The script behaves this way because `upDownColor()` uses the history-referencing operator on the `source` series, which is *local* to the function's scope, and the script does not call the function on *every* execution. When the value of `remainder` is zero, the *first* expression in the ternary condition evaluates to `true`, and therefore the second branch of the ternary expression, which contains the function call, does *not* execute.

The compiler issues the following warning about the function directly in the Pine Editor:

```
The function `upDownColor()` should be called on each calculation for consisten
```

The runtime system maintains a separate historical buffer for the local `source` series, but it cannot update that buffer unless the script *calls* the function. On each bar where `remainder` is 0, the call does not occur, and the system has no new value to commit to the time series. Therefore, the `source` buffer does *not* contain a value for that bar. On the bar that follows, the local expression `source[1]` refers to the `source` value from the *last bar* where the `upDownColor()` call occured – *two bars back* – and **not** the value of `remainder` from the previous bar. Because the value from two bars back is *higher* than the current value, the returned color is color.orange instead of color.blue.

We can fix this script's behavior by following the instructions in the compiler warning. Below, we modified the script by moving the `upDownColor()` call *outside* the ternary expression, enabling the script to execute it on *every bar*. The historical buffer for the function's `source` series now contains `remainder` values from *consecutive* bars. With this change, an orange color does not appear because the function consistently compares values from *one* bar back:

```
//@version=6 indicator("Consistent historical references demo") //@function Returns `color.blue` if `source` is above its last committed value; `color.orange` otherwise. // For consistent results, this function should execute on *every bar*, because it uses the // history-referencing operator on the `source` series. upDownColor(float source) => source > source[1] ? color.blue : color.orange //@variable The value by which to divide the `bar_index` value. int divisorInput = input.int(5, "Divisor", 1) //@variable The remainder of dividing `bar_index` by `divisorInput`. float remainder = bar_index % divisorInput //@variable Is `color.blue` if `remainder` is above its previous value, and `color.orange` otherwise. color secondColor = upDownColor(remainder) //@variable Is `color.gray` if `remainder` equals 0, and `secondColor` otherwise. // This ternary operation no longer causes a warning. The scope of the `upDownColor()` call executes on // every bar, meaning its historical buffer consistently includes data for consecutive past bars. color plotColor = remainder == 0 ? color.gray : secondColor // Plot the `remainder` series and color it using `plotColor`. The plot is now blue or gray, but never orange. plot(remainder, "Remainder", plotColor, 5)
```

A similar behavior applies to all built-in functions that reference past values internally, such as those in the `ta` namespace. For example, the ta.sma() function uses the current value of a `source` series and `length - 1` past values from that series to calculate a moving average. If a script calls this function only on *some* bars instead of on *every* bar, the historical buffer for `source` does not contain values for consecutive past bars. Therefore, such a call can cause unintended results, because the call calculates the returned average using an inconsistent history of previous values.

The script below demonstrates how the results of the ta.sma() function can vary with the scope in which the function call occurs. The script declares three global variables to hold calculated SMA values: `controlSMA`, `localSMA`, and `globalSMA`. The script initializes `controlSMA` using the result of a ta.sma() function call, and it initializes the other two variables with na. Within the if structure, the script updates the value of `globalSMA` using `controlSMA`, and it updates `localSMA` using the result of another ta.sma() call with the same arguments as the first call.

As shown below, the `controlSMA` and `globalSMA` variables have the same value. Both hold the result of the *global* ta.sma() call, which executes on *every bar*. The internal historical buffer for `source` in that call thus includes committed values for consecutive past bars. In contrast, the `localSMA` value differs, because the ta.sma() call for that variable does *not* execute on every bar. The buffer for that call's local `source` series contains only the values from bars with an *even* bar_index value:

```
//@version=6 indicator("`ta.*()` functions in scopes demo", overlay = true, behind_chart = false) //@variable Is `true` if the `bar_index` is divisible by 2, and `false` otherwise. bool condition = bar_index % 2 == 0 //@variable The 20-bar moving average of `close` values. // This `ta.sma()` call executes in the global scope, so the script evaluates it on every bar. float controlSMA = ta.sma(close, 20) // Declare two additional variables to modify later within the `if` structure's scope. float globalSMA = na float localSMA = na if condition // Assign the `controlSMA` value to `globalSMA`. This code does not cause a warning. globalSMA := controlSMA // Call `ta.sma()` with the same arguments as before within this block and assign the result to `localSMA`. // The function call causes a warning, because it does not execute in the global scope. // The historical buffers for this `ta.sma()` call contain data only for the bars where `condition` is `true`, // thus leading to a *different* result. localSMA := ta.sma(close, 20) // Plot `controlSMA`, `globalSMA`, and `localSMA` for comparison. plot(controlSMA, "Control SMA", color.blue, 2) plot(globalSMA, "Global SMA", color.purple, 3, style = plot.style_circles) plot(localSMA, "Local SMA", color.gray, 3, style = plot.style_circles)
```

To summarize the behavior of time series in a script's scopes:

- A script evaluates its global scope once on *every execution*. After each script execution on a bar's closing tick, the system commits the data for variables and expressions in the global scope and updates their historical buffers. The resulting buffers thus include data for consecutive past bars, ensuring consistent results for operations and functions that rely on past data.
- A script evaluates local scopes *zero*, *one*, or *several* times per execution. The runtime system **cannot** maintain consistent bar-by-bar historical buffers for scopes that a script does *not* evaluate on every bar, or for scopes that the script evaluates *more than once* on a bar's closing tick. Therefore, using the [] operator on local variables and expressions, or not calling functions that access past data once on each closing tick, can cause **unintended results**.

> Note
>
> Not all built-in functions use past data from historical buffers in their calculations. For example, most functions in the `math` namespace, excluding math.sum(), operate only on the *current* arguments of the function call. Functions that do not interact with their history in any way do not require special treatment to ensure intended results.
>
> If the use of a function call within a local block does *not* cause a compiler warning, it is typically safe to use in that block without affecting the result. However, if the warning occurs, it is usually best to move the call to the *global scope* – outside the operands of the or, and, or ?: operators – to ensure consistent calculations.

Identifiers

Identifiers are names used for user-defined variables and functions:

- They must begin with an uppercase (`A–Z`) or lowercase (`a–z`) letter, or an underscore (_).
- The next characters can be letters, underscores or digits (`0–9`).
- They are case-sensitive.

Here are some examples:

myVar _myVar my123Var functionName MAX_LEN max_len maxLen 3barsDown // NOT VALID!

The Pine Script® Style Guide recommends using uppercase SNAKE_CASE for constants, and camelCase for other identifiers:

GREEN_COLOR = #4CAF50 MAX_LOOKBACK = 100 int fastLength = 7 // Returns 1 if the argument is `true`, 0 if it is `false` or `na`. zeroOne(boolValue) => boolValue ? 1 : 0

Loops

Introduction

Loops are structures that repeatedly execute a block of statements based on specified criteria. They allow scripts to perform repetitive tasks without requiring duplicated lines of code. Pine Script® features three distinct loop types: for, while, and for…in.

Every loop structure in Pine Script consists of two main parts: a *loop header* and a *loop body*. The loop header determines the criteria under which the loop executes. The loop body is the indented block of code (local block) that the script executes on each loop cycle (*iteration*) as long as the header's conditions remain valid. See the Common characteristics section to learn more.

Understanding when and how to use loops is essential for making the most of the power of Pine Script. Inefficient or unnecessary usage of loops can lead to suboptimal runtime performance. However, effectively using loops when necessary enables scripts to perform a wide range of calculations that would otherwise be impractical or impossible without them.

When loops are unnecessary

Pine's execution model and time series structure make loops *unnecessary* in many situations.

When a user adds a Pine script to a chart, it runs within the equivalent of a *large loop*, executing its code once on *every* historical bar and realtime tick in the available data. Scripts can access the values from the executions on previous bars with the history-referencing operator, and calculated values can *persist* across executions when assigned to variables declared with the var or varip keywords. These capabilities enable scripts to utilize bar-by-bar calculations to accomplish various tasks instead of relying on explicit loops.

In addition, several built-ins, such as those in the `ta.*` namespace, are internally optimized to eliminate the need to use loops for various calculations.

Let's consider a simple example demonstrating unnecessary loop usage in Pine Script. To calculate the average close over a specified number of bars, newcomers to Pine may write a code like the following, which uses a for loop to calculate the sum of historical values over `lengthInput` bars and divides the result by the `lengthInput`:

//@version=6 indicator("Unnecessary loops demo", overlay = true) //@variable The number of bars in the calculation window. int lengthInput = input.int(defval = 20, title = "Length") //@variable The sum of `close` values over `lengthInput` bars. float closeSum = 0 // Loop over the most recent `lengthInput` bars, adding each bar's `close` to the `closeSum`. for i = 0 to lengthInput - 1 closeSum += close[i] //@variable The average `close` value over `lengthInput` bars. float avgClose = closeSum / lengthInput // Plot the `avgClose`. plot(avgClose, "Average close", color.orange, 2)

Using a for loop is an **unnecessary**, inefficient way to accomplish tasks like this in Pine. There are several ways to utilize the execution model and the available built-ins to eliminate this loop. Below, we replaced these calculations with a simple call to the ta.sma() function. This code is shorter, and it achieves the same result much more efficiently:

//@version=6 indicator("Unnecessary loops corrected demo", overlay = true) //@variable The number of bars in the calculation window. int lengthInput = input.int(defval = 20, title = "Length") //@variable The average `close` value over `lengthInput` bars. float avgClose = ta.sma(close, lengthInput) // Plot the `avgClose`. plot(avgClose, "Average close", color.blue, 2)

Note that:

- Users can see the substantial difference in efficiency between these two example scripts by analyzing their performance with the Pine Profiler.

When loops are necessary

Although Pine's execution model, time series, and available built-ins often eliminate the need for loops in many cases, not all iterative tasks have loop-free alternatives. Loops *are necessary* for several types of tasks, including:

- Iterating through or manipulating collections (arrays, matrices, and maps)
- Performing calculations that one **cannot** accomplish with loop-free expressions or the available built-ins
- Looking back through history to analyze past bars with a reference value only available on the *current bar*

For example, a loop is *necessary* to identify which past bars' high values are above the current bar's high because the current value is **not** obtainable during a script's executions on previous bars. The script can only access the current bar's value while it executes on that bar, and it must *look back* through the historical series during that execution to compare the previous values.

The script below uses a for loop to compare the high values of `lengthInput` previous bars with the last historical bar's high. Within the loop, it calls label.new() to draw a circular label above each past bar that has a high value exceeding that of the last historical bar:

//@version=6 indicator("Necessary loop demo", overlay = true, max_labels_count = 500) //@variable The number of previous `high` values to compare to the last historical bar's `high`. int lengthInput = input.int(20, "Length", 1, 500) if barstate.islastconfirmedhistory // Draw a horizontal line segment at the last historical bar's `high` to visualize the level. line.new(bar_index - lengthInput, high, bar_index, high, color = color.gray, style = line.style_dashed, width = 2) // Create a `for` loop that counts from 1 to `lengthInput`. for i = 1 to lengthInput // Draw a circular `label` above the bar from `i` bars ago if that bar's `high` is above the current `high`. if high[i] > high label.new(bar_index - i, na, "", yloc = yloc.abovebar, color = color.purple, style = label.style_circle, size = size.tiny) // Highlight the last historical bar. barcolor(barstate.islastconfirmedhistory ? color.orange : na, title = "Last historical bar highlight")

Note that:

- Each *iteration* of the for loop retrieves a previous bar's high with the history-referencing operator [], using the loop's *counter* (`i`) as the historical offset. The label.new() call also uses the counter to determine each label's x-coordinate.
- The indicator() declaration statement includes the argument `max_labels_count = 500`, meaning the script can show up to 500 labels on the chart.
- The script calls barcolor() to highlight the last historical chart bar, and it draws a horizontal line at that bar's high for visual reference.

Common characteristics

The for, while, and for…in loop statements all have similarities in their structure, syntax, and general behavior. Before we explore each specific loop type, let's familiarize ourselves with these characteristics.

Structure and syntax

In any loop statement, programmers define the criteria under which a script remains in a loop and performs *iterations*, where an iteration refers to *one execution* of the code within the loop's local block (*body*). These criteria are part of the *loop header*. A script evaluates the header's criteria *before* each iteration, only allowing new iterations to occur while they remain valid. When the header's criteria are no longer valid, the script *exits* the loop and skips over its body.

The specific header syntax varies with each loop statement (for, while, or for…in) because each uses *distinct* criteria to control its iterations. Effective use of loops entails choosing the structure with control criteria best suited for a script's required tasks. See the `for` loops, `while` loops, and `for…in` loops sections below for more information on each loop statement and its control criteria.

All loop statements in Pine Script follow the same general syntax:

```
[variables = | :=] loop_header
    statements | continue | break
    return_expression
```

Where:

- `loop_header` represents the loop structure's header statement, which defines the criteria that control the loop's iterations.
- `statements` represents the code statements and expressions within the loop's body, i.e., the *indented* block of code beneath the loop header. All code within the body belongs to the loop's local scope.
- `continue` and `break` are loop-specific *keywords* that control the flow of a loop's iterations. The `continue` keyword instructs the script to *skip* the remainder of the current loop iteration and *continue* to the next iteration. The `break` keyword prompts the script to *stop* the current iteration and *exit* the loop entirely. See this section below for more information.
- `return_expression` refers to the *last* code line or block within the loop's body. The loop returns the results from this code after the final iteration. If the loop skips parts of some iterations or stops prematurely due to a `continue` or `break` statement, the returned values or references are those of the latest iteration that evaluated this code. To use the loop's returned results, assign them to a variable or tuple.
- `variables` represents an optional variable or tuple to hold the values or references from the last evaluation of the `return_expression`. The script can assign the loop's returned results to variables only if the results are not void. If the loop's conditions prevent iteration, or if no iterations evaluate the `return_expression`, the variables' assigned values and references are na.

> Note

> If a script initializes a variable with a loop's result using the above syntax, and that variable declaration includes the var or varip keyword, the loop ends *immediately* after its **first** iteration, even if the header's criteria allow more iterations.
>
> To assign a loop's result to a var or varip variable while enabling the loop to perform all required iterations, programmers can do either of the following:
>
> - Declare the variable before the loop, then use the reassignment operator (`:=`) in the above syntax to *update* the variable.
> - Move the loop into a user-defined function, then initialize the variable with the result of a call to that function.

Scope

All code lines that a script executes within a loop must have an indentation of *four spaces* or a *tab* relative to the loop's header. The indented lines following the header define the loop's *body*. This code represents a *local block*, meaning that all the definitions within the body are accessible only during the loop's execution. In other words, the code within the loop's body is part of its *local scope*.

Scripts can modify and reassign most variables from *outer* scopes inside a loop. However, any variables declared within the loop's body strictly belong to that loop's local scope. A script **cannot** access a loop's declared variables *outside* its local block.

Note that:

- Variables declared within a loop's *header* are also part of the local scope. For instance, a script cannot use the *counter variable* in a for loop anywhere but within the loop's local block.

The body of any Pine loop statement can include conditional structures and *nested* loop statements. When a loop includes nested structures, each structure within the body maintains a *distinct* local scope. For example, variables declared within an *outer* loop's scope are accessible to an *inner* loop. However, any variables declared within the inner loop's scope are **not** accessible to the outer loop.

The simple example below demonstrates how a loop's local scope works. This script calls label.new() within a for loop on the last historical bar to draw labels above `lengthInput` past bars. The color of each label depends on the `labelColor` variable declared *within* the loop's local block, and each label's location depends on the loop counter (`i`):

//@version=6 indicator("Loop scope demo", overlay = true) //@variable The number of bars in the calculation. int lengthInput = input.int(20, "Lookback length", 1) if barstate.islastconfirmedhistory for i = 1 to lengthInput //@variable Has a value of `color.blue` if `close[i]` is above the current `close`, `color.orange` otherwise. // This variable is LOCAL to the `for` loop's scope. color labelColor = close[i] > close ? color.blue : color.orange // Display a colored `label` on the historical `high` from `i` bars back, using `labelColor` to set the color. label.new(bar_index - i, high[i], "", color = labelColor, size = size.normal)

In the above code, the `i` and `labelColor` variables are only accessible to the for loop's local scope. They are **not** usable within any outer scopes. Here, we added a label.new() call *after* the loop with `bar_index - i` as

the `x` argument and `labelColor` as the `color` argument. This code causes a *compilation error* because neither `i` nor `labelColor` are valid variables in the outer scope:

//@version=6 indicator("Loop scope demo", overlay = true) //@variable The number of bars in the calculation. int lengthInput = input.int(20, "Lookback length", 1) if barstate.islastconfirmedhistory for i = 1 to lengthInput //@variable Has a value of `color.blue` if `close[i]` is above the current `close`, `color.orange` otherwise. // This variable is LOCAL to the `for` loop's scope. color labelColor = close[i] > close ? color.blue : color.orange // Display a colored `label` on the historical `high` from `i` bars back, using `labelColor` to set the color. label.new(bar_index - i, high[i], "", color = labelColor, size = size.normal) // Call `label.new()` to using the `i` and `labelColor` variables outside the loop's local scope. // This code causes a compilation error because these variables are not accessible in this location. label.new(bar_index - i, low, "Scope test", textcolor = color.white, color = labelColor, style = label.style_label_up)

Keywords and return expressions

Every loop in Pine Script implicitly *returns* values, references, or void. A loop's returned results come from the *latest* execution of the *last* expression or nested structure within its body as of the final iteration. The results are usable only if they are not of the void type. Loops return na results for values or references when no iterations occur. Scripts can add a variable or tuple assignment to a loop statement to hold the returned results for use in additional calculations outside the loop's local scope.

The values or references that a loop returns usually come from evaluating the last written expression or nested code block on the *final* iteration. However, a loop's body can include `continue` and `break` keywords to control the flow of iterations beyond the criteria the loop header specifies, which can also affect the returned results. Programmers often include these keywords within conditional structures to control how iterations behave when certain conditions occur.

The `continue` keyword instructs a script to *skip* the remaining statements and expressions in the current loop iteration, re-evaluate the loop header's criteria, and proceed to the *next* iteration. The script *exits* the loop if the header's criteria do not allow another iteration.

The `break` keyword instructs a script to *stop* the loop entirely and immediately *exit* at that point without allowing any subsequent iterations. After breaking the loop, the script skips any remaining code within the loop's body and *does not* re-evaluate the header's criteria.

If a loop skips parts of iterations or stops prematurely due to a `continue` or `break` statement, it returns the values and references from the *last iteration* where the script *evaluated* the return expression. If the script did not evaluate the return expression across *any* of the loop's iterations, the loop returns na results for all non-void types.

The example below selectively displays numbers from an array within a label on the last historical bar. It uses a for…in loop to iterate through the array's elements and build a "string" to use as the displayed text. The loop's body contains an if statement that controls the flow of specific iterations. If the `number` in the current iteration is 8, the script immediately *exits* the loop using the `break` keyword. Otherwise, if the `number` is even, it *skips* the rest of the current iteration and moves to the next one using the `continue` keyword.

If neither of the if statement's conditions occur, the script evaluates the *last expression* within the loop's body (i.e., the return expression), which converts the current `number` to a "string" and concatenates the result with the `tempString` value. The loop returns the *last evaluated result* from this expression after termination. The

script assigns the returned value to the `finalLabelText` variable and uses that variable as the `text` argument in the label.new() call:

//@version=6 indicator("Loop keywords and variable assignment demo") //@variable An `array` of arbitrary "int" values to selectively convert to "string" and display in a `label`. var array<int> randomArray = array.from(1, 5, 2, -3, 14, 7, 9, 8, 15, 12) // Label creation logic. if barstate.islastconfirmedhistory //@variable A "string" containing representations of selected values from the `randomArray`. string tempString = "" //@variable The final text to display in the `label`. The `for..in` loop returns the result after it terminates. string finalLabelText = for number in randomArray // Stop the current iteration and exit the loop if the `number` from the `randomArray` is 8. if number == 8 break // Skip the rest of the current iteration and proceed to the next iteration if the `number` is even. else if number % 2 == 0 continue // Convert the `number` to a "string", append ", ", and concatenate the result with the current `tempString`. // This code represents the loop's return expression. tempString += str.tostring(number) + ", " // Display the value of the `finalLabelText` within a `label` on the current bar. label.new(bar_index, 0, finalLabelText, color = color.blue, textcolor = color.white, size = size.huge)

Note that:

- The label displays only *odd* numbers from the array because the script does not reassign the `tempString` when the loop iteration's `number` is even. However, it does not include the *last* odd number from the array (15) because the loop stops when `number == 8`, preventing iteration over the remaining `randomArray` elements.
- When the script exits the loop due to the `break` keyword, the loop's return value becomes the last evaluated result from the `tempString` reassignment expression. In this case, the last time that code executes is on the iteration where `number == 9`.

`for` loops

The for loop statement creates a *count-controlled* loop, which uses a *counter* variable to manage the iterative executions of its local code block. The counter starts at a predefined initial value, and the loop increments or decrements the counter by a fixed amount after each iteration. The loop stops its iterations after the counter reaches a specified final value.

Pine Script uses the following syntax to define a for loop:

```
[variables = | :=] for counter = from_num to to_num [by step_num]
    statements | continue | break
    return_expression
```

Where the following parts define the *loop header*:

- `counter` represents the counter variable, which can be any valid identifier. The loop increments or decrements this variable's value from the initial value (`from_num`) to the final value (`to_num`) by a fixed amount (`step_num`) after each iteration. The last possible iteration occurs when the variable's value reaches the `to_num` value.
- `from_num` is the `counter` variable's initial value on the first iteration.
- `to_num` is the *final* `counter` value for which the loop's header allows a new iteration. The loop adjusts the `counter` value by the `step_num` amount until it reaches or passes this value. If the script modifies

the `to_num` during a loop iteration, the loop header uses the new value to control the allowed subsequent iterations.
- `step_num` is a positive value representing the amount by which the `counter` value increases or decreases until it reaches or passes the `to_num` value. If the `from_num` value is greater than the *initial* `to_num` value, the loop *subtracts* this amount from the `counter` value after each iteration. Otherwise, the loop *adds* this amount after each iteration. The default is 1.

Refer to the Common characteristics section above for detailed information about the `variables`, `statements`, `continue`, `break`, and `return_expression` parts of the loop's syntax.

This simple script demonstrates a for loop that draws several labels at future bar indices during its execution on the last historical chart bar. The loop's counter starts at 0, then increases by 1 until it reaches a value of 10, at which point the final iteration occurs:

//@version=6 indicator("Simple `for` loop demo") if barstate.islastconfirmedhistory // Define a `for` loop that iterates from `i == 0` to `i == 10` by 1 (11 total iterations). for i = 0 to 10 // Draw a new label `i` bars ahead of the current bar. label.new(bar_index + i, 0, str.tostring(i), textcolor = color.white, size = size.large)

Note that:

- The `i` variable represents the loop's *counter*. This variable is local to the loop's scope, meaning no *outer scopes* can access it. The code uses the variable within the loop's body to determine the location and text of each label drawing.
- Programmers often use `i`, `j`, and `k` as loop counter identifiers. However, *any* valid variable name is allowed. For example, this code behaves the same if we name the counter `offset` instead of `i`.
- The for loop structure *automatically* manages the counter variable. We do not need to define code in the loop's body to increment its value.

The direction in which a for loop adjusts its counter depends on the *initial* `from_num` and `to_num` values in the loop's header, and the direction does not change across iterations. The loop counts *upward* after each iteration when the `to_num` value is *above* the `from_num` value, as shown in the previous example. If the `to_num` value is *below* the `from_num` value, the loop counts *downward* instead.

The script below calculates and plots the volume-weighted moving average (VWMA) of open prices across a specified number of bars. Then, it uses a downward-counting for loop to compare the last historical bar's value to the values from previous bars, starting with the oldest bar in the specified lookback window. On each loop iteration, the script retrieves a previous bar's `vwmaOpen` value, calculates the difference from the current bar's value, and displays the result in a label at the past bar's opening price:

//@version=6 indicator("`for` loop demo", "VWMA differences", true, max_labels_count = 500) //@variable Display color for indicator visuals. const color DISPLAY_COLOR = color.rgb(17, 127, 218) //@variable The number of bars in the `vwmaOpen` calculation. int maLengthInput = input.int(20, "VWMA length", 1) //@variable The number of past bars to look back through and compare to the current bar. int lookbackInput = input.int(15, "Lookback length", 1, 500) //@variable The volume-weighted moving average of `open` values over `maLengthInput` bars. float vwmaOpen = ta.vwma(open, maLengthInput) if barstate.islastconfirmedhistory // Define a `for` loop that counts *downward* from `i == lookbackInput` to `i == 1`. for i = lookbackInput to 1 //@variable The difference between the `vwmaOpen` from `i` bars ago and the current `vwmaOpen`. float vwmaDifference = vwmaOpen[i] - vwmaOpen //@variable A "string" representation of `vwmaDifference`, rounded to two fractional digits. string displayText = (vwmaDifference > 0 ? "+" : "") + str.tostring(vwmaDifference, "0.00") // Draw a label showing the `displayText` at the `open` of

the bar from `i` bars back. label.new(bar_index - i, open[i], displayText, textcolor = color.white, color = DISPLAY_COLOR, style = label.style_label_lower_right, size = size.normal) // Plot the `vwmaOpen` value. plot(vwmaOpen, "VWMA", color = DISPLAY_COLOR, linewidth = 2)

Note that:

- The script uses the loop's counter (`i`) to within the history-referencing operator to retrieve past values of the `vwmaOpen` series. It also uses the counter to determine the location of each label drawing.
- The loop in this example *decreases* the counter by one on each iteration because the final counter value in the loop's header (`1`) is less than the starting value (`lookbackInput`).

Programmers can use for loops to iterate through collections, such as arrays and matrices. The loop's counter can serve as an *index* for retrieving or modifying a collection's contents. For example, this code block uses array.get() inside a for loop to successively retrieve elements from an array:

int lastIndex = array.size(myArray) - 1 for i = 0 to lastIndex element = array.get(i)

Note that:

- Array *indexing* starts from 0, but the array.size() function *counts* array elements starting from 1. Therefore, we must subtract 1 from the array's size to get the maximum index value. This way, the loop counter avoids representing an out-of-bounds index on the last loop iteration.
- The for…in loop statement is often the *preferred* way to loop through collections. However, programmers may prefer a for loop for some tasks, such as looping through stepped index values, iterating over a collection's contents in reverse or a nonlinear order, and more. See the Looping through arrays and Looping through matrices sections to learn more about the best practices for looping through these collection types.

The script below executes ta.rsi() and ta.mom() calls to calculate the RSI and momentum of close prices over three different lengths (10, 20, and 50), then displays the results using a table on the last chart bar. It stores "string" values for the header title within arrays and the "float" values of the calculated indicators within a 2x3 matrix. The script uses a for loop to access the elements in the arrays and initialize the `displayTable` header cells. It then uses *nested* for loops to iterate over the *row* and *column* indices in the `taMatrix`, access elements, convert their values to strings, and populate the remaining table cells:

//@version=6 indicator("`for` loop with collections demo", "Table of TA Indexes", overlay = true) // Calculate the RSI and momentum of `close` values with constant lengths of 10, 20, and 50. float rsi10 = ta.rsi(close, 10) float rsi20 = ta.rsi(close, 20) float rsi50 = ta.rsi(close, 50) float mom10 = ta.mom(close, 10) float mom20 = ta.mom(close, 20) float mom50 = ta.mom(close, 50) if barstate.islast //@variable A `table` that displays indicator values in the top-right corner of the chart. var table displayTable = table.new(position.top_right, columns = 5, rows = 4, border_color = color.black, border_width = 1) //@variable An array containing the "string" titles to display within the side header of each table row. array<string> sideHeaderTitles = array.from("TA Index", "RSI", "Momentum") //@variable An array containing the "string" titles to representing the length of each displayed indicator. array<string> topHeaderTitles = array.from("10", "20", "50") //@variable A matrix containing the values to display within the table. matrix<float> taMatrix = matrix.new<float>() // Populate the `taMatrix` with indicator values. The first row contains RSI data and the second contains momentum. taMatrix.add_row(0, array.from(rsi10, rsi20, rsi50, mom10, mom20, mom50)) taMatrix.reshape(2, 3) // Initialize top header cells. displayTable.cell(1, 0, "Bars Length", text_color = color.white, bgcolor = color.blue) displayTable.merge_cells(1, 0, 3, 0) // Initialize additional header cells within a `for` loop. for i = 0 to 2 displayTable.cell(0, i + 1, sideHeaderTitles.get(i), text_color = color.white,

bgcolor = color.blue) displayTable.cell(i + 1, 1, topHeaderTitles.get(i), text_color = color.white, bgcolor = color.purple) // Use nested `for` loops to iterate through the row and column indices of the `taMatrix`. for i = 0 to taMatrix.rows() - 1 for j = 0 to taMatrix.columns() - 1 //@variable The value stored in the `taMatrix` at the `i` row and `j` column. float elementValue = taMatrix.get(i, j) // Initialize a cell in the `displayTable` at the `i + 2` row and `j + 1` column showing a "string" // representation of the `elementValue`. displayTable.cell(column = j + 1, row = i + 2, text = str.tostring(elementValue, "#.##"), text_color = chart.fg_color)

Note that:

- Both arrays of header names (`sideHeaderTitles` and `topHeaderTitles`) contain the same number of elements, enabling the script to iterate through their contents simultaneously using a single for loop.
- The nested for loops iterate over *all* the index values in the `taMatrix`. The *outer* loop iterates over each *row* index, and the *inner* loop iterates over every *column* index on each outer loop iteration.
- The script creates and displays the table only on the last historical bar and all realtime bars because the historical states of tables are *never* visible. See the Reducing drawing updates of the Profiling and optimization page for more information.

It's important to note that a for loop's header *dynamically* evaluates the `to_num` value at the start of *every iteration*. If the `to_num` argument is a variable and the script changes its value during an iteration, the loop uses the *new value* to update its stopping condition. Likewise, the stopping condition can change across iterations when the `to_num` argument is an expression or function call that depends on data modified in the loop's scope, such as a call to array.size() on a locally resized array or str.length() on an adjusted string. Therefore, scripts can use for loops to perform iterative tasks where the exact number of required iterations is *not predictable* in advance, similar to while loops.

For example, the following script uses a dynamic for loop to determine the historical offset of the most recent bar whose close differs from the current bar's close by at least one standard deviation. The script declares a `barOffset` variable with an initial value of zero and uses that variable to define the loop counter's `to_num` boundary. Within the loop's scope, the script increments the `barOffset` by one if the referenced bar's `close` is not far enough from the current bar's value. Each time the `barOffset` value increases, the loop increases its final counter value, allowing an *extra iteration*. The script plots the `barOffset` and the corresponding bar's close for visual reference:

//@version=6 indicator("`for` loop with dynamic `to_num` demo") //@variable The length of the standard deviation. int lengthInput = input.int(20, "Length", 1, 4999) //@variable The standard deviation of `close` prices over `lengthInput` bars. float stdev = ta.stdev(close, lengthInput) //@variable The minimum bars back where the past bar's `close` differs from the current `close` by at least `stdev`. // Used as the weight value in the weighted average. int barOffset = 0 // Define a `for` loop that iterates from 0 to `barsBack`. for i = 0 to barOffset // Add 1 for each bar where the distance from that bar's `close` to the current bar's `close` is less than `stdev`. // Each time `barsBack` increases, it changes the loop's `to_num` boundary, allowing another iteration. barOffset += math.abs(close - close[i]) < stdev ? 1 : 0 //@variable A gradient color for the `barOffset` plot. color offsetColor = color.from_gradient(barOffset, 0, lengthInput, color.blue, color.orange) // Plot the `barOffset` in a separate pane. plot(barOffset, "Bar offset", offsetColor, 1, plot.style_columns) // Plot the historical `close` price from `barOffset` bars back in the main chart pane. plot(close[barOffset], "Historical bar's price", color.blue, 3, force_overlay = true)

Note that:

- Changing the `to_num` value on an iteration does not affect the established *direction* in which the loop adjusts its counter variable. For instance, if the loop in this example changed `barOffset` to -1 on any

iteration, it would stop immediately after that iteration ends without reducing the `i` value.
- The script uses `force_overlay = true` in the second plot() call to display the historical closing price on the main chart pane.

`while` loops

The while loop statement creates a *condition-controlled* loop, which uses a *conditional expression* to control the executions of its local block. The loop continues its iterations as long as the specified condition remains `true`.

Pine Script uses the following syntax to define a while loop:

```
[variables = | :=] while condition
    statements | continue | break
    return_expression
```

Where the `condition` in the loop's *header* can be a literal, variable, expression, or function call that returns a "bool" value.

Refer to the Common characteristics section above for detailed information about the `variables`, `statements`, `continue`, `break`, and `return_expression` parts of the loop's syntax.

A while loop's header evaluates its `condition` before each iteration. Consequently, when the script modifies the condition within an iteration, the loop's header reflects those changes on the *next* iteration.

Depending on the specified condition in the loop header, a while loop can behave similarly to a for loop, continuing iteration until a *counter* variable reaches a specified limit. For example, the following script uses a for loop and while loop to perform the same task. Both loops draw a label displaying their respective counter value on each iteration:

//@version=6 indicator("`while` loop with a counter condition demo") if barstate.islastconfirmedhistory // A `for` loop that creates blue labels displaying each `i` value. for i = 0 to 10 label.new(bar_index + i, 0, str.tostring(i), color = color.blue, textcolor = color.white, size = size.large, style = label.style_label_down) //@variable An "int" to use as a counter within a `while` loop. int j = 0 // A `while` loop that creates orange labels displaying each `j` value. while j <= 10 label.new(bar_index + j, 0, str.tostring(j), color = color.orange, textcolor = color.white, size = size.large, style = label.style_label_up) // Update the `j` counter within the local block. j += 1

Note that:

- When a while loop uses count-based logic, it must explicitly manage the user-specified counter within the local block. In contrast, a for loop increments its counter automatically.
- The script declares the variable the while loop uses as a counter *outside* the loop's scope, meaning its value is usable in additional calculations after the loop terminates.
- If this code did not increment the `j` variable within the while loop's body, the value would *never* reach 10, meaning the loop would run *indefinitely* until causing a runtime error.

Because a while loop's execution depends on its condition remaining `true`, and the condition might not change on a specific iteration, the *precise* number of expected iterations might not be knowable *before* the

loop begins. Therefore, while loops are often helpful in scenarios where the exact loop boundaries are *unknown*.

The script below tracks when the chart's close crosses outside Keltner Channels with a user-specified length and channel width. When the price crosses outside the current bar's channel, the script draws a box highlighting all the previous *consecutive* bars with close values within that price window. The script uses a while loop to analyze past bars' prices and incrementally adjust the left side of each new box until the drawing covers all the latest consecutive bars in the current range:

//@version=6 indicator("`while` loop demo", "Price window boxes", true) //@variable The length of the channel. int lengthInput = input.int(20, "Channel length", 1, 4999) //@variable The width multiplier of the channel. float widthInput = input.float(2.0, "Width multiplier", 0) //@variable The `lengthInput`-bar EMA of `close` prices. float ma = ta.ema(close, lengthInput) //@variable The `lengthInput`-bar ATR, multiplied by the `widthInput`. float atr = ta.atr(lengthInput) * widthInput //@variable The lower bound of the channel. float channelLow = ma - atr //@variable The upper bound of the channel. float channelHigh = ma + atr //@variable Is `true` when the `close` price is outside the current channel range, `false` otherwise. bool priceOutsideChannel = close < channelLow or close > channelHigh // Check if the `close` crossed outside the channel range, then analyze the past bars within the current range. if priceOutsideChannel and not priceOutsideChannel[1] //@variable A box that highlights consecutive past bars within the current channel's price window. box windowBox = box.new(bar_index, channelHigh, bar_index, channelLow, border_width = 2, bgcolor = color.new(color.gray, 85)) //@variable The lookback index for box adjustment. The `while` loop increments this value on each iteration. int i = 1 // Use a `while` loop to look backward through close` prices. The loop iterates as long as the past `close` // from `i` bars ago is between the current bar's `channelLow` and `channelHigh`. while close[i] >= channelLow and close[i] <= channelHigh // Adjust the left side of the box. windowBox.set_left(bar_index - i) // Add 1 to the `i` value to check the `close` from the next bar back on the next iteration. i += 1 // Plot the `channelLow` and `channelHigh` for visual reference. plot(channelLow, "Channel low") plot(channelHigh, "Channel high")

Note that:

- The left and right edges of boxes sit within the horizontal *center* of their respective bars, meaning that each drawing spans from the middle of the first consecutive bar to the middle of the last bar within each window.
- This script uses the `i` variable as a history-referencing index within the *conditional expression* the while loop checks on each iteration. The variable **does not** behave as a loop counter, as the iteration boundaries are **unknown**. The loop executes its local block repeatedly until the condition becomes `false`.

`for...in` loops

The for…in loop statement creates a *collection-controlled* loop, which uses the *contents* of a collection to control its iterations. This loop structure is often the preferred approach for looping through arrays, matrices, and maps.

A for…in loop traverses a collection *in order*, retrieving one of its stored items on each iteration. Therefore, the loop's boundaries depend directly on the number of *items* (array *elements*, matrix *rows*, or map *key-value pairs*).

Pine Script features *two* general forms of the for…in loop statement. The *first form* uses the following syntax:

```
[variables = | :=] for item in collection_id
    statements | continue | break
    return_expression
```

Where `item` is a *variable* that holds sequential values or references from the specified `collection_id`. The variable starts with the collection's *first item* and takes on successive items in order after each iteration. This form is convenient when a script must access values from an array or matrix iteratively but does not require the item's *index* in its calculations.

The *second form* has a slightly different syntax that includes a tuple in its *header*:

```
[variables = | :=] for [index, item] in collection_id
    statements | continue | break
    return_expression
```

Where `index` is a variable that contains the *index* or *key* of the retrieved `item`. This form is convenient when a task requires using a collection's items *and* their indices in iterative calculations. This form of the for…in loop is *required* when directly iterating through the contents of a map. See this section below for more information.

Refer to the Common characteristics section above for detailed information about the `variables`, `statements`, `continue`, `break`, and `return_expression` parts of the loop's syntax.

The iterative behavior of a for…in loop depends on the *type* of collection the header specifies as the `collection_id`:

- When using an array in the header, the loop performs *element-wise* iteration, meaning the retrieved `item` on each iteration is one of the array's *elements*.
- When using a matrix in the header, the loop performs *row-wise* iteration, which means that each `item` represents a *row array*.
- When using a map in the header, the loop performs *pair-wise* iteration, which retrieves a *key* and corresponding *value* on each iteration.

> NoteScripts can modify the sizes of arrays and matrices directly within a for…in loop's local scope. When a for…in loop changes the size of a collection during an iteration, the loop's header uses the *updated size* to control subsequent iterations, just like an equivalent for loop that uses `array.size(id) - 1` or `matrix.rows(id) - 1` as the `to_num` argument.

Looping through arrays

Pine scripts can iterate over the elements of arrays using any loop structure. However, the for…in loop is typically the most convenient because it automatically verifies the size of an array when controlling iterations. With other loop structures, programmers must carefully set the header's boundaries or conditions to *prevent* the loop from attempting to access an element at a *nonexistent* index.

For example, a for loop can access an array's elements using the counter variable as the lookup index in functions such as array.get(). However, programmers must ensure the counter always represents a *valid index* to prevent out-of-bounds errors. Additionally, if an array might be *empty*, programmers must set conditions to prevent the loop's execution entirely.

The code below shows a for loop whose counter boundaries depend on the number of elements in an array. If the array is empty, containing zero elements, the header's final counter value is na, which *prevents* iteration. Otherwise, the final value is *one less* than the array's size (i.e., the index of the last element):

for index = 0 to (array.size(myArray) == 0 ? na : array.size(myArray) - 1) element = array.get(myArray, index)

In contrast, a for…in loop automatically validates an array's size and *directly* accesses its elements, providing a more convenient solution than a traditional for loop. The line below achieves the *same effect* as the code above without requiring the programmer to define boundaries explicitly or use the array.get() function to access each element:

for element in myArray

The following example examines bars on a lower timeframe to gauge the strength of *intrabar* trends within each chart bar. The script uses a request.security_lower_tf() call to retrieve an array of intrabar hl2 prices from a calculated `lowerTimeframe`. Then, it uses a for…in loop to access each `price` within the `intrabarPrices` array and compare the value to the current close to calculate the bar's `strength`. The script plots the `strength` as columns in a separate pane:

//@version=6 indicator("`for element in array` demo", "Intrabar strength") //@variable A valid timeframe closest to one-tenth of the current chart's timeframe, "1" if the timeframe is too small. var string lowerTimeframe = timeframe.from_seconds(math.max(int(timeframe.in_seconds() / 10), 60)) //@variable An array of intrabar `hl2` prices calculated from the `lowerTimeframe`. array<float> intrabarPrices = request.security_lower_tf("", lowerTimeframe, hl2) //@variable The excess trend strength of `intrabarPrices`. float strength = 0.0 // Loop directly through the `intrabarPrices` array. Each iteration's `price` represents an array element. for price in intrabarPrices // Subtract 1 from the `strength` if the retrieved `price` is above the current bar's `close` price. if price > close strength -= 1 // Add 1 to the `strength` if the retrieved `price` is below the current bar's `close` price. else if price < close strength += 1 //@variable Is `color.teal` when the `strength` is positive, `color.maroon` otherwise. color strengthColor = strength > 0 ? color.teal : color.maroon // Plot the `strength` as columns colored by the `strengthColor`. plot(strength, "Intrabar strength", strengthColor, 1, plot.style_columns)

The second form of the for…in loop is a convenient solution when a script's calculations require accessing each element *and* corresponding index within an array:

for [index, element] in myArray

For example, suppose we want to display a *numerated* list of array elements within a label while excluding values at specific indices. We can use the second form of the for…in loop structure to accomplish this task. The simple script below declares a `stringArray` variable that references an array of predefined "string" values. On the last historical bar, the script uses a for…in loop to access each `index` and `element` in the `stringArray` to construct the `labelText`, which it uses in a label.new() call after the loop ends:

//@version=6 indicator("`for [index, item] in array` demo", "Array numerated output") //@variable An array of "string" values to display as a numerated list. var array<string> stringArray = array.from("First", "Second", "Third", "Before Last", "Last") if barstate.islastconfirmedhistory //@variable A "string" modified within a loop

to display within the `label`. string labelText = "Array values: \n" // Loop through the `stringArray`, accessing each `index` and corresponding `element`. for [index, element] in stringArray // Skip the third `element` (at `index == 2`) in the `labelText`. Include an "ELEMENT SKIPPED" message instead. if index == 2 labelText += "-- ELEMENT SKIPPED -- \n" continue labelText += str.tostring(index + 1) + ": " + element + "\n" // Display the `labelText` within a `label`. label.new(bar_index, 0, labelText, textcolor = color.white, size = size.huge, style = label.style_label_center, textalign = text.align_left)

Note that:

- This example adds 1 to the `index` in the str.tostring() call to start the numerated list with a value of `"1"`, because array indices always begins at 0.
- On the *third* loop iteration, when `index == 2`, the script adds an `"-- ELEMENT SKIPPED --"` message to the `labelText` instead of the retrieved `element` and uses the `continue` keyword to skip the remainder of the iteration. See this section above to learn more about loop keywords.

Let's explore an advanced example demonstrating the utility of for…in loops. The following indicator draws a fixed number of horizontal lines at pivot high values calculated from a ta.pivothigh() call, and it analyzes the lines within a loop to determine which ones represent active (*uncrossed*) pivots.

Each time the script detects a new pivot high point, it creates a new line, *inserts* that line at the beginning of the `pivotLines` array, then removes the oldest element and deletes its ID using line.delete(). The script accesses each line within the array using a for…in loop, analyzing and modifying the properties of the line referenced on each iteration. When the current high crosses above the `pivotLine`, the script changes its style to signify that it is no longer an active level. Otherwise, it extends the line's `x2` coordinate and uses its price to calculate the average *active* pivot value. The script also plots each pivot high value and the average active pivot value on the chart:

//@version=6 indicator("`for...in` loop with arrays demo", "Active high pivots", true, max_lines_count = 500) //@variable The number of bars required on the left and right to confirm a pivot point. int pivotBarsInput = input.int(5, "Pivot leg length", 1) //@variable The number of recent pivot lines to analyze. Controls the size of the `pivotLines` array. int maxRecentLines = input.int(20, "Maximum recent lines", 1, 500) //@variable An array that acts as a queue holding the most recent pivot high lines. var array<line> pivotLines = array.new<line>(maxRecentLines) //@variable The pivot high price, or `na` if no pivot is found. float highPivotPrice = ta.pivothigh(pivotBarsInput, pivotBarsInput) if not na(highPivotPrice) //@variable The `chart.point` for the start of the line. Does not contain `time` information. firstPoint = chart.point.from_index(bar_index - pivotBarsInput, highPivotPrice) //@variable The `chart.point` for the end of each line. Does not contain `time` information. secondPoint = chart.point.from_index(bar_index, highPivotPrice) //@variable A horizontal line at the new pivot level. line hiPivotLine = line.new(firstPoint, secondPoint, width = 2, color = color.green) // Insert the `hiPivotLine` at the beginning of the `pivotLines` array. pivotLines.unshift(hiPivotLine) // Remove the oldest line from the array and delete its ID. line.delete(pivotLines.pop()) //@variable The sum of active pivot prices. float activePivotSum = 0.0 //@variable The number of active pivot high levels. int numActivePivots = 0 // Loop through the `pivotLines` array, directly accessing each `pivotLine` element. for pivotLine in pivotLines //@variable The `x2` coordinate of the `pivotline`. int lineEnd = pivotLine.get_x2() // Move to the next `pivotline` in the array if the current line is inactive. if pivotLine.get_x2() < bar_index - 1 continue //@variable The price value of the `pivotLine`. float pivotPrice = pivotLine.get_price(bar_index) // Change the style of the `pivotLine` and stop extending its display if the `high` is above the `pivotPrice`. if high > pivotPrice pivotLine.set_color(color.maroon) pivotLine.set_style(line.style_dotted) pivotLine.set_width(1) continue // Extend the `pivotLine` and add the `pivotPrice` to the `activePivotSum` when the loop allows a full iteration.

pivotLine.set_x2(bar_index) activePivotSum += pivotPrice numActivePivots += 1 //@variable The average active pivot high value. float avgActivePivot = activePivotSum / numActivePivots // Plot crosses at the `highPivotPrice`, offset backward by the `pivotBarsInput`. plot(highPivotPrice, "High pivot marker", color.green, 3, plot.style_cross, offset = -pivotBarsInput) // Plot the `avgActivePivot` as a line with breaks. plot(avgActivePivot, "Avg. active pivot", color.orange, 3, plot.style_linebr)

Note that:

- The loop in this example executes on *every bar* because it has to compare active pivot line prices with the current high value, then use the remaining active prices to calculate the bar's `avgActivePivot` value.
- Pine Script features several ways to calculate averages, many of which *do not* require a loop. However, a loop is necessary in this example because the script uses information only available on the **current bar** to determine which prices contribute toward the average.
- The *first* form of the for…in loop is the most convenient option in this example because we need direct access to the lines referenced within the `pivotLines` array, but we do not need the corresponding *index* values.

Looping through matrices

Pine scripts can iterate over the contents of a matrix in several different ways. Unlike arrays, matrices use *two* indices to reference their elements because they store data in a *rectangular* format. The first index refers to *rows*, and the second refers to *columns*. If a programmer opts to use for or while loops to iterate through matrices instead of using for…in, they must carefully define the loop boundaries or conditions to avoid out-of-bounds errors.

This code block shows a for loop that performs *row-wise* iteration, looping through each *row index* in a matrix and using the value in a matrix.row() call to retrieve a row array. If the matrix is empty, the loop statement uses a final loop counter value of na to *prevent* iteration. Otherwise, the final counter is the last row index, which is *one less* than the value returned by matrix.rows():

for rowIndex = 0 to (myMatrix.rows() == 0 ? na : myMatrix.rows() - 1) rowArray = myMatrix.row(rowIndex)

Note that:

- If we replace the matrix.rows() and matrix.row() calls with matrix.columns() and matrix.col(), the loop performs *column-wise* iteration instead.

The for…in loop statement is the more convenient approach to loop over and access the rows of a matrix in order, as it automatically validates the number of rows and retrieves an array of the current row's elements on each iteration:

for rowArray in myMatrix

When a script's calculations require access to each row from a matrix and its corresponding *index*, programmers can use the second form of the for…in loop:

for [rowIndex, rowArray] in myMatrix

Note that:

- The for…in loop only performs **row-wise** iteration on matrices. To *emulate* column-wise iteration, programmers can use a for…in loop on a transposed copy.

The following example creates a custom string representing the rows of a matrix with extra information. When the script executes on the last historical bar, it creates a 3x3 matrix populated with values from math.random() calls. Using the first form of the for…in loop, the script iterates through each row in the matrix to create a "string" value representing the row's contents, its average, and whether the average is above 0.5. Before the end of each iteration, the script concatenates the constructed string with the `labelText` value. After the loop ends, the script creates a label to display the `labelText` variable's final value:

//@version=6 indicator("`for row in matrix` demo", "Custom matrix label") //@variable Generates a pseudorandom value between 0 and 1, rounded to 4 decimal places. rand() => math.round(math.random(), 4) if barstate.islastconfirmedhistory //@variable A matrix of randomized values to format and display in a `label`. matrix<float> randomMatrix = matrix.new<float>() // Add a row of 9 randomized values and reshape the matrix to 3x3. randomMatrix.add_row(0, array.from(rand(), rand(), rand(), rand(), rand(), rand(), rand(), rand(), rand())) randomMatrix.reshape(3, 3) //@variable A custom "string" representation of `randomMatrix` information. Modified within a loop. string labelText = "Matrix rows: \n" // Loop through the rows in the `randomMatrix`. for row in randomMatrix //@variable The average element value within the `row`. float rowAvg = row.avg() //@variable An upward arrow when the `rowAvg` is above 0.5, a downward arrow otherwise. string directionChar = rowAvg > 0.5 ? "⬆" : "⬇" // Add a "string" representing the `row` array, its average, and the `directionChar` to the `labelText`. labelText += str.format("Row: {0} Avg: {1} {2}\n", row, rowAvg, directionChar) // Draw a `label` displaying the `labelText` on the current bar. label.new(bar_index, 0, labelText, color = color.purple, textcolor = color.white, size = size.huge, style = label.style_label_center, textalign = text.align_left)

Working with matrices often entails iteratively accessing their *elements*, not just their rows and columns, typically using *nested loops*. For example, this code block uses an outer for loop to iterate over row indices. The inner for loop iterates over column indices on *each* outer loop iteration and calls matrix.get() to access an element:

for rowIndex = 0 to (myMatrix.rows() == 0 ? na : myMatrix.rows() - 1) for columnIndex = 0 to myMatrix.columns() - 1 element = myMatrix.get(rowIndex, columnIndex)

Alternatively, a more convenient approach for this type of task is to use nested for…in loops. The outer for…in loop in this code block retrieves each row array in a matrix, and the inner for…in statement loops through that array:

for rowArray in myMatrix for element in rowArray

The script below creates a 3x2 matrix, then accesses and modifies its elements within nested for…in loops. Both loops use the second form of the for…in statement to retrieve index values and corresponding items. The outer loop accesses a row index and row array from the matrix. The inner loop accesses each index and respective element from that array.

Within the nested loop's iterations, the script converts each `element` to a "string" and initializes a table cell at the `rowIndex` row and `colIndex` column. Then, it uses the loop header variables within matrix.set() to update the matrix element. After the outer loop terminates, the script displays a "string" representation of the *updated* matrix within a label:

```
//@version=6 indicator("Nested `for...in` loops on matrices demo") if barstate.islastconfirmedhistory //@variable A matrix containing numbers to display. matrix<float> displayNumbers = matrix.new<float>() // Populate the `displayNumbers` matrix and reshape to 3x2. displayNumbers.add_row(0, array.from(1, 2, 3, 4, 5, 6)) displayNumbers.reshape(3, 2) //@variable A table that displays the elements of the `displayNumbers` before modification. table displayTable = table.new( position = position.middle_center, columns = displayNumbers.columns(), rows = displayNumbers.rows(), bgcolor = color.purple, border_color = color.white, border_width = 2 ) // Loop through the `displayNumbers`, retrieving the `rowIndex` and the current `row`. for [rowIndex, row] in displayNumbers // Loop through the current `row` on each outer loop iteration to retrieve the `colIndex` and `element`. for [colIndex, element] in row // Initialize a table cell at the `rowIndex` row and `colIndex` column displaying the current `element`. displayTable.cell(column = colIndex, row = rowIndex, text = str.tostring(element), text_color = color.white, text_size = size.huge ) // Update the `displayNumbers` value at the `rowIndex` and `colIndex`. displayNumbers.set(rowIndex, colIndex, math.round(math.exp(element), 3)) // Draw a `label` to display a "string" representation of the updated `displayNumbers` matrix. label.new( x = bar_index, y = 0, text = "Matrix now modified: \n" + str.tostring(displayNumbers), color = color.orange, textcolor = color.white, size = size.huge, style = label.style_label_up )
```

Looping through maps

The for…in loop statement is the primary, most convenient approach for iterating over the data within Pine Script maps.

Unlike arrays and matrices, maps are *unordered collections* that store data in *key-value pairs*. Rather than traversing an internal lookup index, a script references the *keys* from the pairs within a map to access its *values*. Therefore, when looping through a map, scripts must perform *pair-wise* iteration, which entails retrieving key-value pairs across iterations rather than indexed elements or rows.

Note that:

- Although maps are unordered collections, Pine Script internally tracks the *insertion order* of their key-value pairs.

One way to access the data from a map is to use the map.keys() function, which returns an array containing all the *keys* from the map, sorted in their insertion order. A script can use the for…in structure to loop through the array of keys and call map.get() to retrieve corresponding values:

```
for key in myMap.keys() value = myMap.get(key)
```

However, the more convenient, *recommended* approach is to loop through a map directly *without* creating new arrays. To loop through a map directly, use the second form of the for…in loop statement. Using this loop with a map creates a tuple containing a *key* and respective *value* on each iteration. As when looping through a map.keys() array, this *direct* for…in loop iterates through a map's contents in their insertion order:

```
for [key, value] in myMap
```

Note that:

- The second form of the for…in loop is the **only** way to iterate *directly* through a map. A script cannot directly loop through this collection type without retrieving a key and value on each iteration.

Let’s consider a simple example demonstrating how a for…in loop works on a map. When the script below executes on the last historical bar, it declares a `simpleMap` variable to reference a map of “string” keys and “float” values. The script uses map.put() to insert the keys from the `newKeys` array into the collection with corresponding values from math.random() calls. Then, it uses a for…in loop to iterate through the key-value pairs from the map and construct the `displayText` string. After the loop ends, the script uses a label to visualize the string:

//@version=6 indicator("Looping through map demo") if barstate.islastconfirmedhistory //@variable A map of "string" keys and "float" values to render within a `label`. map<string, float> simpleMap = map.new<string, float>() //@variable An array of "string" values representing the keys to put into the map. array<string> newKeys = array.from("A", "B", "C", "D", "E") // Put key-value pairs into the `simpleMap`. for key in newKeys simpleMap.put(key, math.random(1, 20)) //@variable A "string" representation of the `simpleMap` contents. Modified within a loop. string displayText = "simpleMap content: \n " // Loop through each key-value pair within the `simpleMap`. for [key, value] in simpleMap // Add a "string" representation of the pair to the `displayText`. displayText += key + ": " + str.tostring(value, "#.##") + "\n " // Draw a `label` showing the `displayText` on the current bar. label.new(x = bar_index, y = 0, text = displayText, color = color.green, textcolor = color.white, size = size.huge, textalign = text.align_left, style = label.style_label_center)

Note that:

- This script uses both forms of the for…in loop statement. The first loop iterates through the “string” elements of the `newKeys` array to put key-value pairs into the map referenced by `simpleMap`, and the second iterates directly through the map’s key-value pairs to construct the custom string.

> Notice
>
> In contrast to arrays and matrices, maps cannot change in size while a script iterates through them directly using a for…in loop. Attempting to add or remove a map’s key-value pairs while looping through it with this structure typically causes a *runtime error*.
>
> To correctly modify a map’s size within a loop, programmers can do any of the following:
>
> - Make a copy of the map and loop through that copied instance.
> - Use a for…in loop to iterate through the map.keys() *array*.
> - Use a for or while loop instead of a for…in loop.

ADVANCED

Maps

> TipThis page contains *advanced* material. If you're new to Pine Script®, start by learning about core language components – such as the type system and the basics of the execution model – and explore other, more accessible features before venturing further.

Introduction

Pine Script *maps* are collections that store data in *key-value pairs*. They enable scripts to collect multiple values or references in a single location and associate those elements with specific *unique values (keys)*.

In contrast to arrays and matrices, maps are *unordered* collections. Scripts do not access a map's value elements by traversing an internal index. Instead, they quickly access values by using the *keys* from the collection's key-value pairs as *lookup indices*.

A map's key elements must be of a value type, including any *fundamental type* or enum type. Maps *do not* allow the IDs of any reference type as keys, but they do allow those IDs as *value elements*. Note that, as with other collections, maps cannot directly store the IDs of other collections. However, their value elements can store references to objects of user-defined types that have collection-type fields. See the Maps of other collections section to learn more.

Similar to arrays and matrices, maps can contain up to 100,000 elements in total. However, because each key-value pair in a map consists of *two* elements (a *unique key* and associated *value*), the maximum number of key-value pairs that a map can contain is 50,000.

Declaring a map

Pine Script uses the following syntax for map declarations:

```
[var/varip ][map<keyType, valueType> ]<identifier> = <expression>
```

Where `<keyType, valueType>` is the *type template* that defines the types of *key elements* and *value elements* that the map can contain, and `<expression>` is an expression that returns either the *reference (ID)* of a map or na. See the Collections section of the Type system page to learn about type templates.

When initializing a map variable using na, programmers must prefix the variable declaration with the map keyword followed by a type template to explicitly define the variable's *type identifier* (e.g., `map<int, float>` for a variable that can reference a map of "int" key elements and "float" value elements).

The following line of code declares a `myMap` variable with an initial reference of na. The variable declaration includes the type identifier `map<string, float>`, which tells the compiler that the variable can accept the ID of a map containing "string" keys and corresponding "float" values:

map<string, float> myMap = na

If a map variable is not initialized with na, specifying the type identifier is optional, because the compiler can automatically determine the variable's accepted type from the assigned map ID.

For example, the following line calls `map.new<string, float>()` to create an empty map that stores "string" keys and "float" values, then assigns the resulting ID to the `myMap` variable. An explicit type identifier is optional in this variable declaration, because the compiler uses the assigned ID to determine that the variable's type is `map<string, float>`:

myMap = map.new<string, float>()

Using `var` and `varip` keywords

Users can include the var or varip keywords to instruct their scripts to declare a map variable on only one bar instead of on every execution of the variable's scope. Variables that use these keywords point to the same map instances on each script execution until explicitly reassigned.

For example, this script declares a `colorMap` variable to reference a map that holds pairs of string keys and color values on the first chart bar. The script plots a calculated `oscillator` series in a separate pane. It uses the "color" values put into the map on the *first* bar to color the plot on *every* bar:

//@version=6 indicator("var map demo") //@variable A map associating color values with string keys. var colorMap = map.new<string, color>() // Put `<string, color>` pairs into `colorMap` on the first bar. if bar_index == 0 colorMap.put("Bull", color.green) colorMap.put("Bear", color.red) colorMap.put("Neutral", color.gray) //@variable The 14-bar RSI of `close`. float oscillator = ta.rsi(close, 14) //@variable The color of the `oscillator`. color oscColor = switch oscillator > 50 => colorMap.get("Bull") oscillator < 50 => colorMap.get("Bear") => colorMap.get("Neutral") // Plot the `oscillator` using the `oscColor` from our `colorMap`. plot(oscillator, "Histogram", oscColor, 2, plot.style_histogram, histbase = 50) plot(oscillator, "Line", oscColor, 3)

> Notice
>
> Map variables declared using varip behave similarly to those declared using var, with two key differences. Firstly, the maps that they reference can finalize updates to their key-value pairs on *any* available tick – not only on a bar's closing tick. Secondly, maps referenced by varip variables can store only the following types of data:
>
> - Values of any fundamental type.
> - IDs of the chart.point, footprint, or volume_row type.
> - References to objects of a user-defined type that have fields for storing only data of either of the above types or the IDs of other collections containing only these types.

Reading and writing

Putting and getting key-value pairs

The map.put() function is one that map users will utilize quite often, as it's the primary method to put a new key-value pair into a map. It associates the `key` argument with the `value` argument in the call and adds the pair to the `id` map.

If the `key` argument in the map.put() call already exists in the map's keys, the new pair passed into the function will **replace** the existing one.

To retrieve the value from a map `id` associated with a given `key`, use map.get(). This function returns the value if the `id` map contains the `key`. Otherwise, it returns na.

The following example calculates the difference between the bar_index values from when close was last rising and falling over a given `length` with the help of map.put() and map.get() methods. The script puts a `("Rising", bar_index)` pair into the `data` map when the price is rising and puts a `("Falling", bar_index)` pair into the map when the price is falling. It then puts a pair containing the “Difference” between the “Rising” and “Falling” values into the map and plots that pair’s value on the chart:

//@version=6 indicator("Putting and getting demo") //@variable The length of the `ta.rising()` and `ta.falling()` calculation. int length = input.int(2, "Length") //@variable A map associating `string` keys with `int` values. var data = map.new<string, int>() // Put a new ("Rising", `bar_index`) pair into the `data` map when `close` is rising. if ta.rising(close, length) data.put("Rising", bar_index) // Put a new ("Falling", `bar_index`) pair into the `data` map when `close` is falling. if ta.falling(close, length) data.put("Falling", bar_index) // Put the "Difference" between current "Rising" and "Falling" values into the `data` map. data.put("Difference", data.get("Rising") - data.get("Falling")) //@variable The difference between the last "Rising" and "Falling" `bar_index`. int index = data.get("Difference") //@variable Returns `color.green` when `index` is positive, `color.red` when negative, and `color.gray` otherwise. color indexColor = index > 0 ? color.green : index < 0 ? color.red : color.gray plot(index, color = indexColor, style = plot.style_columns)

Note that:

- This script replaces the values associated with the “Rising”, “Falling”, and “Difference” keys on successive map.put() method calls, as each of these keys is unique and can only appear once in the `data` map.
- Replacing the pairs in a map does not change the internal *insertion order* of its keys. We discuss this further in the next section.

Similar to working with other collections, when putting the reference for an instance of a *special type* (line, linefill, box, polyline, label, table, or chart.point) or a user-defined type into a map, it's important to note the inserted pair's `value` refers to that same object without copying it. Modifying the object referenced by a key-value pair directly modifies the *original* object.

For example, this script contains a custom `ChartData` type with `o`, `h`, `l`, and `c` fields. On the first chart bar, the script declares a `myMap` variable and adds the pair `("A", myData)`, where `myData` is a `ChartData` instance with initial field values of `na`. It adds the pair `("B", myData)` to `myMap` and updates the object from this pair on every bar via the user-defined `update()` method.

Each change to the object corresponding to the “B” key affects the one referenced by the pair with the “A” key, as shown by the candle plot of the “A” object's fields:

//@version=6 indicator("Putting and getting objects demo") //@type A custom type to hold OHLC data. type ChartData float o float h float l float c //@function Updates the fields of a `ChartData` object. method update(ChartData this) => this.o := open this.h := high this.l := low this.c := close //@variable A new `ChartData` instance declared on the first bar. var myData = ChartData.new() //@variable A map associating `string` keys with `ChartData` instances. var myMap = map.new<string, ChartData>() // Put a new pair with the "A" key into `myMap` only on the first bar. if bar_index == 0 myMap.put("A", myData) // Put a pair with

the "B" key into `myMap` on every bar. myMap.put("B", myData) //@variable The `ChartData` value associated with the "A" key in `myMap`. ChartData oldest = myMap.get("A") //@variable The `ChartData` value associated with the "B" key in `myMap`. ChartData newest = myMap.get("B") // Update `newest`. Also affects `oldest` and `myData` since they all reference the same `ChartData` object. newest.update() // Plot the fields of `oldest` as candles. plotcandle(oldest.o, oldest.h, oldest.l, oldest.c)

Note that:

- This script would behave differently if it passed a copy of `myData` into each map.put() call. For more information, see the Copying objects section of the Objects page.

Inspecting keys and values

`map.keys()` and `map.values()`

To retrieve all keys and values put into a map, use map.keys() and map.values(). These functions copy all keys/values within an `id` map to a new array object. Modifying the array returned from either of these functions does not affect the `id` map.

Although maps are *unordered* collections, Pine Script internally maintains the *insertion order* of a map's key-value pairs. As a result, the map.keys() and map.values() functions always return arrays with their elements ordered based on the `id` map's insertion order.

The script below demonstrates this by displaying the key and value arrays from an `m` map in a label once every 50 bars. As we see on the chart, the order of elements in each array returned by `m.keys()` and `m.values()` aligns with the insertion order of the key-value pairs in `m`:

//@version=6 indicator("Keys and values demo") if bar_index % 50 == 0 //@variable A map containing pairs of `string` keys and `float` values. m = map.new<string, float>() // Put pairs into `m`. The map will maintain this insertion order. m.put("First", math.round(math.random(0, 100))) m.put("Second", m.get("First") + 1) m.put("Third", m.get("Second") + 1) //@variable An array containing the keys of `m` in their insertion order. array<string> keys = m.keys() //@variable An array containing the values of `m` in their insertion order. array<float> values = m.values() //@variable A label displaying the `size` of `m` and the `keys` and `values` arrays. label debugLabel = label.new(bar_index, 0, str.format("Pairs: {0}\nKeys: {1}\nValues: {2}", m.size(), keys, values), color = color.navy, style = label.style_label_center, textcolor = color.white, size = size.huge)

Note that:

- The value with the “First” key is a random whole number between 0 and 100. The “Second” value is one greater than the “First”, and the “Third” value is one greater than the “Second”.

It's important to note a map's internal insertion order **does not** change when replacing its key-value pairs. The locations of the new elements in the map.keys() and map.values() arrays will be the same as the old elements in such cases. The only exception is if the script completely removes the key beforehand.

Below, we've added a line of code that calls map.put() as a method to insert a new value with the “Second” key into the `m` map, overwriting the previous value associated with that key. Although the script puts this new key-value pair into the map *after* the inserting one with the “Third” key, the pair's key and value are still second in the `keys` and `values` arrays, because the key is already present in the map *before* the new call:

//@version=6 indicator("Keys and values demo") if bar_index % 50 == 0 //@variable A map containing pairs of `string` keys and `float` values. m = map.new<string, float>() // Put pairs into `m`. The map will maintain this insertion order. m.put("First", math.round(math.random(0, 100))) m.put("Second", m.get("First") + 1) m.put("Third", m.get("Second") + 1) // Overwrite the "Second" pair in `m`. This will NOT affect the insertion order. // The key and value will still appear second in the `keys` and `values` arrays. m.put("Second", -2) //@variable An array containing the keys of `m` in their insertion order. array<string> keys = m.keys() //@variable An array containing the values of `m` in their insertion order. array<float> values = m.values() //@variable A label displaying the `size` of `m` and the `keys` and `values` arrays. label debugLabel = label.new(bar_index, 0, str.format("Pairs: {0}\nKeys: {1}\nValues: {2}", m.size(), keys, values), color = color.navy, style = label.style_label_center, textcolor = color.white, size = size.huge)

> NoticeThe elements in a map.values() array store the same values or references as the `id` map. If a map's value elements are of any reference type, including line, linefill, box, polyline, label, table, chart.point, or a UDT, modifying the instances referenced by the map.values() array also affects those referenced by the original `id` map, because the contents of both collections point to identical objects.

`map.contains()`

To check if a specific `key` exists within an `id` map, use map.contains(). This function is a convenient alternative to calling array.includes() on the array returned from map.keys().

For example, this script checks if various keys exist within an `m` map, then displays the results in a label:

//@version=6 indicator("Inspecting keys demo") //@variable A map containing `string` keys and `string` values. m = map.new<string, string>() // Put key-value pairs into the map. m.put("A", "B") m.put("C", "D") m.put("E", "F") //@variable An array of keys to check for in `m`. array<string> testKeys = array.from("A", "B", "C", "D", "E", "F") //@variable An array containing all elements from `testKeys` found in the keys of `m`. array<string> mappedKeys = array.new<string>() for key in testKeys // Add the `key` to `mappedKeys` if `m` contains it. if m.contains(key) mappedKeys.push(key) //@variable A string representing the `testKeys` array and the elements found within the keys of `m`. string testText = str.format("Tested keys: {0}\nKeys found: {1}", testKeys, mappedKeys) if bar_index == last_bar_index - 1 //@variable Displays the `testText` in a label at the `bar_index` before the last. label debugLabel = label.new(bar_index, 0, testText, style = label.style_label_center, textcolor = color.white, size = size.huge)

Removing key-value pairs

To remove a specific key-value pair from an `id` map, use map.remove(). This function removes the `key` and its associated value from the map while preserving the insertion order of other key-value pairs. It returns the removed value if the map contained the specified `key`. Otherwise, it returns na.

To remove all key-value pairs from an `id` map at once, use map.clear().

The following script creates a new `m` map, puts key-value pairs into the map, uses a map.remove() method call within a loop to remove each valid `key` listed in the `removeKeys` array, then calls map.clear() as a method to remove all remaining key-value pairs. The script uses a custom `debugLabel()` method to display the map.size(), map.keys(), and map.values() results for the map after each change:

//@version=6 indicator("Removing key-value pairs demo") //@function Returns a label to display the keys and values from a map. method debugLabel(map<string, int> this, int barIndex = bar_index, color bgColor =

color.blue, string note = "") => //@variable A string representing the size, keys, and values in `this` map. string repr = str.format("{0}\nSize: {1}\nKeys: {2}\nValues: {3}", note, this.size(), str.tostring(this.keys()), str.tostring(this.values())) label.new(barIndex, 0, repr, color = bgColor, style = label.style_label_center, textcolor = color.white, size = size.huge) if bar_index == last_bar_index - 1 //@variable A map containing `string` keys and `int` values. m = map.new<string, int>() // Put key-value pairs into `m`. for [i, key] in array.from("A", "B", "C", "D", "E") m.put(key, i) m.debugLabel(bar_index, color.green, "Added pairs") //@variable An array of keys to remove from `m`. array<string> removeKeys = array.from("B", "B", "D", "F", "a") // Remove each `key` in `removeKeys` from `m`. for key in removeKeys m.remove(key) m.debugLabel(bar_index + 10, color.red, "Removed pairs") // Remove all remaining keys from `m`. m.clear() m.debugLabel(bar_index + 20, color.purple, "Cleared the map")

Note that:

- Not all strings in the `removeKeys` array are present in the keys of the map. Attempting to remove non-existent keys (“F”, “a”, and the second “B” in this example) has no effect on a map's contents.

Combining maps

Scripts can combine two maps via map.put_all(). This function puts *all* key-value pairs from the `id2` map, in their insertion order, into the `id1` map. As with map.put(), if any keys in `id2` are also present in `id1`, this function **replaces** the key-value pairs that contain those keys without affecting their initial insertion order.

This example contains a user-defined `hexMap()` function that maps decimal int keys to string representations of their hexadecimal forms. The script uses this function to create two maps, `mapA` and `mapB`, then calls map.put_all() as a method to put all key-value pairs from `mapB` into `mapA`.

The script uses a custom `debugLabel()` function to display labels showing the keys and values of `mapA` and `mapB`, then another label displaying the contents of `mapA` after putting all key-value pairs from `mapB` into it:

//@version=6 indicator("Combining maps demo", "Hex map") //@variable An array of string hex digits. var array<string> hexDigits = str.split("0123456789ABCDEF", "") //@function Returns a hexadecimal string for the specified `value`. hex(int value) => //@variable A string representing the hex form of the `value`. string result = "" //@variable A temporary value for digit calculation. int tempValue = value while tempValue > 0 //@variable The next integer digit. int digit = tempValue % 16 // Add the hex form of the `digit` to the `result`. result := hexDigits.get(digit) + result // Divide the `tempValue` by the base. tempValue := int(tempValue / 16) result //@function Returns a map holding the `numbers` as keys and their `hex` strings as values. hexMap(array<int> numbers) => //@variable A map associating `int` keys with `string` values. result = map.new<int, string>() for number in numbers // Put a pair containing the `number` and its `hex()` representation into the `result`. result.put(number, hex(number)) result //@function Returns a label to display the keys and values of a hex map. debugLabel(map<int, string> this, int barIndex = bar_index, color bgColor = color.blue, string style = label.style_label_center, string note = "") => string repr = str.format("{0}\nDecimal: {1}\nHex: {2}", note, str.tostring(this.keys()), str.tostring(this.values())) label.new(barIndex, 0, repr, color = bgColor, style = style, textcolor = color.white, size = size.huge) if bar_index == last_bar_index - 1 //@variable A map with decimal `int` keys and hexadecimal `string` values. map<int, string> mapA = hexMap(array.from(101, 202, 303, 404)) debugLabel(mapA, bar_index, color.navy, label.style_label_down, "A") //@variable A map containing key-value pairs to add to `mapA`. map<int, string> mapB = hexMap(array.from(303, 404, 505, 606, 707, 808)) debugLabel(mapB, bar_index, color.maroon, label.style_label_up, "B") // Put all pairs from `mapB` into `mapA`. mapA.put_all(mapB) debugLabel(mapA, bar_index + 10, color.purple, note = "Merge B into A")

Looping through a map

There are several ways scripts can iteratively access the keys and values in a map. For example, one could loop through the map.keys() array and use map.get() the value for each key, like so:

for key in thisMap.keys() value = thisMap.get(key)

However, we recommend using a `for…in` loop directly on a map, as it iterates over the map's key-value pairs in their insertion order, returning a tuple containing the next pair's key and value on each iteration.

For example, this line of code loops through each key and value in `thisMap`, starting from the first key-value pair put into it:

for [key, value] in thisMap

Let's use this structure to write a script that displays a map's key-value pairs in a table. In the example below, we've defined a custom `toTable()` method that creates a table, then uses a `for...in` loop to iterate over the map's key-value pairs and populate the table's cells. The script uses this method to visualize a map containing `length`-bar averages of price and volume data:

//@version=6 indicator("Looping through a map demo", "Table of averages") //@variable The length of the moving average. int length = input.int(20, "Length") //@variable The size of the table text. string txtSize = input.string(size.huge, "Text size", options = [size.auto, size.tiny, size.small, size.normal, size.large, size.huge]) //@function Displays the pairs of `this` map within a table. //@param this A map with `string` keys and `float` values. //@param position The position of the table on the chart. //@param header The string to display on the top row of the table. //@param textSize The size of the text in the table. //@returns A new `table` object with cells displaying each pair in `this`. method toTable(map<string, float> this, string position = position.middle_center, string header = na, string textSize = size.huge) => // Color variables borderColor = #000000 headerColor = color.rgb(1, 88, 80) pairColor = color.maroon textColor = color.white //@variable A table that displays the key-value pairs of `this` map. table result = table.new(position, this.size() + 1, 3, border_width = 2, border_color = borderColor) // Initialize top and side header cells. result.cell(1, 0, header, bgcolor = headerColor, text_color = textColor, text_size = textSize) result.merge_cells(1, 0, this.size(), 0) result.cell(0, 1, "Key", bgcolor = headerColor, text_color = textColor, text_size = textSize) result.cell(0, 2, "Value", bgcolor = headerColor, text_color = textColor, text_size = textSize) //@variable The column index of the table. Updates on each loop iteration. int col = 1 // Loop over each `key` and `value` from `this` map in the insertion order. for [key, value] in this // Initialize a `key` cell in the `result` table on row 1. result.cell(col, 1, str.tostring(key), bgcolor = color.maroon, text_color = color.white, text_size = textSize) // Initialize a `value` cell in the `result` table on row 2. result.cell(col, 2, str.tostring(value), bgcolor = color.maroon, text_color = color.white, text_size = textSize) // Move to the next column index. col += 1 result // Return the `result` table. //@variable A map with `string` keys and `float` values to hold `length`-bar averages. averages = map.new<string, float>() // Put key-value pairs into the `averages` map. averages.put("Open", ta.sma(open, length)) averages.put("High", ta.sma(high, length)) averages.put("Low", ta.sma(low, length)) averages.put("Close", ta.sma(close, length)) averages.put("Volume", ta.sma(volume, length)) //@variable The text to display at the top of the table. string headerText = str.format("{0} {1}-bar averages", "" + syminfo.tickerid + "", length) // Display the `averages` map in a `table` with the `headerText`. averages.toTable(header = headerText, textSize = txtSize)

Copying a map

Shallow copies

Scripts can make a *shallow copy* of an `id` map by using the map.copy() function. Modifications to a shallow copy do not affect the original `id` map or its internal insertion order.

For example, this script constructs an `m` map with the keys “A”, “B”, “C”, and “D” assigned to four random values between 0 and 10. It then creates an `mCopy` map as a shallow copy of the `m` map and updates the values associated with its keys. The script displays the key-value pairs from the two maps on the chart using our custom `debugLabel()` method:

//@version=6 indicator("Shallow copy demo") //@function Displays the key-value pairs of `this` map in a label. method debugLabel(map<string, float> this, int barIndex = bar_index, color bgColor = color.blue, color textColor = color.white, string note = "") => //@variable The text to display in the label. labelText = note + "\n{" for [key, value] in this labelText += str.format("{0}: {1}, ", key, value) labelText := str.replace(labelText, ", ", "}", this.size() - 1) if barstate.ishistory label result = label.new(barIndex, 0, labelText, color = bgColor, style = label.style_label_center, textcolor = textColor, size = size.huge) if bar_index == last_bar_index - 1 //@variable A map of `string` keys and pseudorandom `float` values. m = map.new<string, float>() // Assign pseudorandom values to an array of keys in `m`. for key in array.from("A", "B", "C", "D") m.put(key, math.random(0, 10)) //@variable A shallow copy of `m`. mCopy = m.copy() // Assign the insertion order value `i` to each `key` in `mCopy`. for [i, key] in mCopy.keys() mCopy.put(key, i) // Display the labels. m.debugLabel(bar_index, note = "Original") mCopy.debugLabel(bar_index + 10, color.purple, note = "Copied and changed")

Deep copies

While a shallow copy will suffice when copying maps that have values of a fundamental type or enum type, it’s crucial to understand that shallow copies of a map with elements of a *reference type* (line, linefill, box, polyline, label, table, chart.point or a UDT) point to the same objects as the original. Modifying the objects referenced by a shallow copy’s elements affect the objects referenced by the original map and vice versa.

To ensure changes to objects referenced by a copied map do not affect instances referenced in other locations, one can make a *deep copy* by creating a new map with key-value pairs containing copies of each value in the original map.

This example creates an `original` map to store string key elements and label value elements, then puts a key-value pair into it. The script copies the map to a `shallow` variable via the built-in map.copy() method, then to a `deep` variable using a custom `deepCopy()` method.

As we see from the chart, changes to the label retrieved from the `shallow` copy also affect the instance referenced by the `original` map, but changes to the one from the `deep` copy do not:

//@version=6 indicator("Deep copy demo") //@function Returns a deep copy of `this` map. method deepCopy(map<string, label> this) => //@variable A deep copy of `this` map. result = map.new<string, label>() // Add key-value pairs with copies of each `value` to the `result`. for [key, value] in this result.put(key, value.copy()) result //Return the `result`. //@variable A map containing `string` keys and `label` values. var original = map.new<string, label>() if bar_index == last_bar_index - 1 // Put a new key-value pair into the `original` map. map.put(original, "Test", label.new(bar_index, 0, "Original", textcolor = color.white, size = size.huge)) //@variable A shallow copy of the `original` map. map<string, label> shallow = original.copy() //@variable A deep copy of the `original` map. map<string, label> deep = original.deepCopy() //@variable The "Test" label from the `shallow` copy. label shallowLabel = shallow.get("Test") //@variable The "Test"

label from the `deep` copy. label deepLabel = deep.get("Test") // Modify the "Test" label's `y` attribute in the `original` map. // This also affects the `shallowLabel`. original.get("Test").set_y(label.all.size()) // Modify the `shallowLabel`. Also modifies the "Test" label in the `original` map. shallowLabel.set_text("Shallow copy") shallowLabel.set_color(color.red) shallowLabel.set_style(label.style_label_up) // Modify the `deepLabel`. Does not modify any other label instance. deepLabel.set_text("Deep copy") deepLabel.set_color(color.navy) deepLabel.set_style(label.style_label_left) deepLabel.set_x(bar_index + 5)

Note that:

- The `deepCopy()` method loops through the `original` map, copying each `value` and putting key-value pairs containing the copies into a new map instance.

Scope and history

As with other collections in Pine, map variables leave historical trails on each bar, allowing a script to access past map instances assigned to a variable using the history-referencing operator []. Scripts can also assign maps to global variables and interact with them from the scopes of user-defined functions, methods, and conditional structures.

As an example, this script uses a global map and its history to calculate an aggregate set of EMAs. It declares a `globalData` map of int keys and float values, where each key in the map corresponds to the length of each EMA calculation. The user-defined `update()` function calculates each `key`-length EMA by mixing the values from the `previous` map assigned to `globalData` with the current `source` value.

The script plots the maximum and minimum values in the global map's map.values() array and the value from `globalData.get(50)` (i.e., the 50-bar EMA):

//@version=6 indicator("Scope and history demo", overlay = true) //@variable The source value for EMA calculation. float source = input.source(close, "Source") //@variable A map containing global key-value pairs. globalData = map.new<int, float>() //@function Calculates a set of EMAs and updates the key-value pairs in `globalData`. update() => //@variable The previous map instance assigned to `globalData`. map<int, float> previous = globalData[1] // Put key-value pairs with keys 10-200 into `globalData` if `previous` is `na`. if na(previous) for i = 10 to 200 globalData.put(i, source) else // Iterate each `key` and `value` in the `previous` map. for [key, value] in previous //@variable The smoothing parameter for the `key`-length EMA. float alpha = 2.0 / (key + 1.0) //@variable The `key`-length EMA value. float ema = (1.0 - alpha) * value + alpha * source // Put the `key`-length `ema` into the `globalData` map. globalData.put(key, ema) // Update the `globalData` map. update() //@variable The array of values from `globalData` in their insertion order. array<float> values = globalData.values() // Plot the max EMA, min EMA, and 50-bar EMA values. plot(values.max(), "Max EMA", color.green, 2) plot(values.min(), "Min EMA", color.red, 2) plot(globalData.get(50), "50-bar EMA", color.orange, 3)

Maps of other collections

Maps cannot directly store references to other maps, arrays, or matrices as value elements, but they can hold references to objects of a user-defined type whose fields can reference other collections.

For example, suppose we want to create a “2D” map that uses string keys to access *nested maps* that hold pairs of string keys and float values. Since maps cannot directly reference other collections, we will first create a *wrapper type* with a field to reference a `map<string, float>` instance, like so:

//@type A wrapper type for maps with `string` keys and `float` values. type Wrapper map<string, float> data

With our `Wrapper` type defined, we can create maps containing string keys and `Wrapper` references as values, where the `data` field of each object referenced by the map points to a `map<string, float>` instance:

mapOfMaps = map.new<string, Wrapper>()

The script below uses this concept to construct a map to reference other maps that hold OHLCV data requested from multiple tickers. The user-defined `requestData()` function requests price and volume data from a ticker, creates a `<string, float>` map, puts the data into it, then returns a `Wrapper` instance containing the new map.

The script puts the results from each call to `requestData()` into the map referenced by the `mapOfMaps` variable, creates a string representation of the nested maps with a user-defined `toString()` method, then displays the string on the chart using a label:

//@version=6 indicator("Nested map demo") //@variable The timeframe of the requested data. string tf = input.timeframe("D", "Timeframe") // Symbol inputs. string symbol1 = input.symbol("EURUSD", "Symbol 1") string symbol2 = input.symbol("GBPUSD", "Symbol 2") string symbol3 = input.symbol("EURGBP", "Symbol 3") //@type A wrapper type for maps with `string` keys and `float` values. type Wrapper map<string, float> data //@function Returns a wrapped map containing OHLCV data from the `tickerID` at the `timeframe`. requestData(string tickerID, string timeframe) => // Request a tuple of OHLCV values from the specified ticker and timeframe. [o, h, l, c, v] = request.security(tickerID, timeframe, [open, high, low, close, volume]) //@variable A map containing requested OHLCV data. result = map.new<string, float>() // Put key-value pairs into the `result`. result.put("Open", o) result.put("High", h) result.put("Low", l) result.put("Close", c) result.put("Volume", v) //Return the wrapped `result`. Wrapper.new(result) //@function Returns a string representing `this` map of `string` keys and `Wrapper` values. method toString(map<string, Wrapper> this) => //@variable A string representation of `this` map. string result = "{" // Iterate over each `key1` and associated `wrapper` in `this`. for [key1, wrapper] in this // Add `key1` to the `result`. result += key1 //@variable A string representation of the `wrapper.data` map. string innerStr = ": {" // Iterate over each `key2` and associated `value` in the wrapped map. for [key2, value] in wrapper.data // Add the key-value pair's representation to `innerStr`. innerStr += str.format("{0}: {1}, ", key2, str.tostring(value)) // Replace the end of `innerStr` with "}" and add to `result`. result += str.replace(innerStr, ", ", "},\n", wrapper.data.size() - 1) // Replace the blank line at the end of `result` with "}". result := str.replace(result, ",\n", "}", this.size() - 1) result //@variable A map of wrapped maps containing OHLCV data from multiple tickers. var mapOfMaps = map.new<string, Wrapper>() //@variable A label showing the contents of the `mapOfMaps`. var debugLabel = label.new(bar_index, 0, color = color.navy, textcolor = color.white, size = size.huge, style = label.style_label_center, text_font_family = font.family_monospace) // Put wrapped maps into `mapOfMaps`. mapOfMaps.put(symbol1, requestData(symbol1, tf)) mapOfMaps.put(symbol2, requestData(symbol2, tf)) mapOfMaps.put(symbol3, requestData(symbol3, tf)) // Update the label. debugLabel.set_text(mapOfMaps.toString()) debugLabel.set_x(bar_index)

ADVANCED

Matrices

> TipThis page contains *advanced* material. If you're new to Pine Script®, start by learning about core language components – such as the type system and the basics of the execution model – and explore other, more accessible features before venturing further.

Introduction

Pine Script *matrices* are collections that store values or references in a rectangular format. They are the equivalent of two-dimensional arrays with functions and methods for inspection, modification, and advanced calculations. As with arrays, all elements within a matrix must be of the same built-in type, user-defined type, or enum type.

Matrices store elements using *two* separate indices. One index specifies which *row* contains the element, and the other specifies the element's *column*. Both indices start at 0 and extend to one less than the total number of rows/columns in the matrix. Matrices in Pine can have dynamic numbers of rows and columns that vary across bars. The total number of elements within a matrix is the *product* of the number of rows and columns it contains (e.g., a five-row, five-column matrix contains 25 elements). Similar to arrays, the total number of elements in a matrix cannot exceed 100,000.

> NoteWe often use the format "MxN matrix" as a shorthand for "M-row, N-column matrix". For example, the phrase "5x5 matrix" refers to a matrix with five rows and five columns.

Declaring a matrix

Pine Script uses the following syntax for matrix declarations:

```
[var/varip ][matrix<type> ]<identifier> = <expression>
```

Where `<type>` is a *type template* that defines the type of elements that the matrix can contain, and `<expression>` is an expression that returns either the *reference (ID)* of a matrix or na. See the Collections section of the Type system page to learn about type templates.

When initializing a matrix variable using na, programmers must prefix the variable declaration with the matrix keyword followed by a type template to explicitly define the variable's *type identifier* (e.g., `matrix<int>` for a variable that can reference a matrix of "int" values).

The following line of code declares a `myMatrix` variable with an initial reference of na. It uses `matrix<float>` as the type identifier, telling the compiler that the variable can accept the ID of a matrix containing "float" elements:

matrix<float> myMatrix = na

If a matrix variable is not initialized with na, specifying a type identifier is optional, because the compiler can automatically determine the variable's accepted type from the assigned matrix ID.

This code line declares a `myMatrix` variable to store the result of a call to `matrix.new<float>()`. The call creates a two-row, two-column "float" matrix with all initial elements set to 0, and then returns the ID of that matrix. An explicit type identifier is optional in this variable declaration, because the compiler uses the assigned ID to determine that the variable's type is `matrix<float>`:

myMatrix = matrix.new<float>(2, 2, 0.0)

Using `var` and `varip` keywords

As with other variables, users can include the var or varip keywords to instruct a script to declare a matrix variable on only one bar instead of on every execution of the variable's scope. A matrix variable declared with this keyword points to the same instance throughout the span of the chart unless the script explicitly assigns another matrix reference to it. This behavior allows a matrix and its elements to persist across bars.

This script declares an `m` variable to reference a matrix that holds a single row of two int elements using the var keyword. On every 20th bar, the script adds 1 to the first element on the first row of the `m` matrix. The plot() call displays this element's value on the chart. As we see from the plot, the value of the matrix.get() method call persists across bars, never returning to the initial value of 0:

//@version=6 indicator("var matrix demo") //@variable A 1x2 rectangular matrix declared only at `bar_index == 0`, i.e., the first bar. var m = matrix.new<int>(1, 2, 0) //@variable Is `true` on every 20th bar. bool update = bar_index % 20 == 0 if update int currentValue = m.get(0, 0) // Get the current value of the first row and column. m.set(0, 0, currentValue + 1) // Set the first row and column element value to `currentValue + 1`. plot(m.get(0, 0), linewidth = 3) // Plot the value from the first row and column.

> Notice
>
> Matrix variables declared using varip behave similarly to those declared using var, with two key differences. Firstly, the matrices that they reference can finalize updates to their elements on *any* available tick – not only on a bar's closing tick. Secondly, matrices referenced by varip variables can contain only the following data:
>
> - Values of any fundamental type.
> - IDs of the chart.point, footprint, or volume_row type.
> - References to objects of a user-defined type that have fields for storing only data of either of the above types or the IDs of other collections containing only these types.

Reading and writing matrix elements

`matrix.get()` and `matrix.set()`

To retrieve the value from a matrix at a specified `row` and `column` index, use matrix.get(). This function locates the specified matrix element and returns its value. Similarly, to overwrite a specific element's value, use matrix.set() to assign the element at the specified `row` and `column` to a new `value`.

The example below defines a square matrix `m` with two rows and columns and an `initial_value` of 0 for all elements on the first bar. The script adds 1 to each element's value on different bars using matrix.get() and matrix.set() method calls. It updates the first row's first value once every 11 bars, the first row's second value once every seven bars, the second row's first value once every five bars, and the second row's second value once every three bars. The script plots each element's value on the chart:

//@version=6 indicator("Reading and writing elements demo") //@variable A 2x2 square matrix of `float` values. var m = matrix.new<float>(2, 2, 0.0) switch bar_index % 11 == 0 => m.set(0, 0, m.get(0, 0) + 1.0) // Adds 1 to the value at row 0, column 0 every 11th bar. bar_index % 7 == 0 => m.set(0, 1, m.get(0, 1) + 1.0) // Adds 1 to the value at row 0, column 1 every 7th bar. bar_index % 5 == 0 => m.set(1, 0, m.get(1, 0) + 1.0) // Adds 1 to the value at row 1, column 0 every 5th bar. bar_index % 3 == 0 => m.set(1, 1, m.get(1, 1) + 1.0) // Adds 1 to the value at row 1, column 1 every 3rd bar. plot(m.get(0, 0), "Row 0, Column 0 Value", color.red, 2) plot(m.get(0, 1), "Row 0, Column 1 Value", color.orange, 2) plot(m.get(1, 0), "Row 1, Column 0 Value", color.green, 2) plot(m.get(1, 1), "Row 1, Column 1 Value", color.blue, 2)

`matrix.fill()`

To overwrite all matrix elements with a specific value, use matrix.fill(). This function points all items in the entire matrix or within the `from_row/column` and `to_row/column` index range to the `value` specified in the call. For example, this snippet declares a 4x4 square matrix, then fills its elements with the result of a math.random() call:

myMatrix = matrix.new<float>(4, 4) myMatrix.fill(math.random())

Note when using matrix.fill() with matrices of *reference types* (line, linefill, box, polyline, label, table, or chart.point) or UDTs, all replaced elements will point to the same object passed in the function call.

This script declares a matrix with four rows and columns of label references, which it fills with a new label reference on the first bar. On each bar, the script sets the `x` property of the label referenced at row 0, column 0 to bar_index, and the `text` property of the one referenced at row 3, column 3 to the number of labels on the chart. Although the matrix can reference 16 (4x4) labels, each element refers to the *same* label object, resulting in only one label on the chart with coordinates and displayed text that update on each bar:

//@version=6 indicator("Object matrix fill demo") //@variable A 4x4 label matrix. var matrix<label> m = matrix.new<label>(4, 4) // Fill `m` with a new label object on the first bar. if bar_index == 0 m.fill(label.new(0, 0, textcolor = color.white, size = size.huge)) //@variable The number of label objects on the chart. int numLabels = label.all.size() // Set the `x` of the label from the first row and column to `bar_index`. m.get(0, 0).set_x(bar_index) // Set the `text` of the label at the last row and column to the number of labels. m.get(3, 3).set_text(str.format("Total labels on the chart: {0}", numLabels))

Rows and columns

Retrieving

Scripts can retrieve all the data from a specific row or column in a matrix via the matrix.row() and matrix.col() functions. These functions return the row or column contents as an array sized according to the other dimension of the matrix. The size of a matrix.row() array equals the number of columns (matrix.columns()), and the size of a matrix.col() array equals the number of rows matrix.rows().

The script below populates a 3x2 `m` matrix with the values 1 - 6 on the first chart bar. It uses matrix.row() and matrix.col() method calls to access the first row and column arrays from the matrix and displays them on the chart in a label along with the array sizes:

//@version=6 indicator("Retrieving rows and columns demo") //@variable A 3x2 rectangular matrix. var matrix<float> m = matrix.new<float>(3, 2) if bar_index == 0 m.set(0, 0, 1.0) // Set row 0, column 0 value to 1. m.set(0, 1, 2.0) // Set row 0, column 1 value to 2. m.set(1, 0, 3.0) // Set row 1, column 0 value to 3. m.set(1, 1, 4.0) // Set row 1, column 1 value to 4. m.set(2, 0, 5.0) // Set row 2, column 0 value to 5. m.set(2, 1, 6.0) // Set row 2, column 1 value to 6. //@variable The first row of the matrix. array<float> row0 = m.row(0) //@variable The first column of the matrix. array<float> column0 = m.col(0) //@variable Displays the first row and column of the matrix and their sizes in a label. var label debugLabel = label.new(0, 0, color = color.blue, textcolor = color.white, size = size.huge) debugLabel.set_x(bar_index) debugLabel.set_text(str.format("Row 0: {0}, Size: {1}\nCol 0: {2}, Size: {3}", row0, m.columns(), column0, m.rows()))

Note that:

- To get the sizes of the arrays displayed in the label, we used the matrix.rows() and matrix.columns() methods rather than array.size() to demonstrate that the size of the `row0` array equals the number of matrix columns and the size of the `column0` array equals the number of matrix rows.

The matrix.row() and matrix.col() functions copy the contents of a row/column to a new array. Modifications to the arrays returned by these functions do not directly affect the elements or the shape of a matrix.

Here, we've modified the previous script to set the first element of `row0` to 10 via the array.set() method before displaying the label. This script also plots the value from row 0, column 0. As we see, the label shows that the first element of the `row0` array is 10. However, the plot shows that the corresponding matrix element still has a value of 1:

//@version=6 indicator("Retrieving rows and columns demo") //@variable A 3x2 rectangular matrix. var matrix<float> m = matrix.new<float>(3, 2) if bar_index == 0 m.set(0, 0, 1.0) // Set row 0, column 0 value to 1. m.set(0, 1, 2.0) // Set row 0, column 1 value to 2. m.set(1, 0, 3.0) // Set row 1, column 0 value to 3. m.set(1, 1, 4.0) // Set row 1, column 1 value to 4. m.set(2, 0, 5.0) // Set row 1, column 0 value to 5. m.set(2, 1, 6.0) // Set row 1, column 1 value to 6. //@variable The first row of the matrix. array<float> row0 = m.row(0) //@variable The first column of the matrix. array<float> column0 = m.col(0) // Set the first `row` element to 10. row0.set(0, 10) //@variable Displays the first row and column of the matrix and their sizes in a label. var label debugLabel = label.new(0, m.get(0, 0), color = color.blue, textcolor = color.white, size = size.huge) debugLabel.set_x(bar_index) debugLabel.set_text(str.format("Row 0: {0}, Size: {1}\nCol 0: {2}, Size: {3}", row0, m.columns(), column0, m.rows())) // Plot the first element of `m`. plot(m.get(0, 0), linewidth = 3)

Although changes to an array constructed from matrix.row() or matrix.col() do not directly affect a parent matrix, it's important to note the resulting array from a matrix containing UDTs or special types, including line, linefill, box, polyline, label, table, or chart.point, behaves as a *shallow copy* of a row/column, i.e., the elements within an array returned from these functions reference the same objects as the corresponding matrix elements.

This script contains a custom `myUDT` type containing a `value` field with an initial value of 0. It declares a 1x1 `m` matrix to hold a single `myUDT` instance on the first bar, then calls `m.row(0)` to copy the first row of the matrix as an array. On every chart bar, the script adds 1 to the `value` field of the first `row` array element. In this case, the `value` field of the matrix element increases on every bar as well, because both elements refer to the same object:

//@version=6 indicator("Row with reference types demo") //@type A custom type that holds a float value. type myUDT float value = 0.0 //@variable A 1x1 matrix of `myUDT` type. var matrix<myUDT> m = matrix.new<myUDT>(1, 1, myUDT.new()) //@variable A shallow copy of the first row of `m`. array<myUDT> row = m.row(0) //@variable The first element of the `row`. myUDT firstElement = row.get(0) firstElement.value += 1.0 // Add 1 to the `value` field of `firstElement`. Also affects the element in the matrix. plot(m.get(0, 0).value, linewidth = 3) // Plot the `value` of the `myUDT` object from the first row and column of `m`.

Inserting

Scripts can add new rows and columns to a matrix via matrix.add_row() and matrix.add_col(). These functions insert the values or references from an array into a matrix at the specified `row/column` index. If the `id` matrix is empty (has no rows or columns), the array referenced by `array_id` in the call can be of any size. If a row/column exists at the specified index, the matrix increases the index value for the existing row/column and all after it by one.

The script below declares an empty `m` matrix and inserts rows and columns by calling matrix.add_row() and matrix.add_col() as methods. It first inserts an array with three elements at row 0, turning `m` into a 1x3 matrix, then another at row 1, changing the shape to 2x3. After that, the script inserts another array at row 0, which changes the shape of `m` to 3x3 and shifts the index of all rows previously at index 0 and higher. It inserts another array at the last column index, changing the shape to 3x4. Finally, it adds an array with four values at the end row index.

The resulting matrix has four rows and columns and contains values 1-16 in ascending order. The script displays the rows of the matrix after each row/column insertion with a user-defined `debugLabel()` function to visualize the process:

//@version=6 indicator("Rows and columns demo") //@function Displays the rows of a matrix in a label with a note. //@param this The matrix to display. //@param barIndex The `bar_index` to display the label at. //@param bgColor The background color of the label. //@param textColor The color of the label's text. //@param note The text to display above the rows. method debugLabel(matrix<float> this, int barIndex = bar_index, color bgColor = color.blue, color textColor = color.white, string note = "") => labelText = note + "\n" + str.tostring(this) if barstate.ishistory label.new(barIndex, 0, labelText, color = bgColor, style = label.style_label_center, textcolor = textColor, size = size.huge) //Create an empty matrix. var m = matrix.new<float>() if bar_index == last_bar_index - 1 debugLabel(m, bar_index - 30, note = "Empty matrix") // Insert an array at row 0. `m` will now have 1 row and 3 columns. m.add_row(0, array.from(5, 6, 7)) debugLabel(m, bar_index - 20, note = "New row at\nindex 0") // Insert an array at row 1. `m` will now have 2 rows and 3 columns. m.add_row(1, array.from(9, 10, 11)) debugLabel(m, bar_index - 10, note = "New row at\nindex 1") // Insert another array at row 0. `m` will now have 3 rows and 3 columns. // The values previously on row 0 will now be on row 1, and the values from row 1 will be on row 2. m.add_row(0, array.from(1, 2, 3)) debugLabel(m, bar_index, note = "New row at\nindex 0") // Insert an array at column 3. `m` will now have 3 rows and 4 columns. m.add_col(3, array.from(4, 8, 12)) debugLabel(m, bar_index + 10, note = "New column at\nindex 3") // Insert an array at row 3. `m` will now have 4 rows and 4 columns. m.add_row(3, array.from(13, 14, 15, 16)) debugLabel(m, bar_index + 20, note = "New row at\nindex 3")

> NoticeJust as the row or column arrays retrieved from a matrix of line, linefill, box, polyline, label, table, chart.point, or UDT references behave as shallow copies, the elements of matrices containing such types

> refer to the same objects as the arrays inserted into them. Modifications to the objects referenced by the elements in either collection affect the objects referenced by the other collection in such cases.

Removing

To remove a specific row or column from a matrix, use matrix.remove_row() and matrix.remove_col(). These functions remove the specified row/column and decrease the index values of all rows/columns after it by one.

For this example, we've added these lines of code to our “Rows and columns demo” script from the Inserting section above:

// Removing example // Remove the first row and last column from the matrix. `m` will now have 3 rows and 3 columns. m.remove_row(0) m.remove_col(3) debugLabel(m, bar_index + 30, color.red, note = "Removed row 0\nand column 3")

This code removes the first row and the last column of the `m` matrix using matrix.remove_row() and matrix.remove_col() method calls, then displays the rows in a label at `bar_index + 30`. As we can see, the matrix has a 3x3 shape after the script executes this block, and the index values for all existing rows are reduced by 1:

Swapping

To swap the rows and columns of a matrix without altering its dimensions, use matrix.swap_rows() and matrix.swap_columns(). These functions swap the positions of the elements at the `row1/column1` and `row2/column2` indices.

Let's add another set of code lines to the example from the removing section. The following lines swap the first and last rows of the `m` matrix and display the changes in a label at `bar_index + 40`:

// Swapping example // Swap the first and last row. `m` retains the same dimensions. m.swap_rows(0, 2) debugLabel(m, bar_index + 40, color.purple, note = "Swapped rows 0\nand 2")

In the new label, we see the matrix has the same number of rows as before, and the first and last rows have traded places:

Replacing

It may be desirable in some cases to completely *replace* a row or column in a matrix. To do so, insert another array’s elements at the desired `row/column` and remove the old elements previously at that index.

In the following code, we've defined a `replaceRow()` method that uses the matrix.add_row() function to insert the new `values` at the `row` index, and the matrix.remove_row() method to remove the old row that moved to the `row + 1` index. This script uses the `replaceRow()` method to fill the rows of a 3x3 matrix with the numbers 1-9. It draws a label on the chart before and after replacing the rows using the custom `debugLabel()` method:

//@version=6 indicator("Replacing rows demo") //@function Displays the rows of a matrix in a label with a note. //@param this The matrix to display. //@param barIndex The `bar_index` to display the label at. //@param bgColor The background color of the label. //@param textColor The color of the label's text. //@param note The text to display above the rows. method debugLabel(matrix<float> this, int barIndex =

bar_index, color bgColor = color.blue, color textColor = color.white, string note = "") => labelText = note + "\n" + str.tostring(this) if barstate.ishistory label.new(barIndex, 0, labelText, color = bgColor, style = label.style_label_center, textcolor = textColor, size = size.huge) //@function Replaces the `row` of `this` matrix with a new array of `values`. //@param row The row index to replace. //@param values The array of values to insert. method replaceRow(matrix<float> this, int row, array<float> values) => this.add_row(row, values) // Inserts a copy of the `values` array at the `row`. this.remove_row(row + 1) // Removes the old elements previously at the `row`. //@variable A 3x3 matrix. var matrix<float> m = matrix.new<float>(3, 3, 0.0) if bar_index == last_bar_index - 1 m.debugLabel(note = "Original") // Replace each row of `m`. m.replaceRow(0, array.from(1.0, 2.0, 3.0)) m.replaceRow(1, array.from(4.0, 5.0, 6.0)) m.replaceRow(2, array.from(7.0, 8.0, 9.0)) m.debugLabel(bar_index + 10, note = "Replaced rows")

Looping through a matrix

`for`

When a script only needs to iterate over the row/column indices in a matrix, the most common method is to use for loops. For example, this line creates a loop with a `row` value that starts at 0 and increases by one until it reaches one less than the number of rows in the `m` matrix (i.e., the last row index):

for row = 0 to m.rows() - 1

To iterate over all index values in the `m` matrix, we can create a *nested* loop that iterates over each `column` index on each `row` value:

for row = 0 to m.rows() - 1 for column = 0 to m.columns() - 1

Let's use this nested structure to create a method that visualizes matrix elements. In the script below, we've defined a `toTable()` method that displays the elements of a matrix within a table object. It iterates over each `row` index and over each `column` index on every `row`. Within the loop, it converts each element to a string to display in the corresponding table cell.

On the first bar, the script creates an empty `m` matrix, populates it with rows, and calls `m.toTable()` to display its elements:

//@version=6 indicator("for loop demo", "Matrix to table") //@function Displays the elements of `this` matrix in a table. //@param this The matrix to display. //@param position The position of the table on the chart. //@param bgColor The background color of the table. //@param textColor The color of the text in each cell. //@param note A note string to display on the bottom row of the table. //@returns A new `table` object with cells corresponding to each element of `this` matrix. method toTable(matrix<float> this, string position = position.middle_center, color bgColor = color.blue, color textColor = color.white, string note = na) => //@variable The number of rows in `this` matrix. int rows = this.rows() //@variable The number of columns in `this` matrix. int columns = this.columns() //@variable A table that displays the elements of `this` matrix with an optional `note` cell. table result = table.new(position, columns, rows + 1, bgColor) // Iterate over each row index of `this` matrix. for row = 0 to rows - 1 // Iterate over each column index of `this` matrix on each `row`. for col = 0 to columns - 1 //@variable The element from `this` matrix at the `row` and `col` index. float element = this.get(row, col) // Initialize the corresponding `result` cell with the `element` value. result.cell(col, row, str.tostring(element), text_color = textColor, text_size = size.huge) // Initialize a merged cell on the bottom row if a `note` is provided. if not na(note) result.cell(0, rows, note, text_color = textColor, text_size = size.huge) result.merge_cells(0, rows, columns - 1, rows) result // Return the `result` table. //@variable A 3x4

matrix of values. var m = matrix.new<float>() if bar_index == 0 // Add rows to `m`. m.add_row(0, array.from(1, 2, 3)) m.add_row(1, array.from(5, 6, 7)) m.add_row(2, array.from(9, 10, 11)) // Add a column to `m`. m.add_col(3, array.from(4, 8, 12)) // Display the elements of `m` in a table. m.toTable()

`for...in`

When a script needs to iterate over and retrieve the rows of a matrix, using the for…in structure is often preferred over the standard for loop. This structure directly references the row arrays in a matrix, making it a more convenient option for such use cases. For example, this line creates a loop that returns the reference of an array representing a row in the `m` matrix on each iteration:

for row in m

The following indicator calculates the moving average of OHLC data with an input `length` and displays the values on the chart. The custom `rowWiseAvg()` method loops through the rows of a matrix using a for…in structure to produce an array containing the array.avg() value for each `row` array.

On the first chart bar, the script creates a new `m` matrix with four rows and `length` columns, which it queues a new column of OHLC data into by calling matrix.add_col() and matrix.remove_col() as methods on each subsequent bar. It uses `m.rowWiseAvg()` to calculate the array of row-wise averages, then it plots the value of each array element on the chart:

//@version=6 indicator("for...in loop demo", "Average OHLC", overlay = true) //@variable The number of terms in the average. int length = input.int(20, "Length", minval = 1) //@function Calculates the average of each matrix row. method rowWiseAvg(matrix<float> this) => //@variable An array with elements corresponding to each row's average. array<float> result = array.new<float>() // Iterate over each `row` of `this` matrix. for row in this // Push the average of each `row` into the `result`. result.push(row.avg()) result // Return the resulting array. //@variable A 4x`length` matrix of values. var matrix<float> m = matrix.new<float>(4, length) // Add a new column containing OHLC values to the matrix. m.add_col(m.columns(), array.from(open, high, low, close)) // Remove the first column. m.remove_col(0) //@variable An array containing averages of `open`, `high`, `low`, and `close` over `length` bars. array<float> averages = m.rowWiseAvg() plot(averages.get(0), "Average Open", color.blue, 2) plot(averages.get(1), "Average High", color.green, 2) plot(averages.get(2), "Average Low", color.red, 2) plot(averages.get(3), "Average Close", color.orange, 2)

Note that:

- The for…in loop structure can also access the *index* value of each row. For example, `for [i, row] in m` creates a tuple containing the `i` row index and the reference of the corresponding `row` array from the `m` matrix on each loop iteration.

Copying a matrix

Shallow copies

Pine scripts can copy matrices via matrix.copy(). This function returns a *shallow copy* of a matrix that does not affect the shape of the original matrix or its contents.

For example, this script assigns a new matrix reference to the `myMatrix` variable and adds two columns. It creates a new `myCopy` matrix from that matrix by calling matrix.copy() as a method, then adds a new row to

the resulting copy. It displays the rows of both matrices in labels via the user-defined `debugLabel()` function:

```
//@version=6 indicator("Shallow copy demo") //@function Displays the rows of a matrix in a label with a note. //@param this The matrix to display. //@param barIndex The `bar_index` to display the label at. //@param bgColor The background color of the label. //@param textColor The color of the label's text. //@param note The text to display above the rows. method debugLabel( matrix<float> this, int barIndex = bar_index, color bgColor = color.blue, color textColor = color.white, string note = "" ) => labelText = note + "\n" + str.tostring(this) if barstate.ishistory label.new( barIndex, 0, labelText, color = bgColor, style = label.style_label_center, textcolor = textColor, size = size.huge ) //@variable A 2x2 `float` matrix. matrix<float> myMatrix = matrix.new<float>() myMatrix.add_col(0, array.from(1.0, 3.0)) myMatrix.add_col(1, array.from(2.0, 4.0)) //@variable A shallow copy of `myMatrix`. matrix<float> myCopy = myMatrix.copy() // Add a row to the last index of `myCopy`. myCopy.add_row(myCopy.rows(), array.from(5.0, 6.0)) if bar_index == last_bar_index - 1 // Display the rows of both matrices in separate labels. myMatrix.debugLabel(note = "Original") myCopy.debugLabel(bar_index + 10, color.green, note = "Shallow Copy")
```

It's important to note that the elements within shallow copies of a matrix have the same values or references as the original matrix. When matrices contain references to special types (line, linefill, box, polyline, label, table, or chart.point) or user-defined types, the elements of a shallow copy reference the same objects as the original matrix.

This script declares a `myMatrix` variable with a `newLabel` as the initial value. It then copies `myMatrix` to a `myCopy` variable by calling the built-in matrix.copy() function in the dot notation form `myMatrix.copy`,and plots the number of labels. As we see below, there's only one label on the chart, as the element in `myCopy` references the same object as the element in `myMatrix`. Consequently, changes to the object referenced in the copied matrix affects the object referenced in the original matrix:

```
//@version=6 indicator("Shallow copy demo") //@variable Initial value of the original matrix elements. var label newLabel = label.new( bar_index, 1, "Original", color = color.blue, textcolor = color.white, size = size.huge ) //@variable A 1x1 matrix containing a new `label` instance. var matrix<label> myMatrix = matrix.new<label>(1, 1, newLabel) //@variable A shallow copy of `myMatrix`. var matrix<label> myCopy = myMatrix.copy() //@variable The first label from the `myCopy` matrix. label testLabel = myCopy.get(0, 0) // Change the `text`, `style`, and `x` values of `testLabel`. Also affects the `newLabel`. testLabel.set_text("Copy") testLabel.set_style(label.style_label_up) testLabel.set_x(bar_index) // Plot the total number of labels. plot(label.all.size(), linewidth = 3)
```

Deep copies

One can produce a *deep copy* of a matrix (i.e., a matrix whose elements refer to copies of the objects referenced by the original matrix) by explicitly copying each element in the matrix.

Here, we've added a `deepCopy()` user-defined method to our previous script. The method creates a new matrix and uses nested `for` loops to assign all elements to copies of the originals. When the script calls this method instead of matrix.copy(), we see that there are now two labels on the chart, and any changes to the label referenced by the copied matrix do not affect the one referenced by the original matrix:

```
//@version=6 indicator("Deep copy demo") //@function Returns a deep copy of a label matrix. method deepCopy(matrix<label> this) => //@variable A deep copy of `this` matrix. matrix<label> that = this.copy() for row = 0 to that.rows() - 1 for column = 0 to that.columns() - 1 // Assign the element at each `row` and `column` of `that` matrix to a copy of the retrieved label. that.set(row, column, that.get(row, column).copy()) that //@variable Initial value of the original matrix. var label newLabel = label.new( bar_index, 2, "Original",
```

color = color.blue, textcolor = color.white, size = size.huge) //@variable A 1x1 matrix containing a new `label` instance. var matrix<label> myMatrix = matrix.new<label>(1, 1, newLabel) //@variable A deep copy of `myMatrix`. var matrix<label> myCopy = myMatrix.deepCopy() //@variable The first label from the `myCopy` matrix. label testLabel = myCopy.get(0, 0) // Change the `text`, `style`, and `x` values of `testLabel`. Does not affect the `newLabel`. testLabel.set_text("Copy") testLabel.set_style(label.style_label_up) testLabel.set_x(bar_index) // Change the `x` value of `newLabel`. newLabel.set_x(bar_index) // Plot the total number of labels. plot(label.all.size(), linewidth = 3)

Submatrices

In Pine, a *submatrix* is a shallow copy of an existing matrix that only includes the rows and columns specified by the `from_row/column` and `to_row/column` parameters. In essence, it is a sliced copy of a matrix.

For example, the script below creates an `mSub` matrix from the `m` matrix via the matrix.submatrix() method, then calls our user-defined `debugLabel()` function to display the rows of both matrices in labels:

//@version=6 indicator("Submatrix demo") //@function Displays the rows of a matrix in a label with a note. //@param this The matrix to display. //@param barIndex The `bar_index` to display the label at. //@param bgColor The background color of the label. //@param textColor The color of the label's text. //@param note The text to display above the rows. method debugLabel(matrix<float> this, int barIndex = bar_index, color bgColor = color.blue, color textColor = color.white, string note = "") => labelText = note + "\n" + str.tostring(this) if barstate.ishistory label.new(barIndex, 0, labelText, color = bgColor, style = label.style_label_center, textcolor = textColor, size = size.huge) //@variable A 3x3 matrix of values. var m = matrix.new<float>() if bar_index == last_bar_index - 1 // Add columns to `m`. m.add_col(0, array.from(9, 6, 3)) m.add_col(1, array.from(8, 5, 2)) m.add_col(2, array.from(7, 4, 1)) // Display the rows of `m`. m.debugLabel(note = "Original Matrix") //@variable A 2x2 submatrix of `m` containing the first two rows and columns. matrix<float> mSub = m.submatrix(from_row = 0, to_row = 2, from_column = 0, to_column = 2) // Display the rows of `mSub` debugLabel(mSub, bar_index + 10, bgColor = color.green, note = "Submatrix")

Scope and history

Matrix variables leave historical trails on each bar, allowing scripts to use the history-referencing operator [] to interact with past matrix instances previously assigned to a variable. Additionally, scripts can modify matrices assigned to global variables from within the scopes of user-defined functions, methods, and conditional structures.

This script calculates the average ratios of body and wick distances relative to the bar range over `length` bars. It displays the data along with values from `length` bars ago in a table. The user-defined `addData()` function adds columns of current and historical ratios to the a matrix created in the global scope, and the `calcAvg()` function references previous matrices assigned to the `globalMatrix` variable using the [] operator to calculate a matrix of averages:

//@version=6 indicator("Scope and history demo", "Bar ratio comparison") int length = input.int(10, "Length", 1) //@variable A global matrix. matrix<float> globalMatrix = matrix.new<float>() //@function Calculates the ratio of body range to candle range. bodyRatio() => math.abs(close - open) / (high - low) //@function Calculates the ratio of upper wick range to candle range. upperWickRatio() => (high - math.max(open, close)) / (high - low) //@function Calculates the ratio of lower wick range to candle range. lowerWickRatio() => (math.min(open, close) - low) / (high - low) //@function Adds data to the `globalMatrix`. addData() => // Add a new column of data at `column` 0. globalMatrix.add_col(0, array.from(bodyRatio(), upperWickRatio(),

lowerWickRatio())) //@variable The column of `globalMatrix` from index 0 `length` bars ago. array<float> pastValues = globalMatrix.col(0)[length] // Add `pastValues` to the `globalMatrix`, or an array of `na` if `pastValues` is `na`. if na(pastValues) globalMatrix.add_col(1, array.new<float>(3)) else globalMatrix.add_col(1, pastValues) //@function Returns the `length`-bar average of matrices assigned to `globalMatrix` on historical bars. calcAvg() => //@variable The sum historical `globalMatrix` matrices. matrix<float> sums = matrix.new<float>(globalMatrix.rows(), globalMatrix.columns(), 0.0) for i = 0 to length - 1 //@variable The `globalMatrix` matrix `i` bars before the current bar. matrix<float> previous = globalMatrix[i] // Break the loop if `previous` is `na`. if na(previous) sums.fill(na) break // Assign the sum of `sums` and `previous` to `sums`. sums := matrix.sum(sums, previous) // Divide the `sums` matrix by the `length`. result = sums.mult(1.0 / length) // Add data to the `globalMatrix`. addData() //@variable The historical average of the `globalMatrix` matrices. globalAvg = calcAvg() //@variable A `table` displaying information from the `globalMatrix`. var table infoTable = table.new(position.middle_center, globalMatrix.columns() + 1, globalMatrix.rows() + 1, bgcolor = color.navy) // Define value cells. for [i, row] in globalAvg for [j, value] in row color textColor = value > 0.333 ? color.orange : color.gray infoTable.cell(j + 1, i + 1, str.tostring(value), text_color = textColor, text_size = size.huge) // Define header cells. infoTable.cell(0, 1, "Body ratio", text_color = color.white, text_size = size.huge) infoTable.cell(0, 2, "Upper wick ratio", text_color = color.white, text_size = size.huge) infoTable.cell(0, 3, "Lower wick ratio", text_color = color.white, text_size = size.huge) infoTable.cell(1, 0, "Current average", text_color = color.white, text_size = size.huge) infoTable.cell(2, 0, str.format("{0} bars ago", length), text_color = color.white, text_size = size.huge)

Note that:

- The `addData()` and `calcAvg()` functions have no parameters, as they directly interact with the `globalMatrix` and `length` variables declared in the outer scope.
- The `calcAvg()` functions calculates the averages by adding `previous` matrices using matrix.sum() and multiplying all elements by `1 / length` using matrix.mult(). We discuss these and other specialized functions in the Matrix calculations section below.

Inspecting a matrix

The ability to inspect the shape of a matrix and patterns within its elements is crucial, as it helps reveal important information about a matrix and its compatibility with various calculations and transformations. Pine Script includes several built-ins for matrix inspection, including matrix.is_square(), matrix.is_identity(), matrix.is_diagonal(), matrix.is_antidiagonal(), matrix.is_symmetric(), matrix.is_antisymmetric(), matrix.is_triangular(), matrix.is_stochastic(), matrix.is_binary(), and matrix.is_zero().

To demonstrate these features, this example contains a custom `inspect()` method that uses conditional blocks with `matrix.is_*()` functions to return information about a matrix. It displays a string representation of an `m` matrix and the description returned from `m.inspect()` in labels on the chart:

//@version=6 indicator("Matrix inspection demo") //@function Inspects a matrix using `matrix.is_*()` functions and returns a `string` describing some of its features. method inspect(matrix<int> this)=> //@variable A string describing `this` matrix. string result = "This matrix:\n" if this.is_square() result += "- Has an equal number of rows and columns.\n" if this.is_binary() result += "- Contains only 1s and 0s.\n" if this.is_zero() result += "- Is filled with 0s.\n" if this.is_triangular() result += "- Contains only 0s above and/or below its main diagonal.\n" if this.is_diagonal() result += "- Only has nonzero values in its main diagonal.\n" if this.is_antidiagonal() result += "- Only has nonzero values in its main antidiagonal.\n" if this.is_symmetric() result += "- Equals its transpose.\n" if this.is_antisymmetric() result += "- Equals the negative of its transpose.\n" if this.is_identity() result += "- Is the identity matrix.\n" result //@variable A 4x4 identity matrix.

matrix<int> m = matrix.new<int>() // Add rows to the matrix. m.add_row(0, array.from(1, 0, 0, 0)) m.add_row(1, array.from(0, 1, 0, 0)) m.add_row(2, array.from(0, 0, 1, 0)) m.add_row(3, array.from(0, 0, 0, 1)) if bar_index == last_bar_index - 1 // Display the `m` matrix in a blue label. label.new(bar_index, 0, str.tostring(m), color = color.blue, style = label.style_label_right, textcolor = color.white, size = size.huge) // Display the result of `m.inspect()` in a purple label. label.new(bar_index, 0, m.inspect(), color = color.purple, style = label.style_label_left, textcolor = color.white, size = size.huge)

Manipulating a matrix

Reshaping

The shape of a matrix can determine its compatibility with various matrix operations. In some cases, it is necessary to change the dimensions of a matrix without affecting the number of elements or the values they reference, otherwise known as *reshaping*. To reshape a matrix in Pine, use the matrix.reshape() function.

This example demonstrates the results of multiple reshaping operations on a matrix. The initial `m` matrix has a 1x8 shape (one row and eight columns). Through successive calls to the matrix.reshape() method, the script changes the shape of `m` to 2x4, 4x2, and 8x1. It displays each reshaped matrix in a label on the chart using the custom `debugLabel()` method:

//@version=6 indicator("Reshaping example") //@function Displays the rows of a matrix in a label with a note. //@param this The matrix to display. //@param barIndex The `bar_index` to display the label at. //@param bgColor The background color of the label. //@param textColor The color of the label's text. //@param note The text to display above the rows. method debugLabel(matrix<float> this, int barIndex = bar_index, color bgColor = color.blue, color textColor = color.white, string note = "") => labelText = note + "\n" + str.tostring(this) if barstate.ishistory label.new(barIndex, 0, labelText, color = bgColor, style = label.style_label_center, textcolor = textColor, size = size.huge) //@variable A matrix containing the values 1-8. matrix<int> m = matrix.new<int>() if bar_index == last_bar_index - 1 // Add the initial vector of values. m.add_row(0, array.from(1, 2, 3, 4, 5, 6, 7, 8)) m.debugLabel(note = "Initial 1x8 matrix") // Reshape. `m` now has 2 rows and 4 columns. m.reshape(2, 4) m.debugLabel(bar_index + 10, note = "Reshaped to 2x4") // Reshape. `m` now has 4 rows and 2 columns. m.reshape(4, 2) m.debugLabel(bar_index + 20, note = "Reshaped to 4x2") // Reshape. `m` now has 8 rows and 1 column. m.reshape(8, 1) m.debugLabel(bar_index + 30, note = "Reshaped to 8x1")

Note that:

- The order of elements in `m` does not change with each `m.reshape()` call.
- When reshaping a matrix, the product of the `rows` and `columns` arguments must equal the matrix.elements_count() value, as matrix.reshape() cannot change the number of elements in a matrix.

Reversing

One can reverse the order of all elements in a matrix using matrix.reverse(). This function moves the references of an m-by-n matrix `id` at the i-th row and j-th column to the m - 1 - i row and n - 1 - j column.

For example, this script creates a 3x3 matrix containing the values 1-9 in ascending order, then uses the matrix.reverse() method to reverse its contents. It displays the original and modified versions of the matrix in labels on the chart via `m.debugLabel()`:

```
//@version=6 indicator("Reversing demo") //@function Displays the rows of a matrix in a label with a note. //@param this The matrix to display. //@param barIndex The `bar_index` to display the label at. //@param bgColor The background color of the label. //@param textColor The color of the label's text. //@param note The text to display above the rows. method debugLabel( matrix<float> this, int barIndex = bar_index, color bgColor = color.blue, color textColor = color.white, string note = "" ) => labelText = note + "\n" + str.tostring(this) if barstate.ishistory label.new( barIndex, 0, labelText, color = bgColor, style = label.style_label_center, textcolor = textColor, size = size.huge ) //@variable A 3x3 matrix. matrix<float> m = matrix.new<float>() // Add rows to `m`. m.add_row(0, array.from(1, 2, 3)) m.add_row(1, array.from(4, 5, 6)) m.add_row(2, array.from(7, 8, 9)) if bar_index == last_bar_index - 1 // Display the contents of `m`. m.debugLabel(note = "Original") // Reverse `m`, then display its contents. m.reverse() m.debugLabel(bar_index + 10, color.red, note = "Reversed")
```

Transposing

Transposing a matrix is a fundamental operation that flips all rows and columns in a matrix about its *main diagonal* (the diagonal vector of all values in which the row index equals the column index). This process produces a new matrix with reversed row and column dimensions, known as the *transpose*. Scripts can calculate the transpose of a matrix using matrix.transpose().

For any m-row, n-column matrix, the matrix returned from matrix.transpose() will have n rows and m columns. All elements in a matrix at the i-th row and j-th column correspond to the elements in its transpose at the j-th row and i-th column.

This example declares a 2x4 `m` matrix, calculates its transpose by calling matrix.transpose() as a method, then displays strings representing both matrices on the chart using our custom `debugLabel()` method. As we can see below, the transposed matrix has a 4x2 shape, and the rows of the transpose match the columns of the original matrix:

```
//@version=6 indicator("Transpose example") //@function Displays the rows of a matrix in a label with a note. //@param this The matrix to display. //@param barIndex The `bar_index` to display the label at. //@param bgColor The background color of the label. //@param textColor The color of the label's text. //@param note The text to display above the rows. method debugLabel( matrix<float> this, int barIndex = bar_index, color bgColor = color.blue, color textColor = color.white, string note = "" ) => labelText = note + "\n" + str.tostring(this) if barstate.ishistory label.new( barIndex, 0, labelText, color = bgColor, style = label.style_label_center, textcolor = textColor, size = size.huge ) //@variable A 2x4 matrix. matrix<int> m = matrix.new<int>() // Add columns to `m`. m.add_col(0, array.from(1, 5)) m.add_col(1, array.from(2, 6)) m.add_col(2, array.from(3, 7)) m.add_col(3, array.from(4, 8)) //@variable The transpose of `m`. Has a 4x2 shape. matrix<int> mt = m.transpose() if bar_index == last_bar_index - 1 m.debugLabel(note = "Original") mt.debugLabel(bar_index + 10, note = "Transpose")
```

Sorting

Scripts can sort a matrix containing “int”, “float”, or “string” values by using the matrix.sort() function. This function rearranges the *rows* of a matrix in a specified order by comparing the elements in a specified *column*.

The `column` parameter specifies the index of the column to use for sorting. The default value is 0, meaning that a call to the function compares elements in the *first* column by default.

The `order` parameter accepts one of the two `order.*` constants. If the argument is order.ascending (the default), a matrix.sort() call sorts the matrix rows in ascending order based on the values from the given column. If the argument is order.descending, it sorts the rows in descending order instead.

If a matrix contains “int” or “float” elements, a call to the matrix.sort() function sorts the rows in the matrix by comparing a column’s numeric values. If the order is ascending, the row of the column element with the lowest value becomes the first row (at row index 0), and the row of the element with the highest value becomes the last row. The opposite applies when sorting in descending order.

For example, the following script sorts a 3x3 “int” matrix in ascending and descending order using values from a specified column. After each step, it creates a string representation of the matrix and displays the result in a separate label:

//@version=6 indicator("Sorting numeric matrix rows demo") //@variable The index of the column to use for sorting. int colInput = input.int(0, "Column", 0, 2) if barstate.islastconfirmedhistory //@variable References a 3x3 matrix of "int" values. matrix<int> numbers = matrix.new<int>() // Insert a row of elements into the matrix, then reshape it to 3x3. numbers.add_row(0, array.from(3, 2, 4, 1, 9, 6, 7, 8, 9)) numbers.reshape(3, 3) // Draw a label at the current `bar_index` value to display the original structure of the matrix. label.new(bar_index, 0, "Unsorted\norder\n" + str.tostring(numbers), style = label.style_label_center, size = 36) // Sort the matrix rows in ascending order (the default) using the column at the `colInput` index. numbers.sort(colInput) // Draw a label at `bar_index + 10` to display the updated structure. label.new(bar_index + 10, 0, str.format("Ascending\n(column {0})\n{1}", colInput, str.tostring(numbers)), color = color.green, style = label.style_label_center, size = 36) // Sort the matrix rows in *descending* order using the column at the `colInput` index. numbers.sort(colInput, order.descending) // Draw a label at `bar_index + 20` to display the updated structure. label.new(bar_index + 20, 0, str.format("Descending\n(column {0})\n{1}", colInput, str.tostring(numbers)), color = color.red, style = label.style_label_center, size = 36)

If a matrix contains “string” elements, a matrix.sort() call sorts the rows in the matrix by comparing the Unicode values of *individual characters* in the strings from the specified column. The sorting algorithm initially compares the *first* character in each string, then compares subsequent characters as necessary if multiple strings have matching characters at the same position. The rows whose column elements contain leading characters with the lowest Unicode values move to the beginning of the matrix if the order is ascending, or to the end of the matrix if the order is descending.

For example, the script version below sorts a 3x3 matrix of strings in ascending and descending order using the elements from a specified column. As with the previous example, the script draws three labels to show the structure of the matrix after each step:

//@version=6 indicator("Sorting string matrix rows demo") //@variable The index of the column to use for sorting. int colInput = input.int(0, "Column", 0, 2) if barstate.islastconfirmedhistory //@variable References a 3x3 matrix of "string" values. matrix<string> strings = matrix.new<string>() // Insert a row of elements into the matrix, then reshape it to 3x3. strings.add_row(0, array.from("A", "E", "H", "C", "D", "I", "B", "F", "G")) strings.reshape(3, 3) // Draw a label at the current `bar_index` value to display the original structure of the matrix. label.new(bar_index, 0, "Unsorted\norder\n" + str.tostring(strings), style = label.style_label_center, size = 36, textalign = text.align_left, text_font_family = font.family_monospace) // Sort the matrix rows in ascending order (the default) using the column at the `colInput` index. strings.sort(colInput) // Draw a label at `bar_index + 10` to display the updated structure. label.new(bar_index + 10, 0, str.format("Ascending\n(column {0})\n{1}", colInput, str.tostring(strings)), color = color.green, style = label.style_label_center, size = 36, textalign = text.align_left, text_font_family = font.family_monospace) //

Sort the matrix rows in descending order using the column at the `colInput` index. strings.sort(colInput, order.descending) // Draw a label at `bar_index + 20` to display the updated structure. label.new(bar_index + 20, 0, str.format("Descending\n(column {0})\n{1}", colInput, str.tostring(strings)), color = color.red, style = label.style_label_center, size = 36, textalign = text.align_left, text_font_family = font.family_monospace)

Note that:

- This example uses strings containing all *uppercase* ASCII letters. Therefore, the effect of the matrix.sort() calls is the same as sorting column strings in *alphabetical* order. However, if we add *lowercase* characters to the start of some strings, the sorting order would *not* be alphabetical, because all uppercase ASCII letters *precede* lowercase letters in the Unicode Standard. See the Sorting section of the Arrays page for an example of this behavior.

> NoteIf an “int”, “float”, or “string” matrix contains elements with `na` values or empty strings in a column used for sorting, a matrix.sort() call moves the corresponding rows to the *end* of the matrix if the `order` argument is order.ascending, or to the *beginning* if the argument is order.descending.

In some cases, a programmer might need to sort the *columns* of a matrix rather than sorting its rows. Although the matrix.sort() function does not support column-wise sorting directly, programmers can achieve this effect by sorting the transpose of the specified matrix. The steps are as follows:

1. Create a transposed copy of the matrix by calling the matrix.transpose() function. The rows of the original matrix become *columns* in the copy, and the original columns become rows.
2. Sort the rows in the transposed matrix using a matrix.sort() function call.
3. Create a transposed copy of the sorted matrix by using a second matrix.transpose() call. This step changes the sorted rows from step 2 back to columns.

The modified example below adds a user-defined function named `sortColumns()` to the first example script in this section, then replaces the script’s matrix.sort() calls with calls to that function. The `sortColumns()` function performs the above steps to create a copy of the matrix with sorted columns. The script displays strings representing the structure of the original matrix and the sorted results in labels on the last historical bar:

//@version=6 indicator("Sorting matrix columns demo") //@function Creates a copy of a specified matrix with sorted columns. //@param id The ID of the matrix to sort. //@param row The index of the row to use for sorting. //@param ascending Optional. If `true`, the function sorts the copy in ascending order. // If `false`, it sorts in descending order. The default is `true`. //@returns The ID of the sorted copy. sortColumns(matrix<int> id, int row, bool ascending = true) => // Create a transposed copy of the matrix. The rows in the copy correspond to the original columns. matrix<int> t = id.transpose() // Sort the rows of the transposed copy. t.sort(row, ascending ? order.ascending : order.descending) // Create a transposed copy of the sorted matrix and return its ID. t.transpose() //@variable The index of the row to use for sorting. int rowInput = input.int(0, "Row", 0, 2) if barstate.islastconfirmedhistory //@variable References a 3x3 matrix of "int" values. matrix<int> numbers = matrix.new<int>() // Insert a row of elements into the matrix, then reshape it to 3x3. numbers.add_row(0, array.from(3, 2, 4, 1, 9, 6, 7, 8, 9)) numbers.reshape(3, 3) // Draw a label at the current `bar_index` value to display the original structure of the matrix. label.new(bar_index, 0, "Unsorted\norder\n" + str.tostring(numbers), style = label.style_label_center, size = 36) // Sort the matrix columns in ascending order using the row at the `rowInput` index. numbers := sortColumns(numbers,

rowInput, true) // Draw a label at `bar_index + 10` to display the updated structure. label.new(bar_index + 10, 0, str.format("Ascending\n(row {0})\n{1}", rowInput, str.tostring(numbers)), color = color.green, style = label.style_label_center, size = 36) // Sort the matrix columns in descending order using the row at the `rowInput` index. numbers := sortColumns(numbers, rowInput, false) // Draw a label at `bar_index + 20` to display the updated structure. label.new(bar_index + 20, 0, str.format("Descending\n(row {0})\n{1}", rowInput, str.tostring(numbers)), color = color.red, style = label.style_label_center, size = 36)

> TipTo release the original matrix from memory after creating a copy with sorted columns, reassign the copy's ID to the variable or field that stores the ID of the original matrix.

Sorting matrices of user-defined types

The matrix.sort() function can also sort matrices whose elements reference objects of user-defined types (UDTs). For such matrices, the function compares values from one of the "int", "float", or "string" *fields* of each object referenced by the elements in a specified column, using the sorting rules described in the Sorting section above.

The function's `sort_field` parameter specifies which object field a call to the function uses to sort the rows in the matrix. The parameter can specify a field using either a *"const int"* or *"const string"* argument:

- A "const int" argument specifies a field by its *field index*, where a value of 0 refers to the *first* field listed in the type declaration, 1 refers to the *second* field, and so on. The value can be any non-negative number up to one less than the total number of fields.
- A "const string" argument specifies a field by its *identifier (name)*. The string must literally match one of the field names listed in the type declaration.

The default `sort_field` argument is 0. Therefore, if a matrix.sort() call does not specify a `sort_field` argument, it attempts to sort rows in the matrix by comparing the first field of each object referenced by a given column.

The following script is a modified form of the *first* example in the Sorting arrays of user-defined types section of the Arrays page. It sorts a 3x2 matrix of object IDs instead of an array of IDs. The script declares a custom type named `myType` with three fields: `field0`, `field1`, and `field2`. Then, it creates six `myType` objects and stores their IDs in a matrix on the last historical bar. The script executes a matrix.sort() call to sort the rows of the matrix by the first, second, or third field of the objects referenced on a given column, depending on the selected inputs. The script loops through the matrix with a for…in loop to create a concatenated string representing its sorted structure, then displays the resulting string's text in a label:

//@version=6 indicator("Sorting UDT matrices demo") //@type A custom type for creating objects that store "float", "string", and "int" values. type myType float field0 // This field's index is 0. string field1 // This field's index is 1. int field2 // This field's index is 2. //@variable A string to indicate whether the script specifies sorting fields by index or name. string specifyInput = input.string("Index", "Specify a field using its", ["Index", "Name"]) //@variable The index of the field to use for sorting if the `specifyInput` value is `"Index"`. int indexInput = input.int(0, "Field index", 0, 2, active = specifyInput == "Index") //@variable The name of the field to use for sorting if the `specifyInput` value is `"Name"`. string nameInput = input.string("field0", "Field name", ["field0", "field1", "field2"], active = specifyInput == "Name") //@variable The index of the column to use for sorting. int colInput = input.int(0, "Column to check", 0, 1) if barstate.islastconfirmedhistory //@variable References an array that stores the IDs of `myType` objects. array<myType> udtArray =

array.from(myType.new(field0 = 2.0, field1 = "C", field2 = 2), myType.new(field0 = 6.0, field1 = "D", field2 = 4), myType.new(field0 = 1.0, field1 = "B", field2 = 3), myType.new(field0 = 5.0, field1 = "E", field2 = 6), myType.new(field0 = 3.0, field1 = "A", field2 = 1), myType.new(field0 = 4.0, field1 = "F", field2 = 5)) //@variable References a 3x2 matrix of `myType` IDs retrieved from the array. matrix<myType> udtMatrix = matrix.new<myType>() // Populate the matrix using the `udtArray` array, then reshape it using `matrix.reshape()`. udtMatrix.add_row(0, udtArray) udtMatrix.reshape(3, 2) // Sort the rows in ascending order. Use the field at the specified index if the `specifyInput` value is `"Index"`. if specifyInput == "Index" switch indexInput 0 => udtMatrix.sort(colInput, sort_field = 0) 1 => udtMatrix.sort(colInput, sort_field = 1) 2 => udtMatrix.sort(colInput, sort_field = 2) // Otherwise, sort using the field with the specified name. else switch nameInput "field0" => udtMatrix.sort(colInput, sort_field = "field0") "field1" => udtMatrix.sort(colInput, sort_field = "field1") "field2" => udtMatrix.sort(colInput, sort_field = "field2") //@variable A string representing the structure of the sorted matrix. string displayStr = switch specifyInput "Index" => str.format("Sorted by column {0} using field at index {1}\n\n", colInput, indexInput) => str.format("Sorted by column {0} using field named "{1}"\n\n", colInput, nameInput) // Concatenate formatted strings to represent the sorted structure. for [i, row] in udtMatrix string tempStr = "[" for [j, id] in row tempStr += str.format("(field0: {0,number,0.0}, field1: {1}, field2: {2}), ", id.field0, id.field1, id.field2) displayStr += str.substring(tempStr, 0, str.length(tempStr) - 2) + "]\n" displayStr := str.substring(displayStr, 0, str.length(displayStr) - 1) // Display the final string's text in a label. label.new(bar_index, 0, displayStr, style = label.style_label_center, size = 24, textalign = text.align_left, text_font_family = font.family_monospace)

Note that:

- The `sort_field` parameter accepts only values that have the *"const"* qualifier; it cannot accept values qualified as "input", "simple", or "series". Therefore, to sort the matrix using an input-specified field, this script uses a *separate* matrix.sort() call for each input combination.

When sorting matrices that store IDs of a user-defined type, it's important to understand the following limitations:

- The `sort_field` argument of a matrix.sort() call must refer to an "int", "float", or "string" field. Attempting to sort UDT matrices using fields of other types causes a *compilation error*.
- While the matrix.sort() function can sort the rows of a matrix using a column containing objects that have *fields* with na values, it **cannot** sort a matrix using columns that contain na *elements*. Elements that are na represent *nonexistent IDs*, meaning that there are *no associated objects* from which to retrieve a field value for sorting. Attempting to sort a UDT matrix using a column with na elements causes a *runtime error*.

Refer to the Sorting arrays of user-defined types section of the Arrays page for examples of errors relating to these limitations and ways to resolve them. The principles explained in those examples also apply to sorting matrices with the matrix.sort() function.

Concatenating

Scripts can *concatenate* two matrices using matrix.concat(). This function appends the rows of an `id2` matrix to the end of an `id1` matrix with the same number of columns.

To create a matrix with elements representing the *columns* of a matrix appended to another, transpose both matrices, use matrix.concat() on the transposed matrices, then transpose() the result.

For example, this script appends the rows of the `m2` matrix to the `m1` matrix and appends their columns using *transposed copies* of the matrices. It displays the `m1` and `m2` matrices and the results after concatenating their rows and columns in labels using the custom `debugLabel()` method:

```
//@version=6 indicator("Concatenation demo") //@function Displays the rows of a matrix in a label with a note. //@param this The matrix to display. //@param barIndex The `bar_index` to display the label at. //@param bgColor The background color of the label. //@param textColor The color of the label's text. //@param note The text to display above the rows. method debugLabel( matrix<float> this, int barIndex = bar_index, color bgColor = color.blue, color textColor = color.white, string note = "" ) => labelText = note + "\n" + str.tostring(this) if barstate.ishistory label.new( barIndex, 0, labelText, color = bgColor, style = label.style_label_center, textcolor = textColor, size = size.huge ) //@variable A 2x3 matrix filled with 1s. matrix<int> m1 = matrix.new<int>(2, 3, 1) //@variable A 2x3 matrix filled with 2s. matrix<int> m2 = matrix.new<int>(2, 3, 2) //@variable The transpose of `m1`. t1 = m1.transpose() //@variable The transpose of `m2`. t2 = m2.transpose() if bar_index == last_bar_index - 1 // Display the original matrices. m1.debugLabel(note = "Matrix 1") m2.debugLabel(bar_index + 10, note = "Matrix 2") // Append the rows of `m2` to the end of `m1` and display `m1`. m1.concat(m2) m1.debugLabel(bar_index + 20, color.blue, note = "Appended rows") // Append the rows of `t2` to the end of `t1`, then display the transpose of `t1`. t1.concat(t2) t1.transpose().debugLabel(bar_index + 30, color.purple, note = "Appended columns")
```

Matrix calculations

Element-wise calculations

Pine scripts can calculate the *average*, *minimum*, *maximum*, and *mode* of all elements within a matrix via matrix.avg(), matrix.min(), matrix.max(), and matrix.mode(). These functions operate the same as their `array.*` equivalents, allowing users to run element-wise calculations on a matrix, its submatrices, and its rows and columns using the same syntax. For example, the built-in `*.avg()` functions called on a 3x3 matrix with values 1-9 and an array with the same nine elements will both return a value of 5.

The script below uses `*.avg()`, `*.max()`, and `*.min()` methods to calculate developing averages and extremes of OHLC data in a period. It adds a new column of open, high, low, and close values to the end of the `ohlcData` matrix whenever `queueColumn` is `true`. When `false`, the script uses the matrix.get() and matrix.set() methods to adjust the elements in the last column for developing HLC values in the current period. It uses the `ohlcData` matrix, a submatrix, and row and column arrays to calculate the developing OHLC4 and HL2 averages over `length` periods, the maximum high and minimum low over `length` periods, and the current period's developing OHLC4 price:

```
//@version=6 indicator("Element-wise calculations example", "Developing values", overlay = true) //@variable The number of data points in the averages. int length = input.int(3, "Length", 1) //@variable The timeframe of each reset period. string timeframe = input.timeframe("D", "Reset Timeframe") //@variable A 4x`length` matrix of OHLC values. var matrix<float> ohlcData = matrix.new<float>(4, length) //@variable Is `true` at the start of a new bar at the `timeframe`. bool queueColumn = timeframe.change(timeframe) if queueColumn // Add new values to the end column of `ohlcData`. ohlcData.add_col(length, array.from(open, high, low, close)) // Remove the oldest column from `ohlcData`. ohlcData.remove_col(0) else // Adjust the last element of column 1 for new highs. if high > ohlcData.get(1, length - 1) ohlcData.set(1, length - 1, high) // Adjust the last element of column 2 for new lows. if low < ohlcData.get(2, length - 1) ohlcData.set(2, length - 1, low) // Adjust the last element of column 3 for the new closing price. ohlcData.set(3, length - 1, close) //@variable The `matrix.avg()` of all elements in `ohlcData`. avgOHLC4 = ohlcData.avg() //@variable The
```

`matrix.avg()` of all elements in rows 1 and 2, i.e., the average of all `high` and `low` values. avgHL2 = ohlcData.submatrix(from_row = 1, to_row = 3).avg() //@variable The `matrix.max()` of all values in `ohlcData`. Equivalent to `ohlcData.row(1).max()`. maxHigh = ohlcData.max() //@variable The `array.min()` of all `low` values in `ohlcData`. Equivalent to `ohlcData.min()`. minLow = ohlcData.row(2).min() //@variable The `array.avg()` of the last column in `ohlcData`, i.e., the current OHLC4. ohlc4Value = ohlcData.col(length - 1).avg() plot(avgOHLC4, "Average OHLC4", color.purple, 2) plot(avgHL2, "Average HL2", color.navy, 2) plot(maxHigh, "Max High", color.green) plot(minLow, "Min Low", color.red) plot(ohlc4Value, "Current OHLC4", color.blue)

Note that:

- In this example, we used `array.*()` and `matrix.*()` methods interchangeably to demonstrate their similarities in syntax and behavior.
- Users can calculate the matrix equivalent of array.sum() by multiplying the values of matrix.avg() and matrix.elements_count().

Special calculations

Pine Script features several built-in functions for performing essential matrix arithmetic and linear algebra operations, including matrix.sum(), matrix.diff(), matrix.mult(), matrix.pow(), matrix.det(), matrix.inv(), matrix.pinv(), matrix.rank(), matrix.trace(), matrix.eigenvalues(), matrix.eigenvectors(), and matrix.kron(). These functions are advanced features that facilitate a variety of matrix calculations and transformations.

Below, we explain a few fundamental functions with some basic examples.

`matrix.sum()` and `matrix.diff()`

Scripts can perform addition and subtraction of two matrices with the same shape or a matrix and a scalar value using the matrix.sum() and matrix.diff() functions. These functions use the values from the `id2` matrix or scalar to add to or subtract from the elements in `id1`.

This script demonstrates a simple example of matrix addition and subtraction in Pine. It creates a 3x3 matrix, calculates its transpose, then calculates the matrix.sum() and matrix.diff() results using the two matrices. This example displays the original matrix, its transpose, and the resulting sum and difference matrices in labels on the chart:

//@version=6 indicator("Matrix sum and diff example") //@function Displays the rows of a matrix in a label with a note. //@param this The matrix to display. //@param barIndex The `bar_index` to display the label at. //@param bgColor The background color of the label. //@param textColor The color of the label's text. //@param note The text to display above the rows. method debugLabel(matrix<float> this, int barIndex = bar_index, color bgColor = color.blue, color textColor = color.white, string note = "") => labelText = note + "\n" + str.tostring(this) if barstate.ishistory label.new(barIndex, 0, labelText, color = bgColor, style = label.style_label_center, textcolor = textColor, size = size.huge) //@variable A 3x3 matrix. m = matrix.new<float>() // Add rows to `m`. m.add_row(0, array.from(0.5, 1.0, 1.5)) m.add_row(1, array.from(2.0, 2.5, 3.0)) m.add_row(2, array.from(3.5, 4.0, 4.5)) if bar_index == last_bar_index - 1 // Display `m`. m.debugLabel(note = "A") // Get and display the transpose of `m`. matrix<float> t = m.transpose() t.debugLabel(bar_index + 10, note = "A^T") // Calculate the sum of the two matrices. The resulting matrix is symmetric. matrix.sum(m, t).debugLabel(bar_index + 20, color.green, note = "A + A^T") // Calculate the difference between the two matrices. The resulting matrix is antisymmetric. matrix.diff(m, t).debugLabel(bar_index + 30, color.red, note = "A - A^T")

Note that:

- In this example, we've labeled the original matrix as “A” and the transpose as “A^T”.
- Adding “A” and “A^T” produces a *symmetric* matrix, and subtracting them produces an *antisymmetric* matrix. The functions matrix.is_symmetric() and matrix.is_antisymmetric() test a matrix for these conditions.

`matrix.mult()`

Scripts can multiply two matrices via the matrix.mult() function. This function can also multiply a matrix by an array or a scalar value.

In the case of multiplying two matrices, unlike addition and subtraction, matrix multiplication does not require two matrices to share the same shape. However, the number of columns in the first matrix must equal the number of rows in the second one. The resulting matrix returned by matrix.mult() will contain the same number of rows as the `id1` matrix and the same number of columns as the `id2` matrix. For instance, a 2x3 matrix multiplied by a 3x4 matrix will produce a matrix with two rows and four columns, as shown below. Each value within the resulting matrix is the dot product of the corresponding row in the `id1` maxtrix and column in the `id2` matrix:

//@version=6 indicator("Matrix mult example") //@function Displays the rows of a matrix in a label with a note. //@param this The matrix to display. //@param barIndex The `bar_index` to display the label at. //@param bgColor The background color of the label. //@param textColor The color of the label's text. //@param note The text to display above the rows. method debugLabel(matrix<float> this, int barIndex = bar_index, color bgColor = color.blue, color textColor = color.white, string note = "") => labelText = note + "\n" + str.tostring(this) if barstate.ishistory label.new(barIndex, 0, labelText, color = bgColor, style = label.style_label_center, textcolor = textColor, size = size.huge) //@variable A 2x3 matrix. a = matrix.new<float>() //@variable A 3x4 matrix. b = matrix.new<float>() // Add rows to `a`. a.add_row(0, array.from(1, 2, 3)) a.add_row(1, array.from(4, 5, 6)) // Add rows to `b`. b.add_row(0, array.from(0.5, 1.0, 1.5, 2.0)) b.add_row(1, array.from(2.5, 3.0, 3.5, 4.0)) b.add_row(0, array.from(4.5, 5.0, 5.5, 6.0)) if bar_index == last_bar_index - 1 //@variable The result of `a` * `b`. matrix<float> ab = a.mult(b) // Display `a`, `b`, and `ab` matrices. debugLabel(a, note = "A") debugLabel(b, bar_index + 10, note = "B") debugLabel(ab, bar_index + 20, color.green, note = "A * B")

Note that:

- In contrast to the multiplication of scalars, matrix multiplication is *non-commutative*, i.e., `matrix.mult(a, b)` does not necessarily produce the same result as `matrix.mult(b, a)`. In the context of our example, the latter will raise a runtime error because the number of columns in `b` doesn't equal the number of rows in `a`.

When multiplying a matrix and an array, this function treats the operation the same as multiplying the `id1` matrix by a single-column matrix, but it returns an array with the same number of elements as the number of matrix rows. When matrix.mult() passes a scalar as its `id2` value, the function returns a new matrix whose elements are the elements in the `id1` matrix multiplied by the `id2` value.

`matrix.det()`

A *determinant* is a scalar value associated with a square matrix that describes some of its characteristics, namely its invertibility. If a matrix has an inverse, its determinant is nonzero. Otherwise, the matrix is *singular*

(non-invertible). Scripts can calculate the determinant of a matrix via matrix.det().

Programmers can use determinants to detect similarities between matrices, identify *full-rank* and *rank-deficient* matrices, and solve systems of linear equations, among other applications.

For example, this script uses determinants to solve a system of linear equations with a matching number of unknown values using Cramer's rule. The user-defined `solve()` function returns the reference of an array containing solutions for each unknown value in the system, where the n-th element of the array is the determinant of the coefficient matrix with the n-th column replaced by the column of constants divided by the determinant of the original coefficients.

In this script, we've defined the matrix `m` that holds coefficients and constants for these three equations:

```
3 * x0 + 4 * x1 - 1 * x2 = 8
5 * x0 - 2 * x1 + 1 * x2 = 4
2 * x0 - 2 * x1 + 1 * x2 = 1
```

The solution to this system is `(x0 = 1, x1 = 2, x2 = 3)`. The script calculates these values from `m` via `m.solve()` and plots them on the chart:

//@version=6 indicator("Determinants example", "Cramer's Rule") //@function Solves a system of linear equations with a matching number of unknowns using Cramer's rule. //@param this An augmented matrix containing the coefficients for each unknown and the results of // the equations. For example, a row containing the values 2, -1, and 3 represents the equation // `2 * x0 + (-1) * x1 = 3`, where `x0` and `x1` are the unknown values in the system. //@returns An array containing solutions for each variable in the system. solve(matrix<float> this) => //@variable The coefficient matrix for the system of equations. matrix<float> coefficients = this.submatrix(from_column = 0, to_column = this.columns() - 1) //@variable The array of resulting constants for each equation. array<float> constants = this.col(this.columns() - 1) //@variable An array containing solutions for each unknown in the system. array<float> result = array.new<float>() //@variable The determinant value of the coefficient matrix. float baseDet = coefficients.det() matrix<float> modified = na for col = 0 to coefficients.columns() - 1 modified := coefficients.copy() modified.add_col(col, constants) modified.remove_col(col + 1) // Calculate the solution for the column's unknown by dividing the determinant of `modified` by the `baseDet`. result.push(modified.det() / baseDet) result //@variable A 3x4 matrix containing coefficients and results for a system of three equations. m = matrix.new<float>() // Add rows for the following equations: // Equation 1: 3 * x0 + 4 * x1 - 1 * x2 = 8 // Equation 2: 5 * x0 - 2 * x1 + 1 * x2 = 4 // Equation 3: 2 * x0 - 2 * x1 + 1 * x2 = 1 m.add_row(0, array.from(3.0, 4.0, -1.0, 8.0)) m.add_row(1, array.from(5.0, -2.0, 1.0, 4.0)) m.add_row(2, array.from(2.0, -2.0, 1.0, 1.0)) //@variable An array of solutions to the unknowns in the system of equations represented by `m`. solutions = solve(m) plot(solutions.get(0), "x0", color.red, 3) // Plots 1. plot(solutions.get(1), "x1", color.green, 3) // Plots 2. plot(solutions.get(2), "x2", color.blue, 3) // Plots 3.

Note that:

- Solving systems of equations is particularly useful for *regression analysis*, e.g., linear and polynomial regression.
- Cramer's rule works fine for small systems of equations. However, it's computationally inefficient on larger systems. Other methods, such as Gaussian elimination, are often preferred for such use cases.

`matrix.inv()` and `matrix.pinv()`

For any non-singular square matrix, there is an inverse matrix that yields the identity matrix when multiplied by the original. Inverses have use in various matrix transformations and solving systems of equations. Scripts can calculate the inverse of a matrix **when one exists** via the matrix.inv() function.

For singular (non-invertible) matrices, one can calculate a generalized inverse (pseudoinverse), regardless of whether the matrix is square or has a nonzero determinant, via the matrix.pinv() function. Keep in mind that unlike a true inverse, the product of a pseudoinverse and the original matrix does not necessarily equal the identity matrix unless the original matrix *is invertible*.

The following example forms a 2x2 `m` matrix from user inputs, then calls matrix.inv() and matrix.pinv() as methods to calculate the inverse or pseudoinverse of `m`. The script displays strings representing the original matrix, its inverse or pseudoinverse, and their product in labels on the chart:

```
//@version=6 indicator("Inverse example") // Element inputs for the 2x2 matrix. float r0c0 = input.float(4.0, "Row 0, Col 0") float r0c1 = input.float(3.0, "Row 0, Col 1") float r1c0 = input.float(2.0, "Row 1, Col 0") float r1c1 = input.float(1.0, "Row 1, Col 1") //@function Displays the rows of a matrix in a label with a note. //@param this The matrix to display. //@param barIndex The `bar_index` to display the label at. //@param bgColor The background color of the label. //@param textColor The color of the label's text. //@param note The text to display above the rows. method debugLabel( matrix<float> this, int barIndex = bar_index, color bgColor = color.blue, color textColor = color.white, string note = "" ) => labelText = note + "\n" + str.tostring(this) if barstate.ishistory label.new( barIndex, 0, labelText, color = bgColor, style = label.style_label_center, textcolor = textColor, size = size.huge ) //@variable A 2x2 matrix of input values. m = matrix.new<float>() // Add input values to `m`. m.add_row(0, array.from(r0c0, r0c1)) m.add_row(1, array.from(r1c0, r1c1)) //@variable Is `true` if `m` is square with a nonzero determinant, indicating invertibility. bool isInvertible = m.is_square() and m.det() != 0 //@variable The inverse or pseudoinverse of `m`. mInverse = isInvertible ? m.inv() : m.pinv() //@variable The product of `m` and `mInverse`. Returns the identity matrix when `isInvertible` is `true`. matrix<float> product = m.mult(mInverse) if bar_index == last_bar_index - 1 // Display `m`, `mInverse`, and their `product`. m.debugLabel(note = "Original") mInverse.debugLabel(bar_index + 10, color.purple, note = isInvertible ? "Inverse" : "Pseudoinverse") product.debugLabel(bar_index + 20, color.green, note = "Product")
```

Note that:

- This script calls `m.inv()` only when `isInvertible` is `true`, i.e., when `m` is square and has a nonzero determinant. Otherwise, it uses `m.pinv()` to calculate the generalized inverse.

`matrix.rank()`

The *rank* of a matrix represents the number of linearly independent vectors (rows or columns) it contains. In essence, matrix rank measures the number of vectors one cannot express as a linear combination of others, or in other words, the number of vectors that contain **unique** information. Scripts can calculate the rank of a matrix via matrix.rank().

This script identifies the number of linearly independent vectors in two 3x3 matrices (`m1` and `m2`) using matrix.rank() and plots the values in a separate pane. As we see on the chart, the `m1.rank()` value is 3 because each vector is unique. The `m2.rank()` value, on the other hand, is 1 because it has just one unique vector:

//@version=6 indicator("Matrix rank example") //@variable A 3x3 full-rank matrix. m1 = matrix.new<float>() //@variable A 3x3 rank-deficient matrix. m2 = matrix.new<float>() // Add linearly independent vectors to `m1`. m1.add_row(0, array.from(3, 2, 3)) m1.add_row(1, array.from(4, 6, 6)) m1.add_row(2, array.from(7, 4, 9)) // Add linearly dependent vectors to `m2`. m2.add_row(0, array.from(1, 2, 3)) m2.add_row(1, array.from(2, 4, 6)) m2.add_row(2, array.from(3, 6, 9)) // Plot `matrix.rank()` values. plot(m1.rank(), color = color.green, linewidth = 3) plot(m2.rank(), color = color.red, linewidth = 3)

Note that:

- The highest rank value a matrix can have is the minimum of its number of rows and columns. A matrix with the maximum possible rank is known as a *full-rank* matrix, and any matrix without full rank is known as a *rank-deficient* matrix.
- The determinants of full-rank square matrices are nonzero, and such matrices have inverses. Conversely, the determinant of a rank-deficient matrix is always 0.
- For any matrix that contains nothing but the same value in each of its elements (e.g., a matrix filled with 0), the rank is always 0 since none of the vectors hold unique information. For any other matrix with distinct values, the minimum possible rank is 1.

Error handling

In addition to usual **compiler** errors, which occur during a script's compilation due to improper syntax, scripts using matrices can raise specific **runtime** errors during their execution. When a script raises a runtime error, it displays a red exclamation point next to the script title. Users can view the error message by clicking this icon.

In this section, we discuss runtime errors that users may encounter while utilizing matrices in their scripts.

The row/column index (xx) is out of bounds, row/column size is (yy).

This runtime error occurs when trying to access indices outside the matrix dimensions with functions including matrix.get(), matrix.set(), matrix.fill(), and matrix.submatrix(), as well as some of the functions relating to the rows and columns of a matrix.

For example, this code contains two lines that will produce this runtime error. The `m.set()` method references a `row` index that doesn't exist (2). The `m.submatrix()` method references all column indices up to `to_column - 1`. A `to_column` value of 4 results in a runtime error because the last column index referenced (3) does not exist in `m`:

//@version=6 indicator("Out of bounds demo") //@variable A 2x3 matrix with a max row index of 1 and max column index of 2. matrix<float> m = matrix.new<float>(2, 3, 0.0) m.set(row = 2, column = 0, value = 1.0) // The `row` index is out of bounds on this line. The max value is 1. m.submatrix(from_column = 1, to_column = 4) // The `to_column` index is invalid on this line. The max value is 3. if bar_index == last_bar_index - 1 label.new(bar_index, 0, str.tostring(m), color = color.navy, textcolor = color.white, size = size.huge)

Users can avoid this error in their scripts by ensuring their function calls do not reference indices greater than or equal to the number of rows/columns.

The array size does not match the number of rows or columns in the matrix.

When using matrix.add_row() and matrix.add_col() functions to insert rows and columns into a non-empty matrix, the size of the inserted array must align with the matrix dimensions. The size of an inserted row must match the number of columns, and the size of an inserted column must match the number of rows. Otherwise, the script will raise this runtime error. For example:

//@version=6 indicator("Invalid array size demo") // Declare an empty matrix. m = matrix.new<float>() m.add_col(0, array.from(1, 2)) // Add a column. Changes the shape of `m` to 2x1. m.add_col(1, array.from(1, 2, 3)) // Raises a runtime error because `m` has 2 rows, not 3. plot(m.col(0).get(1))

Note that:

- When `m` is empty, one can insert a row or column array of *any* size, as shown in the first `m.add_col()` line.

Cannot call matrix methods when the ID of matrix is 'na'.

When a matrix variable is assigned to `na`, it means that the variable doesn't reference an existing object. Consequently, one cannot use built-in `matrix.*()` functions and methods with it. For example:

//@version=6 indicator("na matrix methods demo") //@variable A `matrix` variable assigned to `na`. matrix<float> m = na mCopy = m.copy() // Raises a runtime error. You can't copy a matrix that doesn't exist. if bar_index == last_bar_index - 1 label.new(bar_index, 0, str.tostring(mCopy), color = color.navy, textcolor = color.white, size = size.huge)

To resolve this error, assign `m` to a valid matrix instance before using `matrix.*()` functions.

Matrix is too large. Maximum size of the matrix is 100,000 elements.

The total number of elements in a matrix (matrix.elements_count()) cannot exceed **100,000**, regardless of its shape. For example, this script will raise an error because it inserts 1000 rows with 101 elements into the `m` matrix:

//@version=6 indicator("Matrix too large demo") var matrix<float> m = matrix.new<float>() if bar_index == 0 for i = 1 to 1000 // This raises an error because the script adds 101 elements on each iteration. // 1000 rows * 101 elements per row = 101000 total elements. This is too large. m.add_row(m.rows(), array.new<float>(101, i)) plot(m.get(0, 0))

The row/column index must be 0 <= from_row/column < to_row/column.

When using `matrix.*()` functions with `from_row/column` and `to_row/column` indices, the `from_*` values must be less than the corresponding `to_*` values, with the minimum possible value being 0. Otherwise, the script will raise a runtime error.

For example, this script shows an attempt to declare a submatrix from a 4x4 `m` matrix with a `from_row` value of 2 and a `to_row` value of 2, which will result in an error:

//@version=6 indicator("Invalid from_row, to_row demo") //@variable A 4x4 matrix filled with a pseudorandom value. matrix<float> m = matrix.new<float>(4, 4, math.random()) matrix<float> mSub = m.submatrix(from_row = 2, to_row = 2) // Raises an error. `from_row` can't equal `to_row`. plot(mSub.get(0, 0))

Matrices 'id1' and 'id2' must have an equal number of rows and columns to be added.

When using matrix.sum() and matrix.diff() functions, the `id1` and `id2` matrices must have the same number of rows and the same number of columns. Attempting to add or subtract two matrices with mismatched dimensions will raise an error, as demonstrated by this code:

//@version=6 indicator("Invalid sum dimensions demo") //@variable A 2x3 matrix. matrix<float> m1 = matrix.new<float>(2, 3, 1) //@variable A 3x4 matrix. matrix<float> m2 = matrix.new<float>(3, 4, 2) mSum = matrix.sum(m1, m2) // Raises an error. `m1` and `m2` don't have matching dimensions. plot(mSum.get(0, 0))

The number of columns in the 'id1' matrix must equal the number of rows in the matrix (or the number of elements in the array) 'id2'.

When using matrix.mult() to multiply an `id1` matrix by an `id2` matrix or array, the matrix.rows() or array.size() of `id2` must equal the matrix.columns() in `id1`. If they don't align, the script will raise this error.

For example, this script tries to multiply two 2x3 matrices. While *adding* these matrices is possible, *multiplying* them is not:

//@version=6 indicator("Invalid mult dimensions demo") //@variable A 2x3 matrix. matrix<float> m1 = matrix.new<float>(2, 3, 1) //@variable A 2x3 matrix. matrix<float> m2 = matrix.new<float>(2, 3, 2) mSum = matrix.mult(m1, m2) // Raises an error. The number of columns in `m1` and rows in `m2` aren't equal. plot(mSum.get(0, 0))

Operation not available for non-square matrices.

Some matrix operations, including matrix.inv(), matrix.det(), matrix.eigenvalues(), and matrix.eigenvectors() only work with **square** matrices, i.e., matrices with the same number of rows and columns. When attempting to execute such functions on non-square matrices, the script will raise an error stating the operation isn't available or that it cannot calculate the result for the matrix `id`. For example:

//@version=6 indicator("Non-square demo") //@variable A 3x5 matrix. matrix<float> m = matrix.new<float>(3, 5, 1) plot(m.det()) // Raises a runtime error. You can't calculate the determinant of a 3x5 matrix.

ADVANCED

Methods

> TipThis page contains *advanced* material. If you're new to Pine Script®, start by learning about core language components – such as the type system and the basics of the execution model – and explore other, more accessible features before venturing further.

Introduction

Pine Script methods are specialized functions associated with values of specific built-in types, user-defined types, or enum types. They behave the same as regular functions in most regards while offering a shorter, more convenient syntax. Users can access methods using *dot notation* syntax on variables of the associated type, similar to accessing the fields of a Pine Script object.

Built-in methods

Pine Script features built-in methods for most special types, including array, matrix, map, line, linefill, box, polyline, label, table, chart.point, footprint, and volume_row. These methods provide users with a more concise way to call specialized routines for these types within their scripts.

When using these special types, the expressions:

```
<namespace>.<functionName>([paramName =] <objectName>, …)
```

and:

```
<objectName>.<functionName>(…)
```

are equivalent. For example, rather than using:

array.get(id, index)

to get the value from an array `id` at the specified `index`, we can simply use:

id.get(index)

to achieve the same effect. This notation eliminates the need for users to reference the function's namespace, as get() is a method of `id` in this context.

Written below is a practical example to demonstrate the usage of built-in methods in place of functions.

The following script computes Bollinger Bands over a specified number of prices sampled once every `n` bars. It calls array.push() and array.shift() to queue `sourceInput` values through the `sourceArray`, then array.avg() and array.stdev() to compute the `sampleMean` and `sampleDev`. The script then uses these values to calculate the `highBand` and `lowBand`, which it plots on the chart along with the `sampleMean`:

//@version=6 indicator("Custom Sample BB", overlay = true) float sourceInput = input.source(close, "Source") int samplesInput = input.int(20, "Samples") int n = input.int(10, "Bars") float multiplier = input.float(2.0, "StdDev") var array<float> sourceArray = array.new<float>(samplesInput) var float sampleMean = na var float sampleDev = na // Identify if `n` bars have passed. if bar_index % n == 0 // Update the queue. array.push(sourceArray, sourceInput) array.shift(sourceArray) // Update the mean and standard deviation values. sampleMean := array.avg(sourceArray) sampleDev := array.stdev(sourceArray) * multiplier // Calculate bands. float highBand = sampleMean + sampleDev float lowBand = sampleMean - sampleDev plot(sampleMean, "Basis", color.orange) plot(highBand, "Upper", color.lime) plot(lowBand, "Lower", color.red)

Let's rewrite this code to utilize methods rather than built-in functions. In this version, we have replaced all built-in `array.*()` function calls in the script with equivalent method calls:

//@version=6 indicator("Custom Sample BB", overlay = true) float sourceInput = input.source(close, "Source") int samplesInput = input.int(20, "Samples") int n = input.int(10, "Bars") float multiplier = input.float(2.0, "StdDev") var array<float> sourceArray = array.new<float>(samplesInput) var float sampleMean = na var float sampleDev = na // Identify if `n` bars have passed. if bar_index % n == 0 // Update the queue. sourceArray.push(sourceInput) sourceArray.shift() // Update the mean and standard deviation values. sampleMean := sourceArray.avg() sampleDev := sourceArray.stdev() * multiplier // Calculate band values. float highBand = sampleMean + sampleDev float lowBand = sampleMean - sampleDev plot(sampleMean, "Basis", color.orange) plot(highBand, "Upper", color.lime) plot(lowBand, "Lower", color.red)

Note that:

- We call the array methods using `sourceArray.*` rather than referencing the array namespace.
- We do not include `sourceArray` as a parameter when we call the methods since they already reference the object.

User-defined methods

Pine Script allows users to define custom methods for use with objects of any built-in or user-defined type. Defining a method is essentially the same as defining a function, but with two key differences:

- The method keyword must be included before the function name.
- The type of the first parameter in the signature must be explicitly declared, as it represents the type of object that the method will be associated with.

```
[export] method <functionName>(<paramType> <paramName> [= <defaultValue>], …) =>
    <functionBlock>
```

Let's apply user-defined methods to our previous Bollinger Bands example to encapsulate operations from the global scope, which will simplify the code and promote reusability. See this portion from the example:

// Identify if `n` bars have passed. if bar_index % n == 0 // Update the queue. sourceArray.push(sourceInput) sourceArray.shift() // Update the mean and standard deviation values. sampleMean := sourceArray.avg()

sampleDev := sourceArray.stdev() * multiplier // Calculate band values. float highBand = sampleMean + sampleDev float lowBand = sampleMean - sampleDev

We will start by defining a simple method to queue values through an array in a single call.

This `maintainQueue()` method invokes the push() and shift() methods on a `srcArray` when `takeSample` is true and returns the object:

// @function Maintains a queue of the size of `srcArray`. // It appends a `value` to the array and removes its oldest element at position zero. // @param srcArray (array<float>) The array where the queue is maintained. // @param value (float) The new value to be added to the queue. // The queue's oldest value is also removed, so its size is constant. // @param takeSample (bool) A new `value` is only pushed into the queue if this is true. // @returns (array<float>) `srcArray` object. method maintainQueue(array<float> srcArray, float value, bool takeSample = true) => if takeSample srcArray.push(value) srcArray.shift() srcArray

Note that:

- Just as with user-defined functions, we use the `@function` compiler annotation to document method descriptions.

Now we can replace `sourceArray.push()` and `sourceArray.shift()` with `sourceArray.maintainQueue()` in our example:

// Identify if `n` bars have passed. if bar_index % n == 0 // Update the queue. sourceArray.maintainQueue(sourceInput) // Update the mean and standard deviation values. sampleMean := sourceArray.avg() sampleDev := sourceArray.stdev() * multiplier // Calculate band values. float highBand = sampleMean + sampleDev float lowBand = sampleMean - sampleDev

From here, we will further simplify our code by defining a method that handles all Bollinger Band calculations within its scope.

This `calcBB()` method invokes the avg() and stdev() methods on a `srcArray` to update `mean` and `dev` values when `calculate` is true. The method uses these values to return a tuple containing the basis, upper band, and lower band values respectively:

// @function Computes Bollinger Band values from an array of data. // @param srcArray (array<float>) The array where the queue is maintained. // @param multiplier (float) Standard deviation multiplier. // @param calcuate (bool) The method will only calculate new values when this is true. // @returns A tuple containing the basis, upper band, and lower band respectively. method calcBB(array<float> srcArray, float mult, bool calculate = true) => var float mean = na var float dev = na if calculate // Compute the mean and standard deviation of the array. mean := srcArray.avg() dev := srcArray.stdev() * mult [mean, mean + dev, mean - dev]

With this method, we can now remove Bollinger Band calculations from the global scope and improve code readability:

// Identify if `n` bars have passed. bool newSample = bar_index % n == 0 // Update the queue and compute new BB values on each new sample. [sampleMean, highBand, lowBand] = sourceArray.maintainQueue(sourceInput, newSample).calcBB(multiplier, newSample)

Note that:

- Rather than using an `if` block in the global scope, we have defined a `newSample` variable that is only true once every `n` bars. The `maintainQueue()` and `calcBB()` methods use this value for their respective `takeSample` and `calculate` parameters.
- Since the `maintainQueue()` method returns the object that it references, we're able to call `calcBB()` from the same line of code, as both methods apply to `array<float>` instances.

Here is how the full script example looks now that we've applied our user-defined methods:

//@version=6 indicator("Custom Sample BB", overlay = true) float sourceInput = input.source(close, "Source") int samplesInput = input.int(20, "Samples") int n = input.int(10, "Bars") float multiplier = input.float(2.0, "StdDev") var array<float> sourceArray = array.new<float>(samplesInput) // @function Maintains a queue of the size of `srcArray`. // It appends a `value` to the array and removes its oldest element at position zero. // @param srcArray (array<float>) The array where the queue is maintained. // @param value (float) The new value to be added to the queue. // The queue's oldest value is also removed, so its size is constant. // @param takeSample (bool) A new `value` is only pushed into the queue if this is true. // @returns (array<float>) `srcArray` object. method maintainQueue(array<float> srcArray, float value, bool takeSample = true) => if takeSample srcArray.push(value) srcArray.shift() srcArray // @function Computes Bollinger Band values from an array of data. // @param srcArray (array<float>) The array where the queue is maintained. // @param multiplier (float) Standard deviation multiplier. // @param calcuate (bool) The method will only calculate new values when this is true. // @returns A tuple containing the basis, upper band, and lower band respectively. method calcBB(array<float> srcArray, float mult, bool calculate = true) => var float mean = na var float dev = na if calculate // Compute the mean and standard deviation of the array. mean := srcArray.avg() dev := srcArray.stdev() * mult [mean, mean + dev, mean - dev] // Identify if `n` bars have passed. bool newSample = bar_index % n == 0 // Update the queue and compute new BB values on each new sample. [sampleMean, highBand, lowBand] = sourceArray.maintainQueue(sourceInput, newSample).calcBB(multiplier, newSample) plot(sampleMean, "Basis", color.orange) plot(highBand, "Upper", color.lime) plot(lowBand, "Lower", color.red)

Method overloading

User-defined methods can override and overload existing built-in and user-defined methods with the same identifier. This capability allows users to define multiple routines associated with different parameter signatures under the same method name.

As a simple example, suppose we want to define a method to identify a variable's type. Since we must explicitly specify the type of object associated with a user-defined method, we will need to define overloads for each type that we want it to recognize.

Below, we have defined a `getType()` method that returns a string representation of a variable's type with overloads for the five primitive types:

// @function Identifies an object's type. // @param this Object to inspect. // @returns (string) A string representation of the type. method getType(int this) => na(this) ? "int(na)" : "int" method getType(float this) => na(this) ? "float(na)" : "float" method getType(bool this) => // "bool" values only have two states, `true` and `false`, but never `na`. "bool" method getType(color this) => na(this) ? "color(na)" : "color" method getType(string this) => na(this) ? "string(na)" : "string"

Now we can use these overloads to inspect some variables. This script uses str.format() to format the results from calling the `getType()` method on five different variables into a single `results` string, then displays the

string in the `lbl` label using the built-in set_text() method:

//@version=6 indicator("Type Inspection") // @function Identifies an object's type. // @param this Object to inspect. // @returns (string) A string representation of the type. method getType(int this) => na(this) ? "int(na)" : "int" method getType(float this) => na(this) ? "float(na)" : "float" method getType(bool this) => na(this) ? "bool(na)" : "bool" method getType(color this) => na(this) ? "color(na)" : "color" method getType(string this) => na(this) ? "string(na)" : "string" a = 1 b = 1.0 c = true d = color.white e = "1" // Inspect variables and format results. results = str.format("a: {0}\nb: {1}\nc: {2}\nd: {3}\ne: {4}", a.getType(), b.getType(), c.getType(), d.getType(), e.getType()) var label lbl = label.new(0, 0) lbl.set_x(bar_index) lbl.set_text(results)

Note that:

- The underlying type of each variable determines which overload of `getType()` the compiler will use.
- The method will append “(na)” to the output string when a variable is `na` to demarcate that it is empty.

Advanced example

Let's apply what we've learned to construct a script that estimates the cumulative distribution of elements in an array, meaning the fraction of elements in the array that are less than or equal to any given value.

There are many ways in which we could choose to tackle this objective. For this example, we will start by defining a method to replace elements of an array, which will help us count the occurrences of elements within a range of values.

Written below is an overload of the built-in fill() method for `array<float>` instances. This overload replaces elements in a `srcArray` within the range between the `lowerBound` and `upperBound` with an `innerValue`, and replaces all elements outside the range with an `outerValue`:

// @function Replaces elements in a `srcArray` between `lowerBound` and `upperBound` with an `innerValue`, // and replaces elements outside the range with an `outerValue`. // @param srcArray (array<float>) Array to modify. // @param innerValue (float) Value to replace elements within the range with. // @param outerValue (float) Value to replace elements outside the range with. // @param lowerBound (float) Lowest value to replace with `innerValue`. // @param upperBound (float) Highest value to replace with `innerValue`. // @returns (array<float>) `srcArray` object. method fill(array<float> srcArray, float innerValue, float outerValue, float lowerBound, float upperBound) => for [i, element] in srcArray if (element >= lowerBound or na(lowerBound)) and (element <= upperBound or na(upperBound)) srcArray.set(i, innerValue) else srcArray.set(i, outerValue) srcArray

With this method, we can filter an array by value ranges to produce an array of occurrences. For example, the expression:

srcArray.copy().fill(1.0, 0.0, min, val)

copies the `srcArray` object, replaces all elements between `min` and `val` with 1.0, then replaces all elements above `val` with 0.0. From here, it's easy to estimate the output of the cumulative distribution function at the `val`, as it's simply the average of the resulting array:

srcArray.copy().fill(1.0, 0.0, min, val).avg()

Note that:

- The compiler will only use this `fill()` overload instead of the built-in when the user provides `innerValue`, `outerValue`, `lowerBound`, and `upperBound` arguments in the call.
- If either `lowerBound` or `upperBound` is `na`, its value is ignored while filtering the fill range.
- We are able to call `copy()`, `fill()`, and `avg()` successively on the same line of code because the first two methods return an `array<float>` instance.

We can now use this to define a method that will calculate our empirical distribution values. The following `eCDF()` method estimates a number of evenly spaced ascending `steps` from the cumulative distribution function of a `srcArray` and pushes the results into a `cdfArray`:

// @function Estimates the empirical CDF of a `srcArray`. // @param srcArray (array<float>) Array to calculate on. // @param steps (int) Number of steps in the estimation. // @returns (array<float>) Array of estimated CDF ratios. method eCDF(array<float> srcArray, int steps) => float min = srcArray.min() float rng = srcArray.range() / steps array<float> cdfArray = array.new<float>() // Add averages of `srcArray` filtered by value region to the `cdfArray`. float val = min for i = 1 to steps val += rng cdfArray.push(srcArray.copy().fill(1.0, 0.0, min, val).avg()) cdfArray

Lastly, to ensure that our `eCDF()` method functions properly for arrays containing small and large values, we will define a method to normalize our arrays.

This `featureScale()` method uses array min() and range() methods to produce a rescaled copy of a `srcArray`. We will use this to normalize our arrays prior to invoking the `eCDF()` method:

// @function Rescales the elements within a `srcArray` to the interval [0, 1]. // @param srcArray (array<float>) Array to normalize. // @returns (array<float>) Normalized copy of the `srcArray`. method featureScale(array<float> srcArray) => float min = srcArray.min() float rng = srcArray.range() array<float> scaledArray = array.new<float>() // Push normalized `element` values into the `scaledArray`. for element in srcArray scaledArray.push((element - min) / rng) scaledArray

Note that:

- This method does not include special handling for divide by zero conditions. If `rng` is 0, the value of the array element will be `na`.

The full example below queues a `sourceArray` of size `length` with `sourceInput` values using our previous `maintainQueue()` method, normalizes the array's elements using the `featureScale()` method, then calls the `eCDF()` method to get an array of estimates for `n` evenly spaced steps on the distribution. The script then calls a user-defined `makeLabel()` function to display the estimates and prices in a label on the right side of the chart:

//@version=6 indicator("Empirical Distribution", overlay = true) float sourceInput = input.source(close, "Source") int length = input.int(20, "Length") int n = input.int(20, "Steps") // @function Maintains a queue of the size of `srcArray`. // It appends a `value` to the array and removes its oldest element at position zero. // @param srcArray (array<float>) The array where the queue is maintained. // @param value (float) The new value to be added to the queue. // The queue's oldest value is also removed, so its size is constant. // @param takeSample (bool) A new `value` is only pushed into the queue if this is true. // @returns (array<float>) `srcArray` object. method maintainQueue(array<float> srcArray, float value, bool takeSample = true) => if takeSample srcArray.push(value) srcArray.shift() srcArray // @function Replaces elements in a `srcArray` between `lowerBound` and `upperBound` with an `innerValue`, // and replaces elements outside the range with an `outerValue`. // @param srcArray (array<float>) Array to modify. // @param innerValue

(float) Value to replace elements within the range with. // @param outerValue (float) Value to replace elements outside the range with. // @param lowerBound (float) Lowest value to replace with `innerValue`. // @param upperBound (float) Highest value to replace with `innerValue`. // @returns (array<float>) `srcArray` object. method fill(array<float> srcArray, float innerValue, float outerValue, float lowerBound, float upperBound) => for [i, element] in srcArray if (element >= lowerBound or na(lowerBound)) and (element <= upperBound or na(upperBound)) srcArray.set(i, innerValue) else srcArray.set(i, outerValue) srcArray // @function Estimates the empirical CDF of a `srcArray`. // @param srcArray (array<float>) Array to calculate on. // @param steps (int) Number of steps in the estimation. // @returns (array<float>) Array of estimated CDF ratios. method eCDF(array<float> srcArray, int steps) => float min = srcArray.min() float rng = srcArray.range() / steps array<float> cdfArray = array.new<float>() // Add averages of `srcArray` filtered by value region to the `cdfArray`. float val = min for i = 1 to steps val += rng cdfArray.push(srcArray.copy().fill(1.0, 0.0, min, val).avg()) cdfArray // @function Rescales the elements within a `srcArray` to the interval [0, 1]. // @param srcArray (array<float>) Array to normalize. // @returns (array<float>) Normalized copy of the `srcArray`. method featureScale(array<float> srcArray) => float min = srcArray.min() float rng = srcArray.range() array<float> scaledArray = array.new<float>() // Push normalized `element` values into the `scaledArray`. for element in srcArray scaledArray.push((element - min) / rng) scaledArray // @function Draws a label containing eCDF estimates in the format "{price}: {percent}%" // @param srcArray (array<float>) Array of source values. // @param cdfArray (array<float>) Array of CDF estimates. // @returns (void) makeLabel(array<float> srcArray, array<float> cdfArray) => float max = srcArray.max() float rng = srcArray.range() / cdfArray.size() string results = "" var label lbl = label.new(0, 0, "", style = label.style_label_left, text_font_family = font.family_monospace) // Add percentage strings to `results` starting from the `max`. cdfArray.reverse() for [i, element] in cdfArray results += str.format("{0}: {1}%\n", max - i * rng, element * 100) // Update `lbl` attributes. lbl.set_xy(bar_index + 1, srcArray.avg()) lbl.set_text(results) var array<float> sourceArray = array.new<float>(length) // Add background color for the last `length` bars. bgcolor(bar_index > last_bar_index - length ? color.new(color.orange, 80) : na) // Queue `sourceArray`, feature scale, then estimate the distribution over `n` steps. array<float> distArray = sourceArray.maintainQueue(sourceInput).featureScale().eCDF(n) // Draw label. makeLabel(sourceArray, distArray)

ADVANCED

Objects

> TipThis page contains *advanced* material. If you're new to Pine Script®, start by learning about core language components – such as the type system and the basics of the execution model – and explore other, more accessible features before venturing further.

Introduction

Pine Script objects are instances of *user-defined types* (UDTs). They are the equivalent of variables containing parts called *fields*, each able to hold independent values that can be of various types.

Experienced programmers can think of UDTs as methodless classes. They allow users to create custom types that organize different values under one logical entity.

Creating objects

Before an object can be created, its type must be defined. The User-defined types section of the Type system page explains how to do so.

Let's define a `pivotPoint` type to hold pivot information:

```
type pivotPoint int x float y string xloc = xloc.bar_time
```

Note that:

- We use the type keyword to declare the creation of a UDT.
- We name our new UDT `pivotPoint`.
- After the first line, we create a local block containing the type and name of each field.
- The `x` field will hold the x-coordinate of the pivot. It is declared as an "int" because it will hold either a timestamp or a bar index of "int" type.
- `y` is a "float" because it will hold the pivot's price.
- `xloc` is a field that will specify the units of `x`: xloc.bar_index or xloc.bar_time. We set its default value to xloc.bar_time by using the `=` operator. When an object is created from that UDT, its `xloc` field will thus be set to that value.

Now that our `pivotPoint` UDT is defined, we can proceed to create objects from it. We create objects using the UDT's `new()` built-in method. To create a new `foundPoint` object from our `pivotPoint` UDT, we use:

```
foundPoint = pivotPoint.new()
```

We can also specify field values for the created object using the following:

```
foundPoint = pivotPoint.new(time, high)
```

Or the equivalent:

```
foundPoint = pivotPoint.new(x = time, y = high)
```

At this point, the `foundPoint` object's `x` field will contain the value of the time built-in when it is created, `y` will contain the value of high and the `xloc` field will contain its default value of xloc.bar_time because no value was defined for it when creating the object.

Object placeholders can also be created by declaring na object names using the following:

pivotPoint foundPoint = na

This example displays a label where high pivots are detected. The pivots are detected `legsInput` bars after they occur, so we must plot the label in the past for it to appear on the pivot:

//@version=6 indicator("Pivot labels", overlay = true) int legsInput = input(10) // Define the `pivotPoint` UDT. type pivotPoint int x float y string xloc = xloc.bar_time // Detect high pivots. pivotHighPrice = ta.pivothigh(legsInput, legsInput) if not na(pivotHighPrice) // A new high pivot was found; display a label where it occurred `legsInput` bars back. foundPoint = pivotPoint.new(time[legsInput], pivotHighPrice) label.new(foundPoint.x, foundPoint.y, str.tostring(foundPoint.y, format.mintick), foundPoint.xloc, textcolor = color.white)

Take note of this line from the above example:

foundPoint = pivotPoint.new(time[legsInput], pivotHighPrice)

This could also be written using the following:

pivotPoint foundPoint = na foundPoint := pivotPoint.new(time[legsInput], pivotHighPrice)

When using the var keyword while declaring a variable assigned to an object of a user-defined type, the keyword automatically applies to all the object's fields:

//@version=6 indicator("Objects using `var` demo") //@type A custom type to hold index, price, and volume information. type BarInfo int index = bar_index float price = close float vol = volume //@variable A `BarInfo` instance whose fields persist through all iterations, starting from the first bar. var BarInfo firstBar = BarInfo.new() //@variable A `BarInfo` instance declared on every bar. BarInfo currentBar = BarInfo.new() // Plot the `index` fields of both instances to compare the difference. plot(firstBar.index) plot(currentBar.index)

It's important to note that assigning an object to a variable that uses the varip keyword does *not* automatically allow the object's fields to persist without rolling back on each *intrabar* update. One must apply the keyword to each desired field in the type declaration to achieve this behavior. For example:

//@version=6 indicator("Objects using `varip` fields demo") //@type A custom type that counts the bars and ticks in the script's execution. type Counter int bars = 0 varip int ticks = 0 //@variable A `Counter` object whose reference persists throughout all bars. var Counter counter = Counter.new() // Add 1 to the `bars` and `ticks` fields. The `ticks` field is not subject to rollback on unconfirmed bars. counter.bars += 1 counter.ticks += 1 // Plot both fields for comparison. plot(counter.bars, "Bar counter", color.blue, 3) plot(counter.ticks, "Tick counter", color.purple, 3)

Note that:

- We used the var keyword to specify that the `Counter` object assigned to the `counter` variable persists throughout the script's execution.
- The `bars` field rolls back on realtime bars, whereas the `ticks` field does not since we included varip in its declaration.

Changing field values

The value of an object's fields can be changed using the := reassignment operator.

This line of our previous example:

foundPoint = pivotPoint.new(time[legsInput], pivotHighPrice)

Could be written using the following:

foundPoint = pivotPoint.new() foundPoint.x := time[legsInput] foundPoint.y := pivotHighPrice

Collecting objects

Pine Script *collections* (arrays, matrices, and maps) can contain references to UDT objects, enabling programmers to add virtual dimensions to their data structures. To create a collection of a user-defined type, call the collection type's `*.new*()` function with the UDT name in the function's *type template*.

The following line of code declares a variable that holds the ID of an empty array that can store references to objects of a `pivotPoint` user-defined type:

pivotHighArray = array.new<pivotPoint>()

To explicitly declare the type of a variable as an array, matrix, or map of a user-defined type, prefix the variable declaration with collection's *type keyword* followed by its *type template*. For example:

var array<pivotPoint> pivotHighArray = na pivotHighArray := array.new<pivotPoint>()

See the Collections section of the Type system page to learn about type templates.

Let's use what we have learned to create a script that detects high pivot points. The script first collects historical pivot information in an array. It then loops through the array on the last historical bar, creating a label for each pivot and connecting the pivots with lines:

//@version=6 indicator("Pivot Points High", overlay = true) int legsInput = input(10) // Define the `pivotPoint` UDT containing the time and price of pivots. type pivotPoint int openTime float level // Create an empty `pivotPoint` array. var pivotHighArray = array.new<pivotPoint>() // Detect new pivots (`na` is returned when no pivot is found). pivotHighPrice = ta.pivothigh(legsInput, legsInput) // Add a new `pivotPoint` object to the end of the array for each detected pivot. if not na(pivotHighPrice) // A new pivot is found; create a new object of `pivotPoint` type, setting its `openTime` and `level` fields. newPivot = pivotPoint.new(time[legsInput], pivotHighPrice) // Add the new pivot object to the array. array.push(pivotHighArray, newPivot) // On the last historical bar, draw pivot labels and connecting lines. if barstate.islastconfirmedhistory var pivotPoint previousPoint = na for eachPivot in pivotHighArray // Display a label at the pivot point. label.new(eachPivot.openTime, eachPivot.level, str.tostring(eachPivot.level, format.mintick), xloc.bar_time, textcolor = color.white) // Create a line between pivots. if not na(previousPoint) // Only create a line starting at the loop's second iteration because lines connect two pivots. line.new(previousPoint.openTime, previousPoint.level, eachPivot.openTime, eachPivot.level, xloc = xloc.bar_time) // Save the pivot for use in the next iteration. previousPoint := eachPivot

Copying objects

In Pine, objects are assigned by reference. When an existing object is assigned to a new variable, both point to the same object.

In the example below, we create a `pivot1` object and set its `x` field to 1000. Then, we declare a `pivot2` variable containing the reference to the `pivot1` object, so both point to the same instance. Changing `pivot2.x` will thus also change `pivot1.x`, as both refer to the `x` field of the same object:

//@version=6 indicator("") type pivotPoint int x float y pivot1 = pivotPoint.new() pivot1.x := 1000 pivot2 = pivot1 pivot2.x := 2000 // Both plot the value 2000. plot(pivot1.x) plot(pivot2.x)

To create a copy of an object that is independent of the original, we can use the built-in `copy()` method in this case.

In this example, we declare the `pivot2` variable referring to a copied instance of the `pivot1` object. Now, changing `pivot2.x` will not change `pivot1.x`, as it refers to the `x` field of a separate object:

//@version=6 indicator("") type pivotPoint int x float y pivot1 = pivotPoint.new() pivot1.x := 1000 pivot2 = pivotPoint.copy(pivot1) pivot2.x := 2000 // Plots 1000 and 2000. plot(pivot1.x) plot(pivot2.x)

It's important to note that the built-in `copy()` method produces a *shallow copy* of an object. If an object contains fields that reference objects of user-defined types or built-in *special types* (array, matrix, map, line, linefill, box, polyline, label, table, chart.point, footprint, or volume_row), those fields in a shallow copy of the object will point to the same instances as the original.

In the following example, we have defined an `InfoLabel` type with a label as one of its fields. The script instantiates a `shallow` copy of the `parent` object, then calls a user-defined `set()` method to update the `info` and `lbl` fields of each object. Since the `lbl` field of both objects points to the same label instance, changes to this field in either object affect the other:

//@version=6 indicator("Shallow Copy") type InfoLabel string info label lbl method set(InfoLabel this, int x = na, int y = na, string info = na) => if not na(x) this.lbl.set_x(x) if not na(y) this.lbl.set_y(y) if not na(info) this.info := info this.lbl.set_text(this.info) var parent = InfoLabel.new("", label.new(0, 0)) var shallow = parent.copy() parent.set(bar_index, 0, "Parent") shallow.set(bar_index, 1, "Shallow Copy")

To produce a *deep copy* of an object with all of its special type fields pointing to independent instances, we must explicitly copy those fields as well.

In this example, we have defined a `deepCopy()` method that instantiates a new `InfoLabel` object with its `lbl` field pointing to a copy of the original's field. Changes to the `deep` copy's `lbl` field will not affect the `parent` object, as it points to a separate instance:

//@version=6 indicator("Deep Copy") type InfoLabel string info label lbl method set(InfoLabel this, int x = na, int y = na, string info = na) => if not na(x) this.lbl.set_x(x) if not na(y) this.lbl.set_y(y) if not na(info) this.info := info this.lbl.set_text(this.info) method deepCopy(InfoLabel this) => InfoLabel.new(this.info, this.lbl.copy()) var parent = InfoLabel.new("", label.new(0, 0)) var deep = parent.deepCopy() parent.set(bar_index, 0, "Parent") deep.set(bar_index, 1, "Deep Copy")

Shadowing

To avoid potential conflicts in the eventuality where namespaces added to Pine Script in the future would collide with UDT names in existing scripts; as a rule, UDT names shadow the language's namespaces. For

example, a UDT can have the same name as some built-in types, such as line or table.

However, scripts cannot use the following keywords for fundamental types as names for UDTs: int, float, string, bool, and color.

Operators

Introduction

Some operators are used to build *expressions* returning a result:

- Arithmetic operators
- Comparison operators
- Logical operators
- The ?: ternary operator
- The [] history-referencing operator

Other operators are used to assign values to variables:

- `=` is used to assign a value to a variable, **but only when you declare the variable** (the first time you use it)
- `:=` is used to assign a value to a **previously declared variable**. The following operators can also be used in such a way: `+=`, `-=`, `*=`, `/=`, `%=`

As is explained in the Type system page, *qualifiers* and *types* play a critical role in determining the type of results that expressions yield. This, in turn, has an impact on how and with what functions you will be allowed to use those results. Expressions always return a value with the strongest qualifier used in the expression, e.g., if you multiply an “input int” with a “series int”, the expression will produce a “series int” result, which you will not be able to use as the argument to `length` in ta.ema().

This script will produce a compilation error:

//@version=6 indicator("") lenInput = input.int(14, "Length") factor = year > 2020 ? 3 : 1 adjustedLength = lenInput * factor ma = ta.ema(close, adjustedLength) // Compilation error! plot(ma)

The compiler will complain: *Cannot call 'ta.ema' with argument 'length'='adjustedLength'. An argument of 'series int' type was used but a 'simple int' is expected;*. This is happening because `lenInput` is an “input int” but `factor` is a “series int” (it can only be determined by looking at the value of year on each bar). The `adjustedLength` variable is thus assigned a “series int” value. Our problem is that the Reference Manual entry for ta.ema() tells us that its `length` parameter requires a “simple” value, which is a weaker qualifier than “series”, so a “series int” value is not allowed.

The solution to our conundrum requires:

- Using another moving average function that supports a “series int” length, such as ta.sma(), or
- Not using a calculation producing a “series int” value for our length.

Arithmetic operators

There are five arithmetic operators in Pine Script®:

Operator	Meaning
+	Addition and string concatenation
-	Subtraction

Operator	Meaning
*	Multiplication
/	Division
%	Modulo (remainder after division)

The arithmetic operators above are all *binary*, meaning they need two *operands* — or values — to work on, as in the example operation `1 + 2`. The `+` and `-` can also be *unary* operators, which means they work on one operand, as in the example values `-1` or `+1`.

If both operands are numbers but at least one of these is of float type, the result will also be a float. If both operands are of int type, the result will also be an int. If at least one operand is na, the result is also na.

Note that when using the division operator with “int” operands, if the two “int” values are not evenly divisible, the result of the division is always a number with a fractional value, e.g., `5/2 = 2.5`. To discard the fractional remainder, wrap the division with the int() function, or round the result using math.round(), math.floor(), or math.ceil().

The `+` operator also serves as the concatenation operator for strings. `"EUR"+"USD"` yields the `"EURUSD"` string.

The `%` operator calculates the modulo by rounding down the quotient to the lowest possible value. Here is an easy example that helps illustrate how the modulo is calculated behind the scenes:

//@version=6 indicator("Modulo function") modulo(series int a, series int b) => a - b * math.floor(nz(a/b)) plot(modulo(-1, 100))

Comparison operators

There are six comparison operators in Pine Script:

Operator	Meaning
<	Less Than
<=	Less Than or Equal To
!=	Not Equal
==	Equal
>	Greater Than
>=	Greater Than or Equal To

Comparison operations are binary, and return a result of type “bool”, i.e., true or false. The == equal and `!=` not equal operators can work with operands of any fundamental type, such as colors and strings, while the other comparison operators are only applicable to numerical values. Therefore, `"a" != "b"` is a valid comparison, but `"a" > "b"` is invalid.

Examples:

1 > 2 // false 1 != 1 // false close >= open // Depends on values of `close` and `open`

Logical operators

There are three logical operators in Pine Script:

Operator	Meaning
`not`	Negation
`and`	Logical Conjunction
`or`	Logical Disjunction

The operator `not` is unary. When applied to a `true`, operand the result will be `false`, and vice versa.

`and` operator truth table:

a	b	a and b
true	true	true
true	false	false
false	true	false
false	false	false

`or` operator truth table:

a	b	a or b
true	true	true
true	false	true
false	true	true
false	false	false

`?:` ternary operator

The ?: ternary operator is used to create expressions of the form:

condition ? valueWhenConditionIsTrue : valueWhenConditionIsFalse

The ternary operator returns a result that depends on the value of `condition`. If it is `true`, then it returns `valueWhenConditionIsTrue`. Otherwise, if `condition` is `false`, then it returns `valueWhenConditionIsFalse`.

A combination of ternary expressions can be used to achieve the same effect as a switch structure, e.g.:

timeframe.isintraday ? color.red : timeframe.isdaily ? color.green : timeframe.ismonthly ? color.blue : na

The example is calculated from left to right:

- If timeframe.isintraday is `true`, then `color.red` is returned. If it is `false`, then timeframe.isdaily is evaluated.
- If timeframe.isdaily is `true`, then `color.green` is returned. If it is `false`, then timeframe.ismonthly is evaluated.
- If timeframe.ismonthly is `true`, then `color.blue` is returned, otherwise na is returned.

Note that, in contrast to conditional structures, the ternary operator does *not* create local scopes.

[] history-referencing operator

It is possible to refer to past values of time series using the [] history-referencing operator. Past values are values a variable had on bars preceding the bar where the script is currently executing — the *current bar*. See the Execution model page for more information about the way scripts are executed on bars.

The [] operator is used after a variable, expression or function call. The value used inside the square brackets of the operator is the offset in the past we want to refer to. To refer to the value of the volume built-in variable two bars away from the current bar, one would use `volume[2]`.

Because series grow dynamically, as the script calculates on successive bars, a constant historical offset refers to different bars. Let's see how the value returned by the same offset is dynamic, and why series are very different from arrays. In Pine Script, the close variable, or `close[0]` which is equivalent, holds the value of the current bar's "close". If your code is now executing on the **third** bar of the *dataset* (the set of all bars on your chart), `close` will contain the price at the close of that bar, `close[1]` will contain the price at the close of the preceding bar (the dataset's second bar), and `close[2]`, the first bar. `close[3]` will return na because no bar exists in that position, and thus its value is *not available*.

When the same code is executed on the next bar, the **fourth** in the dataset, `close` will now contain the closing price of that bar, and the same `close[1]` used in your code will now refer to the "close" of the third bar in the dataset. The close of the first bar in the dataset will now be `close[3]`, and this time `close[4]` will return na.

In the Pine Script runtime environment, as your code is executed once for each historical bar in the dataset, starting from the left of the chart, Pine Script is adding a new element in the series at index 0 and pushing the pre-existing elements in the series one index further away. Arrays, in comparison, can have constant or variable sizes, and their content or indexing structure is not modified by the runtime environment. Pine Script series are thus very different from arrays and only share familiarity with them through their indexing syntax.

When the market for the chart's symbol is open and the script is executing on the chart's last bar, the *realtime bar*, close returns the value of the current price. It will only contain the actual closing price of the realtime bar the last time the script is executed on that bar, when it closes.

Pine Script has a variable that contains the number of the bar the script is executing on: bar_index. On the first bar, bar_index is equal to 0 and it increases by 1 on each successive bar the script executes on. On the last bar, bar_index is equal to the number of bars in the dataset minus one.

There is another important consideration to keep in mind when using the `[]` operator in Pine Script. We have seen cases when a history reference may return the na value. na represents a value which is not a number and using it in any expression will produce a result that is also na (similar to NaN). Such cases often happen during the script's calculations in the early bars of the dataset, but can also occur in later bars under certain conditions. If your code does not explicitly handle these special cases using the na() and nz() functions, na values can introduce invalid results in your script's calculations that can affect calculations all the way to the realtime bar.

These are all valid uses of the [] operator:

high[10] ta.sma(close, 10)[1] ta.highest(high, 10)[20] close > nz(close[1], open)

Note that the [] operator can only be used once on the same value. This is not allowed:

close[1][2] // Error: incorrect use of [] operator

Operator precedence

The order of calculations is determined by the operators' precedence. Operators with greater precedence are calculated first. Below is a list of operators sorted by decreasing precedence:

Precedence	Operator
9	`[]`
8	unary `+`, unary `-`, `not`
7	`*`, `/`, `%`
6	`+`, `-`
5	`>`, `<`, `>=`, `<=`
4	`==`, `!=`
3	`and`
2	`or`
1	`?:`

If in one expression there are several operators with the same precedence, then they are calculated left to right.

If the expression must be calculated in a different order than precedence would dictate, then parts of the expression can be grouped together with parentheses.

`=` assignment operator

The `=` operator assigns an initial value or reference to a declared variable. It means *this is a new variable, and it starts with this value*.

These are all valid variable declarations:

i = 1 MS_IN_ONE_MINUTE = 1000 * 60 showPlotInput = input.bool(true, "Show plots") pHi = ta.pivothigh(5, 5) plotColor = color.green

See the Variable declarations page for more information on how to declare variables.

`:=` reassignment operator

The `:=` is used to *reassign* a value to an existing variable. It says *use this variable that was declared earlier in my script, and give it a new value*.

Variables which have been first declared, then reassigned using `:=`, are called *mutable* variables. All the following examples are valid variable reassignments. You will find more information on how var works in the section on the `var` declaration mode:

//@version=6 indicator("", "", true) // Declare `pHi` and initilize it on the first bar only. var float pHi = na // Reassign a value to `pHi` pHi := nz(ta.pivothigh(5, 5), pHi) plot(pHi)

Note that:

- We declare `pHi` with this code: `var float pHi = na`. The var keyword tells Pine Script that we only want that variable initialized with na on the dataset's first bar. The `float` keyword tells the compiler we are declaring a variable of type “float”. This is necessary because, contrary to most cases, the compiler cannot automatically determine the type of the value on the right side of the `=` sign.
- While the variable declaration will only be executed on the first bar because it uses var, the `pHi := nz(ta.pivothigh(5, 5), pHi)` line will be executed on all the chart's bars. On each bar, it evaluates if the ta.pivothigh() call returns na because that is what the function does when it hasn't found a new pivot. The nz() function is the one doing the “checking for na” part. When its first argument (`ta.pivothigh(5, 5)`) is na, it returns the second argument (`pHi`) instead of the first. When ta.pivothigh() returns the price point of a newly found pivot, that value is assigned to `pHi`. When it returns na because no new pivot was found, we assign the previous value of `pHi` to itself, in effect preserving its previous value.

The output of our script looks like this:

Note that:

- The line preserves its previous value until a new pivot is found.
- Pivots are detected five bars after the pivot actually occurs because our `ta.pivothigh(5, 5)` call says that we require five lower highs on both sides of a high point for it to be detected as a pivot.

See the Variable reassignment section for more information on how to reassign values to variables.

Compound assignment operators

A *compound assignment operator* combines an arithmetic operator with the reassignment operator. It provides a shorthand way to perform an arithmetic calculation on a variable and then assign the result back to that same variable.

For example, `counter += 1` adds 1 to the current value of a `counter` variable and assigns the new incremented value back to `counter`. This operation is equivalent to `counter := counter + 1`. Note that a variable must be declared before a script can use a compound assignment operator on it.

There are five compound assignment operators in Pine Script:

Operator	Meaning
`+=`	Addition assignment and string concatenation
`-=`	Subtraction assignment
`*=`	Multiplication assignment
`/=`	Division assignment
`%=`	Modulo (remainder after division) assignment

This example executes various compound assignment operations on one “float” variable, `x`, and traces how each operation changes the variable’s stored value. The script draws a table to show each operation and its resulting value of `x` after reassignment. A float input can change the initial value assigned to `x`, which in turn changes the result of each row’s calculation:

```
//@version=6 indicator("Compound assignment operators demo") //@variable The initial value assigned to the `x` variable. float initialInput = input.float(12, "Initial value of `x`", minval = 0) //@variable A `table` that displays the executed operations and traces their results on the `x` variable. var table resultsTable = table.new(position.middle_center, 3, 7, color.white, color.black, 1, color.white, 2) //@function Initializes a `resultsTable` row to show an `operation` and the resulting `x` variable value. displayResult(int rowID, string operation, string description, float x, bool initialRow = false) => //@variable Is yellow only for initial row. Otherwise, alternates row colors: blue if `rowID` is even, white if odd. color rowColor = initialRow ? color.yellow : rowID % 2 == 0 ? color.rgb(33, 149, 243, 75) : color.white // Display the `operation` in the row's first cell. resultsTable.cell(0, rowID, operation, bgcolor = rowColor, text_font_family = font.family_monospace) // Display the operation's `description` in the row's second cell. resultsTable.cell(1, rowID, description, bgcolor = rowColor, text_halign = text.align_left) // Show the result of the `operation` on the `x` variable by outputting its current value in the row's third cell. resultsTable.cell(2, rowID, str.format(" x = {0}", x), bgcolor = rowColor, text_halign = text.align_left) if barstate.islastconfirmedhistory // Display the table's header cells. resultsTable.cell(0, 0, "OPERATION") resultsTable.cell(1, 0, "DESCRIPTION") resultsTable.cell(2, 0, "RESULT") // Declare and initialize the `x` variable, and display an initial row in the table to show its starting value. float x = initialInput displayResult(1, "float x = initialInput", "Declares \"float\" variable `x` and initializes to input value", x, true) // Execute various compound assignments successively on `x`, displaying each operation and its result in the table. x += 6 displayResult(2, "x += 6", "Adds 6 to `x` and reassigns result", x) x -= 3 displayResult(3, "x -= 3", "Subtracts 3 from `x` and reassigns result", x) x *= 4 displayResult(4, "x *= 4", "Multiplies `x` by 4 and reassigns result", x) x /= 8 displayResult(5, "x /= 8", "Divides `x` by 8 and reassigns result", x) x %= 5 displayResult(6, "x %= 5", "Calculates remainder of dividing `x` by 5 and reassigns result", x)
```

The `+=` operator also acts as a concatenation operator when both operands are strings. For example, if a `symTicker` variable holds the string `"NASDAQ:"`, then `symTicker += "AAPL"` appends the `"AAPL"` characters to the `"NASDAQ:"` characters to create a new “string” value `"NASDAQ:AAPL"`, which is then assigned back to `symTicker`.

Script structure

A Pine script follows this general structure:

```
<version>
<declaration_statement>
<code>
```

Version

A compiler annotation in the following form tells the compiler which of the versions of Pine Script® the script is written in:

//@version=6

- The version number is a number from 1 to 6.
- The compiler annotation is not mandatory. When omitted, version 1 is assumed. It is strongly recommended to always use the latest version of the language.
- While it is synctactically correct to place the version compiler annotation anywhere in the script, it is much more useful to readers when it appears at the top of the script.

Notable changes to the current version of Pine Script are documented in the Release notes.

Declaration statement

All Pine scripts must contain one declaration statement, which is a call to one of these functions:

- indicator()
- strategy()
- library()

The declaration statement:

- Identifies the type of the script, which in turn dictates which content is allowed in it, and how it can be used and executed.
- Sets key properties of the script such as its name, where it will appear when it is added to a chart, the precision and format of the values it displays, and certain values that govern its runtime behavior, such as the maximum number of drawing objects it will display on the chart. With strategies, the properties include parameters that control backtesting, such as initial capital, commission, slippage, etc.

Each script type has distinct basic requirements. Scripts that do not meet these criteria cause a compilation error:

- Indicators must call at least one function that creates a script output, such as plot(), plotshape(), barcolor(), line.new(), log.info(), alert(), etc.
- Strategies must call at least one order placement command or other output function.
- Libraries must export at least one user-defined function, method, type, or enum.

Code

Lines in a script that are not comments or compiler annotations are *statements*, which implement the script's algorithm. A statement can be one of these:

- variable declaration
- variable reassignment
- function definition
- built-in function call, user-defined function call or a library function call
- if, for, while, switch, type, or enum *structure*.

Statements can be arranged in multiple ways:

- Some statements can be expressed in one line, like most variable declarations, lines containing only a function call or single-line function declarations. Lines can also be wrapped (continued on multiple lines). Multiple one-line statements can be concatenated on a single line by using the comma as a separator.
- Others statements such as structures or multiline function definitions always require multiple lines because they require a *local block*. A local block must be indented by a tab or four spaces. Each local block defines a distinct *local scope*.
- Statements in the *global scope* of the script (i.e., which are not part of local blocks) cannot begin with white space (a space or a tab). Their first character must also be the line's first character. Lines beginning in a line's first position become by definition part of the script's *global scope*.

A simple valid Pine Script indicator can be generated in the Pine Script Editor by using the “Open” button and choosing “New blank indicator”:

//@version=6 indicator("My Script") plot(close)

This indicator includes three local blocks, one in the `barIsUp()` function declaration, and two in the variable declaration using an if structure:

//@version=6 indicator("", "", true) // Declaration statement (global scope) barIsUp() => // Function declaration (global scope) close > open // Local block (local scope) plotColor = if barIsUp() // Variable declaration (global scope) color.green // Local block (local scope) else color.red // Local block (local scope) bgcolor(color.new(plotColor, 70)) // Call to a built-in function (global scope)

You can bring up a simple Pine Script strategy by selecting “New blank strategy” instead:

//@version=6 strategy("My Strategy", overlay=true, margin_long=100, margin_short=100) longCondition = ta.crossover(ta.sma(close, 14), ta.sma(close, 28)) if (longCondition) strategy.entry("My Long Entry Id", strategy.long) shortCondition = ta.crossunder(ta.sma(close, 14), ta.sma(close, 28)) if (shortCondition) strategy.entry("My Short Entry Id", strategy.short)

Comments

Double slashes (`//`) define comments in Pine Script. Comments can begin anywhere on the line. They can also follow Pine Script code on the same line:

//@version=6 indicator("") // This line is a comment a = close // This is also a comment plot(a)

The Pine Editor has a keyboard shortcut to comment/uncomment lines: `ctrl` + `/`. You can use it on multiple lines by highlighting them first.

Line wrapping

Scripts can use *line wrapping* to define a long *single line* of code across *multiple* lines. Generally, each wrapped line after the first can use any indentation length *except* multiples of four, because Pine uses four-space or tab indentations to define local code blocks.

For example, consider the following line of code:

float x = open + high + low + close

We can distribute any part of this single line of code across two or more lines. Within a wrapped statement or expression, the subsequent lines can use different indentation lengths, and can also include comments without disrupting the code:

float x = open + high + // Indented by 2 spaces. low + // Indented by 5 spaces. close // Indented by 10 spaces.

If parts of a line-wrapped statement or expression are within *parentheses* `( )` or *square brackets* `[ ]`, all code between those characters *does not* have any restriction on indentation length; the code can use *any* number of spaces for indentation, including zero and any multiple of four.

For example, this script demonstrates various ways in which code enclosed in parentheses or square brackets can wrap across multiple lines:

//@version=6 indicator("Line wrapping within parentheses and brackets demo") // We can enclose expressions in parentheses to wrap them across multiple lines using a four-space indentation. float x = (open + high + low + close) // We can wrap a long function call across two lines, for a minimal line wrapping style. plot(ta.sma(close, 14), title = "Avg close", color = color.new(color.purple, 70), style = plot.style_area, force_overlay = true, display = display.all - display.status_line) // No indentation. // We can apply any indentation to line wraps in square brackets as well. [macd, signal, hist] = ta.macd(x, 12, 26, 9) // We can also apply different indentation lengths across the same line-wrapped expression. // The enclosing parentheses or square brackets can start and end on separate lines from the wrapped content. float osc = (signal // Indented by one space. + 2 * hist // Indented by four spaces. * math.abs(ta.change(macd, 10) / ta.stdev(macd, 10)) // Indented by eight spaces.) // No indentation. plot(series = osc, title = "Custom osc", // No indentation. color = osc > osc[// Indented by two spaces. 1 // Indented by 12 spaces.] ? color.green : color.red, // Indented by four spaces. linewidth = 3 // Indented by six spaces.) // Indented by nine spaces.

Expressions inside *local* code blocks can also use line wrapping. A local block requires indenting each line that belongs to its scope by four spaces or a tab relative to the local block's header. Therefore, we recommend indenting any wrapped lines inside a local block by a *larger* number of spaces than that of the block to preserve scope readability. For example:

upDown(float s) => // These lines are indented by four spaces relative to the `upDown()` function header because they belong to the function's scope. var int ud = 0 bool isEqual = s == s[1] bool isGrowing = s > s[1]

// Within the local block, this statement wraps across multiple lines, where each part uses // an indentation length that is larger than the required indentation for the function block. ud := isEqual ? 0 : isGrowing ? (ud <= 0 ? 1 : ud + 1) : (ud >= 0 ? -1 : ud - 1)

Scripts can also create line-wrapped expressions by using multiline strings. A multiline string, enclosed by three pairs of quotation marks (e.g., `"""..."""`) or apostrophes (e.g., `'''...'''`), can occupy multiple lines in the Pine Editor, where each part of the code between the """ or ''' delimiters represents *literal text*. If an expression uses a string spanning multiple lines as an operand or argument, Pine still treats that expression as part of a *single* line of code. For example:

//@version=6 indicator("Line wrapping with multiline strings demo", overlay = true) //@variable A string indicating whether the bar is rising, falling, or neutral. // The strings in the ternary operations span multiple visible lines, but the entire expression is // considered part of a single line. string labelText = close > open ? """ Bar is rising. """ : close < open ? """ Bar is falling. """ : """ Bar is neutral. """ // Draw a label to display the text from the current `labelText` string. label.new(bar_index, close, labelText)

Note that:

- A multiline string treats *all* characters between the """ or ''' delimiters as literal text, including the *leading spaces* on each visible code line. If a line in a multiline string definition is indented by any number of spaces, relative to *column 0* in the Pine Editor, the line in the resulting string also includes that indentation.

Compiler annotations

Compiler annotations are comments that issue special instructions for a script:

- `//@version=` specifies the PineScript version that the compiler will use. The number in this annotation should not be confused with the script's version number, which updates on every saved change to the code.
- `//@description` sets a custom description for scripts that use the library() declaration statement.
- `//@function`, `//@param` and `//@returns` add custom descriptions for a user-defined function or method, its parameters, and its result when placed above the function declaration.
- `//@type` adds a custom description for a user-defined type (UDT) when placed above the type declaration.
- `//@enum` adds a custom description for an enum types when placed above the enum declaration.
- `//@field` adds a custom description for the field of a user-defined type (UDT) or an enum types when placed above the type or enum declaration.
- `//@variable` adds a custom description for a variable when placed above its declaration.
- `//@strategy_alert_message` provides a default message for strategy scripts to pre-fill the “Message” field in the alert creation dialog.

The Pine Editor also features two specialized annotations, `//#region` and `//#endregion`, that create *collapsible* code regions. Clicking the dropdown arrow next to a `//#region` line collapses all the code between that line and the nearest `//#endregion` annotation below it.

This example draws a triangle using three interactively selected points on the chart. The script illustrates how one can use compiler and Editor annotations to document code and make it easier to navigate:

```
//@version=6 indicator("Triangle", "", true) //#region ———————————————————————————— Constants and inputs int TIME_DEFAULT = 0 float PRICE_DEFAULT = 0.0 x1Input = input.time(TIME_DEFAULT, "Point 1", inline = "1", confirm = true) y1Input = input.price(PRICE_DEFAULT, "", inline = "1", tooltip = "Pick point 1", confirm = true) x2Input = input.time(TIME_DEFAULT, "Point 2", inline = "2", confirm = true) y2Input = input.price(PRICE_DEFAULT, "", inline = "2", tooltip = "Pick point 2", confirm = true) x3Input = input.time(TIME_DEFAULT, "Point 3", inline = "3", confirm = true) y3Input = input.price(PRICE_DEFAULT, "", inline = "3", tooltip = "Pick point 3", confirm = true) //#endregion //#region ———————————————————————————— Types and functions // @type Used to represent the coordinates and color to draw a triangle. // @field time1 Time of first point. // @field time2 Time of second point. // @field time3 Time of third point. // @field price1 Price of first point. // @field price2 Price of second point. // @field price3 Price of third point. // @field lineColor Color to be used to draw the triangle lines. type Triangle int time1 int time2 int time3 float price1 float price2 float price3 color lineColor //@function Draws a triangle using the coordinates of the `t` object. //@param t (Triangle) Object representing the triangle to be drawn. //@returns The ID of the last line drawn. drawTriangle(Triangle t) => line.new(t.time1, t.price1, t.time2, t.price2, xloc = xloc.bar_time, color = t.lineColor) line.new(t.time2, t.price2, t.time3, t.price3, xloc = xloc.bar_time, color = t.lineColor) line.new(t.time1, t.price1, t.time3, t.price3, xloc = xloc.bar_time, color = t.lineColor) //#endregion //#region ———————————————————————————— Calculations // Draw the triangle only once on the last historical bar. if barstate.islastconfirmedhistory //@variable Used to hold the Triangle object to be drawn. Triangle triangle = Triangle.new() triangle.time1 := x1Input triangle.time2 := x2Input triangle.time3 := x3Input triangle.price1 := y1Input triangle.price2 := y2Input triangle.price3 := y3Input triangle.lineColor := color.purple drawTriangle(triangle) //#endregion
```

Type system

Introduction

Pine Script® uses a system of *types* and *type qualifiers* to categorize the data in a script and indicate where and how the script can use it. This system applies to all values and references in a script, and to the variables, function parameters, and fields that store them.

Types in Pine Script indicate the kinds of information that a script's data represents. Some types directly represent *values*, such as numbers, logical conditions, colors, or text, while others define *structures* for special tasks, such as displaying visuals on the chart. Qualifiers indicate when the values of any given type are accessible, and whether those values can change across script executions.

The combination of a type and a qualifier forms a *qualified type*, which determines the operations and functions with which a value or reference is compatible.

> NoteFor the sake of brevity, we often use the term "type" when referring to a qualified type.

The type system closely connects to the execution model and its time series structure – together, they determine how a script behaves as it runs on a dataset. Although it's possible to write simple scripts without understanding these foundational topics, learning about them and their nuances is key to mastering Pine Script.

Qualifiers

Pine's type qualifiers (const, input, simple, and series) indicate *when* values in a script are accessible – either at compile time, input time, or runtime – and whether those values can change across script executions:

`"const"`

Established at *compile time*, when the user saves the script in the Pine Editor or applies the script to a dataset. Values qualified as "const" remain *constant* during every script execution.

`"input"`

Established at *input time*, when the system confirms input data from the script's "Settings/Inputs" tab or the chart. Similar to "const" values, "input" values *do not change* as the script runs on the dataset.

`"simple"`

Established by the script at runtime, on the *first* available bar. On all subsequent bars, values qualified as "simple" do not change.

`"series"`

Dynamic. Values qualified as "series" are available at runtime, and they are the **only** values that can change across bars.

> NoteThe “const”, “input”, and “simple” qualifiers apply exclusively to value types. Pine’s reference types, such as label and array types, automatically inherit the “series” qualifier.

Pine Script uses the following *qualifier hierarchy* to determine the compatibility of values in a script’s calculations:

```
const < input < simple < series
```

In this hierarchy, “const” is the *lowest* (*weakest*) qualifier, and “series” is the *highest* (*strongest*). Any variable, parameter, or operation that accepts a value with a specific qualifier also allows a value of the same type with a *weaker* qualifier, but **not** one that is stronger.

For instance, a function parameter that accepts a value of the “simple int” qualified type also allows a value of the “input int” or “const int” type, because “const” and “input” are *lower* than “simple” in the qualifier hierarchy. However, the parameter *cannot* accept a “series int” value, because “series” is higher in the hierarchy than “simple”.

Pine also uses this hierarchy to determine the qualifiers assigned to the results of expressions, i.e., function calls and operations. The returned types of an expression always inherit the *strongest* qualifier in the calculation. For example, an expression that performs a calculation using “input” and “simple” values returns “simple” results, because “simple” is a *stronger* qualifier than “input”.

Note that a script **cannot** change the qualifier of a returned value to one that is lower in the hierarchy to make it compatible with specific operations or functions. For instance, if a calculation returns a value qualified as “series”, the script cannot modify that value’s qualifier later to enable using it in code that requires “simple” or “const” values.

The following sections explain the behavior of each type qualifier, as well as the built-in keywords that programmers can use to specify qualifiers in their code.

const

Values qualified as “const” are available at *compile time*, before the script starts its first execution. Compilation occurs when the user saves the script in the Pine Editor, and immediately before a script starts to run on the chart or in another location. Values with the “const” qualifier remain constant after compilation; they do not change during any script execution. All *literal values* and the results of expressions that use only values qualified as “const” automatically inherit the “const” qualifier.

The following list shows a few values of each fundamental type. All of these represent literal values if a script includes them directly in its source code:

- *literal int*: `1`, `-1`, `42`
- *literal float*: `1.`, `1.0`, `3.14`, `6.02E-23`, `3e8`
- *literal bool*: `true`, `false`
- *literal color*: `#FF55C6`, `#FF55C6ff`
- *literal string*: `"A text literal"`, `"Embedded single quotes 'text'"`, `'Embedded double quotes "text"'`

Scripts can declare variables that hold “const” values, and use those variables to calculate other constants. In the example below, we use “const” variables to set the title of a script and its plots. The script compiles successfully, because the indicator() and plot() calls used in the code both require a `title` argument of the *“const string”* qualified type:

//@version=6 // All of the following global variables automatically inherit the "const" qualifier, // because their "string" values are constants that are available at *compile time*. //@variable The indicator's title. INDICATOR_TITLE = "const demo" //@variable The title of the first plot. var PLOT1_TITLE = "High" //@variable The title of the second plot. PLOT2_TITLE = "Low" //@variable The title of the third plot. PLOT3_TITLE = "Midpoint between " + PLOT1_TITLE + " and " + PLOT2_TITLE // Set the title of the indicator using `INDICATOR_TITLE`. indicator(title = INDICATOR_TITLE, overlay = true) // Set the title of each plot using the `PLOT*_TITLE` variables. plot(high, PLOT1_TITLE) plot(low, PLOT2_TITLE) plot(hl2, PLOT3_TITLE)

Note that:

- All the variables above the indicator() call in this script have the “const” qualifier, because they hold a literal value or the result of operations that use only “const” values.
- All our “const” variables in this example have names in *uppercase snake case* so that they are easy to distinguish in the code, as recommended by our Style guide.
- Although the “const” variables in this script hold constant values, the script initializes them on *every bar*. The only exception is `PLOT1_TITLE`, which the script initializes only on the *first* bar, because its declaration includes the var keyword. See the Declaration modes section of the Variable declarations page to learn more.

Any variable or function parameter that requires a “const” value *cannot* accept a value with the “input”, “simple”, or “series” qualifier, because “const” is the *lowest* qualifier in the qualifier hierarchy.

For example, the following script combines a literal string with the value of syminfo.ticker to set the value of a `scriptTitle` variable. Then, it attempts to use the variable as the `title` argument of the indicator() declaration statement, causing a *compilation error*. The `title` parameter requires a “const string” argument, but `scriptTitle` holds a value of the type *“simple string”*:

//@version=6 //@variable Holds a value intended for use as the `title` argument in the `indicator()` call. // However, this variable's type is "simple string", not "const string", because // the value of `syminfo.ticker` is not available until *after* compilation. var scriptTitle = "My indicator for " + syminfo.ticker // This statement causes an error. The `indicator()` statement cannot use a "simple string" // value as its `title` argument. It requires a "const string" value. indicator(title = scriptTitle) plot(close - open)

Note that:

- The syminfo.ticker variable holds a “simple string” value because it depends on data that is available only at *runtime*. Combining this value with a literal string produces another “simple string” value, because “simple” is a stronger qualifier than “const”.
- We did not name the `scriptTitle` variable using snake case, because our Style guide recommends using *lower camel case* to name variables that do not hold “const” values.

Programmers can restrict the behavior of a variable and force *constant assignments* on each execution by prefixing its declaration with the const keyword, followed by a type keyword or type identifier. If a variable

includes const in its declaration, the script cannot *change* its value with the reassignment or compound assignment operators. This restriction applies even if the new assigned value is still a constant.

For example, the script below declares a `myVar` variable using the const keyword. Then, it attempts to change the variable's value with the += operator, causing a compilation error:

//@version=6 indicator("Cannot reassign `const` variable demo") //@variable A "float" variable declared using the `const` keyword. const float myVar = 0.0 // Using the `+=` operator on `myVar` causes an error, because scripts *cannot* modify variables declared using `const`. myVar += 1.0 plot(myVar)

For a variable of any value type, applying the const keyword to the declaration prevents the script from assigning a value qualified as "input", "simple", or "series". Likewise, if a user-defined function parameter of these types includes the keyword in its declaration, it accepts only "const" values.

> NoteUsing the const keyword in a script is usually optional. However, the keyword is required when defining *exported variables* in libraries.

The following script attempts to use the value of the close variable as the initial value of a `myVar` variable declared using the const keyword. However, close is *not compatible* with the variable, so a compilation error occurs. The value of close is of the type *"series float"*, because it updates from bar to bar, but the `myVar` variable requires a "const float" value:

//@version=6 indicator("Cannot assign values with stronger qualifiers demo") // This declaration causes an error. The value of `close` is of the type "series float", // but `myVar` accepts only a "const float" value. const float myVar = close plot(myVar)

Note that:

- If we remove the const keyword from the variable declaration, the `myVar` variable automatically inherits the "series" qualifier, and no error occurs.

> NoteScripts can also use the const keyword when declaring variables of most special types, such as line, label, and array types. However, this keyword **does not** set the *qualifier* of these variables; it only prevents the script from changing the variable's assigned *reference (ID)* during each execution. Special types and other reference types *always* inherit the "series" qualifier. See the Value vs. reference types section for advanced details.

input

Values qualified as "input" are established at *input time*. They are similar to "const" values, because they are available before the first script execution and never change during runtime. However, unlike "const" values, "input" values depend on user input.

All function parameters that have the "input" qualifier can accept only "input" or "const" values; they do not allow values qualified as "simple" or "series".

Most of the built-in `input.*()` functions return values qualified as “input”. These functions create adjustable inputs in the script’s “Settings/Inputs” tab, enabling users to change specific values in a script without altering its source code. Each time the user changes the value of an input, the script *reloads* across all bars on the chart – from the first available bar to the most recent bar – to update its results using the specified value, as explained in the Execution model page.

> NoteThe only `input*()` function that does *not* return a value qualified as “input” is input.source(). That function returns the value of a built-in price series, such as close, or a value from another script’s plots. Therefore, its return type is *“series float”*, which is not compatible with code that requires “input float” values. See the Source input section of the Inputs page to learn more.

The following script requests the value of an RSI calculated on the dataset for a specific symbol and timeframe, and then plots the result on the chart as columns. The script includes two string inputs that specify the context of the request, and it uses a float input value to set the base of the plotted columns. If the user changes any of these inputs in the “Settings/Inputs” tab, the script reloads to update its results for every bar:

//@version=6 indicator("'input' values demo") //@variable An "input string" value representing the requested symbol. symbolInput = input.string("AAPL", "Symbol") //@variable An "input string" value representing the timeframe of the requested data. timeframeInput = input.string("1D", "Timeframe") //@variable An "input float" value for specifying the base of the plotted columns. colBaseInput = input.float(50.0, "Column base", 0.0, 100.0) //@variable The value of an RSI calculated on the `symbolInput` symbol and `timeframeInput` timeframe. // The `request.security()` function's `symbol` and `timeframe` parameters accept "series string" values, // so they also allow weaker qualified types such as "input string". requestedRSI = request.security(symbol = symbolInput, timeframe = timeframeInput, expression = ta.rsi(close, 14)) // Plot the `requestedRSI` value as columns with the base defined by `colBaseInput`. // This call works, because `histbase` requires an "int" or "float" value with the "const" or "input" qualifier. plot(requestedRSI, "RSI", color.purple, 1, plot.style_columns, histbase = colBaseInput)

Note that:

- The plot() function’s `histbase` parameter, which sets the base of the plotted columns, has the expected type “input int” or “input float”. As such, it cannot accept “simple int/float” or “series int/float” values, because “simple” and “series” are stronger qualifiers than “input”.
- The request.security() function requests data from a specified dataset. Its `symbol` and `timeframe` parameters, which define the context of the request, accept “series string” values by default. Therefore, these parameters also accept “input string” values. See the Other timeframes and data page to learn more about `request.*()` functions.

Some built-in `chart.*` variables also hold “input” values, because these variables update at input time based on changes to the *chart*. Scripts that use these variables reload, executing across the entire dataset again, if any chart changes affect their values.

The example below uses some of these variables to display a gradient background color that incrementally changes over the chart’s visible bars. It uses chart.left_visible_bar_time and chart.right_visible_bar_time to get the timestamps of the leftmost and rightmost visible bars for its calculation, and it uses chart.bg_color

and chart.fg_color to define the start and end colors of the gradient. If the user scrolls or zooms on the chart, or changes the chart's background color, the script reloads to generate new results:

//@version=6 indicator("Built-in 'input' variables demo") //@variable The difference between `time` and the leftmost visible bar's time, relative to the visible range. // The `chart.*` variables in this calculation depend on input data from the visible chart, so their type is // "input int". gradientValue = (time - chart.left_visible_bar_time) / (chart.right_visible_bar_time - chart.left_visible_bar_time) //@variable The gradient color. The `chart.*` variables in this calculation are of the type "input color", because // they depend on the "Background" inputs in the "Canvas" tab of the chart's settings. gradientColor = color.from_gradient(gradientValue, 0, 1, chart.bg_color, chart.fg_color) // Color the background using the `gradientColor` value. bgcolor(gradientColor)

simple

Values qualified as "simple" are established at runtime, while the script executes on the *first* available bar. Similar to values qualified as "input" or "const", "simple" values *do not change* across bars.

All variables and function parameters that have the "simple" qualifier can accept only "simple", "input", or "const" values; they do not allow values qualified as "series".

Many built-in variables, such as most `syminfo.*` and `timeframe.*` variables, hold "simple" values instead of "const" or "input" because they depend on information that a script can obtain only *after* it starts running on a dataset. Likewise, various built-in function parameters require values with the "simple" qualifier or a weaker one.

The following script uses request.security() with a `calc_bars_count` argument to retrieve a limited history of daily close values. It determines the number of historical days in the request based on the "simple string" value of syminfo.type. For cryptocurrency symbols, the call requests 14 days of historical data. For other symbols, it requests 10 days of data. The script compiles successfully because the `reqDays` variable holds the type "simple int", which matches the expected type for the `calc_bars_count` parameter:

//@version=6 indicator("'simple' values demo") //@variable The number of historical days in the data request. This variable's type is "simple int", // because the strongest qualified type in the calculation is "simple string". reqDays = syminfo.type == "crypto" ? 14 : 10 //@variable The `close` value from the "1D" timeframe. // This call works because `calc_bars_count` expects a "simple int" argument. requestedClose = request.security(syminfo.tickerid, "1D", close, calc_bars_count = reqDays) plot(requestedClose)

Programmers can explicitly define variables and parameters that require "simple" values, or values with a weaker qualifier, by prefixing their declaration with the simple keyword, followed by a type keyword. Variables declared with this keyword can hold runtime-calculated values that do not change across bars. These variables cannot accept values qualified as "series", even if those values remain consistent on every bar.

The script below attempts to assign the result of a math.random() call to a `rand` variable declared with the simple keyword, causing a compilation error. The math.random() function returns a *different value* on each call, meaning its return type is "series float". However, the simple keyword forces the `rand` variable to require a "simple float", "input float", or "const float" value:

//@version=6 indicator("Cannot assign a 'series' value demo") // This declaration causes an error. `math.random()` returns a "series float" value, but the `rand` variable // requires a "float" value with the

"simple" qualifier or a weaker one. simple float rand = math.random() plot(rand)

> NoteUsing the simple keyword is optional in most cases. However, the keyword is required to define exported *library functions* that accept only arguments with “simple” or weaker qualifiers and return “simple” results. See the Libraries page to learn more.

series

Values qualified as “series” provide the most flexibility in a script’s calculations. These values are available at runtime, and they are the **only** values that can *change* from bar to bar.

All variables and function parameters that accept a “series” value also allow values with any other qualifier, because “series” is the *highest* qualifier in the qualifier hierarchy.

All built-in variables that store bar information – such as open, high, low, close, volume, time, bar_index, and barstate.isconfirmed – always hold “series” values. The same applies to variables that store data from internal calculations that update from bar to bar, such as ta.vwap and ta.pvi.

If an expression’s result *can vary* on any execution, it automatically inherits the “series” qualifier. Similarly, even if an expression returns an unchanging result on every bar, that result is still qualified as “series” if the calculation depends on at least one “series” value.

> NoteSpecial types and user-defined types automatically inherit the “series” qualifier, meaning any calculation involving these types returns “series” results. Scripts **cannot** create instances of these types with weaker qualifiers such as “simple” or “const”. See the reference types section for more information.

The following script calculates highest and lowest values from a `sourceInput` series and a “const float” value over `lengthInput` bars. The `highest` and `lowest` variables automatically inherit the “series” qualifier because the ta.highest() and ta.lowest() functions always return “series” results. These functions never return a value with a weaker qualifier, even if they calculate on a constant, because their `source` parameter is of the type “series float”:

//@version=6 indicator("'series' values demo", overlay = true) //@variable The source series to process in the `ta.highest()` call. // This variable's type is "series float", because `input.source()` returns "series" values. sourceInput = input.source(close, "Source") //@variable The number of bars to analyze. This variable's type is "input int". lengthInput = input.int(20, "Length") //@variable The highest `sourceInput` value over `lengthInput` bars. This variable's type is "series float". highest = ta.highest(source = sourceInput, length = lengthInput) //@variable The result of calculating the lowest value from a constant. This variable's type is also "series float". // The `source` parameter's type is "series float", so the function returns a "series float" value, regardless // of the argument's type qualifier. lowest = ta.lowest(source = 100.0, length = lengthInput) plot(highest, "Highest", color.green) plot(lowest, "Lowest", color.red)

Programmers can use the series keyword to explicitly define variables and parameters that accept “series” values. A script cannot use a variable declared using this keyword in any part of the code that requires “simple” or weaker qualifiers, even if the variable’s assigned value never changes.

For example, the script below declares a `lengthInput` variable with the series keyword. Then, it attempts to use the variable as the `length` argument of a ta.ema() function call, causing a compilation error. Although the variable's value comes from an integer input, the series keyword causes its type to become "series int" instead of "input int". This type is not compatible with the ta.ema() function's `length` parameter, because the strongest qualified type that the parameter accepts is "simple int":

//@version=6 indicator("`series` keyword demo", overlay = true) //@variable Holds a value intended for use as the `length` argument in `ta.ema()`. // Although the variable's value is from an input, its type is "series int" because the declaration uses the // `series` keyword. series int lengthInput = input.int(20, "Length") // This function call causes an error. The `length` parameter requires a "simple int", "input int", or "const int" // argument; it cannot accept a "series int" value. ema = ta.ema(close, length = lengthInput) plot(ema)

Types

Types define the *categories* of values in a script and determine the kinds of functions and operations with which those values are compatible. Each type represents a different kind of data. The primary types available in Pine Script consist of the following:

- Fundamental types: int, float, bool, color, and string
- Enum types (enums)
- Special types: plot, hline, line, linefill, box, polyline, label, table, chart.point, footprint, volume_row, array, matrix, and map
- User-defined types (UDTs)
- void

Fundamental types and enum types are also known as value types. Variables of these types directly hold values. Additionally, value types can inherit any type qualifier, depending on their use in the code. By contrast, special types and UDTs are reference types. Variables of these types do not store direct values; they hold *references* (sometimes referred to as *IDs*) that provide access to data stored *elsewhere* in memory. Instances of these types always inherit the "series" qualifier, regardless of how the script uses them.

> NotePine Script also includes a set of *unique value types*. These types are compatible only with specific built-in parameters and operators. For example, plot.style_line and other `plot.style_*` constants are of the *"plot_style"* type. A value of this type is required only by the `style` parameter of the plot() function; other built-ins *cannot* use it. The only other way scripts can use `plot.style_*` constants is by assigning their values to separate variables or comparing them with the == or != operators. See the "Constants" section of the Reference Manual to learn about other unique types and their uses.

Programmers can explicitly define the type of a variable, function parameter, or field by prefixing its declaration with a *type keyword* (e.g., int) or a *type identifier* (e.g., `array<int>`). Specifying types in code is usually optional, because the compiler can automatically determine type information in most cases. However, type specification is *required* when:

- Declaring variables, user-defined function parameters, or UDT fields with initial `na` values.
- Defining the parameters of exported library functions, or declaring exported constants.
- Using qualifier keywords in a variable or parameter declaration.

- Declaring the first parameter of a user-defined method.

> TipEven when specifying a type in the code is not mandatory, doing so helps to promote code readability. Additionally, using type keywords helps the Pine Editor provide relevant code suggestions.

The example below calculates a moving average and detects when the close series crosses over the value. The script uses values of different fundamental types in its calculations. It includes the int, float, bool, color, and string keywords in its variable declarations to specify which type each variable accepts:

//@version=6 indicator("Type keywords demo", overlay = true) // The `string`, `int`, `float`, `bool` and `color` keywords are *optional* in the following variable declarations: string MA_TITLE = "MA" int lengthInput = input.int(100, "Length", minval = 2) float ma = ta.sma(close, lengthInput) bool crossUp = ta.crossover(close, ma) color maColor = close > ma ? color.lime : color.fuchsia // Specifying a type is required in this declaration, because the variable's initial value is `na`. // The `float` keyword tells the compiler that the variable accepts "float" values. var float crossValue = na // Update the `crossValue` variable based on the `crossUp` condition. if crossUp crossValue := close plot(ma, MA_TITLE, maColor) plot(crossValue, "Cross value", style = plot.style_circles) plotchar(crossUp, "Cross Up", "▲", location.belowbar, size = size.small)

Note that:

- The first five variables in this script *do not* require type keywords in their declarations, but including them helps promote readability. However, the `crossValue` variable *does* require a specified type in its declaration because its initial value is na.

> TipTo confirm a variable's type, hover over its identifier in the Pine Editor. The editor displays a pop-up window containing the name of the variable's type and additional information about the variable.

The sections below explain the different types available in Pine Script and how they work.

Value types

The types covered in the following sections are *value types*. These types directly represent values, such as numbers, logical conditions, colors, or text sequences. Value types are compatible with any type qualifier, depending on their use in the code. Additionally, value types, unlike reference types, are compatible with arithmetic and logical operators.

int

Values of the "int" type represent *integers*: whole numbers *without* fractional parts.

Literal integers in a script are sequences of decimal digits without a decimal point (`.`). These literals can also include the *unary* + or - operators at the beginning of the sequence to specify their sign (positive or negative).

Below are a few examples of literal integers:

1 -1 750

Many built-in variables hold “int” values, including bar_index, time, timenow, dayofmonth, and strategy.wintrades.

float

Values of the “float” type represent *floating-point* numbers. In contrast to “int” values, “float” values represent the whole *and* fractional parts of a number.

Literal floating-point values in Pine have two different formats:

- A sequence of decimal digits that contains a decimal point (`.`) to separate the number’s whole and fractional parts. This format can include a unary + or - operator at the beginning to specify the number’s sign.
- A number, with an *optional* decimal point, followed by `e` or `E` and an additional *whole number*. The number before *and* after `e` or `E` can include the unary + or - operator. This format represents a floating-point number in E notation. It translates to *“X multiplied by 10 raised to the power of Y”*, where “X” is the number before `e` or `E`, and “Y” is the number that follows. This format provides a compact way to represent very large or very small values.

Below are a few examples of floating-point literals:

3.14159 // Rounded value of Pi (π) -3.0 6.02e23 // 6.02 * 10^23 (a very large number) 1.6e-19 // 1.6 * 10^-19 (a very small number)

The internal precision of “float” values in Pine Script is 1e-16. Floating-point values in Pine cannot precisely represent numbers with more than 16 fractional digits. However, note that comparison operators automatically round “float” operands to *nine* fractional digits.

Many built-in variables store “float” values, including close, hlcc4, volume, ta.vwap, and strategy.position_size.

> NotePine Script automatically converts “int” values to the “float” type if a script passes those values to variables or function parameters that require “float” values. Likewise, Pine converts “int” values to “float” in arithmetic or comparison operations that include a “float” operand. See the Type casting section to learn more.

bool

Values of the “bool” type represent the Boolean truth values of conditions (*true* or *false*). Scripts use these values in conditional structures and expressions to trigger specific calculations in the code. All *comparison* and *logical* operators return “bool” values.

There are only two possible “bool” literals in Pine Script:

true // true value false // false value

In contrast to most other types, values of the “bool” type are *never* na. Any expression or structure with the “bool” return type returns `false` instead of na if data is *not available*.

For example, if a script uses the history-referencing operator to retrieve the value of a “bool” variable from a previous bar that does not exist, that operation returns `false`. Likewise, an if statement with a return expression of the “bool” type returns `false` if none of its *local blocks* activate. By contrast, expressions and structures with other return types, excluding void, return na if there is no available data.

All built-in variables that represent conditions store “bool” values, including barstate.isfirst, chart.is_heikinashi, session.ismarket, and timeframe.isdaily.

> NoteIn contrast to some other languages, Pine Script does *not* automatically convert other types to the “bool” type in logical expressions. Scripts can explicitly convert “int” or “float” values to the “bool” type by using the bool() function. See the Type casting section to learn more about type conversions.

color

Values of the “color” type represent *RGB colors*, which scripts use to define the colors of chart visuals. Color literals in Pine have the format `#RRGGBB` or `#RRGGBBAA`, where:

- Each symbol after the number sign (`#`) represents a hexadecimal digit, which is a *numeral* from `0` to `9` or a *letter* from `A` (for 10) to `F` (for 15). Each set of *two digits* represents one of the color’s *component values*, ranging from 0 (`00`) to 255 (`FF`).
- The `RR`, `GG`, and `BB` parts represent the color’s *red*, *green*, and *blue* components, respectively. The last pair of digits, `AA`, is optional; it specifies the color’s opacity (*alpha*). If the pair is `00`, the color is *transparent*. If `FF` or not specified, the color is *fully opaque*.
- All letters in the literal value can be uppercase or lowercase.

Below are several examples of literal “color” values:

#000000 // Black #FF0000 // Red #00FF00 // Green #0000FF // Blue #FFFFFF // White #808080 // A shade of gray #3ff7a0 // A custom green-cyan color #FF000080 // 50% transparent red #FF0000ff // Equivalent to #FF0000; fully opaque red #FF000000 // Completely transparent (invisible) red

Pine Script also includes several built-in color constants, such as color.green, color.orange, color.red, and color.blue. Note that color.blue is the default color for plots, and it is the default value for several *color properties* of drawing types.

The `color` namespace contains functions for retrieving color components, modifying colors, and creating new colors. For instance, scripts can use color.new() to define a copy of a built-in color with different transparency, or use color.rgb() to create a new color with specific red, green, blue, and transparency components.

Note that the `red`, `green`, and `blue` parameters of the color.rgb() function expect a number from *0 to 255*, where 0 means no intensity and 255 means maximum intensity. The `transp` parameter of color.rgb() and color.new() expects a value from *0 to 100*, where 0 means fully opaque and 100 means completely transparent. Both functions automatically *clamp* arguments to these ranges, and they round the specified values to *whole numbers*.

The example below creates a new “color” value with color.rgb(), modifies the color’s transparency based on the current day of the week with color.new(), and then displays the resulting color in the chart’s background:

//@version=6 indicator("`color.*()` functions demo") //@variable A color with custom red, green, and blue components. The variable's type is "const color". color BASE_COLOR = color.rgb(0, 99, 165) //@variable A calculated transparency value based on the current day of the week. This variable's type is "series int". int transparency = 50 + int(40 * dayofweek / 7) //@variable A modified copy of `BASE_COLOR` with dynamic transparency. // This variable's type is "series color", because its calculation depends on a "series int" value. color modifiedColor = color.new(BASE_COLOR, transparency) // Color the background using the `modifiedColor` value. bgcolor(modifiedColor)

Note that:

- The value stored by `BASE_COLOR` is of the type “const color” because it depends on only “const” values. However, the modified color returned by color.new() is of the type “series color”, because the dayofweek variable used in the calculation has the “series” qualifier.

To learn more about working with colors in Pine, see the Colors page.

string

Values of the “string” type contain sequences of encoded characters representing text, including letters, digits, symbols, spaces, or other Unicode characters. Scripts use strings in many ways, such as to define titles, express symbols and timeframes, create alerts and debug messages, and display text on the chart.

Literal strings in Pine Script are sequences of characters enclosed by two ASCII quotation marks (`"`) or apostrophes (`'`). For example:

"This is a literal string enclosed in quotation marks." 'This is a literal string enclosed in apostrophes.'

Quotation marks and apostrophes are functionally similar when used as the enclosing delimiters of literal strings. A string enclosed in quotation marks can contain any number of apostrophes. Likewise, a string enclosed in apostrophes can contain any number of quotation marks. For example:

"It's an example" 'The "Star" indicator'

A literal string can prefix some characters with the backslash character (`\`) to *change* their meaning. For example, applying a backslash to a quotation mark or apostrophe adds that character directly into a literal string’s sequence instead of treating the character as the *end* of the string:

'It\'s an example' "The \"Star\" indicator"

Applying a backslash to the `n` or `t` characters in a literal string creates *escape sequences* for multiline text or indentation respectively, which scripts can render using `plot*()` functions, Pine Logs, or some drawing types. For example, this string represents multiline text with a single word per line:

"This\nString\nContains\nOne\nWord\nPer\nLine"

Scripts can use two operators, + and +=, to *concatenate* (combine) two separate strings. These operators create a new string containing the first operand’s character sequence followed by the second operand’s sequence. For example:

"This creates a " + "concatenated string."

The `str` namespace contains several built-in functions that perform string-based calculations or create new strings. For example, the script below calls str.format() on each bar to create a *formatted string* containing representations of "float" price values, and it displays the result as multiline text in a label positioned at the bar's high value:

//@version=6 indicator("Formatted string demo", overlay = true) //@variable A "series string" value containing representations of the bar's OHLC prices. string ohlcString = str.format("Open: {0}\nHigh: {1}\nLow: {2}\nClose: {3}", open, high, low, close) // Draw a label to display the `ohlcString` value as multiline text at the bar's `high` value. label.new(bar_index, high, ohlcString, textcolor = color.white)

Several built-in variables that contain symbol and timeframe information store "string" values, e.g., syminfo.tickerid, syminfo.currency, and timeframe.period.

For detailed information about Pine strings and the built-in `str.*()` functions, refer to the Strings page. To learn more about displaying text from strings, see the Text and shapes and Debugging pages.

Enum types

The enum keyword enables the creation of an *enum*, otherwise known as an *enumeration*, *enumerated type*, or *enum type*. An enum is a unique type that contains distinct *named fields*. These fields represent the *members* (i.e., possible values) of the enum type. Programmers can use enums to maintain strict control over the values accepted by variables, parameters, conditional expressions, collections, and the fields of UDT objects. Additionally, scripts can use the input.enum() function to create enum-based dropdown inputs in the "Settings/Inputs" tab.

The syntax to declare an enum is as follows:

```
[export ]enum <enumName>
    <field_1>[ = <title_1>]
    <field_2>[ = <title_2>]
    ...
    <field_N>[ = <title_N>]
```

Where:

- export is the optional keyword for exporting the enum from a library, enabling its use in other scripts. See the Enum types section of the Libraries page to learn more about exporting enums.
- `enumName` is the name of the enum type. Scripts can use the enum's name as the *type keyword* in variable declarations, parameter and field declarations, and the *type templates* of collections.
- `field_*` is the name of an enum field. The field represents a *named member* (value) of the `enumName` type. Each field must have a *unique* name that does not match the name or title of any other member in the enum. To retrieve an enum member, use *dot notation* syntax on the enum's name (e.g., `enumName.field_1`).
- `title_*` is a "const string" value representing the *title* of an enum member. If the enum declaration does not specify a member's title, its title is the "string" representation of its name. The input.enum() function displays enum member titles within a dropdown input in the "Settings/Inputs" tab. To retrieve the "string" title of an enum member, use the str.tostring() function on that member (e.g.,

`str.tostring(enumName.field_1)`). As with member names, each enum member's title must be *unique*; it cannot match the name or title of another member in the same enum.

The following code block declares an enum named `maChoice`. Each field within the declaration represents a unique, constant member of the `maChoice` enum type with a distinct title:

//@enum An enumeration of named values for moving average selection. //@field sma Specifies a Simple Moving Average. //@field ema Specifies an Exponential Moving Average. //@field wma Specifies a Weighted Moving Average. //@field hma Specifies a Hull Moving Average. enum maChoice sma = "Simple Moving Average" ema = "Exponential Moving Average" wma = "Weighted Moving Average" hma = "Hull Moving Average"

The following script uses the input.enum() function to create a dropdown input from our `maChoice` enum in the "Settings/Inputs" tab. The dropdown displays each field's *title* as a possible choice. The value of `maInput` is the `maChoice` *member* corresponding to the selected title. The script compares the `maChoice` value inside a switch structure to determine which `ta.*()` function it uses to calculate a moving average:

//@version=6 indicator("Enum types demo", overlay = true) //@enum An enumeration of named values for moving average selection. //@field sma Specifies a Simple Moving Average. //@field ema Specifies an Exponential Moving Average. //@field wma Specifies a Weighted Moving Average. //@field hma Specifies a Hull Moving Average. enum maChoice sma = "Simple Moving Average" ema = "Exponential Moving Average" wma = "Weighted Moving Average" hma = "Hull Moving Average" //@variable The `maChoice` member representing a selected moving average name. // This variable's type is "input maChoice". maChoice maInput = input.enum(maChoice.sma, "Moving average type") //@variable The length of the moving average. int lengthInput = input.int(20, "Length", 1, 4999) //@variable The moving average corresponding to the selected enum member. float selectedMA = switch maInput maChoice.sma => ta.sma(close, lengthInput) maChoice.ema => ta.ema(close, lengthInput) maChoice.wma => ta.wma(close, lengthInput) maChoice.hma => ta.hma(close, lengthInput) // Plot the `selectedMA` value. plot(selectedMA, "Selected moving average", color.teal, 3)

See the Enums page and the Enum input section of the Inputs page to learn more about using enums and enum inputs.

Reference types

All the types covered in the following sections are *reference types*. These types *do not* directly represent values. Instead, scripts use them to create *objects*: logical entities that store data in a distinct location. Variables of reference types hold *references*, also known as *IDs*, that identify objects in memory and enable access to their data.

In contrast to value types, which support *any* type qualifier, instances of a reference type automatically inherit the "series" qualifier, because each instance is *unique*. Additionally, because reference types do not represent values, they are *not* compatible with any arithmetic or logical operators.

For advanced information about how these types differ from value types, see the Value vs. reference types section at the bottom of the page.

plot and hline

Pine Script uses the “plot” and “hline” types to display plots and horizontal levels on the chart. The plot() and hline() functions create instances of these types. Each call to these functions returns a *reference (ID)* to a specific “plot” or “hline” instance. Scripts can assign the references returned by these functions to variables for use with the fill() function, which colors the space between two displayed plots or levels.

> NoteOnly the plot() and hline() functions return usable IDs. All other plot-related functions – including plotchar(), plotshape(), plotarrow(), plotbar(), plotcandle(), barcolor(), and bgcolor() – return void, because they produce *only* visual outputs. Scripts *cannot* use data from these functions in other parts of the code.

The following example calculates two EMAs, and then uses two plot() calls to display their values on the chart. It assigns the “plot” IDs returned by the function calls to variables, then uses those variables in a call to fill() to color the visual space between the displayed plots:

//@version=6 indicator("plot fill demo", overlay = true) //@variable A "series float" value representing a 10-bar EMA of `close`. float emaFast = ta.ema(close, 10) //@variable A "series float" value representing a 20-bar EMA of `close`. float emaSlow = ta.ema(close, 20) //@variable Holds the ID of the plot that displays the `emaFast` series. emaFastPlot = plot(emaFast, "Fast EMA", color.orange, 3) //@variable Holds the ID of the plot that displays the `emaSlow` series. emaSlowPlot = plot(emaSlow, "Slow EMA", color.gray, 3) // Color the space between the outputs from the "plot" objects referenced by `emaFastPlot` and `emaSlowPlot`. fill(emaFastPlot, emaSlowPlot, color.new(color.purple, 50), "EMA Fill")

Note that:

- Pine does *not* include type keywords for specifying variables of the “plot” or “hline” type. Variables of these types never hold na, so Pine can always determine their type information automatically.
- A single fill() function call cannot use both a “plot” and “hline” ID. The function requires two IDs of the *same type*.

In addition to displaying the complete history of “series” values on the chart, “plot” objects enable *indicator-on-indicator* functionality. Scripts can access values from *another* script’s plots for their calculations by using the input.source() function. See the Source input section of the Inputs page to learn more.

> NoteIn contrast to variables of all other reference types, variables of the “plot” or “hline” type cannot refer to different plots or levels across bars. All variables of these types must consistently hold the references returned by the *same* plot() or hline() calls on *every execution*. Additionally, the functions that create “plot” and “hline” objects work only in the *global scope*; scripts cannot use them in the *local scopes* of user-defined functions, conditional structures, or loops.

Drawing types

Pine’s drawing types serve as structures for creating *drawing objects*, which scripts use to display custom chart visuals. The available drawing types are line, linefill, box, polyline, label, and table.

Each drawing type has an associated *namespace* with the *same name*. This namespace contains all the available built-ins for creating and managing drawing objects. For example, the `label` namespace contains all the built-in functions and variables for creating and managing labels. To create new instances of any

drawing type, scripts can use the following `*.new()` functions from each type's namespace: line.new(), linefill.new(), box.new(), polyline.new(), label.new(), and table.new().

Each of these `*.new()` functions creates a new drawing object on every call, and it returns the *ID (reference)* of that specific object. The other functions in the type's namespace require this ID to access and delete, copy, or modify the drawing. For example, a script can use the ID returned by line.new() later to delete the underlying line object with line.delete(), copy the object with line.copy(), or update the drawing's color with line.set_color().

For detailed information about lines, boxes, and polylines, see the Lines and boxes page. To learn more about tables and labels, see the Tables page and the Labels section of the Text and shapes page.

Chart points

The chart.point type is a special type that scripts use to generate *chart points*. Chart points are objects that contain *chart coordinates*. Scripts use information from these objects to position lines, boxes, polylines, and labels on the chart.

Objects of the chart.point type contain three *fields*: `time`, `index`, and `price`. The `time` and `index` fields both represent horizontal locations (*x-coordinates*). The `price` field represents the vertical location (*y-coordinate*). Whether a drawing instance uses the `time` or `index` field from a chart point as an x-coordinate depends on the drawing's `xloc` property. By default, drawings use the `index` field from a chart point and *ignore* the `time` field.

Multiple functions in the `chart.point` *namespace* create chart points:

- The chart.point.new() function creates a new chart point containing specified `time`, `index`, and `price` values.
- The chart.point.now() function creates a chart point with a specified `price` value. The object's `time` and `index` field automatically contain the time and bar_index values from the bar on which the function call occurs.
- The chart.point.from_index() function creates a chart point with only specified `price` and `index` values. The `time` field of the created object is na. Therefore, all chart points from this function are intended for use with drawings whose `xloc` property is xloc.bar_index.
- The chart.point.from_time() function creates a chart point with only specified `price` and `time` values. The `index` field of the created object is na. Therefore, all chart points from this function are intended for use with drawings whose `xloc` property is xloc.bar_time.
- The chart.point.copy() function creates a new chart point with the *same* `time`, `index`, and `price` values as the one referenced by the specified `id` argument.

The following script draws a new line from the previous bar's high value to the current bar's low value on each execution. It also displays labels at both points of the line. The script sets the coordinates of the line and label drawings using data from chart points created by the chart.point.from_index() and chart.point.now() functions:

//@version=6 indicator("Chart points demo", overlay = true) //@variable References a chart point containing the previous bar's `bar_index` and `high` values. firstPoint = chart.point.from_index(bar_index - 1, high[1]) //@variable References a chart point containing the current bar's `bar_index`, `time`, and `low` values. chart.point secondPoint = chart.point.now(low) //@variable References a line connecting the coordinates from the objects referenced by `firstPoint` and `secondPoint`. line myLine = line.new(firstPoint, secondPoint,

color = color.purple, width = 3) // Draw a label at the `index` and `price` coordinates of the chart point referenced by `firstPoint`. // The label displays a string representing the first chart point's `price` value. label.new(firstPoint, str.tostring(firstPoint.price), color = color.green, style = label.style_label_down, textcolor = color.white) // Draw a label at the `index` and `price` coordinates of the chart point referenced by `secondPoint`. // The label displays a string representing the second chart point's `price` value. label.new(secondPoint, str.tostring(secondPoint.price), color = color.red, style = label.style_label_up, textcolor = color.white)

Refer to the Lines and boxes page for additional examples of using chart points.

footprint and volume_row

The footprint and volume_row types are special data types that scripts use when requesting volume footprint information with the request.footprint() function. An object of the footprint type stores the available volume footprint data for a specific bar. A volume_row object stores the data for an *individual row* within a bar's volume footprint.

The only way to create objects of the footprint type is by calling the request.footprint() function. A call to the function returns either the *reference (ID)* of a footprint object that contains the retrieved volume footprint data for the current bar, or na if no footprint data is available.

Scripts can use footprint IDs in calls to the functions from the `footprint` *namespace* to retrieve the calculated volume footprint data. Each function has an `id` parameter that requires a non-na ID of the footprint type.

Some of the available `footprint.*()` functions return values representing overall metrics from a specific bar's volume footprint:

- The footprint.buy_volume() function calculates the total "buy" volume for the volume footprint.
- The footprint.sell_volume() function calculates the total "sell" volume for the volume footprint.
- The footprint.total_volume() function calculates the sum of the footprint's total "buy" volume and total "sell" volume.
- The footprint.delta() function calculates the volume footprint's overall volume delta. The value represents the difference between the footprint's total "buy" volume and total "sell" volume. A positive value indicates that the total "buy" volume is greater than the total "sell" volume, and a negative value indicates the opposite.

The other `footprint.*()` functions retrieve the *IDs* of volume_row objects that contain data for individual rows in the volume footprint represented by a footprint object:

- The footprint.poc() function finds the *Point of Control (POC)* row of the volume footprint and returns the ID of a volume_row object containing data for that row. The POC is the footprint row that has the largest total volume.
- The footprint.vah() function finds the *Value Area High (VAH)* row of the volume footprint and returns a volume_row ID for that row. The VAH row is the highest one in the footprint's *Value Area*.
- The footprint.val() function finds the *Value Area Low (VAL)* row of the volume footprint and returns a volume_row ID for that row. The VAL row is the lowest one in the footprint's Value Area.
- The footprint.get_row_by_price() function searches the volume footprint to find the row whose price range includes a specified price level. If the price belongs to one of the footprint's rows, the function

returns the ID of the volume_row object that contains the data for that row. If the price level does *not* belong to any row in the footprint, the function returns na.
- The footprint.rows() function creates an array that contains the volume_row IDs for *every row* within the volume footprint, sorted in *ascending order* by the rows' price levels. The first element refers to the volume_row object for the *lowest* row, and the last refers to the one for the *highest* row. The array's *type identifier* is `array<volume_row>`. See the Collections section below to learn more about collection type identifiers.

> NoteThe `va_percent` argument of a request.footprint() call specifies the percentage of the total volume that the resulting footprint object uses for the volume footprint's Value Area. Therefore, changes to the argument directly affect the results of footprint.vah() and footprint.val() calls that use the returned ID.

The only way to access objects of the volume_row type is by calling any of the above functions using a valid footprint ID. Scripts can retrieve data from objects of this type for detailed footprint analysis by using their IDs in calls to the functions in the `volume_row` *namespace*. Each function has an `id` parameter that requires a non-na ID of the volume_row type:

- The volume_row.up_price() function returns the upper price level of the footprint row.
- The volume_row.down_price() function returns the lower price level of the footprint row.
- The volume_row.buy_volume() function calculates the total "buy" volume for the footprint row.
- The volume_row.sell_volume() function calculates the total "sell" volume for the footprint row.
- The volume_row.total_volume() function calculates the sum of the footprint row's total "buy" volume and total "sell" volume.
- The volume_row.delta() function calculates the volume delta for the footprint row. The value represents the difference between the row's "buy" volume and "sell" volume. A positive value indicates that the row's "buy" volume exceeds its "sell" volume, and a negative value indicates the opposite.
- The volume_row.has_buy_imbalance() function checks whether the footprint row has a *buy imbalance*, based on the `imbalance_percent` argument of the original request.footprint() call. It returns `true` if the row's "buy" volume exceeds the "sell" volume of the row *below* it by the specified percentage, and `false` otherwise.
- The volume_row.has_sell_imbalance() function checks whether the footprint row has a *sell imbalance*, based on the `imbalance_percent` argument of the original request.footprint() call. It returns `true` if the row's "sell" volume exceeds the "buy" volume of the row *above* it by the specified percentage, and `false` otherwise.

> NoteThe `imbalance_percent` argument of a request.footprint() call determines the percentage difference that the resulting footprint uses for detecting volume imbalances. Changing the argument directly affects the results of the volume_row.has_buy_imbalance() and volume_row.has_sell_imbalance() calls that use volume_row IDs from the footprint object created by the request.

See the `request.footprint()` section of the Other timeframes and data page for more information about footprint requests, and for examples that demonstrate how to use the `footprint.*()` and `volume_row.*()` functions to retrieve footprint data.

To learn more about volume footprints and how they work, refer to the Volume footprint charts article in our Help Center.

Collections

Pine Script *collections* (arrays, matrices, and maps) are objects that store values or the *IDs (references)* of other objects as *elements*. Collection types enable scripts to group multiple values or IDs in a single location and perform advanced calculations. Arrays and matrices contain elements of *one* specific type. Maps can contain data of *two* types: one type for the *keys*, and another for the corresponding *value elements*. The `array`, `matrix`, and `map` *namespaces* include all the built-in functions for creating and managing collections.

A collection's *type identifier* consists of two parts: a *keyword* defining the collection's *category* (array, matrix, or map), and a *type template* specifying the *types of elements* that the collection stores. The type template for array or matrix types consists of a single type keyword enclosed in angle brackets (e.g., `<int>` for a collection of "int" values). The type template for a map type consists of *two* comma-separated keywords surrounded by angle brackets (e.g., `<string, int>` for a map of "string" keys and "int" values).

Below, we list some examples of collection type identifiers and the types that they represent:

- `array<int>` – an array type for storing "int" values.
- `array<label>` – an array type for storing label IDs.
- `array<myUDT>` – an array type for storing references to objects of a `myUDT` user-defined type.
- `matrix<float>` – a matrix type for storing "float" values.
- `matrix<line>` – a matrix type for storing line IDs.
- `map<string, float>` – a map type for storing key-value pairs with "string" keys and "float" value elements.
- `map<int, myUDT>` – a map type for storing "int" values as keys, and references to `myUDT` objects as value elements.

Scripts also use type templates in the `*.new*()` functions that create new collections. For example, a call to `array.new<int>()` creates an array that stores "int" values, and a call to `map.new<int, color>()` creates a map that stores "int" keys and corresponding "color" values.

Programmers can explicitly define variables, parameters, and fields that accept references to objects of specific collection types by using the type identifier as the *type keyword* in the declaration. The following code snippet declares variables that hold references to collections of the type `array<int>`, `array<float>`, and `matrix<float>`:

```
//@variable References an "int" array with a single element. array<int> myIntArray = array.new<int>(1, 10)
//@variable Holds an initial reference of `na`. Can reference an array of "float" values. array<float>
myFloatArray = na //@variable References a "float" matrix with two rows and three columns. matrix<float>
myFloatMatrix = matrix.new<float>(2, 3, 0.0)
```

> Notice
>
> The `array` namespace also includes *legacy functions* for creating arrays of specific built-in types. For example, array.new_float() creates a "float" array, just like `array.new<float>()`. However, we recommend using the general-purpose array.new<type>() function, because it can create arrays of *any* supported type.

> An alternative way to specify an array variable's type is to prefix its declaration with the *element* type keyword, followed by empty *square brackets* (`[]`). For example, a variable whose declaration includes `int[]` as the type keyword accepts a reference to a collection of the type `array<int>`. However, this legacy format is *deprecated*; future versions of Pine Script might not support it. Therefore, we recommend using the `array<type>` format to define type identifiers for readability and consistency.
>
> Note that there are no alternative `*.new*()` functions or type declaration formats for matrices or maps.

Scripts can construct collections and type templates for most available types, including:

- All value types: int, float, bool, color, string, and enum types.
- The following *special types*: line, linefill, box, polyline, label, table, chart.point, footprint, and volume_row.
- User-defined types (UDTs).

Note that maps can use any of these types as value elements, but they can store only *value types* as *keys*. See the Maps page to learn more.

Collections *cannot* store elements of any of the following types:

- The *unique types* for specific built-ins, such as "plot_style", "plot_display", and "barmerge_gaps".
- The "plot" or "hline" type.
- Any collection type.

> TipAlthough collections cannot directly store the IDs of other collections, they *can* store references to *user-defined type* instances that contain collection IDs in their fields. See the next section to learn more about UDTs.

User-defined types

The type keyword enables the creation of *user-defined types (UDTs)*. UDTs are composite types; they can contain an arbitrary number of *fields* that can be of *any* supported type, including collection types and other user-defined types. Scripts use UDTs to create custom objects that can store multiple types of data in a single location.

The syntax to declare a user-defined type is as follows:

```
[export ]type <UDT_identifier>
    [varip ][field_type ]<field_name>[ = <value>]
    ...
```

Where:

- export is the optional keyword for exporting the UDT from a *library*, enabling its use in other scripts. See the User-defined types and objects section of the Libraries page to learn more.
- `UDT_identifier` is the *name* of the user-defined type.
- varip is an optional keyword that enables the field's assigned data to *persist* across all ticks within a bar, similar to a `varip` variable.
- `field_type` is a type keyword or identifier, which specifies the field's type.
- `field_name` is the name of the field.
- `value` is an optional *default value* for the field. Each time that the script creates a new instance of the UDT, it initializes the field with the specified value. If not specified, the field's default value is na, or `false` if the field's type is "bool". Note that the default value *cannot* be the result of a function call or any other expression; only a *literal value* or a compatible *built-in variable* is allowed.

The following example declares a UDT named `pivotPoint`. The type contains two fields for storing pivot data: `pivotTime` and `priceLevel`. The `pivotTime` field is of the type "int", and `priceLevel` is of the type "float":

//@type A custom type for creating objects that store pivot information. //@field pivotTime Stores the pivot's timestamp. //@field priceLevel Stores the pivot's price. type pivotPoint int pivotTime float priceLevel

User-defined types can contain fields for referencing other UDT objects. Additionally, UDTs support *type recursion*, meaning a UDT can include fields for referencing objects of the *same* UDT. Below, we added a `nextPivot` field to our `pivotPoint` type. Objects of this version of the UDT can store a *reference (ID)* to a separate object of the same `pivotPoint` type in this field:

//@type A custom type for creating objects that store pivot information. //@field pivotTime Stores the pivot's timestamp. //@field priceLevel Stores the pivot's price. //@field nextPivot Stores the reference to *another* instance of the `pivotPoint` type. type pivotPoint int pivotTime float priceLevel pivotPoint nextPivot

Every user-defined type includes built-in `*.new()` and `*.copy()` functions for creating objects or copying existing ones. Both functions construct a new object on every call and return that object's ID. For example, `pivotPoint.new()` creates a new instance of our `pivotPoint` type and returns its ID for use in other parts of the script.

To learn more about objects of UDTs and how to use them, see the Objects page.

void

Pine Script includes some built-in functions that produce *side effects* – such as creating triggers for alerts, generating chart visuals, or modifying collections – *without* returning any value or reference. The return type of these functions is **"void"**, which represents the *absence* of usable data. The "void" type applies to every function that performs actions without returning anything that the script can use elsewhere in the code.

For example, plotshape() performs an action (plots shapes on the chart), but it does *not* return a usable ID like the plot() function does. Therefore, its return type is "void". Another example is the alert() function. The function creates an alert trigger without returning any data that the script can use elsewhere, so it also has the "void" return type.

Because "void" represents the absence of usable data, scripts *cannot* call functions that return "void" in other calculations or assign their results to variables. Additionally, there is no available keyword to specify that an expression returns the "void" type.

`na` value

Pine Script includes a special value called na, which is an abbreviation for *"not available"*. Scripts use na to represent an *undefined* value or reference. It is similar to `null` in Java or `NONE` in Python.

Pine can automatically *cast* na values to almost any type. The type assigned to an na value depends on how the code uses it. However, in some cases, more than one type might be valid for a piece of code that includes na, and the compiler cannot determine which type to assign in those cases.

For example, this line of code declares a `myVar` variable with an initial value of na. This line causes a *compilation error*, because the type of data the variable might hold later is *uncertain*. It might store a "float" value for plotting, a "string" value for setting text in a label, or maybe even a *reference* to a drawing object:

// This declaration causes an error, because the type that `myVar` accepts is *uncertain*. myVar = na

To resolve this error, we must explicitly define the variable's type in the code. For instance, if the `myVar` variable will store "float" values, we can prefix the variable with the float keyword to specify its type as "float":

// It is clear to the compiler that this variable accepts "float" values, so this declaration does not cause an error. float myVar = na

Alternatively, we can use the float() function to explicitly cast the na value's type to "float", causing the variable to automatically inherit the "float" type:

// This declaration does not cause an error, because `na` is cast to "float", and `myVar` inherits the type. myVar = float(na)

Scripts can test whether the result from a variable or expression is na by using the na() function. The function returns `true` if the value or reference is *undefined*. Otherwise, it returns `false`. For example, the following ternary operation returns 0 if the value of `myVar` is na, or close if the value is defined:

//@variable Holds 0 if the the value of `myVar` is `na`; `close` otherwise. float myClose = na(myVar) ? 0 : close

It is crucial to note that scripts **cannot** directly *compare* values to na, because by definition, na values are undefined. The ==, != operators, and all other comparison operators always return `false` if at least one of the operands is a variable with an na value. Therefore, na comparisons can cause *unexpected results*. Additionally, if a script attempts to use na directly as an operand in any comparison operation, it causes a *compilation error*. For example:

// This line causes an error, because using `na` directly as an operand for the `==` operator is *not allowed*. float myClose = myVar == na ? 0 : close

Best practices often involve *replacing* occurrences of undefined values in the code to prevent them from propagating in a script's calculations. There are three ways to replace na values with defined values in a script's calculations, depending on the type:

- For the "int", "float", and "color" types, scripts can use the nz() function to replace na values with a type-specific default value (`0` for "int", `0.0` for "float", or `#00000000` for "color") or a specified replacement.
- Alternatively, scripts can use the fixnan() function to replace na values of the above types in a series with the latest non-na value from that series' history.

- For other types such as “string”, scripts can test for an undefined value using the na() function and replace it if the function returns `true`.

The following line of code uses the nz() function to replace the value of `close[1]` with the current bar’s open value if the expression’s result is na. This logic prevents the code from returning na on the first bar, where there is *no* previous close value for the [] operator to access:

//@variable Holds `true` if the current `close` value is above the previous `close` (or the current `open` if the previous `close` is `na`). bool risingClose = close > nz(close[1], open)

Replacing na values to avoid unintended results is especially helpful when a calculation involves data that can *persist* across bars.

For example, the script below declares a global `allTimeHigh` variable using the var keyword, meaning the variable is initialized only on the first bar and persists on all subsequent bars. On each bar, the script updates the variable with the result of a math.max() call that returns the maximum between the current `allTimeHigh` and high values, then plots the result.

This script plots na instead of the chart’s all-time high on every bar. The `allTimeHigh` variable has an initial value of na, and the math.max() function *cannot* compare na to the current value of high. Therefore, the function call consistently returns na:

//@version=6 indicator("Persistent `na` result demo", overlay = true) //@variable A variable intended to store the chart's all-time high as of the current bar, with an initial value of `na`. var float allTimeHigh = na // Compare the current `allTimeHigh` and `high` values, and update the `allTimeHigh` with the maximum. // This line does not assign the current all-time high to the variable; the value remains `na` on *every bar*. // The `math.max()` function cannot compare undefined values, so it returns `na` if at least one argument is `na`. allTimeHigh := math.max(allTimeHigh, high) plot(allTimeHigh)

To fix this script’s behavior and enable it to calculate the chart’s all-time high as intended, we must stop the script from passing an na value to the math.max() call. In the version below, we included the expression `nz(allTimeHigh, high)` as the first argument in the function call. Now, on any execution where the `allTimeHigh` value is na, the script replaces it with the value of high, preventing na values from persisting in the calculation:

//@version=6 indicator("Replaced `na` demo", overlay = true) //@variable Stores the chart's all-time high value as of the current bar. var float allTimeHigh = na // The `nz()` call in this line replaces `allTimeHigh` with `high` when the variable's value is `na`. Now, the // `math.max()` function never receives an `na` argument, so the `na` result no longer persists. allTimeHigh := math.max(nz(allTimeHigh, high), high) plot(allTimeHigh)

Note that:

- An alternative way to fix this script’s behavior is to initialize the `allTimeHigh` variable using the value of high. The fix works in this case because the script does not use na later in its calculations.

> NoteSome built-in functions automatically *ignore* na values in their calculations, preventing them from continuously returning na results in most cases. For example, ta.max() calculates the all-time high of a series *without* considering na values. Check the “Remarks” section of a function’s entry in the Reference Manual to confirm whether it ignores na in its calculations.

Type casting

Pine Script can convert (cast) values of one type to another type either by using specific functions, or automatically.

The *automatic* type-casting process can cast “int” values to the “float” type when necessary. All variables, function parameters, fields, and expressions that allow the “float” type can also accept “int” values, because any integer is equivalent to a floating-point number with its fractional part set to 0. If a script assigns an “int” value to a variable, function parameter, or field declared with the float keyword, the assigned value’s type automatically changes to “float”. Likewise, Pine converts “int” values to “float” in arithmetic or comparison operations that include a “float” operand.

For example, the following line of code uses the addition operator + with “int” and “float” operands. Pine automatically casts the “int” value to the “float” type before performing the addition operation, and thus the expression returns a “float” result:

// This variable holds a "float" value, because any arithmetic operation with "int" and "float" operands // returns a "float" result. myVar = bar_index + close

Sometimes, a script must cast data of one type to another. Scripts can cast na values, or numeric values, to specific types by using the following *type-casting functions*: int(), float(), bool(), color(), string(), line(), linefill(), label(), box(), and table().

For example, the script below declares a `LENGTH` variable of the “const float” type, then attempts to use that variable as the `length` argument of a ta.sma() function call:

//@version=6 indicator("Invalid type demo", overlay = true) //@variable Holds a "const float" value intended for use in the `length` argument of `ta.sma()`. float LENGTH = 10.0 // This line causes an error, because the `length` parameter has the expected type "series int". float sma = ta.sma(close, length = LENGTH) plot(sma)

The above code causes the following compilation error:

```
Cannot call `ta.sma()` with the argument `length = LENGTH`. An argument of "con
```

This error tells us that the code uses a “float” value where an “int” value is required. There is no automatic rule to cast “float” to “int”, so we must resolve the error manually. In this version of the code, we used the int() function to cast the “float” value of the `LENGTH` variable to the “int” type in the ta.sma() call. Now, the script compiles successfully:

//@version=6 indicator("Explicit casting demo") //@variable Holds a "const float" value intended for use in the `length` argument of `ta.sma()`. float LENGTH = 10.0 // This line does not cause an error, because the `int()` function converts the `length` argument's type to "const int". float sma = ta.sma(close, length = int(LENGTH)) plot(sma)

Note that:

- The int() function removes all fractional information from a “float” value *without* rounding. For instance, a call such as `int(10.5)` returns a value of 10, not 11. To round a “float” value to the nearest whole number before converting it to “int”, use math.round().

For most available types, explicit type casting is useful when defining variables that have initial na values or references, as explained in the previous section, na value.

For example, a script can declare a variable that holds an na reference of the label type in either of the following, equivalent ways:

// Explicitly specify that the variable can reference `label` objects. label myLabel = na // Explicitly cast the `na` instance to the `label` type, causing `myLabel` to inherit the type. myLabel = label(na)

Tuples

A *tuple* is a *comma-separated list* of expressions or identifiers enclosed in square brackets (e.g., `[expr1, expr2, expr3]`). If a structure that creates a local scope, such as a function, method, conditional structure, or loop, returns more than one result, the code lists the expressions for all the results in the form of a tuple at the end of the structure's local block.

For example, the following user-defined function returns a tuple containing two values. The first item in the tuple is the sum of the function's `a` and `b` arguments, and the second is the product of those two values:

//@function Calculates the sum and product of two "float" values. calcSumAndProduct(float a, float b) => //@variable The sum of `a` and `b`. float sum = a + b //@variable The product of `a` and `b`. float product = a * b // Return a tuple containing the `sum` and `product` values. [sum, product]

When calling this function later in the code, the script must use a tuple declaration to declare one new variable for each value returned by the function to use its data. For example, the `hlSum` and `hlProduct` variables in the following tuple declaration hold the `sum` and `product` values returned by a `calcSumAndProduct()` call:

// Declare a tuple containing a variable for each value returned by the `calcSumAndProduct()` call. [hlSum, hlProduct] = calcSumAndProduct(high, low)

Note that:

- In contrast to individual variable declarations, tuple declarations *do not* support type keywords. The compiler automatically determines the type of each variable in a declared tuple.

If a script's calculations do not require *all* the values returned by a function or structure, programmers can use an underscore as the *identifier* for one or more returned items in the tuple declaration. If a variable's identifier is an underscore, that variable is not usable elsewhere in the code, such as in comparisons or arithmetic expressions.

For example, if we do not require the `product` value returned by our `calcSumAndProduct()` function, we can replace the `hlProduct` variable with _ in our declared tuple:

// Declare a tuple with `_` as the second identifier, signifying that the script does not use the second returned value. // The `_` identifier in this tuple is *not* usable elsewhere in the code. [hlSum, _] = calcSumAndProduct(high, low)

In the above examples, the resulting tuple contains two items of the same type ("float"). However, Pine does not restrict tuples to only one type; a single tuple can contain multiple items of *different types*. For example,

the custom `chartInfo()` function below returns a five-item tuple containing “int”, “float”, “bool”, “color”, and “string” values:

//@function Returns information about the current chart. chartInfo() => //@variable The first visible bar's UNIX time value. int firstVisibleTime = chart.left_visible_bar_time //@variable The `close` value at the `firstVisibleTime`. float firstVisibleClose = ta.valuewhen(ta.cross(time, firstVisibleTime), close, 0) //@variable Is `true` if the chart has a standard chart type; `false` otherwise. bool isStandard = chart.is_standard //@variable The foreground color of the chart. color fgColor = chart.fg_color //@variable The ticker ID of the current chart. string symbol = syminfo.tickerid // Return a tuple containing the values. [firstVisibleTime, firstVisibleClose, isStandard, fgColor, symbol]

Scripts can also pass tuples to the `expression` parameter of `request.*()` functions, enabling them to retrieve *multiple* series from a single function call. A single call to request.security() or another `request.*()` function that requests a tuple of data still counts as *one* data request, not multiple. See the Other timeframes and data page to learn more about `request.*()` functions and the data that they can retrieve.

The following code snippet defines a `roundedOHLC()` function that returns a tuple of OHLC prices rounded to the nearest values that are divisible by the symbol’s minimum tick size. We use this function as the `expression` argument in a request.security() call to retrieve a tuple containing the symbol’s rounded price values on the “1D” timeframe:

//@function Returns a tuple of OHLC values, rounded to the nearest tick. roundedOHLC() => [math.round_to_mintick(open), math.round_to_mintick(high), math.round_to_mintick(low), math.round_to_mintick(close)] [op, hi, lo, cl] = request.security(syminfo.tickerid, "1D", roundedOHLC())

An alternative way to perform the same request is to pass the tuple of rounded values *directly* to the `expression` parameter of the request.security() call. For example:

[op, hi, lo, cl] = request.security(syminfo.tickerid, "1D", [math.round_to_mintick(open), math.round_to_mintick(high), math.round_to_mintick(low), math.round_to_mintick(close)])

Note that:

- Only the `request.*()` functions that have an `expression` parameter and the `input.*()` functions that include an `options` parameter support this argument format. No other functions can use tuples as arguments.

Conditional structures and loops can use tuples as their return expressions, enabling them to return multiple values at once after the script exits their scopes. For example, the if statement below returns a two-item tuple from one of its local blocks:

[v1, v2] = if close > open [high, close] else [close, low]

The following switch statement is equivalent to the above if statement:

[v1, v2] = switch close > open => [high, close] => [close, low]

It’s crucial to emphasize that only the *local scopes* of functions, conditional structures, or loops can return tuples. In contrast to if and switch statements, ternary operations are **not** conditional structures; they are *expressions* that *do not* create local scopes. Therefore, they cannot return tuples.

For example, this line of code attempts to return a tuple from a ternary operation, causing a *compilation error*:

// Causes an error. Only local scopes can return tuples, and the `?:` operator does not create new scopes. [v1, v2] = close > open ? [high, close] : [close, low]

Although all items in a tuple do not have to be of the same *type*, it's important to note that every item inherits the **same** type qualifier. All items within a tuple *returned* by a local scope inherit either the "simple" or "series" qualifier, depending on the structure and the items' types. Therefore, because "series" is the stronger qualifier, all other items in the returned tuple automatically inherit the "series" qualifier if at least one item is qualified as "series".

For example, the script below defines a `getParameters()` function that returns a two-item tuple. The script attempts to use the values returned by the function as arguments in a ta.ema() function call, causing a compilation error. The ta.ema() function requires a `length` argument of the type "simple int", but the `len` variable's type is *"series int"*. The value assigned to `len` automatically inherits the "series" qualifier because the `source` argument of the `getParameters()` call is of the type "series float":

//@version=6 indicator("Qualified types in tuples demo") getParameters(float source, simple int length) => // Although the expected type of the `length` parameter is "simple int", the // `length` value in the returned tuple inherits the "series" qualifier if the // `source` value has that qualifier, because all items in a tuple inherit the *same* qualifier. [source, length] // Declare a tuple containing the values returned by a `getParameters()` call. // Both variables in this tuple have the "series" qualifier, because `close` is of the type "series float". [src, len] = getParameters(source = close, length = 20) // This line causes an error. `ta.ema()` expects a "simple int" `length` argument, but `len` has the type "series int". plot(ta.ema(source = src, length = len))

Value vs. reference types

> TipThis section contains advanced details about the differences between value and reference types. To make the most of this information, we recommend that newcomers to Pine Script start by reading about the available types, and then come back to this section to learn more about their differences.

Every type in Pine Script, excluding void, is either a value type or a reference type.

All fundamental types, enum types, and the *unique types* for specific function parameters are *value types*. These types directly *represent* values, which scripts can use in arithmetic, comparison, or logical operations. Variables of these types store values. Likewise, expressions that return these types return values. Values can become available at compile time, input time, or runtime. Therefore, they can inherit *any* type qualifier, depending on their use in the code.

By contrast, user-defined types (UDTs) and *special types* – including label, line, linefill, polyline, box, table, chart.point, and collection types – are *reference types*. These types serve as structures for creating *objects*. An object is **not** a value; it is a logical entity that stores data in a distinct memory location. Each separate object has a unique associated *reference*, similar to a pointer, which identifies the object in memory and enables the script to access its data. Variables of reference types hold these object references; they **do not** store objects directly.

> NoteFor simplicity and ease of discussion, we sometimes use the term *“ID”* as a substitute for *“object reference”*.

Scripts create objects exclusively at *runtime*, using the available constructor functions from the type’s namespace (e.g., label.new()). Every call to these functions creates a *new object* with a *unique reference*. Therefore, unlike value types, reference types automatically inherit the “series” qualifier; they never inherit *weaker* qualifiers such as “simple” or “const”.

For example, the following script declares a `myLabel` variable and assigns it the result of a label.new() function call with constant `x` and `y` arguments on the first bar. Although the script calls label.new() only *once*, with “const” arguments, the variable’s type is *“series label”*. The type is **not** “const label”, because every call to the function returns a new, unique label reference, which no other call can reproduce:

//@version=6 indicator("'series' reference demo") //@variable References a label created on the first bar using "const" arguments. // Although the script creates only one label, using constant values, this variable's type is "series label" // because the assigned `label` reference is *unique*. No additional function calls can create the same label // instance or return the same reference. var label myLabel = label.new(0, 0, "A new 'series' label")

Note that:

- The script creates a label only on the first bar because the variable that stores its reference is declared in the *global scope* using the var keyword. See the Declaration modes section of the Variable declarations page to learn more.

Modifying variables vs. objects

Each variable of a value type holds an independent value, and the only way to modify that variable’s data is by using the reassignment or compound assignment operators. Each use of these operators directly overwrites the stored value, thus removing it from the current execution.

Scripts can also modify variables of reference types with the := operator, but not a compound assignment operator such as +=, because object references are *not compatible* with arithmetic or logical operations. However, it’s crucial to note that reassigning a variable of a reference type *does not* directly affect any object; it only assigns *another reference* to that variable. The object referenced before the operation *can* remain in memory and affect the script’s results, depending on the type and the script’s logic.

To understand this distinction, consider the following script, which uses a variable to store label references on the last historical bar. First, the script initializes the `myLabel` variable with the result of one label.new() call. Then, it uses the := operator to assign the variable the result of a *second* label.new() call. Reassigning the `myLabel` variable only changes the variable’s stored reference; it *does not* overwrite the *first* label object. The first label *still exists* in memory. Consequently, this script displays *two* separate drawings:

//@version=6 indicator("Reassigning reference-type variables demo") if barstate.islastconfirmedhistory // Create a new `label` object and assign its reference to `myLabel`. label myLabel = label.new(bar_index, 0, "First label") // Create another `label` object and assign that object's reference to the variable. // This reassignment operation does not overwrite the first label. It only changes the variable's assigned // reference. The first object still exists and produces an output on the chart. myLabel := label.new(bar_index, 20, "Second label")

Note that:

- Objects remain in memory until a script no longer uses them. For drawing types, the runtime system automatically maintains a limited number of active objects. It begins deleting those objects, starting with the oldest ones, only if a script reaches its *drawing limit* (e.g., ~50 labels by default).
- A script can also explicitly delete objects of drawing types by using the built-in `*.delete()` functions, such as label.delete(). For example, if we add the call `label.delete(myLabel)` before the final line in the code above, the script removes the first label before assigning the second label's reference to the `myLabel` variable.

Because objects are not values, but entities that store data separately, scripts do not modify their data by reassigning the variables that reference them. To access or modify an object's data, programmers must do either of the following, depending on the type:

- Use the built-in *getter* and *setter* functions available for most special types. For example, label.get_x() retrieves the `x` value from a label object, and label.set_x() updates a label's `x` value.
- Use *dot notation* syntax on a variable of a UDT or the chart.point type to access the object's *field*. Then, to change the field's assigned data, use a reassignment or compound assignment operator after the syntax. For example, `myObj.price` retrieves the `price` field of the object referenced by the `myObj` variable, and `myObj.price := 10` sets that field's value to 10.

> NoteBecause reference types always inherit the "series" qualifier, all data retrieved from an object also inherits the qualifier. Values stored by an object never qualify as "simple", "input", or "const", even if the script constructs the object using values with those weaker qualifiers.

The example below creates a chart.point and a label instance on the first bar, and then modifies the two objects on every bar. With each execution, the script updates the `price` ("float") and `index` ("int") fields of the chart point, then uses its reference in a label.set_point() call to change the label's coordinates. Lastly, the script uses label.get_y() to get the label's `y` value ("float"), then uses a plot to display the value:

//@version=6 indicator("Modifying objects demo", overlay = true) //@variable Maintains a persistent reference to one `chart.point` object with an initial `price` field of `na`. var chart.point myPoint = chart.point.now(na) //@variable Maintains a persistent reference to one `label` object initialized using the `myPoint` chart point. var label myLabel = label.new(myPoint, "Persistent label") // Update the chart point referenced by `myPoint` on each bar by reassigning the object's *fields*. myPoint.index := bar_index myPoint.price := close // Update the label referenced by `myLabel` using a call to `label.set_point()`. The call uses the `index` field of // the chart point for the label's x-coordinate, and the `price` field for the y-coordinate. label.set_point(myLabel, myPoint) // Retrieve the y-coordinate from the `myLabel` label, confirming that both persistent objects were modified. plot(label.get_y(myLabel), "Label y-coordinate")

Note that:

- The label.set_point() call in this example uses the `index` field of the chart point to set the label's `x` value, and it uses the `price` field to set the `y` value. It does not use the `time` field from the chart point for the `x` value, because the default `xloc` property for labels is xloc.bar_index.

Modifying global data in local scopes

Every script has one *global* scope, and it includes zero or more *local* scopes from any conditional structures, loops, user-defined functions or methods, or other structures. Most structures that create local scopes can access and use any global variables declared above them in the source code, because a script's local scopes *embed* into the global scope.

Conditional structures and loops defined in the global scope, as well as the nested structures within them, can also contain reassignment or compound assignment operations that modify global variables. In other words, either of these structures can directly change the data associated with global variables of value types and reference types.

For example, the script below declares a persistent "int" variable named `counter` in the global scope. Then, it uses the += and := operators inside nested if statements to update the variable's assigned value based on cyclic occurrences of a pseudorandom condition:

//@version=6 indicator("Modifying global variables in conditional structures demo") //@variable The number of conditions that occur before the counter value resets. int cycleSizeInput = input.int(10, "Cycle size", 1) //@variable A persistent global variable for counting occurrences of a condition in cycles. var int counter = 0 // Logic to update `counter` based on a pseudorandom condition. if math.random() < 0.5 // Increase the `counter` value by one when the condition occurs. counter += 1 // Reset the `counter` value to 1 if it exceeds the value of `cycleSizeInput`. if counter > cycleSizeInput counter := 1 // Plot the `counter` value. plot(counter, "Counter value")

By contrast, user-defined functions and methods *cannot* use the reassignment or compound assignment operators on global variables, because variables declared outside a function scope cannot accept *different* values *or* references during the execution of a *function call*. Consequently, functions and methods *cannot modify* the data associated with global variables of value types.

> NoteThe same limitation applies to function *parameters*. Function definitions cannot contain reassignment or compound assignment operations that modify declared parameters, because the *arguments* for those parameters cannot change while a function call executes.

Below, we edited the previous script to demonstrate this limitation. The following script version defines an `updateCounter()` function that attempts to modify the global `counter` variable from inside its scope using the same += and := operations as the example above. However, because the variable exists *outside* the function's definition, the function *cannot* change its value. As such, a *compilation error* occurs:

//@version=6 indicator("Cannot modify global variables in functions demo") //@variable The number of conditions that occur before the counter value resets. int cycleSizeInput = input.int(10, "Cycle size", 1) //@variable A persistent global variable for counting occurrences of a condition in cycles. var int counter = 0 //@function Attempts to increment and cyclically reset the `counter` variable based on a pseudorandom condition. // This function *does not* compile, because modifying global variables in function scopes is *not allowed*. updateCounter() => if math.random() < 0.5 // Attempting to increment `counter` causes a compilation error. // The variable's value *cannot change* during the execution of an `updateCounter()` call. counter += 1 if counter > cycleSizeInput // Reassigning the `counter` variable causes the same error. counter := 1 updateCounter() // This call does not work. plot(counter, "Counter value")

To modify global data from within the scope of a function call, programmers can use global variables of reference types instead of value types in the function's definition. As explained in the previous section,

scripts do *not* modify objects of these types by reassigning the variables that reference them. Instead, they *reassign fields* or use *setter functions*, depending on the type, to update the data that an object stores *elsewhere* in memory. Therefore, because a variable's assigned reference *does not change* after a script modifies an object, functions *can* change the data associated with global variables of reference types, unlike those of value types.

For example, the script version below declares a user-defined type named `Counter` with an "int" field named `value`. Then, it creates a new object of that type with a call to `Counter.new()`, and assigns the returned reference to a persistent global variable named `myCounter`. The `updateCounter()` function in this script uses the `+=` and `:=` operators on the `value` *field* of the `Counter` object referenced by the `myCounter` variable rather than directly reassigning the variable. Although the `value` field's assigned value can change during the execution of an `updateCounter()` call, the global variable itself remains unchanged; it still holds the reference to the *same* `Counter` object while the call executes. As a result, the script compiles successfully:

//@version=6 indicator("Modifying globally referenced objects in functions demo") //@variable The number of conditions that occur before the counter value resets. int cycleSizeInput = input.int(10, "Cycle size", 1) //@type A custom type for creating objects that store counter data. //@field value The counter value, initialized to 0 by default. type Counter int value = 0 //@variable A persistent global variable that holds the reference of a `Counter` object. var Counter myCounter = Counter.new() //@function Increments and cyclically resets the `value` field of the object referenced by `myCounter` based on a // pseudorandom condition. // This function does *not* cause an error, because it does not modify the global variable. updateCounter() => if math.random() < 0.5 // Increase the `value` *field* of the `Counter` object referenced by `myCounter` when the condition occurs. myCounter.value += 1 // Reset the `value` field to 1 if it exceeds the value of `cycleSizeInput`. if myCounter.value > cycleSizeInput myCounter.value := 1 // Modify the object referenced by `myCounter`. This function call works without issue. updateCounter() // Plot the value of the object's `value` field, i.e., the condition counter. plot(myCounter.value, "Counter value")

Copies vs. shared references

Variables of value types hold values that act as *independent copies*, because the only way to modify their data is through reassignment. If a script directly assigns one variable's value to another variable, it can change either variable's data later without affecting the other variable's data in any way.

For example, the following script initializes a `myVar1` variable with a value of 10, and then initializes a `myVar2` variable using `myVar1`. Afterward, the script adds 10 to `myVar1` with the `+=` operator, and plots the values of both variables on the chart. The script plots two different values (20 and 10), because changes to the value of `myVar1` do not affect the data accessed by `myVar2`:

//@version=6 indicator("Value type independence demo") // Initialize the first variable with a value of 10. int myVar1 = 10 // Initialize the second variable using the first. This variable's value is now 10. int myVar2 = myVar1 // Increase the first variable's value by 10. Now, the value of `myVar1` is 20, but the value of `myVar2` is still 10. myVar1 += 10 // Plot both values for comparison. plot(myVar1, "First variable", color.blue, 3) plot(myVar2, "Second variable", color.purple, 3)

The same behavior does not apply to variables of reference types. Assigning the reference stored by one variable to another **does not** create a new *copy* of an object. Instead, both variables refer to the **same** object in memory. As a result, the script can access or change that object's data through *either* variable and produce the same results.

The following example demonstrates this behavior. On the last historical bar, the script creates a new label with label.new() and assigns the returned reference to the `myLabel1` variable. Then, it initializes the `myLabel2` variable using `myLabel1`. The script calls label.set_color() to modify the label referenced by `myLabel1`, and then calls label.set_style() and label.set_text() to modify the one referenced by `myLabel2`.

A newcomer to reference types might expect this script to display *two* separate labels, with different colors, orientation, and text. However, the script shows only **one** label on the chart, and that label includes the changes from all `label.set_*()` calls. Modifying the label referenced by `myLabel2` directly affects the one referenced by `myLabel1`, and vice versa, because both variables refer to the **same** label object:

//@version=6 indicator("Shared object references demo") if barstate.islastconfirmedhistory // Create a new label and assign its reference to a variable. label myLabel1 = label.new(bar_index, 0, "First label", color = color.green, size = size.large) // Initialize a second variable using the `myLabel1` variable. // This variable declaration *does not* copy the label referenced by `myLabel1`; it only copies that variable's // *reference* to the new variable. label myLabel2 = myLabel1 // Change the color of the label referenced by `myLabel1`. label.set_color(myLabel1, color.red) // Update the style and text of the label referenced by `myLabel2`. // Because both variables refer to the *same object*, all the label changes affect that one object, // regardless of which variable the script uses in the `label.set_*()` calls. label.set_style(myLabel2, label.style_label_up) label.set_text(myLabel2, "Second label")

Most reference types, including user-defined types, feature a built-in `*.copy()` function. This function creates a new, *independent* object that contains the same data as the original object, and that new object has a *unique reference*. The script can modify the copied object's data without directly affecting the original.

In the following example, we changed the previous script to initialize `myLabel2` using the expression `label.copy(myLabel1)`, which creates an independent copy of the label referenced by `myLabel1` and returns a new reference. Now, `myLabel1` and `myLabel2` refer to two *separate* labels, and changes to the label referenced by one of the variables do not affect the other:

//@version=6 indicator("Copied objects demo") if barstate.islastconfirmedhistory // Create a new label and assign its reference to a variable. label myLabel1 = label.new(bar_index, 0, "First label", color = color.green, size = size.large) // Initialize a second variable using `label.copy(myLabel1)`. This variable now references an independent copy // of the initial label instead of pointing to the same object as `myLabel1`, and the script now displays two labels // on the chart. label myLabel2 = label.copy(myLabel1) // Now that `myLabel2` refers to a different `label` object than `myLabel1`, this call does not affect that object. label.set_color(myLabel1, color.red) // Likewise, these two calls do not affect the label referenced by `myLabel1`. label.set_style(myLabel2, label.style_label_up) label.set_text(myLabel2, "Second label")

> NoteThe `*.copy()` function creates a *shallow copy* of an object, not a *deep copy*. If a script uses this function to copy a collection or UDT instance that stores *other object references*, the contents of the copied instance refer to the **same objects** as the original instance. See the Copying objects section of the Objects page for more information.

Using `const` with reference types

Scripts can use the const keyword when declaring variables of most reference types, except for plot, hline, and user-defined types. However, with reference types, the keyword behaves *differently* than it does with value types.

Recall that for a variable of a value type, the const keyword directly *restricts* the qualifier of that variable to "const", *and* it prevents the script from using the reassignment or compound assignment operators to modify that variable – even if the assigned value from those operations is otherwise a constant.

For variables of reference types, using the const keyword to declare them also prevents a script from reassigning those variables. However, in contrast to its behavior with value types, the keyword **does not** set the *qualifier* of a reference-type variable to "const". As explained in previous sections, reference types automatically inherit the "series" qualifier, because each call to a function that creates objects produces a *new* object with a *unique* reference – any call to the function in the code never returns the same object reference more than once.

For example, the script below creates an array of pseudorandom "float" values using array.from(), and then assigns the returned reference to a variable declared using the const keyword on each bar. During each execution, the array.from() call creates a *new array* and returns a unique "series" reference. However, this script does *not* cause an error, even though the variable's *qualifier* is "series", because the variable's assigned reference remains *consistent* for the rest of each execution:

//@version=6 indicator("Using `const` with reference types demo") //@variable Holds a reference to an array of three pseudorandom "float" values. // Although the variable is declared using `const`, the reference returned by `array.from()` has the "series" // qualifier, because each execution creates a new, unique array object. Additionally, all elements in the // array are of the type "series float". // This *does not* cause an error, because the script does not *reassign* the variable during any execution. const array<float> randArray = array.from(math.random(), math.random(), math.random()) // Plot the sum of the `randArray` elements. plot(randArray.sum())

However, if we use the := operator to reassign the `randArray` variable, a compilation error occurs, because the const keyword prevents the script from assigning *another* array reference to the variable during each execution. For example:

//@version=6 indicator("Invalid reassignment demo") //@variable Holds a reference to an array of three pseudorandom "float" values. // Although the variable is declared using `const`, the reference returned by `array.from()` has the "series" // qualifier, because each execution creates a new, unique array object. Additionally, all elements in the // array are of the type "series float". const array<float> randArray = array.from(math.random(), math.random(), math.random()) // This line causes an error, because the `const` keyword prevents reassignment operations on the `randArray` variable. randArray := array.new<float>(3, 0.0) // Plot the sum of the `randArray` elements. plot(randArray.sum())

It's important to note that the const keyword *does not* directly prevent a script from modifying a collection or drawing object referenced by a variable or function parameter. Scripts can still use the available *setter functions* to change that object's data, because those functions *do not* affect the identifier's associated reference.

Below, we edited our script by including a call to array.set(). The call sets the first element of the array referenced by `randArray` to 0. Although the contents of the array change *after* each `randArray` declaration, the variable's assigned reference does not, so no error occurs:

//@version=6 indicator("Valid modification demo") //@variable Holds a reference to an array of three pseudorandom "float" values. // Although the variable is declared using `const`, the reference returned by `array.from()` has the "series" // qualifier, because each execution creates a new, unique array object. Additionally, all elements in the // array are of the type "series float". const array<float> randArray =

```
array.from(math.random(), math.random(), math.random()) // This line *does not* cause an error, even though it changes the array's contents, because `randArray` still refers // to the *same* array instance for the rest of the execution. array.set(randArray, 0, 0.0) // Plot the sum of the `randArray` elements. plot(randArray.sum())
```

User-defined functions

Introduction

User-defined functions are functions written by programmers, as opposed to the built-in functions provided by Pine Script®. They help to encapsulate custom calculations that scripts perform conditionally or repeatedly, or to isolate logic in a single location for modularity and readability. Programmers often write functions to extend the capabilities of their scripts when no existing built-ins fit their needs.

A function definition consists of two main parts: a *header* and a *body*.

Header

The header declares the function's *signature*, i.e., its *name* and *parameters*. A script *calls* the function by creating an expression containing the function's name followed by parentheses (e.g., `f()`). If the function has declared parameters, calls to the function list *arguments* (values or references) for those parameters within the parentheses (e.g., `f(x = 1)`).

Body

The function's body is the code that follows the header. Each *call* to the function performs the tasks defined by the expressions and statements in the function's body.

Function definitions in Pine can use either of the following formats:

- Single-line format, where the function's header and body occupy only *one* line of code. This format is best suited for defining compact functions containing only minimal statements or expressions.
- Multiline format, where the *first* line defines the function's header, and all *indented* lines that follow define the function's body. This format is optimal for functions that require conditional structures, loops, or multiple other statements.

Programmers can define functions for use only in a specific script, or create *library functions* for use in other scripts. Refer to our Style guide for recommendations on where to include function definitions in a source code. To learn more about library functions and their unique requirements, see the Libraries page.

Regardless of format, location, or use, several common characteristics and limitations apply to every user-defined function, including the following:

- The function's definition must be in the global scope. Programmers cannot define functions inside the body of another function or the local blocks of any other structure.
- The function cannot modify its declared parameters or any global variables.
- The function can include calls to most other functions within its body, but it *cannot* include calls to *itself* or to functions that must be called from the *global scope*.
- Each written call to the function must have consistent parameter types, and therefore consistent *argument* types, across all executions.
- Each call returns the result of evaluating the *final* statement or separate expression defined in the function's body, and that result inherits the *strongest* type qualifier used in the call's calculations. As with parameter types, the call's returned types must be consistent across executions.
- Each written call to the function establishes a *new scope* from the function's definition. The parameters, variables, and expressions created in that scope are *unique* and have an *independent*

history; other calls to the function do not directly affect them.

Structure and syntax

A function definition can occupy a single line of code or multiple lines, depending on the expressions and statements that the function requires. The single-line and multiline formats are similar, with the key difference being the placement of the function's *body*.

Single-line functions define their header and body on the same line of code:

```
<functionHeader> => <functionBody>
```

In contrast, multiline functions define the body on separate lines of code following the header. The code block following the header line has an indentation of *four spaces* or a single tab:

```
<functionHeader> =>
    <functionBody>
```

Both formats use the following syntax for defining the function's *header*:

```
[export ]<functionName>([[[paramQualifier ]<paramType> ]<paramName>[ = defaultV
```

Where:

- All parts within square brackets (`[]`) represent *optional* syntax, and all parts within angle brackets (`<>`) represent *required* syntax.
- export is the optional keyword for exporting the function from a library, enabling its use in other scripts. See the Libraries page to learn more.
- `functionName` is the function's identifier (name). The script calls the function by referencing this identifier, followed by parentheses.
- `paramName` is the identifier for a declared *parameter*. The script can supply a specific *argument* (value or reference) to the parameter in each function call. A function header can contain zero or more parameter declarations.
- `defaultValue` is the parameter's default argument. If not specified, each call to the function requires an argument for the parameter. Otherwise, supplying an argument is optional.
- `paramQualifier` and `paramType` are qualifier and type *keywords*, which together specify the parameter's qualified type. Using these keywords is optional in most cases. If the declaration does not include them, the compiler determines the parameter's type information automatically. See the Declaring parameter types section to learn more.

> TipProgrammers can also place the method keyword immediately before a function's name to declare the function as a *method* for a *specific type*. All methods must include at least *one* parameter, and the first parameter *requires* a declared type. Refer to the Methods page for more information.

Below is an example of a simple function header. The header declares that the function's name is `myFunction`, and that the function has two parameters named `param1` and `param2`:

myFunction(param1, param2) =>

Note that:

- Neither parameter has a *default* argument. Therefore, the script must supply arguments to these parameters in *every* `myFunction()` call.
- Because the function's parameters do not include type keywords, they inherit the *same* qualified type as their argument in each separate call.

The following two sections explain the body structure of single-line and multiline functions.

Single-line functions

A single-line function's body begins and ends on the *same* line of code as the header. This format is convenient for defining compact functions that execute only simple statements and do not use conditional structures or loops. The syntax to define a single-line function is as follows:

```
<functionHeader> => {statement, }<returnExpression>
```

Where:

- `functionHeader` declares the function's *name* and *parameters*, as explained in the previous section.
- `statement`, in curly brackets, represents zero or more statements or expressions that the function evaluates *before* returning a result. The function must separate all individual statements in its body with *commas*.
- `returnExpression` is the *final* expression, variable, or tuple in the function's body. Each function call *returns* the result of evaluating this code.

The following example defines an `add()` function in single-line format. The function includes two parameters named `val1` and `val2`. The body of the function contains a single + operation that *adds* or concatenates the parameter values, depending on their types. Each call to the function returns the result of that operation:

add(val1, val2) => val1 + val2

A script that includes this function definition can call `add()` with different arguments for `val1` and `val2`. The type of value returned by each call depends on these arguments. For example, the script below executes a few calls to `add()`, then passes their results to the `series`, `title`, and `linewidth` parameters in a call to plot():

//@version=6 indicator("Simple single-line function demo") add(val1, val2) => val1 + val2 float a = add(open, close) // `open + close` ("series float") int b = add(bar_index, 1) // `bar_index + 1` ("series int") int c = add(2, 3) // 5 ("const int") string d = add(add("Test", " "), "plot") // `"Test plot"` ("const string") // Create a plot using the `a`, `b`, `c`, and `d` values. plot(a / b, title = d, linewidth = c)

Note that:

- The `val1` and `val2` parameters automatically *inherit* the qualified types of their arguments in each `add()` call, because the function's header does not declare their types using type and qualifier keywords.
- Although the parameters accept arguments of any type, except for void, an `add()` function call compiles successfully only if the arguments have types that are *compatible* with the + operator, such as "int", "float", or "string". As shown above, if the arguments are numbers ("int" or "float" values), the function performs *addition*. If they are strings, the function performs *concatenation*.
- The value returned by each `add()` call inherits the *strongest* type qualifier used in the calculation. The first two calls return "series" results because both use at least one "series" value. In contrast, the other calls return "const" results, because all values in their calculations are *constants*.

The body of a single-line function can contain a *comma-separated list* of statements and expressions. Each call to the function evaluates the list from left to right, treating each item as a separate line of code. The call returns the result of evaluating the *final* expression or statement in the list.

For example, the following script contains a `zScore()` function defined in single-line format. The function computes the z-score of a `source` series over `length` bars. Its body declares two variables, `mean` and `sd`, to hold the average and standard deviation of the series. The final expression in the body uses these variables to calculate the function's returned value. On each bar, the script calls the `zScore()` function using close as the `source` argument and 20 as the `length` argument, then plots the result:

//@version=6 indicator("Single-line function with more than one statement demo") //@function Calculates the z-score of a `source` series over `length` bars. zScore(float source, int length) => mean = ta.sma(source, length), sd = ta.stdev(source, length), (source - mean) / sd //@variable The 20-bar z-score of `close` values. float osc = zScore(close, 20) // Plot the `osc` series as color-coded columns. plot(osc, "Z-score", osc > 0 ? color.green : color.red, style = plot.style_columns)

Note that:

- The `source` parameter requires an "int" or "float" value because its declaration includes the float keyword. The `length` parameter requires an "int" value because its declaration uses the int keyword. See the Declaring parameter types section to learn more.
- The `//@variable` and `//@function` comments are annotations that document identifiers in the code. The `//@function` annotation provides documentation for the `zScore()` function. Users can hover over the function's name in the Pine Editor, or write a function call, to view the annotation's formatted text in a pop-up window. See the Documenting functions section for more information.

> TipAlthough it is possible to write single-line functions containing multiple statements, as shown above, using the multiline format is often preferred for readability. Programmers can list the statements on separate lines and add comments for each one.

Multiline functions

A multiline function defines its body using a *block* of code following the header line. The general syntax is as follows:

```
<functionHeader> =>
    [statements]
    …
    <returnExpression>
```

Where:

- `functionHeader` declares the function's *name* and *parameters*. See the Structure and syntax section above to learn the syntax for function headers.
- `statements` is an optional block of statements and expressions that the function evaluates *before* returning a result. Single lines in the body can also contain *multiple* statements separated by commas.
- `returnExpression` is the *final* statement, expression, variable, or tuple at the end of the body. Each call to the function returns the result of this code. If the end of the function's body is a comma-separated list of statements or expressions, `returnExpression` is the code at the end of that list. If the final statement is a conditional structure or loop, the function returns that structure's result.
- Each line of code following the function's header has an indentation of *four spaces* or a tab, signifying that it belongs to the function's scope. The function's body ends on the final indented line; all non-indented code following the definition belongs to the *global* scope.

The multiline format is well-suited for defining functions that perform multiple tasks. Programmers can organize all the function's statements across different lines and document them with comments for readability.

In the following example, we modified the second script from the Single-line functions section to define an equivalent `zScore()` function using the multiline format. The function's variable declarations and return expression are now on *separate* indented lines, and we've added comments inside the body to describe the function's calculations:

//@version=6 indicator("Multiline function demo") //@function Calculates the z-score of a `source` series over `length` bars. zScore(float source, int length) => // Calculate the mean and standard deviation of the series. float mean = ta.sma(source, length) float sd = ta.stdev(source, length) // Compute the z-score using the `mean` and `sd` values, and return the result. (source - mean) / sd //@variable The 20-bar z-score of `close` values. float osc = zScore(close, 20) // Plot the `osc` series as color-coded columns. plot(osc, "Z-score", osc > 0 ? color.green : color.red, style = plot.style_columns)

Note that:

- We added the float keyword to the `mean` and `sd` declarations to declare their types in the code. The keyword is *optional* in both variable declarations because the compiler can determine the correct types from the assigned values, but including it helps promote readability.

Programmers often define multiline functions to encapsulate complex or logical tasks involving loops or conditional structures. If the final part of a function's body contains one of these structures, a call to the function returns the result of evaluating that structure.

For example, the `smoothMedian()` function in the script below calculates the median of a `source` series over `length` bars, then smooths the result using a moving average specified by the `avgType` parameter. The function compares the `avgType` value in a switch statement to select the type of average that it returns. The

script calls the function to calculate the median of close values over 10 bars, smoothed by an EMA with the same length, and then plots the result on the chart:

//@version=6 indicator("Multiline function with conditional structure demo", overlay = true, behind_chart = false) //@function Calculates the median of `source` over `length` bars, then returns a moving average of the median. smoothMedian(float source, int length, string avgType = "ema") => //@variable The median of the `source` series. float median = ta.median(source, length) // Calculate the EMA, SMA, and WMA of `median`. float ema = ta.ema(median, length) float sma = ta.sma(median, length) float wma = ta.wma(median, length) // Return `ema`, `sma`, or `wma`, depending on the `avgType` value. switch avgType "ema" => ema "sma" => sma "wma" => wma //@variable The EMA of the 10-bar median of `close` values. float medSmooth = smoothMedian(close, 10) // Plot the `medSmooth` series as a color-coded line. plot(medSmooth, "Smoothed median", close > medSmooth ? color.teal : color.maroon, 3)

Note that:

- The `smoothMedian()` call in this example does not supply an argument to the `avgType` parameter. Specifying an argument is *optional*, because the parameter declaration includes a *default argument* (`"ema"`).
- We included empty lines between each section for readability. The lines that follow an empty line are considered part of the function as long as they are indented correctly.

Functions that return multiple results

Sometimes, a function must perform multiple calculations and return separate results for each one. Programmers can write functions that return *multiple* values or references by using a tuple as the *final statement* in the function's body. A tuple is a comma-separated list of expressions enclosed in square brackets (`[]`). Every call to a function with a tuple at the end of its body returns the result of evaluating each listed expression in order.

For example, the `sumDiff()` function below calculates both the sum and difference of the values passed to its `val1` and `val2` parameters. Its body consists of a two-item tuple containing the addition and subtraction operations:

//@function Returns a tuple containing the sum and difference between the `val1` and `val2` values, respectively. sumDiff(float val1, float val2) => [val1 + val2, val1 - val2]

Note that:

- The `val1` and `val2` parameters include the float keyword in their declarations. Therefore, both parameters are of the type "float", and they accept only "float" or "int" arguments. See the Declaring parameter types section for more information.

Because the end of the function's body is a two-item tuple, each call to the function always returns two separate values. To assign the function's results to variables, the script must use a tuple declaration containing one *new* variable for each returned item. It is *not* possible to reassign existing tuples or to use previously declared variables within them.

The following example calls `sumDiff()` to calculate the sum and difference between two pseudorandom values. The script uses a tuple declaration containing two variables, `sum` and `diff`, to store the values returned by the call. Then, it plots the values of those variables on the chart:

```
//@version=6 indicator("Functions that return multiple results demo") //@function Returns a tuple containing the sum and difference between the `val1` and `val2` values, respectively. sumDiff(float val1, float val2) => [val1 + val2, val1 - val2] // Call `sumDiff()` with two pseudorandom arguments, and use a tuple declaration with one variable for each separate result. [sum, diff] = sumDiff(math.random(), math.random()) // Plot the `sum` and `diff` series. plot(sum, "Sum", color.teal, 3) plot(diff, "Difference", color.maroon, 3)
```

In some cases, a script might not require *all* the results from the tuple returned by a function call. Instead of writing unique identifiers for every variable in the tuple declaration, programmers can use an underscore as the identifier for each variable that the script does not require. All variables with the name _ are *not usable* in the script's calculations.

For example, if we change `sum` to _ in the previous script's tuple declaration, the first value returned by the `sumDiff()` call is *inaccessible* to the script. Attempting to use the identifier elsewhere in the code causes a *compilation error*, as shown by the first plot() call in the script version below:

```
//@version=6 indicator("Using `_` in a tuple declaration demo") //@function Returns a tuple containing the sum and difference between the `val1` and `val2` values, respectively. sumDiff(float val1, float val2) => [val1 + val2, val1 - val2] // Declare a tuple with `_` as the first identifier, meaning the first item returned by `sumDiff()` (the sum) is // inaccessible to the script. [_, diff] = sumDiff(math.random(), math.random()) // The first `plot()` call now causes an error, because all `_` variables are *not usable* in a script's calculations. plot(_, "Sum", color.teal, 3) plot(diff, "Difference", color.maroon, 3)
```

> NoteThe items in a function's returned tuple can have *different* types. However, functions *cannot* return multiple results with different type qualifiers. Therefore, *all* items in a returned tuple automatically inherit the *same qualifier*. If the tuple does not rely on a "series" value, all the items in the tuple typically inherit the *"simple"* qualifier. Otherwise, all items inherit the *"series"* qualifier. See the Tuples section of the Type system page for an example.

Declaring parameter types

User-defined function definitions can prefix each parameter declaration with type and qualifier *keywords*, enabling strict control over the qualified types that a script can pass to the parameter in any function call. If a declaration does not include these keywords, the compiler automatically determines the parameter's qualified type based on its arguments and the function's structure.

The sections below explain how type and qualifier keywords affect the behavior of function parameters. For detailed information about Pine's types and qualifiers, refer to the Type system page.

Type keywords

Parameter declarations prefixed by *type keywords* – such as int, float, string, or label – declare the types of data that the parameters represent in any function call. If a parameter declaration includes a type keyword, it accepts only arguments of that type, or arguments that Pine can automatically cast to that type.

If a function parameter does *not* have a type keyword in its declaration, its type is initially *undefined*. In each separate call to the function, the parameter automatically inherits the *same* type as its specified argument. In other words, the parameter can take on *any type*, except for void, depending on the function call.

The following example demonstrates this behavior. The user-defined `pass()` function in the script below returns the value of the `source` parameter without performing additional calculations. The parameter's declaration does not include a type keyword. The script executes five calls to the function with different argument types and then uses their results in code that accepts those types. This script compiles successfully, because each call's version of the `source` parameter inherits its argument's type:

//@version=6 indicator("Undefined parameter types demo") //@function Returns the value of the `source` argument without modification. // Each written call to the function can accept an argument of *any* type except for "void". pass(source) => source plotSeries = pass(bar_index) // "series int" lineWidth = pass(3) // "const int" plotTitle = pass("Test plot") // "const string" plotColor = pass(chart.fg_color) // "input color" plotDisplay = pass(display.all - display.status_line) // "const plot_display" // Create a plot using the values from the five `pass()` calls. plot(plotSeries, plotTitle, plotColor, lineWidth, display = plotDisplay)

We can restrict the `source` parameter's type, and thus the arguments it can accept, by including a type keyword in its declaration. For example, in the modified script below, we added the int keyword to the declaration to specify that the parameter's type is "int". With this change, the last three `pass()` calls now cause a compilation error, because the parameter no longer allows "string", "color", or "plot_display" arguments:

//@version=6 indicator("Declared parameter type demo") //@function Returns the value of the `source` argument. // The argument in each call must be an integer. No other type is allowed. pass(int source) => source // These two calls are valid, because their arguments are "int" values. plotSeries = pass(bar_index) lineWidth = pass(3) // These three calls cause an error, because the `pass()` function no longer allows non-integer arguments. plotTitle = pass("Test plot") plotColor = pass(chart.fg_color) plotDisplay = pass(display.all - display.status_line) // Create a plot using the values from the five `pass()` calls. plot(plotSeries, plotTitle, plotColor, lineWidth, display = plotDisplay)

Note that:

- If a parameter declaration includes a type keyword *without* a qualifier keyword, such as const or simple, the compiler automatically sets the qualifier to *"series"* or *"simple"*, depending on the function's calculations. In this example, the `source` parameter's type is "series int", because the function's logic does not require a "simple" or weaker qualifier. See the Qualifier keywords section to learn more.

In most cases, type keywords are *optional* in parameter declarations. However, specifying a parameter's type is *required* if:

- The parameter's default argument is an `na` value.
- The parameter declaration includes a *qualifier keyword*.
- The function definition includes the export keyword. Exported functions require declared types for *every* parameter. See the Libraries page to learn more.
- The function definition includes the method keyword, and the parameter is the *first* one listed in the header. See the User-defined methods section of the Methods page for more information.

> TipEven when not required, we recommend declaring parameter types where possible. Type keywords help promote readability, and they enable the Pine Editor to provide relevant code suggestions. Additionally, parameters with declared types help prevent *unintended* arguments in function calls.

Qualifier keywords

A *qualifier keyword* (const, simple, or series) preceding a type keyword in a parameter declaration specifies the parameter's type qualifier. The keyword also indicates when the *argument* for the parameter must be accessible, and whether that argument can *change* across bars:

`const`

The parameter has the "const" qualifier. Its argument must be available at *compile time*, and that argument cannot change during any execution. Only the parameters of *non-exported* functions can use the const keyword.

`simple`

The parameter has the "simple" qualifier. Its argument must be a "simple" value or a value with a *weaker* qualifier ("input" or "const"). The value cannot change after the *first bar*.

`series`

The parameter has the "series" qualifier. It can accept an argument with *any* type qualifier, because "series" is the *highest* one in Pine's qualifier hierarchy. The argument for the parameter in each function call *can change* on any execution.

> NoteQualifier keywords affect only the parameters that accept value types. They **do not** affect those that accept reference types. Instances of reference types *always* have the "series" qualifier, regardless of how the script uses them. Therefore, all parameters of these types automatically inherit the "series" qualifier. See the Type system page to learn more.

Qualifier keywords are always *optional* in parameter declarations. If a declaration does not include a qualifier keyword, the compiler uses the following logic to assign a qualifier to the parameter:

- If the declaration does *not* include a type keyword, the compiler assigns the parameter the *same* qualifier *and* type as its argument in each written function call. For example, if the script passes a "const int" argument in one function call, the parameter of that call inherits the type "const int". If it uses a "series float" value in another call to the function, the parameter of that call inherits the type "series float".
- If the declaration *does* include a type keyword, the compiler first tests whether the parameter can use the *"series"* qualifier. If the function contains a call to another function in its body, and that call *cannot* accept a "series" value, the compiler then checks whether the *"simple"* qualifier works. If neither the "series" nor "simple" qualifier is compatible with the function's calculations, a *compilation error* occurs.

To demonstrate these behaviors, let's revisit the `pass()` function from the Type keywords section. The function returns the value of its `source` parameter without additional calculations. If the parameter does not include *any* keywords in its declaration, its qualified type is that of its *argument* in each separate function call:

//@function Returns the value of the `source` argument without modification. // Each written call to the function can accept an argument of *any* type except for "void". pass(source) => source

The script below calls `pass()` using an “int” value with the “const” qualifier, then uses the returned value as the `length` argument in a call to ta.ema() and plots the result. This script compiles successfully because our `pass()` call returns the same qualified type as its argument (“const int”), and the `length` parameter of ta.ema() can accept a value of that type:

```
//@version=6 indicator("Qualifier inheritance demo") //@function Returns the value of the `source` argument without modification. // Each written call to the function can accept an argument of *any* type except for "void". pass(source) => source //@variable The EMA smoothing length. // This `pass()` call's `source` parameter automatically inherits the "const" qualifier from its argument. // Therefore, the returned type is "const int". int lengthVal = pass(14) //@variable The EMA of `close - open`. // This call works as expected, because the `length` parameter of `ta.ema()` can accept "int" values with // "simple" or weaker qualifiers. float emaDiff = ta.ema(close - open, length = lengthVal) // Plot the `emaDiff` series. plot(emaDiff, "Smoothed difference", color.purple, 3)
```

If we add int to the `source` declaration, the parameter then requires an “int” value, but it **does not** directly inherit the *same* type qualifier as its argument. Instead, the compiler first checks if it can assign *“series”* to the parameter, then tries using *“simple”* if “series” does not work.

Our `pass()` function does not use the `source` parameter in any local function calls that require a “simple int” value, so the compiler sets its qualifier to **“series”**. Consequently, the function’s returned type is always *“series int”*, even if the `source` argument is a “const” value. Adding this change to the previous script thus causes a *compilation error*, because the `length` parameter of ta.ema() cannot accept a “series” argument; only “simple” or weaker qualifiers are allowed:

```
//@version=6 indicator("Qualifier promotion demo") //@function Returns the value of the `source` parameter. Requires an integer. pass(int source) => source //@variable The EMA smoothing length. // This `pass()` call's `source` parameter **does not** inherit the "const" qualifier. // Its type is promoted to "series int". int lengthVal = pass(14) // This call causes a *compilation error*. The `length` parameter cannot accept a "series int" value. float emaDiff = ta.ema(close - open, length = lengthVal) // Plot the `emaDiff` series. plot(emaDiff, "Smoothed difference", color.purple, 3)
```

We can restrict the type qualifier of our function’s `source` parameter by adding a *qualifier keyword* to its declaration. In the script version below, we prefixed the declaration with the simple keyword. Now, the `source` parameter’s type is *“simple int”* instead of “series int”. With this change, the script *does not* cause an error, because the `pass()` function’s returned type is now compatible with the `length` parameter of ta.ema():

```
//@version=6 indicator("Declared parameter qualifier demo") //@function Returns the value of the `source` argument. Requires an "int" value with // "simple" or a weaker qualifier. pass(simple int source) => source //@variable The EMA smoothing length. // This `pass()` call's result is a "simple int" value, because the function definition explicitly // declares that the `source` parameter's qualifier is "simple". int lengthVal = pass(14) // This call does not cause an error. The qualified type of `lengthVal` matches the `length` parameter's type. float emaDiff = ta.ema(close - open, length = lengthVal) // Plot the `emaDiff` series. plot(emaDiff, "Smoothed difference", color.teal, 3)
```

For some types of calculations, using “series” values might not cause a compilation error, but if the values do not remain consistent across all bars, the calculations produce incorrect or unintended results. When wrapping such calculations in a function, declaring the relevant parameters with the simple keyword – or const when appropriate – ensures that only *unchanging* arguments are allowed in each call, preventing unintended behavior.

Let's look at an example. The following `calcAvg()` function calculates a moving average of a `source` series over `length` bars. The function compares the value of its `avgType` parameter in a switch statement to select a built-in `ta.*()` call to use for the average calculation:

//@function Calculates a moving average of a `source` series over `length` bars. // The selected average calculation depends on the `avgType` value. calcAvg(float source, int length, string avgType) => switch avgType "ema" => ta.ema(source, length) "sma" => ta.sma(source, length) "wma" => ta.wma(source, length) "hma" => ta.hma(source, length)

The compiler raises a *warning* about this function's structure inside the Pine Editor, because using `calcAvg()` with a *dynamic* `avgType` argument can cause *unintended results*. If the `ta.*()` call executed by the function changes on any bar, it affects the *history* of values used in the average calculations. See the Time series in scopes section of the Execution model page for advanced details about this behavior.

The script below executes two `calcAvg()` calls and plots their returned values. The first call consistently uses `"ema"` as its `avgType` argument, and the other alternates between using `"ema"` and `"sma"` as the argument. The second call's result *does not* often align with the first call's result, even on the bars where its `avgType` argument is `"ema"`, because both `ta.*()` calls require *consistent* evaluation to calculate the averages for *consecutive* bars:

//@version=6 indicator("Inconsistent behavior demo", overlay = true, behind_chart = false) //@function Calculates a moving average of a `source` series over `length` bars. // The selected average calculation depends on the `avgType` value. calcAvg(float source, int length, string avgType) => switch avgType "ema" => ta.ema(source, length) "sma" => ta.sma(source, length) "wma" => ta.wma(source, length) "hma" => ta.hma(source, length) //@variable The 10-bar EMA of *consecutive* `close` values. float avg1 = calcAvg(close, 10, "ema") //@variable The EMA of `close` values from *even* bar indices, or the SMA of values from *odd* bar indices. // Either average uses only the values from *every other bar* in its calculation, **not** consecutive bars. float avg2 = calcAvg(close, 10, bar_index % 2 == 0 ? "ema" : "sma") // Plot `avg1` and `avg2` for comparison. plot(avg1, "Consistent EMA", color.blue, 4) plot(avg2, "Inconsistent EMA/SMA", color.purple, 3)

To ensure that any `calcAvg()` call calculates consistent averages without modifying the function's logic, we can prevent it from using *dynamic* `avgType` arguments by prefixing the parameter declaration with the simple keyword. With this change, the compiler does *not* raise a warning about the function – as long as the script evaluates calls to the function on *every bar* – because any `calcAvg()` call must always use the same `ta.*()` function. Now, if the script attempts to pass a "series string" value to `avgType`, a compilation error occurs:

//@version=6 indicator("Promoting consistency with qualifier keywords demo", overlay = true, behind_chart = false) //@function Calculates a moving average of a `source` series over `length` bars. // The selected average calculation depends on the `avgType` value. // The average type *cannot change* across bars. calcAvg(float source, int length, simple string avgType) => switch avgType "ema" => ta.ema(source, length) "sma" => ta.sma(source, length) "wma" => ta.wma(source, length) "hma" => ta.hma(source, length) // This `calcAvg()` call works successfully, because the `avgType` argument is a "const string" value. float avg1 = calcAvg(close, 10, "ema") // This `calcAvg()` call now causes an error, because a dynamic `avgType` argument is *not allowed*. float avg2 = calcAvg(close, 10, bar_index % 2 == 0 ? "ema" : "sma") // Plot `avg1` and `avg2` for comparison. plot(avg1, "Consistent EMA", color.blue, 4) plot(avg2, "Inconsistent EMA/SMA", color.purple, 3)

Documenting functions

Pine Script features annotations that programmers can use to document a function, its parameters, and its result directly in the source code. Annotations are *comments* that issue special *instructions* to the compiler or the Pine Editor. The editor can display the formatted text from function annotations in a pop-up window as users work with the function in their code. Additionally, the “Publish script” window uses the text from function annotations to generate default descriptions for libraries.

To annotate a function, add comment lines containing valid annotation syntax directly above the function’s header. Each annotation comment must start with `@`, immediately followed by the keyword that indicates its purpose. Below is the syntax for all annotations that apply exclusively to function and method definitions:

```
//@function <description>
```

The //@function annotation defines the *main description* of the function or method. This annotation is where the programmer documents the function’s purpose and key behaviors. The Pine Editor displays the formatted `description` text in its pop-up window while the user hovers over the function’s name or writes a function call.

```
//@param <parameterName> <description>
```

The //@param annotation defines the description of a specific function *parameter*. The Pine Editor displays the `description` text beneath the parameter’s type in its pop-up window while the user *writes an argument* for that parameter in a function call.

In this syntax, `parameterName` is the name of one of the function’s *parameters*. If the annotation does not include a parameter name, or if the specified name does not match one of the listed parameters, the annotation is *ignored*.

```
//@returns <description>
```

The //@returns annotation defines the description of the function’s *returned data*. The Pine Editor displays the `description` text at the bottom of the pop-up window that appears while the user hovers over the function’s name.

> NoteRedundant annotations are automatically *ignored*. If two or more `//@function` or `//@returns` annotations are above a function header, only the **last** one adds a description to the function or its return expression. If two or more `//@param` annotations above the header share the same parameter name, only the **first** one defines the corresponding parameter’s description.

The following code block defines a `mixEMA()` function, which calculates an EMA of a `source` series and then mixes the EMA with that series by a specified amount (`mix`). Above the function definition, we included `//@function`, `//@param`, and `//@returns` annotations to document its purpose, parameters, and result, respectively. Users can view the formatted text from these annotations by hovering over the `mixEMA` identifier in the Pine Editor or writing a `mixEMA()` function call:

//@function Calculates an EMA of a source series, and mixes it with the `source` value. //@param source The series of values to process. //@param length The length of the EMA's smoothing parameter. //@param mix Optional. The mix ratio. Requires a value from 0 to 1. The default is 1. //@returns The mixture between the `source` value and its EMA. mixEMA(float source, int length, float mix = 1.0) => // Raise an error if the `mix` value is outside the supported range. if mix < 0 or mix > 1 runtime.error("The `mix` value must be a

number between 0 and 1.") // Calculate the EMA of `source`. float ma = ta.ema(source, length) // Mix the `source` and `ma` values and return the result. float result = (1.0 - mix) * source + mix * ma

Note that:

- The pop-up window that shows annotation text automatically *removes* most leading or repeated whitespaces in its formatted result.

The description attached to any function annotation can occupy *more than one* comment line in the source code, but only if there are *no blank lines* between each comment line. For example, the following code block contains `//@function` and `//@param` annotations that both define descriptions across *three* comment lines:

//@function Calculates an EMA of a source series with a specified `length`, and then // computes a mixture between the EMA and the original value based on the `mix` value. // For consistency, calls to this function should execute on every bar. //@param source The series of values to process. //@param length The length of the EMA's smoothing parameter. //@param mix Optional. The mix ratio. Requires a value from 0 to 1. If 0, the result is the `source` value. // If 1, the result is the EMA. If between 0 and 1, the result is a mixture between the two values. // The default is 1. //@returns The mixture between the `source` value and its EMA. mixEMA(float source, int length, float mix = 1.0) => // Raise an error if the `mix` value is out of the supported range. if mix < 0 or mix > 1 runtime.error("The `mix` value must be a number between 0 and 1.") // Calculate the EMA of `source`. float ma = ta.ema(source, length) // Mix the `source` and `ma` values and return the result. float result = (1.0 - mix) * source + mix * ma

Note that each separate comment line in an annotation block *does not* typically create a new line of text in the displayed description. Programmers can create annotations with *multiline* descriptions by adding *empty* comment lines to the annotation block. For example:

//@function Calculates an EMA of a source series with a specified `length`, and then // computes a mixture between the EMA and the original value based on the `mix` value. // // (**New text line**) For consistency, calls to this function should execute on every bar. mixEMA(float source, int length, float mix = 1.0) => // Raise an error if the `mix` value is out of the supported range. if mix < 0 or mix > 1 runtime.error("The `mix` value must be a number between 0 and 1.") // Calculate the EMA of `source`. float ma = ta.ema(source, length) // Mix the `source` and `ma` values and return the result. float result = (1.0 - mix) * source + mix * ma

Annotations support a limited range of Markdown syntax, which enables *custom text formats* in the Pine Editor's pop-up window. The example below shows some of the syntax that the window can render. To view the annotation's results, copy the code and hover over the `f` identifier inside the Pine Editor:

//@function This annotation shows some common Markdown syntax that the Pine Editor can render in its pop-up window. // // --- // // `Monospace with gray background` // // *Italic text* // // **Bold text** // // ***Bold and italic*** // // ~Strikethrough~ // // --- // // > Block quotation // // --- // // Bulleted list: // - Item 1 // - Item 2 // // --- // // Numbered list: // 1. Item 1 // 1. Item 2 // // --- // // ``` // // Code block format // float x = 1.5 // ``` // // --- // // Hyperlink: // // The [@function](https://www.tradingview.com/pine-script-reference/v6/#an_@function) annotation is very flexible. // // --- // // # Heading 1 // ## Heading 2 // ### Heading 3 // // --- f() => int(na)

> NoteThe annotation syntax in the above example affects only the appearance of text displayed by the Pine Editor's autosuggest feature; it **does not** affect text formatting in *script publications*. See the Title and description section of the Publishing scripts page to learn the formatting syntax for publication descriptions.

Function scopes

All variables, expressions, and statements have a designated scope, which refers to the part of the script where they are defined and accessible. Every script has one *global* scope and zero or more *local* scopes.

Each function definition creates a distinct local scope. All code written in the function's header and body belongs exclusively to that definition; no code outside the function can access its declared variables or parameters.

For example, the following script defines a function named `myFun()`. The function's body contains a variable named `result`. The script attempts to use a plot() call, in the global scope, to display the value of that variable. This script causes a compilation error, because the global scope *cannot* access the function's *local* identifiers, and there is no global variable named `result`:

//@version=6 indicator("Inaccessible scope demo") //@function Multiplies `x` by a pseudorandom value and returns the result. myFun(float x = 10) => // This variable belongs to the function definition; other scopes cannot access it. float result = math.random() * x // This `plot()` call causes a compilation error. There is no *global* `result` variable available for plotting. plot(result)

The only way for the script to use the function's code is to execute a *function call*. Additionally, because the code in the function definition is *inaccessible* to other scopes, the script can declare separate variables in other parts of the code with the *same* identifiers as the function's local variables and parameters. For example, the following script executes a call to the `myFun()` function and assigns its returned value to a new, *global* variable *also* named `result`. The plot() call in this script does not cause an error, because only the global `result` variable is available to that call:

//@version=6 indicator("Function variable vs. global variable demo") //@function Multiplies `x` by a pseudorandom value and returns the result. myFun(float x = 10) => // This variable belongs to the function definition; other scopes cannot access it. float result = math.random() * x // This variable stores the result returned by a `myFun()` call. // Although it shares the same name as the variable in `myFun()`, it is a completely separate variable declared // in the global scope, after the function definition. float result = myFun() // This `plot()` call does not cause an error, because `result` refers to the *global* variable. plot(result, "Global `result` series", linewidth = 3)

All local scopes in a script, including the scope of a function definition, are embedded into the script's global scope. Therefore, while the global scope cannot access any variables within a function's scope, the function *can* access any global variables declared *above* its definition.

For instance, the following script declares a `globalVar` variable before defining the `myFun()` function, then uses that variable in the function's body. This script compiles successfully, because the function definition has access to any global variables declared before its location in the code:

//@version=6 indicator("Global variables in functions demo") //@variable A global variable that holds a pseudorandom value. var float globalVar = math.random() //@function Multiplies `x` by a pseudorandom value, adds the value of `globalVar`, then returns the result. // This function can use `globalVar` in its body because the script declares the variable *first*. myFun(float x = 10) => math.random() * x + globalVar // Assign the result of a default `myFun()` call to a new variable. The call uses `globalVar` internally. float result = myFun() // Plot the `result` series. plot(result, "`result` series", color.teal, 3)

It's important to note that all global variables used in a function's body cannot change their assigned values or references during any execution of a function call. Therefore, although user-defined functions can *access* some global variables, they *cannot modify* those variables using the reassignment or compound assignment operators. Similarly, a function cannot reassign its declared parameters, because the parameters of each function call must always hold the same values or references passed to them from the scope where that call occurs.

The example script below demonstrates this limitation. It declares a global variable named `counter` with an initial value of 0. Then, it defines an `increment()` function that uses the `+=` operator to change the `counter` variable's assigned value, causing a compilation error:

//@version=6 indicator("Cannot modify global variables demo") //@variable A global variable to hold a counter value. var int counter = 0 //@function Attempts to use `+=` to increment the global `counter` value by the value of `amountToAdd`. // This definition causes an error, because global variables cannot be modified // during the execution of a user-defined function call. increment(int amountToAdd = 1) => counter += amountToAdd plot(increment())

> TipIt is possible to modify the data accessed by a global variable from inside a function's scope if that variable is of a reference type, such as a collection type or user-defined type. Instead of reassigning the variable, the script can use *setter functions* or *field reassignments*, depending on the type, to modify the *object* that the variable *refers* to. For examples of this advanced technique, see the Extracting data from local scopes section of the Debugging page and the scope-related sections of the Arrays, Matrices, and Maps pages.

Scope of a function call

A function's definition acts as a *template* for each function call. Every call *written* in the source code establishes a *separate* local scope using that definition. The parameters, variables, and expressions created for each function call are *unique* to that call, and they leave *independent* historical trails in the script's time series. Because each written call has a separate scope, with an independent history, no two calls to the same function directly affect each other.

For example, the script below contains a user-defined function named `accumulate()`. The function's structure declares a *persistent* variable named `total` with an initial value of 0, then adds the value of its `source` parameter to that variable and returns the result. The script uses two separate `accumulate()` calls inside a ternary operation, and then plots the result returned by that expression. The operation triggers the first written call on every *third* bar, and the second on all other bars. Both calls use a value of 1 as their `source` argument:

//@version=6 indicator("Independent call scopes demo") //@function Accumulates the values from a `source` series across executions. // Each written call to this function defines a *separate* scope with unique versions of `source` and `total`. accumulate(float source) => //@variable A *persistent* variable declared on only one bar, initialized with a value of 0. var float total = 0.0 // Add the `source` value to `total` and return the variable's new value. total += source // Use two calls to `accumulate()` with the same `source` argument inside a ternary expression. // Each call has its own version of `total`, which updates only on bars where the script *evaluates* that call. // These two calls thus return **different** results, because the script does not evaluate both of them on the // *same* number of bars. float altCallSeries = bar_index % 3 == 0 ? accumulate(1) : accumulate(1) plot(altCallSeries, "Alternating call results", color.purple, 3)

Note that:

- The `total` variable and its assigned value *persist* across bars because the variable declaration includes the var keyword. See the Declaration modes section of the Variable declarations page to learn more.

Both `accumulate()` calls in this script might seem identical. However, each one has a *separate* scope with distinct versions of the `source` parameter and the `total` variable. The first call **does not** affect the second, and vice versa. As shown below, the script's plotted value *decreases* on every third bar before increasing again on subsequent bars. This behavior occurs because each `accumulate()` call's version of `total` increases its value only on bars where the script *evaluates* that call, and the script evaluates the first call on about *half* as many bars as the second:

It is crucial to emphasize that each function call *written* in the source code has **one** unique scope created from the function's definition. Repeated evaluations of the same written call **do not** create additional scopes with separate local variables and history. For example, a function call written in the body of a loop performs calculations using the *same* local series on *every* iteration; it does *not* calculate on different versions of those series for each separate iteration.

The following script demonstrates this behavior. The source code includes two written calls to our previous `accumulate()` function. The script evaluates the first call in the global scope, and the second inside the body of a for loop that performs 10 iterations per execution. It then plots the results of both calls in a separate pane:

//@version=6 indicator("Written call scopes in structures demo") //@function Accumulates the values from a `source` series across executions. // Each written call to this function defines a **separate** scope with unique versions of `source` and `total`. accumulate(float source) => //@variable A *persistent* variable declared on only one bar, initialized with a value of 0. var float total = 0.0 // Add the `source` value to `total` and return the variable's new value. total += source //@variable Holds the result of evaluating `accumulate(1)` in the global scope. The value increases by *one* on each bar. float globalCallRes = accumulate(1) //@variable Initialized with `na`, then reassigned the result of an `accumulate()` call within a loop. float loopedCallRes = na for _ = 1 to 10 // Reassign the `loopedCallRes` variable using the result of `accumulate(1)`. // This written call establishes **one** new scope with a unique version of the `total` variable. // The script evaluates that scope on every loop iteration, thus adding 1 to the *same* version of `total` // *10 times* per bar. loopedCallRes := accumulate(1) // Plot the results of global and looped calls for comparison. plot(globalCallRes, "Global call result", color.teal, 3) plot(loopedCallRes, "Looped call result", color.red, 3)

A newcomer to Pine might expect the results of both `accumulate()` calls to be equal. However, the result of the call inside the loop is *10 times* that of the call evaluated in the global scope. This difference occurs because the call written in the loop does not have separate scopes for each iteration; every evaluation of that call modifies the *same* version of the persistent `total` variable. Consequently, the value returned by the loop's `accumulate()` call increases by **10** instead of one on each bar:

Note that:

- Both `accumulate()` calls return the *same* result if we remove var from the `total` variable declaration. Without the keyword, each call's version of the `total` variable no longer *persists*. Instead, every evaluation of the call *re-declares* the variable and initializes it to 0 before adding the value of the `source` argument.

Function overloading

Function *overloads* are *unique versions* of a function that share the same name but differ in their required parameter types. Overloading enables function calls using the same identifier to perform *different tasks* based on their specified arguments. Programmers often write overloads to group similar calculations for specific types under a single function name, offering a convenient alternative to defining several related functions with unique names.

In Pine Script, a function overload is valid only if its definition satisfies one of the following conditions:

- It has a different number of *required* parameters (parameters *without* default arguments) than that of any other defined overload.
- It has the same number of required parameters as another overload, but at least *one* required parameter has a different qualified type than the one declared at the *same position* in the other overload's header.

The following code block defines three overloads of a custom `negate()` function. Each overload has a different parameter type and performs a distinct task. The first overload uses the `-` operator to negate a "float" or "int" value; the second uses the `not` operator to negate a "bool" value; and the third calculates the sRGB negative of a "color" value using the built-in `color.*()` functions:

//@function (Overload 1) Changes the sign of a number. //@param value The "float" or "int" value to process. //@returns The negated number. negate(float value) => -value //@function (Overload 2) Negates a Boolean value. //@param value The "bool" value to process. //@returns `false` if the value is `true`, or `true` if the value is `false`. negate(bool value) => not value //@function (Overload 3) Calculates the sRGB negative (i.e., complement) of a color. //@param value The "color" value to process. //@returns The "color" value whose red, green, and blue components are opposites of the `value` color's components // in the sRGB color space. negate(color value) => color.rgb(255 - color.r(value), 255 - color.g(value), 255 - color.b(value), color.t(value))

Note that:

- The `value` parameter of each `negate()` overload automatically inherits the *"series"* qualifier, because its declaration includes a type keyword *without* a qualifier keyword, and the overload does not use the parameter in a local function call that requires a "simple" or weaker qualifier.
- We included annotations to document each separate overload. As a user writes a `negate()` call, the Pine Editor shows the documentation for *one* of the overloads in a pop-up window. While the window is open, the user can view the documentation for the *other* overloads by using the Up and Down arrow keys.

With our function overloads defined, we can use `negate()` calls for different type-specific tasks. The script below uses a call to each overload. First, it uses the second overload to calculate the opposite of the condition `close > open`. It then uses the negated condition to determine the value of a plotted series. The plotted value is the result of `negate(close)` if the condition is `true`, and the value of close otherwise. The script colors the plot using the negative of the chart's background color (`negate(chart.bg_color)`):

//@version=6 indicator("Function overloading demo") //@function (Overload 1) Changes the sign of a number. //@param value The "float" or "int" value to process. //@returns The negated number. negate(float value) => -value //@function (Overload 2) Negates a Boolean value. //@param value The "bool" value to process. //@returns `false` if the value is `true`, or `true` if the value is `false`. negate(bool value) => not

value //@function (Overload 3) Calculates the sRGB negative (i.e., complement) of a color. //@param value The "color" value to process. //@returns The "color" value whose red, green, and blue components are opposites of the `value` color's components // in the sRGB color space. negate(color value) => color.rgb(255 - color.r(value), 255 - color.g(value), 255 - color.b(value), color.t(value)) //@variable `true` if the current `close` value is *not* greater than the value of `open`. Otherwise, `false`. bool notUpBar = negate(close > open) //@variable The `-close` value if the `notUpBar` condition is `true`, or the `close` value if the condition is `false`. float plotSeries = notUpBar ? negate(close) : close //@variable The sRGB negative of the chart's background color. color plotColor = negate(chart.bg_color) // Plot the value of `plotSeries` and color it using `plotColor`. plot(plotSeries, "Test plot", plotColor, style = plot.style_area)

It's important to emphasize that a function overload is valid only if its signature contains a *unique* set of qualified types for its *required* parameters. Differences in *optional* parameters (the parameters that have default arguments) **do not** affect the validity of overloads. The compiler raises an *error* if two overloads share the same required parameter types and differ only in optional parameters, because it cannot determine which overload to use for *all* possible function calls.

For example, the following code block defines a fourth `negate()` overload with two "float" parameters. The overload's first parameter requires an argument, but the second one *does not* because it has a default value. With this addition, any `negate()` call with a single "float" argument becomes *ambiguous*. Such a call might refer to the first overload or the fourth, and the compiler *cannot* confirm which one to use in that case. Therefore, the extra overload causes a compilation error:

//@function (Overload 1) Changes the sign of a number. //@param value The "float" or "int" value to process. //@returns The negated number. negate(float value) => -value //@function (Overload 2) Negates a Boolean value. //@param value The "bool" value to process. //@returns `false` if the value is `true`, or `true` if the value is `false`. negate(bool value) => not value //@function (Overload 3) Calculates the sRGB negative (i.e., complement) of a color. //@param value The "color" value to process. //@returns The "color" value whose red, green, and blue components are opposites of the `value` color's components // in the sRGB color space. negate(color value) => color.rgb(255 - color.r(value), 255 - color.g(value), 255 - color.b(value), color.t(value)) // This overload is **invalid**. // Although the function has two parameters, `scale` has a default argument. The only parameter that requires an // argument is `value`, and that parameter shares the same type as the required parameter in Overload 1. // Any `negate()` call with a single "float" argument is *ambiguous* in this case, so the compiler raises an error. negate(float value, float scale = 1.0) => -value * scale

Another crucial limitation to note is that the *names* of parameters **do not** affect the validity of overloads. The compiler validates overloads by analyzing only the *qualified types* of their required parameters at each *position* in the function headers, because function calls can use *positional* or *named* arguments for each parameter. Named arguments specify the parameter to which they apply. For example, `f(x = 10)` passes the value 10 to the `x` parameter of the `f()` call. In contrast, positional arguments omit names and apply to the parameters in order: the first argument to the first parameter, the second to the second parameter, and so on.

If two overloads have the same parameter types but different parameter names, a compilation error occurs, because any call that uses positional arguments is ambiguous. For example, the code block below defines a separate `negate()` overload with a single "float" parameter named `comparedValue`. That overload causes a compilation error, as the required parameter's qualified type matches that of the `value` parameter in the first overload:

//@function (Overload 1) Changes the sign of a number. //@param value The "float" or "int" value to process. //@returns The negated number. negate(float value) => -value //@function (Overload 2) Negates a Boolean value. //@param value The "bool" value to process. //@returns `false` if the value is `true`, or `true` if the value is `false`. negate(bool value) => not value //@function (Overload 3) Calculates the sRGB negative (i.e., complement) of a color. //@param value The "color" value to process. //@returns The "color" value whose red, green, and blue components are opposites of the `value` color's components // in the sRGB color space. negate(color value) => color.rgb(255 - color.r(value), 255 - color.g(value), 255 - color.b(value), color.t(value)) // This overload is **invalid**. // Although the required parameter has a different name from that of Overload 1, it is possible to call the // function using a *positional* argument. The compiler cannot determine whether calls such as `negate(close)` // should refer to the first overload or this one, so it raises an error. negate(float comparedValue) => not (comparedValue > open)

In Pine, function overloads can contain calls to overloads with the *same* function name in their bodies, but only if those overloads are defined first. However, just like non-overloaded functions, an overload *cannot* use calls to *itself* within its body.

For example, the script version below defines a fourth `negate()` overload with two required “float” parameters. The new overload calculates the product of its arguments, then calls the *first* overload of `negate()` to change the result’s sign. This script compiles successfully, because the fourth overload uses a *separate* `negate()` implementation in its scope, *not* a call to itself:

//@version=6 indicator("Overloads calling other overloads demo") //@function (Overload 1) Changes the sign of a number. //@param value The "float" or "int" value to process. //@returns The negated number. negate(float value) => -value //@function (Overload 2) Negates a Boolean value. //@param value The "bool" value to process. //@returns `false` if the value is `true`, or `true` if the value is `false`. negate(bool value) => not value //@function (Overload 3) Calculates the sRGB negative (i.e., complement) of a color. //@param value The "color" value to process. //@returns The "color" value whose red, green, and blue components are opposites of the `value` color's components // in the sRGB color space. negate(color value) => color.rgb(255 - color.r(value), 255 - color.g(value), 255 - color.b(value), color.t(value)) //@function (Overload 4) Changes the sign of the product of two numbers. //@param value1 The first "float" or "int" value to process. //@param value2 The second value to process. //@returns The negated product. negate(float value1, float value2) => // Using this call in the overload's body does not cause an error, because it is *non-recursive*. // It refers to the *first* `negate()` implementation, not the fourth. negate(value1 * value2) //@variable `true` if the current `close` value is *not* greater than the value of `open`. Otherwise, `false`. bool notUpBar = negate(close > open) //@variable `-close * 2` if the `notUpBar` condition is `true`, or `close * 2` if the condition is `false`. float plotSeries = notUpBar ? negate(close, 2) : negate(close, -2) //@variable The sRGB negative of `color.green`. color plotColor = negate(color.green) // Plot the value of `plotSeries` and color it using `plotColor`. plot(plotSeries, "Test plot", plotColor, style = plot.style_area)

Note that:

- If we change the fourth overload to use a `negate()` call with *two* “float” arguments, a compilation error occurs because the overload calls *itself* in that case. Even if Pine allowed recursive functions, using such an overload would cause an error because it creates the equivalent of an *infinite loop*, where a single call activates a second call, the second call activates a third one, and so on.

Limitations

This section explains several key limitations common to all user-defined functions. These same limitations also apply to user-defined methods.

No global-only built-in function calls

The body of a user-defined function can contain calls to most other functions or methods, or previously defined overloads with the same function name. However, a function *cannot* use calls to the following *built-in* functions inside its scope:

- Declaration statements: indicator(), strategy(), and library().
- Plot-related functions: plot(), hline(), fill(), plotshape(), plotchar(), plotarrow(), plotbar(), plotcandle(), barcolor(), bgcolor(), and alertcondition().

Global variables can be assigned values returned by functions in order to be used in calls to the above functions. However, the calls to these functions are allowed only in the script's global scope, outside the operands of ternary operations or other conditional expressions.

Consistent types for each call

Each written call to a user-defined function must have *consistent* parameter types and return types during each evaluation of that call. The call *cannot* accept varying argument types throughout the script's runtime, nor can it change its return type. Similar limitations also apply to other structures and expressions, including loops, conditional structures, and ternary operations.

Cannot modify global variables or parameters

The body of a user-defined function cannot use the *reassignment* or *compound assignment* operators to modify variables declared in the *global scope*. Likewise, it cannot use these operators to reassign function *parameters*. The value or reference assigned to a global variable or the parameter of a function call *cannot change* while the script evaluates the call. See the Function scopes section for an example.

No nested definitions

The body of a user-defined function *cannot* contain the definition of another user-defined function. Likewise, loops and conditional structures cannot include function definitions in their local blocks. Function and method definitions are allowed only in the script's *global scope*.

For example, the following `f()` function definition causes a compilation error, because it attempts to define another function, `g()`, within its body:

//@version=6 indicator("Invalid nested definition demo") f(x, y) => // Defining a separate function inside the body of `f()` is not allowed. g(a, b) => math.sqrt(a * a + b * b) g(x, y) / (x + y) plot(f(open, close))

Instead of attempting to define `g()` within the scope of `f()`, we can move the definition of `g()` above `f()` in the global scope:

//@version=6 indicator("Moving nested definitions to the global scope demo") // If we move `g()` to the global scope, above `f()`, the `f()` function can use it, and no error occurs. g(a, b) => math.sqrt(a * a + b * b) f(x, y) => g(x, y) / (x + y) plot(f(open, close))

No recursive functions

In contrast to functions in some programming languages, user-defined functions in Pine Script *cannot* contain calls to *themselves* within their bodies. Instead of using recursive function structures, programmers can replace those structures with equivalent loop calculations.

For instance, the following `gcd()` function causes a compilation error because it includes a `gcd()` call within its body:

//@version=6 indicator("Invalid recursive function demo") gcd(int x, int y) => // This expression causes an error. `gcd()` cannot call itself. y == 0 ? x : gcd(y, x % y) plot(gcd(15, 20))

Instead of attempting to use recursion, we can reformat the function's structure to perform the necessary calculations using *iteration*. The example version below achieves our intended calculations using a while loop and no recursive `gcd()` calls:

//@version=6 indicator("Replacing recursion with loops demo") // This function performs the same intended calculations as the invalid recursive structure. // It uses separate local variables to track values, and modifies those variables in a `while` loop // until it finds the greatest common divisor for two positive integers. gcd(int x, int y) => int a = x int b = y while b != 0 int remainder = a % b a := b b := remainder a plot(gcd(15, 20))

Variable declarations

Introduction

Variables are *named containers* that store calculated values or other data for a script to access and use within a given scope. Variables in Pine Script® can hold data of any available type that is not void, including the direct values of value types, and the *IDs* (references) of drawings, collections, plots or other instances of reference types.

A variable in Pine Script consists of three main parts:

- An identifier (name), which represents the variable in the source code.
- A qualified type, which determines the kind of data the variable stores and whether the data can change.
- An assigned value or reference.

Programmers write *variable declarations* to create *custom* variables for working with data of specific types when the available built-in variables do not suffice. A variable declaration is a statement specifying that, from a particular point onward in a specific *scope*, an identifier refers to a variable with a given initial value or reference. The script accesses the saved value or reference while evaluating expressions or statements that use the variable's identifier.

There are two forms of variable declarations in Pine Script:

- Single-variable declarations declare and initialize *one* variable. Programmers can include optional keywords in the statement to define the variable's type and its declaration behavior, or to export the variable from a library.
- Tuple declarations declare and initialize *multiple* variables using a *tuple* format. Programmers use these statements to declare variables that hold the data from *function calls*, conditional structures, or loops that return tuples of data.

All of the statements in the following code block are examples of valid variable declarations. Each identifier to the left of an = operator in the code is the *name* of a *new variable*, and the expression or structure to the right determines that variable's initial value or reference:

// Declares a variable named `oc2` that holds a "series float" value. oc2 = (open + close) / 2 // Declares a variable named `MULT` that holds a "const float" value. // The `const` and `float` keywords are optional. Using `const` prevents the script from changing the value later. const float MULT = 2.5 // Declares *three* variables named `basis`, `upper`, and `lower` to hold all the values returned by `ta.bb()`. // This declaration format does not support keywords; each variable inherits the type of its assigned value. [basis, upper, lower] = ta.bb(oc2, 20, MULT) // Declares a variable named `ratio`. The type is "series float". // The `float` keyword is optional, but helps promote readability. float ratio = math.pow((oc2 - basis) / (upper - lower), 3) // Declares a `ratioColor` variable to hold a "series color" value returned by a `switch` structure. // The `series` and `color` keywords are optional. series color ratioColor = switch ratio > 0.05 => color.green ratio < -0.05 => color.red => color.gray // Declares a variable named `historyBarsStr` on the *first* bar only. Its type is "series string". // The `var` keyword causes the variable and its value to *persist* across subsequent bars. var historyBarsStr = "Historical bars: " + str.tostring(last_bar_index + 1) // Declares a persistent `timeLabel` variable to hold a `label` ID. The `label` keyword is optional. var label timeLabel = label.new(last_bar_time, 0, historyBarsStr, xloc.bar_time, yloc.price, color.blue, label.style_label_left) // Declares a persistent "float"

variable named `highLevel`. // The `float` keyword is *required* because the initial value is *undefined*. // The `varip` keyword causes the variable to persist across *every* execution, not just every bar. varip float highLevel = na // These statements declare variables to hold "plot" IDs, which the script can use in `fill()` function calls. ratioPlot = plot(ratio, "Ratio", ratioColor, 3) basisPlot = plot(0, "Zero")

Regardless of format, several key characteristics and limitations apply to user-defined variables:

- Every variable has *one* qualified type, even if its declaration does not explicitly specify the type in the code. Variables declared without type keywords or qualifier keywords *inherit* type information from their assigned data. A variable's qualified type *never changes* across script executions.
- Most custom variables are *mutable*. Scripts can reassign mutable variables by using the reassignment or compound assignment operators. However, they *cannot* reassign any *global* variables from inside user-defined functions or methods.
- The scope of a variable depends on the location of its declaration in the code. The scope determines which parts of the script can *access* that variable. A variable is available to all parts of a script *after* its declaration in the *same scope* or a *nested scope*, but **not** to any part that is *before* the declaration or in an *outer scope*.
- Variables in *different* scopes can have the *same name*, but all variables in the *same* scope require *unique names*. The only exception is for variables whose identifier is an underscore (_), which makes them *unusable* in any expressions or statements.
- If a variable in a nested scope has the same name as one in an outer scope, that variable shadows the outer scope's variable. In other words, the script *cannot access* the outer scope's variable in any part of the nested scope following the inner variable's declaration.
- By default, a script declares and initializes a variable anew during *each execution* of its scope. However, a single-variable declaration can include the var or varip keyword to set an alternative declaration mode, causing the variable and its data to *persist* across bars or ticks.

Single-variable declarations

A single-variable declaration is a statement that creates one new variable, names it, and assigns it an initial value or reference. The statement can include *keywords* to specify the variable's qualified type and declaration mode, or to export the variable. The syntax is as follows:

```
[export ][var |varip ][[qualifier ]<type> ]<identifier> = <expression>|<structu
```

Where:

- The `|` character represents *OR*, all parts enclosed in angle brackets (`<>`) represent *required* syntax, and all parts in square brackets (`[]`) represent *optional* syntax.
- export is the optional keyword for exporting the variable from a library, enabling its use in other scripts. Exporting is allowed only if the variable is of a fundamental type and the declaration includes the const keyword.
- var and varip are optional keywords that cause the variable and its data to *persist* across bars or ticks. If the declaration does not include either keyword, the script *reinitializes* the variable during *every* execution of the variable's scope. Refer to the Declaration modes section for more information.
- `qualifier` and `type` refer to *keywords* for specifying the variable's qualified type. These keywords are usually optional. If the declaration does not include them, the variable's assigned data determines its

type information. See the Declaring qualified types section to learn more.
- `identifier` is the variable's *name*.
- = is the assignment operator. The `expression` or `structure` part to the right of the operator determines the initial value or reference that it assigns to the new variable. `expression` refers to a literal value, the identifier of another variable, an operation, or a *function* or method call that returns a single value or reference. `structure` refers to any conditional structure or loop that returns a single value or reference.

The example below demonstrates a single-variable declaration that declares a "float" variable named `median` to hold the current value returned by a ta.median() function call:

//@variable Holds the 20-bar median of `hl2` values as of the current bar. float median = ta.median(hl2, 20)

Note that:

- This statement initializes the `median` variable anew on *every* execution, because it does not specify a different declaration mode with the var or varip keyword. Each execution thus *updates* the variable with the function call's latest result for the current bar.
- The `//@variable` comment is an optional annotation that *documents* the declared variable in the code. Users can hover over the `median` identifier in the Pine Editor to view a pop-up window that displays the specified line of text.

After a script declares a variable, it can then use that variable in any subsequent part of the code in the same scope or a nested scope. The variable's identifier serves as a *placeholder* for a specific value or reference in the script's logic. When the script evaluates an expression that contains the identifier, it retrieves the variable's saved data and uses that data in the calculation.

For example, the following script calculates the median of hl2 values over a specified number of bars, then plots the median on the chart as a color-coded line. It declares variables to store the median and other values for the calculations, and uses three of the variables as arguments for the plot() call at the end:

//@version=6 indicator("Single-variable declarations demo", overlay = true) //@variable Holds a "const string" value for use as the `title` argument in the `plot()` call. const string PLOT_TITLE = "Median" //@variable Holds an "input int" value for use as the `length` argument in the `ta.median()` call. int lengthInput = input.int(20, "Length", 1) //@variable Stores the current median of `hl2` values over `lengthInput` bars. The value updates on every bar. // The script uses it to calculate `isUptrend`, and to define the `series` argument of the `plot()` call. float median = ta.median(hl2, length = lengthInput) //@variable Holds `true` if the last change in the `median` value was positive, and `false` otherwise. bool isUptrend = ta.valuewhen(median != median[1], median > median[1], 0) //@variable Holds the value of `color.green` if the value of `isUptrend` is `true`, and `color.red` otherwise. color plotColor = if isUptrend color.green else color.red // Plot the current `median` value. Set the plot's title and color using the values of `PLOT_TITLE` and `plotColor`. plot(series = median, title = PLOT_TITLE, color = plotColor, linewidth = 3)

Note that:

- The const keyword specifies that the script cannot reassign the variable. For value types such as "string", it also declares that the variable's qualifier is "const", meaning that it accepts only *constant* values that are available at *compile time*.
- The script uses the int, float, bool, color, and string keywords to specify the type of each variable. Using type keywords is optional in all the above declarations, because the compiler can automatically

determine the appropriate types, but doing so helps promote readability. See the Declaring qualified types section to learn more about type and qualifier keywords.
- The script can assign the result of the if structure to a variable because both of the structure's local blocks return the same type ("color"). See the Matching local block type requirement section of the Conditional structures page to learn more.

It's important to note that a script *cannot* use a custom variable in any expressions or statements that *precede* the variable's declaration, because the variable is *not available* at that point in the code. Attempting to use a variable in any code before its declaration causes a compilation error.

For example, moving the `median` declaration in the previous script to the end of the source code causes an error, because the script can no longer access the variable for the `isUptrend` calculation or the plot() call:

//@version=6 indicator("Inaccessible variable demo", overlay = true) const string PLOT_TITLE = "Median" int lengthInput = input.int(20, "Length", 1) bool isUptrend = ta.valuewhen(median != median[1], median > median[1], 0) color plotColor = if isUptrend color.green else color.red plot(series = median, title = PLOT_TITLE, color = plotColor, linewidth = 3) // Moving this statement to the bottom of the code makes `median` *inaccessible* to all code above. // This change causes an error, because `isUptrend` and the `plot()` call both require the variable's value. float median = ta.median(hl2, length = lengthInput)

> NoteThe for loop structure uses a single-variable declaration in its header to declare a local counter variable. Likewise, the for…in structure can declare a single local variable in its header to store items from a collection. These types of declarations use a different syntax than that of the declarations described above. See the Loops page to learn more.

Tuple declarations

Some conditional structures, loops, and function or method calls return tuples containing *multiple* values or references. To use the data returned from such expressions and structures, programmers must write *tuple declarations*, which are single statements that declare multiple variables using a tuple format.

The syntax for a tuple declaration is as follows:

```
<tuple_of_identifiers> = <function_call>|<structure>
```

Where:

- The `|` character represents *OR*, and all parts enclosed in angle brackets (<>) represent required syntax.
- `tuple_of_identifiers` represents a comma-separated list of variable names enclosed in square brackets (e.g., `[x, y, z]`). The tuple must contain *one* new identifier for *each* returned value or reference.
- = is the assignment operator. The `function_call` or `structure` part to the right of the operator determines the initial data that it assigns to each new variable. `function_call` refers to a call to a built-in function or user-defined function, or method, that returns a tuple. Likewise, `structure` refers to a loop or conditional structure that returns a tuple.

Some built-in functions in the `ta` namespace return a tuple instead of a single value. Therefore, scripts must use tuple declarations to create variables that store the data from calls to those functions. For example, the ta.bb() function returns a tuple containing all *three* values of the Bollinger Bands indicator in the following order: the basis moving average, the upper band, and the lower band. Therefore, a script must use a tuple declaration, such as the following, to declare one new variable for *each* returned value:

// Declares three variables named `bbMiddle`, `bbUpper`, and `bbLower` to hold the values returned by `ta.bb()`. // `bbMiddle` stores the middle band (SMA), `bbUpper` stores the upper band, and `bbLower` stores the lower band. [bbMiddle, bbUpper, bbLower] = ta.bb(close, 5, 4)

> TipFunction signatures displayed by the Pine Editor's autosuggest feature or the Reference Manual list return types in square brackets if the function returns a tuple. For instance, the signature for ta.bb() shows the return type `[series float, series float, series float]`, indicating that a call to the function returns a tuple of three "series float" values.

Programmers often use tuples in user-defined functions and methods to return multiple values for use later in a script's calculations. A user-defined function returns a tuple only if the *final code* in its body is a tuple of expressions.

For example, the code block below defines a `calcWidthAndColor()` function that returns a two-item tuple. The tuple contains a "float" value representing the width between two bands, and a "color" value based on the width value. The code then calls that function using variables from the previous example declaration as arguments, and uses a tuple declaration to declare two new variables to store the returned values:

//@function Calculates the width between two bands, and a gradient color based on the normalized width over a // specified length. calcWidthAndGradient(float upper, float lower, int length, color upperColor, color lowerColor) => float width = upper - lower float normWidth = width / ta.highest(width, length) color gradient = color.from_gradient(normWidth, 0, 1, lowerColor, upperColor) // Return a tuple containing the values of `width` ("series float") and `gradient` ("series color"). [width, gradient] [bbMiddle, bbUpper, bbLower] = ta.bb(close, 5, 4) // Declares two variables named `bandWidth` and `widthColor` to store the values returned by `calcWidthAndGradient()`. // `bandWidth` stores the returned `width` value, and `widthColor` stores the returned `gradient` value. [bandWidth, widthColor] = calcWidthAndGradient(bbUpper, bbLower, 5, color.orange, color.purple)

Note that:

- The `upper`, `lower`, `length`, `upperColor`, and `lowerColor` identifiers in the function definition represent *parameters*, which determine the types of *arguments* that a call to the function requires.
- The function definition uses single-variable declarations in its body to create variables that store the necessary data for the function's calculations. Those variables are available only inside the function definition; a script *cannot* access them in any other scope.

Programmers often use tuple declarations to declare multiple variables that store results returned by conditional structures. Similar to a function, a conditional structure returns a tuple if the final code in *each local block* is a tuple of expressions.

For example, the following code block declares two variables, `lowColor` and `highColor`, to hold "color" values returned by a switch structure based on the value of a string input:

//@variable Holds a string to specify a colorful or grayscale style. string styleInput = input.string("Color", "Style", ["Color", "Grayscale"]) // Declares two variables named `lowColor` and `highColor` to hold the two values returned by the `switch` structure. // The value of `lowColor` is `color.purple` or `#606060`, and the value of `highColor` is `color.orange` or `#b1b1b1`. [lowColor, highColor] = switch styleInput "Color" => [color.purple, color.orange] => [#606060, #b1b1b1]

The following script combines all three examples above to calculate a set of Bollinger Bands, their width, and a gradient color, then plots all the values on the chart:

//@version=6 indicator("Tuple declarations demo") //@variable Holds a string to specify a colorful or grayscale style. string styleInput = input.string("Color", "Style", ["Color", "Grayscale"]) //@function Calculates the width between two bands, and a gradient color based on the normalized width over a // specified length. calcWidthAndGradient(float upper, float lower, int length, color upperColor, color lowerColor) => float width = upper - lower float normWidth = width / ta.highest(width, length) color gradient = color.from_gradient(normWidth, 0, 1, lowerColor, upperColor) // Return a tuple containing the values of `width` ("series float") and `gradient` ("series color"). [width, gradient] // Declares two variables named `lowColor` and `highColor` to hold the two values returned by the `switch` structure. // The value of `lowColor` is `color.purple` or `#606060`, and the value of `highColor` is `color.orange` or `#b1b1b1`. [lowColor, highColor] = switch styleInput "Color" => [color.purple, color.orange] => [#606060, #b1b1b1] // Declares three variables named `bbMiddle`, `bbUpper`, and `bbLower` to hold the values returned by `ta.bb()`. // `bbMiddle` stores the middle band (SMA), `bbUpper` stores the upper band, and `bbLower` stores the lower band. [bbMiddle, bbUpper, bbLower] = ta.bb(close, 5, 3) // Declares two variables named `bandWidth` and `widthColor` to store the values returned by `calcWidthAndGradient()`. // `bandWidth` stores the returned `width` value, and `widthColor` stores the returned `gradient` value. [bandWidth, widthColor] = calcWidthAndGradient(bbUpper, bbLower, 5, highColor, lowColor) // Plot the `bbMiddle`, `bbUpper`, and `bbLower` series on the main pane, using `widthColor` as each plot's color. plot(bbMiddle, "Average", widthColor, 3, force_overlay = true) plot(bbUpper, "Upper band", widthColor, 3, force_overlay = true) plot(bbLower, "Lower band", widthColor, 3, force_overlay = true) // Plot the `bandWidth` series as columns in a separate pane, and color the plot using `widthColor`. plot(bandWidth, "Band width", widthColor, style = plot.style_area)

> NoteThe for…in loop structure can also use a tuple declaration in its header to declare two local variables for tracking items from a collection and their indices. However, the syntax differs from other tuple declarations described above. See the `for...in` loops section of the Loops page to learn more.

Using an underscore as an identifier

Scripts can declare variables using a *single underscore* (_) as the identifier to mark those variables as *unused*. A script *cannot* access data from any variables named _ or use those variables in other expressions or statements after their declaration. Programmers can write any number of _ variable declarations anywhere in a script, including multiple times in the same scope.

This behavior is useful in cases where a function call returns a tuple of multiple values, but the script requires only *some* of those values in its calculations. Rather than specifying unique names for all the unused variables from a tuple declaration, programmers can *discard* those variables by using _ as the name for each one.

For example, the following script calculates the highest and lowest prices across the chart's visible bars. It imports the VisibleChart library from PineCoders and calls the library's `ohlcv()` function to perform the calculation. The call returns a tuple of five values: the visible chart range's open, high, low, close, and cumulative volume. However, our script requires only the high and low. Instead of specifying unique names for all the unused variables, we use _ as each unused variable's identifier:

//@version=6 indicator("Underscores in tuple declarations demo", overlay = true) // Import version 5 of the `VisibleChart` library from PineCoders. import PineCoders/VisibleChart/5 as visChart // Declare a tuple of variables for all values returned by the imported `ohlcv()` function. // This function returns five values in the following order: open, high, low, close, and cumulative volume. // We require only the high and low, so we use `_` to discard the other returned values. [_, visibleHigh, visibleLow, _, _] = visChart.ohlcv() // Plot the values of the `visibleHigh` and `visibleLow` variables on the chart. plot(visibleHigh, "Visible high", color.green, 3) plot(visibleLow, "Visible low", color.red, 3)

Programmers also occasionally use _ when writing a loop whose calculations do not require the variables declared in the loop's header. For example, the script below calculates the sum of 20 pseudorandom values from math.random() calls using a for loop. The calculation does not require the loop's *counter* variable, so we used _ as the variable's name to mark it as unused:

//@version=6 indicator("Underscores for loop variables demo") //@variable Stores a pseudorandom value from a Bates distribution. float sample = 0.0 // Calculate the sum of 20 `math.random()` values in a `for` loop. // The calculation does not require the counter variable from the loop's header, so we set its identifier to `_`. for _ = 1 to 20 sample += math.random() // Divide by 20 to calculate the final sample. sample /= 20 // Plot the resulting value. plot(sample, "Pseudorandom sample")

Note that:

- The += and /= operators in this script *reassign* the value of the `sample` variable after initialization. See the Variable reassignment section to learn more.

Declaring qualified types

Every variable has an assigned type and a type qualifier, which together define the variable's *qualified type*. A variable's type determines *what kind* of data the variable represents in the script's calculations, as well as the types of data that the script can pass to the variable. A variable's qualifier indicates *when* the assigned data is available and whether it can *change* across executions.

> TipProgrammers can inspect a variable's qualified type by hovering over its identifier in the Pine Editor. The editor displays a pop-up window that shows the variable's type information below its defined description.

By default, the Pine Script compiler automatically determines the qualified type of a variable based on its assigned data. However, in single-variable declarations, programmers can override this behavior and specify qualified types directly by prefixing the declared identifiers with type keywords and qualifier keywords.

The following sections explain how these keywords affect declared variables. For detailed information about Pine's types and qualifiers, and how they work, refer to the Type system page.

NoteTuple declarations do not support extra keywords for specifying qualified types. As such, each variable from a tuple declaration automatically inherits the same type as its assigned value or reference, and all the variables inherit the *strongest* type qualifier used by the function call or structure. See the Tuples section of the Type system page to learn more.

Type keywords

A variable declaration that prefixes the variable's identifier with a *type keyword* specifies the type of data that the variable represents in the script's calculations.

Programmers can use any of the following as the type keyword in a single-variable declaration to set the variable's type:

- Built-in type keywords: int, float, bool, color, string, line, linefill, box, polyline, label, table, chart.point, footprint, and volume_row.
- Collection type identifiers, which contain the array, matrix, or map keyword followed by a *type template* (e.g., `array<int>`, `matrix<float>`, `map<string, color>`).
- The names of enum types or user-defined types.

NoteNot all built-in types have corresponding keywords. For example, there are no keywords for the "plot" and "hline" types, or for the unique value types such as "plot_style".

Including a type keyword in a variable declaration is usually *optional*, because the Pine Script compiler can automatically determine a variable's type based on its assigned value or reference. However, a variable declaration *requires* a type keyword if any of the following conditions apply:

- The declaration includes a qualifier keyword.
- The variable is a constant exported by a library.
- The variable's initial value is na (undefined), and the statement does not cast it to a valid type using the available type-casting functions (e.g., int()). See the `na` value section of the Type system page for more information.

TipEven when type keywords are not required, we recommend using them in variable declarations when possible. Type keywords help promote readability, and they help the Pine Editor provide type-specific code suggestions.

If a variable declaration does *not* include a type keyword, the variable automatically inherits the *same type* as the data that the script uses to initialize it.

For example, the script below declares a variable named `myVar` without using a type keyword. It initializes the variable using the result of the expression `last_bar_index - bar_index`, which returns an "int" value. Therefore, the variable automatically inherits the "int" type:

//@version=6 indicator("Type inheritance demo") //@variable Counts the number of bars remaining until the script reaches the latest bar. // The expression returns a "series int" value. Therefore, the variable

automatically inherits the "int" type. // You can hover over the `myVar` identifier to confirm the type. myVar = last_bar_index - bar_index // Plot the value on the chart. plot(myVar, "Bars remaining", color.purple, 3)

Note that:

- The variable's *qualified type* is "series int", because the built-in variables in the expression store "series" values that change from bar to bar. See the Qualifiers section of the Type system page and the Qualifier keywords section below to learn more.

After a variable inherits a type, the script can assign only data of the inherited type or data that Pine Script can cast to that type, because a variable's assigned type *cannot change* after initialization.

For example, the following script attempts to reassign the `myVar` variable using an expression that returns a "float" value after initializing the variable with an "int" value. This script causes a *compilation error*, because it cannot automatically cast a "float" value to the "int" type that the `myVar` variable requires:

//@version=6 indicator("Cannot change an inherited type demo") //@variable The natural logarithm of bars remaining until the script reaches the latest bar. // The expression returns a "series int" value. Therefore, the variable automatically inherits the "int" type. // You can hover over the `myVar` identifier to confirm the type. myVar = last_bar_index - bar_index // This line causes a compilation error. The `myVar` variable already inherited the type "int", so the script cannot // later assign it the "float" value returned by `math.log()`. myVar := nz(math.log(myVar)) // Plot the value on the chart. plot(myVar, "Log of bars remaining", color.purple, 3)

If a variable declaration *does* include a type keyword, the compiler assigns the specified type directly to the variable instead of using the type of the initial value or reference. The script can then assign the variable only data of the specified type, or data that Pine can cast to that type.

For example, if we add the float type keyword to the `myVar` declaration in the previous example, no compilation error occurs. The keyword directly sets that variable's type to "float". Variables of the "float" type can accept "float" or "int" values without errors, because Pine automatically casts "int" values to the "float" type when necessary:

//@version=6 indicator("Explicit typing with a type keyword demo") //@variable The natural logarithm of bars remaining until the script reaches the latest bar. // Although the initial expression returns an "int" value, the `float` keyword directly sets the variable's // type to "float". float myVar = last_bar_index - bar_index // This line does not cause an error, because the expression's returned type ("float") matches the type of the variable. myVar := nz(math.log(myVar)) // Plot the value on the chart. plot(myVar, "Log of bars remaining", color.purple, 3)

Qualifier keywords

A single-variable declaration that includes a *qualifier keyword* (const, simple, or series) before the type keyword specifies the variable's type qualifier. A variable's type qualifier indicates *when* the assigned value must be accessible, and whether the value can *change* during or across script executions. Qualifier keywords are almost always *optional*. The only exception is for a library's exported variables, which require the const keyword in their declarations.

Below, we list how each qualifier keyword affects declared variables of value types:

`const`

The variable has the "const" qualifier. It accepts only a "const" value, which is a compile-time constant that never changes at runtime. Additionally, the keyword *prevents* the script from reassigning the variable. Other code that requires any value of the type specified by the type keyword can use the variable, because the "const" qualifier is the *weakest* in Pine's qualifier hierarchy.

`simple`

The variable has the "simple" qualifier. It accepts a "simple" value, which becomes available at *runtime*, during script executions on the *first bar* of the dataset, and remains *consistent* across all subsequent bars. It can also accept a value with a *weaker* qualifier ("input" or "const"). The script can use the variable in any code that allows "simple" values of the given type, but *not* in any code that requires values with the "input" or "const" qualifiers.

`series`

The variable has the "series" qualifier. It can accept values with *any* type qualifier, because "series" is the *strongest* qualifier in Pine's qualifier hierarchy. The variable's value is available at runtime and *can change* on any bar. The script can use the variable in code that allows "series" values of the given type, but *not* in any code that requires a value with a weaker qualifier.

> NoteIt is possible to use a qualifier keyword when declaring variables of most reference types. However, in such declarations, the keyword **does not** directly define the variable's type qualifier. Instances of these types always have the *"series"* qualifier. Therefore, the variables that store their IDs automatically inherit the "series" qualifier, regardless of any qualifier keyword. See the Type system page to learn more.

If the declaration of a value-type variable does *not* include a qualifier keyword, the compiler automatically assigns the variable the *strongest* type qualifier used by the expressions and structures that determine its value, including those that the script uses to reassign the variable after declaring it.

> NotePine Script does not include a keyword for the "input" qualifier. A variable inherits the "input" qualifier only if its declaration does not use a qualifier keyword and the script uses "input" expressions to determine its value.

For example, the following script calculates and plots the RMA of the close series with a specified length when it runs on a standard chart. It declares multiple variables of value types without using qualifier keywords. Therefore, each variable automatically inherits a qualifier based on its assigned data:

//@version=6 indicator("Qualifier inheritance demo", overlay = true) //@variable Holds a string for use as the `title` argument in `input.int()`. // The assigned literal string has the "const string" qualified type. // Therefore, this variable automatically inherits the "const" qualifier. string INPUT_TITLE = "Length" //@variable Holds an integer for calculating the `length` argument for the `ta.rma()` call. // All `input*()` functions except for `input.source()` return a value qualified as "input". // Therefore, this variable inherits the "input" qualifier. int lengthInput = input.int(10, title = INPUT_TITLE, minval = 1) //@variable Holds the value of `lengthInput` if the chart is a standard type, and 1 otherwise. // The `chart.is_standard` variable has the "simple" qualifier, because it depends on data that does not change // but is available only at runtime. The other parts of the expression have weaker qualifiers. // Therefore, the expression returns a "simple" value, and the variable

inherits the "simple" qualifier. int lengthVal = chart.is_standard ? lengthInput : 1 //@variable Stores the RMA of `close` calculated using `lengthVal` as the `length` argument. // The `close` variable is of the type "series float", and `ta.rma()` always returns a "series" result. // Therefore, this variable inherits the "series" qualifier. float rma = ta.rma(close, length = lengthVal) //@variable Stores a "color" value for the plot. // The variable is initialized using the value of `color.gray`, which is of the type "const color". // However, the variable does **not** inherit the "const" qualifier, because the script *reassigns* the // variable later in an `if` structure with logic that depends on a "series" value. // Therefore, this variable's qualifier is "series". color plotColor = color.gray // If we remove this structure from the code, the `plotColor` variable's qualified type becomes "const color". if ta.change(rma) > 0 plotColor := color.green else plotColor := color.red // Plot the `rma` series. plot(rma, "RMA", plotColor, 3)

Note that:

- If a reassignment or compound assignment operation modifies any variable declared without a qualifier keyword, and the operation depends on a value with a stronger type qualifier than that of the variable's initial value, the variable automatically *inherits* that stronger qualifier. For instance, the `plotColor` variable has the *"series"* qualifier, even though the script initializes it using a "const color" value, because the if structure where the script reassigns the value depends on a *"series bool"* expression (`ta.change(rma) > 0`).

If a value-type variable declaration *does* include a qualifier keyword, the compiler assigns the specified qualifier directly to the variable. The variable can accept a value of the specified type with the given qualifier or a *weaker* one, but it *cannot* accept a value with a *stronger* qualifier.

Below, we modified the previous example from this section to demonstrate how qualifier keywords restrict assigned values. Each declaration after the first includes a qualifier keyword that represents a *weaker* qualifier than that of the variable's assigned value, causing a *compilation error*:

//@version=6 indicator("Invalid qualifier keywords demo", overlay = true) // The `const` keyword sets the variable's qualifier to "const", which matches the qualifier of the assigned value. // Therefore, no error occurs here. const string INPUT_TITLE = "Length" // The `const` keyword causes a compilation error here. A "const" variable cannot accept a value qualified as "input". const int lengthInput = input.int(10, title = INPUT_TITLE, minval = 1) // The `const` keyword also causes an error in this declaration, as a "const" variable cannot hold a "simple" value. const int lengthVal = chart.is_standard ? lengthInput : 1 // Using `simple` in this declaration causes an error, because "series" values cannot be stored by "simple" variables. simple float rma = ta.rma(close, length = lengthVal) // Using `simple` in this declaration causes compilation errors in the `if` structure below, because that structure // depends on a "series" value. simple color plotColor = color.gray // The reassignment operations here attempt to assign a "series" value to a "simple" variable. Such operations are not // allowed. if ta.change(rma) > 0 plotColor := color.green else plotColor := color.red // Plot the `rma` series. plot(rma, "RMA", plotColor, 3)

In addition to restricting when a variable's value must be available and whether it can change, a qualifier keyword restricts *how* the script can use the variable. Scripts can pass a variable only to code that accepts the variable's qualified type, or to code that allows a value of the same type with a *stronger* qualifier. If a script attempts to use the variable in code that requires a value with a *weaker* qualifier, a compilation error occurs.

For example, the script version below uses the simple keyword for the `INPUT_TITLE` declaration. This change causes an error in the input.int() call. The simple keyword sets the `INPUT_TITLE` variable's type to "simple

string", but the `title` parameter of the input.int() function *requires* an argument of the type "const string". The parameter cannot accept "string" arguments with any other type qualifier:

//@version=6 indicator("Invalid argument qualifier demo", overlay = true) // The `simple` keyword explicitly sets the variable's qualifier to "simple". simple string INPUT_TITLE = "Length" // The `input.int()` call causes a compilation error. The `title` parameter requires a "const" argument. It cannot // accept an argument with a stronger qualifier such as "simple". int lengthInput = input.int(10, title = INPUT_TITLE, minval = 1) int lengthVal = chart.is_standard ? lengthInput : 1 float rma = ta.rma(close, length = lengthVal) color plotColor = color.gray if ta.change(rma) > 0 plotColor := color.green else plotColor := color.red // Plot the `rma` series. plot(rma, "RMA", plotColor, 3)

Variable reassignment

In Pine Script, most variables declared by a script are *mutable*, meaning that the script can *change (reassign)* their assigned values or references (IDs) after their declarations. The only exception is for variables that a script declares using the const keyword, because that keyword explicitly *prevents* the script from reassigning those variables.

Scripts can reassign custom variables of most available types by using the reassignment operator (:=). The operator directly *replaces* the variable's assigned value or reference with the one returned by the specified expression or structure.

For example, the following script declares a variable named `myVar` with an initial value of 0. Then, it uses the := operator to reassign the variable a value of 10 and plots the result. The script plots a consistent value of 10, not 0, because the := operation *overwrites* the variable's initial value:

//@version=6 indicator("Variable reassignment demo") //@variable Stores an initial value of 0. int myVar = 0 // This operation changes the variable's value to 10. The previous value of 0 is no longer stored by the variable. myVar := 10 // This call plots a consistent value of 10, not 0. plot(myVar, "Plotted value", color.teal, 4)

Note that scripts cannot reassign variables *before* declaring those variables. Similarly, they cannot reassign *local* variables while executing code in an outer scope or a separate local scope. See the Scopes section below for more information.

For example, a compilation error occurs if we move the := operation *above* the `myVar` declaration in the previous script, because the variable is *not available* at that point in the global scope:

//@version=6 indicator("Invalid reassignment of undeclared variable demo") // This operation causes a compilation error. // The `myVar` identifier does not refer to a valid variable in this part of the code. myVar := 10 // The reassignment operation must occur *after* this declaration. int myVar = 0 plot(myVar, "Plotted value", color.teal, 4)

Scripts can also reassign variables of specific value types by using the compound assignment operators. These operators perform an *arithmetic* operation using the value of a variable and another specified value, and then reassign the result directly to the original variable:

- Addition/concatenation assignment (+=)
- Subtraction assignment (-=)
- Multiplication assignment (*=)

- Division assignment (/=)
- Modulo (remainder) assignment (%=)

> NoteMost compound assignment operators are compatible with variables or fields of only the “int” and “float” types. However, the += operator is also compatible with variables of the “string” type. Additionally, scripts can use the += and -= operators to modify variables that store “plot_display” values from expressions that use the built-in `display.*` constants.

The following example calculates an EMA of the close series with a user-specified length using reassignment and compound assignment operations. It declares a variable named `ema` and initializes it with a value of 0, and then reassigns the variable to store the value of `nz(ema[1], close)`. Afterward, the script uses the *= operator to multiply the variable’s value by the value of `(1.0 - alpha)`, and then calculates the final value by using the += operator to add the result of `alpha * close`:

//@version=6 indicator("Reassigning with compound assignment operators demo", overlay = true) //@variable Stores the length for the smoothing factor of the EMA (`alpha`). int lengthInput = input.int(20, "Length", 1) //@variable The EMA's smoothing factor. float alpha = 2.0 / (lengthInput + 1.0) //@variable Stores an initial value of 0, and is modified through reassignment. float ema = 0.0 // Reassign the `ema` variable the previous bar's `ema` value, or the `close` value if the previous value is `na`. // The variable no longer stores a value of 0. ema := nz(ema[1], close) // Multiply and reassign the `ema` variable's value. // After this operation, the value equals the result of `nz(ema[1], close) * (1.0 - alpha)`. ema *= (1.0 - alpha) // Add and reassign the variable's value. // After this operation, the value equals the result of `nz(ema[1], close) * (1.0 - alpha) + alpha * close`. // This result on the current bar is what the `ema[1]` operation retrieves on the next bar. ema += alpha * close // Plot the final value of the `ema` variable for the current bar. plot(ema, "EMA", color.blue, 3)

Note that:

- This script uses compound assignment operators for demonstration purposes. An equivalent way to calculate the `ema` value with *fewer* lines of code is to use a single := operation to reassign the variable the result of `(1.0 - alpha) * nz(ema[1], close) + alpha * close`.
- Reassigning a variable can affect its type qualifier. For example, although the script initializes the `ema` variable using a “const” value, it also reassigns the variable using *“series”* expressions. Therefore, the variable inherits the “series” qualifier, because that qualifier is *stronger* than “const”.

Variables declared in tuple declarations are also compatible with reassignment or compound assignment operators. For example, the script below uses a tuple declaration to declare three variables that hold the result of a ta.macd() call, and then uses the := operator on the declared `macd` variable to assign it a new value. It plots the value of the variable before *and* after the operation for comparison:

//@version=6 indicator("Reassigning tuple variables demo") //@variable Stores a multiplier to apply to the histogram value while modifying the `macd` variable. float factorInput = input.float(2.0, "Factor") // Declare three variables to store the MACD, signal, and histogram values from `ta.macd()`. [macd, sig, hist] = ta.macd(close, 12, 26, 9) // Plot the initial value of the `macd` variable. The reassignment operation below does not affect this plot. plot(macd, "Initial `macd` value", color.blue, 2) // Reassign the `macd` variable a new value. macd := sig + hist * factorInput // Plot the new value of the `macd` variable. plot(macd, "Modified `macd` value", color.orange, 2)

Note that:

- A reassignment or compound assignment operation does not apply to a variable in any code before that operation in the script. The two plot() calls demonstrate this behavior. Although both calls use the `macd` variable, they show different results because the first call uses the variable's *initial* value, and the second uses the variable's value *after* executing the := operation.

Scopes

The *scope* of a variable refers to the region of a script in which the script can use the declared identifier to access that variable and its data. Every script has one *global* scope and zero or more *local* scopes.

The location of a variable declaration in a source code determines the resulting variable's scope:

- A variable declared *inside* the code block of a conditional structure, a loop, or a user-defined function or method definition belongs to a unique local scope.
- All variables declared outside these structures, as signified by *non-indented* lines of code, belong to the script's global scope.

The global scope is the *outermost* scope; it encloses all parts of the script defined in the source code. Every local scope is an *inner* scope, nested into an outer scope, that encloses only the parts of the script defined within a specific *structure*. In general, declared variables that belong to a given outer scope are *accessible* to the inner scopes defined within that scope, but only if the structures that create those scopes are *below* the variable declarations in the code. However, variables that belong to an inner scope are **not** accessible to any outer scope.

For example, the following script declares a variable named `counter` and increments its assigned value inside the scope of an if structure, then attempts to use that local variable's identifier for the `series` argument of a plot() call in the global scope. This script causes a compilation error, because only the if structure can use the `counter` identifier to access the variable declared within it. In the global scope, the identifier *does not* refer to a valid variable:

//@version=6 indicator("Visibility of inner scopes demo") if close > open //@variable A persistent *local* variable that tracks the number of upward bars. // Only the `if` structure's scope can access this variable. The variable is inaccessible to other scopes. var int counter = 0 // Increment the variable's value by 1 in this scope. counter += 1 // The use of `counter` in this `plot()` call causes a compilation error. // No variable with that name exists in this scope, and the outermost (global) scope cannot access local variables. plot(counter, "Up bar count", color.teal, 3)

By contrast, if we move the `counter` variable declaration *above* the if structure in the previous code, the variable then belongs to the *global scope*. With this change, the plot() call can now use the variable. Additionally, the script can still use the identifier in the if structure's += operation to reassign the variable without causing an error, because a global variable *is accessible* to the local scopes of the structures defined after its declaration:

//@version=6 indicator("Visibility of outer scopes demo") //@variable A persistent *global* variable that tracks the number of upward bars. // The global scope encloses all parts of the script. Therefore, the `if` structure below can access // this variable. var int counter = 0 if close > open // Increment the variable's value by 1 in this scope. This does not cause an error, as `counter` refers to the // global variable declared above. counter += 1 // Plot the global `counter` series. plot(counter, "Up bar count", color.teal, 3)

Note that:

- The script uses the var keyword to enable the `counter` variable and its value to *persist* across bars. To learn more about this keyword, see the Declaration modes section below.

Each variable in a script is a *unique container* that stores a specific value or reference (ID). As such, every variable that belongs to the *same* scope must have a *unique* identifier, because using two variables with identical names in the same scope causes ambiguity. The only exception is for variables that have a single underscore (_) as their identifier, because the identifier makes those variables *unusable*.

However, variables that belong to *different* scopes can have the *same* identifier, even if they differ in their qualified types or declaration modes, because the identifier refers to only one specific variable while the script executes the scope where each variable declaration occurs.

For example, the script below calculates the percentage difference between the total number of rising and falling bars. It declares three variables named `counter` for its calculations. The script declares the first two inside the separate if structures, and the last one *below* those structures in the global scope. Although multiple variables share the `counter` identifier, each exists in a *different* scope, and the identifier refers to only *one* of those variables at a time. Therefore, the script compiles successfully:

//@version=6 indicator("Identical variable names in different scopes demo") //@variable Stores the total number of up bars if `close > open`, and `na` otherwise. int risingCount = if close > close[1] // Declare a local variable named `counter` and increment its value. // This variable is a unique entity that exists only in this `if` structure's scope. No other scopes can access it. var int counter = 0 counter += 1 // Modifies the variable declared on line 8. //@variable Stores the total number of down bars if `close < open`, and `na` otherwise. int fallingCount = if close < close[1] // Declare another variable named `counter` and increment its value. // Although it has the same name as the variable in the `if` block above, it exists only in this structure's scope. var int counter = 0 counter += 1 // Modifies the variable declared on line 15. Does not affect the variable on line 8. // Declare a global variable named `counter` and increment its value. The identifier here does not refer to either of // the local variables above, because this declaration is in the outermost scope. var int counter = 0 counter += 1 // Modifies the variable declared on line 20. Does not affect the ones on lines 8 and 15. //@variable The percentage difference between the total number of up bars and down bars. // The `counter` identifier in the expression refers to the variable declared on line 20. float diff = 100 * (fixnan(risingCount) - fixnan(fallingCount)) / counter // Plot the `diff` series as a color-coded line. plot(diff, "Up bars - down bars %", diff > 0 ? color.teal : color.maroon, 3)

Note that:

- The script declares the global `risingCount` and `fallingCount` variables to store the values returned by the if structures, because the local `counter` variables are not accessible to the expressions outside their local scopes. When either structure executes its scope, it returns the result of its `counter += 1` statement. Otherwise, it returns na.
- If we move the variable declaration on line 20 in this script *above* the two if statements, a *compiler warning* occurs because the local variables named `counter` *shadow* the global variable. See the Shadowing section below to learn more.

Shadowing

Variable shadowing refers to the behavior in which a variable in a specific scope *prevents access* to a variable with the *same name* in an outer scope. If a script declares a variable within an inner scope and

assigns it the same identifier as a variable declared before it in an outer scope, the script **cannot** use the identifier to access the outer variable while executing the rest of the inner scope. In other words, the inner variable *shadows* the outer variable.

In most cases, variable shadowing is *unintentional*. It typically occurs in parts of a script where the programmer intends to reassign a variable instead of creating a new one. Therefore, the compiler displays a *warning* in the Pine Editor to inform the programmer when it detects a local variable that shadows an outer-scope variable.

Consider the following script, which checks for engulfing candlestick patterns on the chart. It declares a global variable named `isEngulf` with an initial conditional value of `true` or `false`. Then, the script uses the `isEngulf` identifier in an if structure to filter the condition using criteria based on inputs, and draws a diamond label if the filtered condition remains true. Lastly, the script uses the identifier in a barcolor() call to highlight bars in yellow or orange if the global variable's value is `true`.

A newcomer to Pine might expect the script to color the same bars for which it also draws a label, and not others. However, the script colors *every* bar where the expression on line 16 returns `true`, and the inputs intended to filter the condition do not affect that output:

//@version=6 indicator("Shadowing demo", overlay = true, max_labels_count = 500) // Declare "input" variables specifying allowed directions, and whether only strong patterns appear in the result. bool bullInput = input.bool(true, "Include bullish patterns") bool bearInput = input.bool(false, "Include bearish patterns") bool showStrongInput = input.bool(true, "Show strong patterns only") // Declare variables to store candle body information for pattern detection. float bodyLow = math.min(close, open) float bodyHigh = math.max(close, open) float bodyDir = math.sign(close - open) float bodyRange = bodyHigh - bodyLow //@variable Holds a "bool" value indicating whether an engulfing pattern is detected on the current bar. bool isEngulf = (bodyDir != bodyDir[1] and bodyLow <= bodyLow[1] and bodyHigh >= bodyHigh[1] and bodyRange > bodyRange[1]) if isEngulf // The following statement *does not* modify the variable declared on line 16. // Instead, it declares a *new local variable* named `isEngulf`, causing a compiler warning. After the // declaration, the local variable *shadows* the global variable with the same name, making that variable // *inaccessible* to all code that follows in the enclosing `if` structure's local block. isEngulf = switch bodyDir 1 => bullInput and (showStrongInput ? bodyHigh >= high[1] : true) -1 => bearInput and (showStrongInput ? bodyLow <= low[1] : true) // `isEngulf` in this nested `if` statement refers to the *local* variable above, **not** the one from line 16. if isEngulf label.new(bar_index, bodyDir == 1 ? low : high, style = label.style_diamond, size = size.small) // This call colors *all* bars with an engulfing pattern, regardless of the specified inputs, // because the `isEngulf` identifier here refers to the *global* variable, and that variable is *not* affected by // the `if` structure's logic. barcolor(isEngulf ? (bodyDir == 1 ? color.yellow : color.orange) : na, title = "Engulfing bar color")

This behavior occurs because the script uses the = operator with the `isEngulf` identifier inside the if structure, then uses the identifier further in the local block to specify the condition that controls the label drawings. That = operation declares a new, *local* variable named `isEngulf`, and the new variable *shadows* the global variable declared on line 16. Consequently, the logic of the structure does not affect the value of the global `isEngulf` variable. The compiler also displays a warning on line 25 in the code, where the local `isEngulf` declaration occurs.

We can align the script's visuals and resolve the compiler warning by replacing the = operator with the reassignment operator (:=) in the if structure. This simple change causes the script to *reassign* the global `isEngulf` variable using the switch statement's result rather than creating a new local variable. Because the

script directly changes the value of the global variable in the if structure and uses that variable to control both the label and the bar color, both outputs now occur on the same recent bars:

//@version=6 indicator("Avoiding shadowing demo", overlay = true, max_labels_count = 500) // Declare "input" variables specifying allowed directions, and whether only strong patterns appear in the result. bool bullInput = input.bool(true, "Include bullish patterns") bool bearInput = input.bool(false, "Include bearish patterns") bool showStrongInput = input.bool(true, "Show strong patterns only") // Declare variables to store candle body information for pattern detection. float bodyLow = math.min(close, open) float bodyHigh = math.max(close, open) float bodyDir = math.sign(close - open) float bodyRange = bodyHigh - bodyLow //@variable Holds a "bool" value indicating whether an engulfing pattern is detected on the current bar. bool isEngulf = (bodyDir != bodyDir[1] and bodyLow <= bodyLow[1] and bodyHigh >= bodyHigh[1] and bodyRange > bodyRange[1]) if isEngulf // This statement uses the `:=` operator instead of the `=` operator. Now, the script directly modifies the // global variable from line 16 instead of creating a new variable that shadows it. isEngulf := switch bodyDir 1 => bullInput and (showStrongInput ? bodyHigh >= high[1] : true) -1 => bearInput and (showStrongInput ? bodyLow <= low[1] : true) // `isEngulf` in this nested statement now refers to the global variable. if isEngulf label.new(bar_index, bodyDir == 1 ? low : high, style = label.style_diamond, size = size.small) // Now that the `if` structure modifies the global `isEngulf` variable, this call colors the same recent bars where // a label drawing occurs. barcolor(isEngulf ? (bodyDir == 1 ? color.yellow : color.orange) : na, title = "Engulfing bar color")

It is also possible for custom variables in a script to shadow some built-in variables. If a script declares a variable with the same identifier as a built-in variable, the identifier refers exclusively to that variable for the remainder of the scope. As with custom variables, shadowing a built-in variable causes a compiler warning.

For example, the script below declares a variable named `close` and assigns it the value of the built-in open variable, then plots the values associated with the two identifiers. Both plots show the *same* values, because the variable declaration makes the built-in close variable *inaccessible* to the script:

//@version=6 indicator("Shadowing a built-in demo", overlay = true) //@variable Stores the value of the `open` variable. float close = open // Plot the values associated with the `close` and `open` identifiers. // Both plots show the *same* value, because the script *cannot access* the built-in `close` variable in this part of // the global scope. plot(close, "`close` value", color.purple, 5) plot(open, "`open` value", color.orange, 2)

However, shadowing a built-in variable is possible only if the script does not use the identifier to represent the built-in anywhere in the code. If a script already uses the built-in variable, creating a custom variable that shadows it causes a *compilation error*. For example:

//@version=6 indicator("Cannot use and shadow a built-in demo", overlay = true) // The `switch` structure for this declaration uses `close` to refer to the *built-in* variable. float ma = switch chart.is_standard => ta.sma(close, 20) => close // This declaration now causes a compilation error. A script cannot use the identifier of a built-in to access that // built-in and then use the identifier for a custom variable later. This applies regardless of the scope where the // script accesses the built-in. float close = open plot(close, "`close` value", color.purple, 5) plot(open, "`open` value", color.orange, 2) plot(ma)

Some variables can also have the same names as *namespaces*. This naming does not typically result in shadowing. For example, a script can name a variable `barstate` and still access variables from the `barstate` namespace. However, if a variable is of a user-defined type (UDT), a compilation error occurs if its name matches a namespace. Such an identifier is *not* allowed for the type because it can cause *obscuring*, where the namespace becomes inaccessible or the use of the name becomes ambiguous. For example:

//@version=6 indicator("Cannot obscure namespaces demo") //@type A custom type with a single "int" field named `tickerid`. type myType int tickerid = 1 // This declaration causes an error. // A UDT variable with the name `syminfo` can obscure the `syminfo` namespace. myType syminfo = myType.new() log.info(str.tostring(syminfo.tickerid))

> TipRegardless of errors or warnings, we recommend declaring variables with names that do not conflict with identifiers for built-ins of any kind or custom variables from outer scopes. In addition to preventing shadowing and obscuring, using unique identifiers often helps promote readability and makes code simpler to maintain.

Declaration modes

A variable's *declaration mode* defines whether and how the variable and its data *persist* across script executions. By default, declared variables *do not* persist beyond a single execution; the script declares and initializes them anew during *every* execution of their scopes.

Programmers can override this behavior and specify an alternative mode in a single-variable declaration by including the var or varip keyword in the statement:

- If the declaration includes the var keyword, the resulting variable persists *across bars* after the first execution of its scope on a bar's closing tick. After initialization on a closing tick, the variable remains initialized and preserves any data changes that occur on the close of each subsequent bar. However, it does *not* preserve any changes that occur on a bar *before* the bar's closing tick.
- If the declaration includes the varip keyword, the resulting variable persists *across every execution*. The variable remains initialized after the *first execution* of its scope, even if that execution occurs *before* the bar's closing tick. After initialization, the variable preserves all changes that occur on *any* execution, including on those for the incoming ticks of open realtime bars.

> TipThe sections below provide detailed explanations of the default, `var`, and `varip` declaration modes. Understanding this information requires some prior knowledge of Pine's Execution model and Type system. Therefore, we recommend reviewing the basics of the execution model, reading about the available types, and then coming back to this part to learn more about each declaration mode.

Default

If a script declares a variable without using the var or varip keyword, it declares and initializes that variable anew during *every* execution of the variable's scope. In other words, the variable *resets* and holds a new value or reference on each new execution, without preserving the data stored during the scope executions on previous bars or ticks.

> NoteTuple declarations do not support extra keywords. Therefore, all variables created by a tuple declaration *always* use the default declaration mode.

The following example demonstrates the default declaration behavior. The script declares a variable named `count` with an initial value of 0, then uses the += operator to increase its value by one and plots the result. Because the variable declaration does not include the var or varip keyword, the script *reinitializes* the variable with a value of 0 on every execution. Therefore, the += operation reassigns a constant value of 1 to the variable across the entire dataset:

//@version=6 indicator("Default declaration mode demo") // This declaration does not use `var` or `varip`. // Therefore, the script reinitializes the variable to 0 on every execution. int count = 0 // Increment the `count` variable by one. Because the variable resets to 0, this operation consistently reassigns it // a value of 1. count += 1 // Plot the variable's final value. plot(count, "Constant value", color.blue, 3)

Although a variable that uses the default declaration mode does not persist across executions, Pine's runtime system *commits (saves)* a script's calculated data from each execution on a bar's *closing tick*, including the data for all the script's variables, to internal time series structures. Scripts can access a variable's *previous* saved values or references (IDs) by using the history-referencing operator or the built-in functions that retrieve history internally.

For example, the script version below retrieves the last saved value of the `count` variable from *one bar back* using the expression `nz(count[1])`, then increments that value by one and reassigns the result to the `count` variable on the current bar using the := operator. The plot now shows a value that *increases* by one on each bar rather than remaining at a constant value, because the final value of the `count` variable on each bar is one greater than the retrieved value for the previous bar:

//@version=6 indicator("Using past values of a variable demo") // This declaration does not use `var` or `varip`. // Therefore, the script reinitializes the variable to 0 on every execution. int count = 0 // Retrieve the value of the `count` variable from the previous bar, or 0 if it is not available, add 1 to that value, // then reassign the result to the `count` variable on the current bar. The script accesses this result with // the `count[1]` operation while executing on the next bar. // Therefore, the current value of the variable is always one greater than the value on the previous bar. count := nz(count[1]) + 1 // Plot the variable's final value. plot(count, "Bar counter", color.blue, 3)

Note that:

- The expression `count[1]` returns na on the *first* bar of the dataset, because there is no previous bar for the script to access at that point. Therefore, we use the nz() function to replace na with 0 in the calculation. See the `na` value section of the Type system page to learn more.
- A simpler way to achieve the same plotted result is to add var to the `counter` variable declaration in the previous example script. See the `var` section to learn more.

> NoteAlthough the [] operator can retrieve reference-type IDs for previous bars, scripts cannot *modify* all types of historical objects. For example, a script cannot modify a collection that it accesses using an ID retrieved for a previous bar. To maintain and update collections across bars, the simplest approach is to declare *persistent* variables to store their IDs. The `var` section below includes a basic example.

`var`

A variable declaration that includes the var keyword creates a variable that persists *across bars*. The variable *remains initialized* after the *first* execution of its scope on a bar's *closing tick*. From that bar onward,

the variable automatically preserves its assigned value or reference until the script explicitly reassigns it.

In the following example, we modified the first example from the Default section above by adding the var keyword to the `count` variable declaration. With this change, the script no longer reinitializes the variable on every bar. Instead, the variable becomes *permanently* initialized as of the closing tick of the dataset's *first bar*. On each subsequent bar, the += operation increases the variable's value by one, and that new value persists into the next execution. The script now plots a value that changes to 1 on the first bar, then to 2 on the second, and so on:

//@version=6 indicator("Persistence across bars demo") //@variable A persistent variable initialized to 0 on the first bar, and then modified on each bar. // The script does not reinitialize this variable after the first bar. var int count = 0 // Increment the `count` variable by one. Because the `count` variable persists, it preserves the result of this // operation on the close of each bar. Therefore, on each bar, the variable's current value is one greater than the // value on the previous bar. count += 1 // Plot the variable's final value. plot(count, "Bar counter", color.teal, 3)

Scripts can use the var keyword to declare persistent variables of most available types, including reference types. If a variable declared with var stores the reference (ID) of an *object*, such as a collection, changes to that object's saved data also persist across bars.

For example, the script below declares a variable named `myArray` using var and initializes it with the ID of an empty array created from a call to `array.new<float>()`. Then, it uses the variable in a call to array.push() to add a *new element* to the array once every five bars, and plots the array's size on the chart. The plotted size increases by one on every fifth bar without resetting to zero, because assigning an array's ID to a var variable causes that array to persist while the variable continues to reference it:

//@version=6 indicator("Persistent collection demo") //@variable A persistent variable that stores the ID of an array created on the first bar. var array<float> myArray = array.new<float>() // Push the current `close` value into the end of the array once every five bars. if bar_index % 5 == 0 array.push(myArray, close) // Plot the size of the array referenced by `myArray`. plot(array.size(myArray), "Persistent array's size", linewidth = 3)

> Note
>
> The data associated with a variable declared using var can change during executions that occur *before* a bar's closing tick, including those on the ticks of an open *realtime bar*. However, such changes are **temporary**. Pine's *rollback* process *reverts* the variable and its data to their last *confirmed* states, as of the previous bar's close, before executing on the current bar again. Only the *final* changes to the variable's data on the bar's *closing tick* persist across subsequent bars. See the Realtime bars section of the Execution model page to learn more.
>
> To create a variable that preserves changes across *every* tick, not just every closing tick, use the varip keyword instead of var in the declaration. See the `varip` section below for more information.

The var keyword is often helpful when working with instances of drawing types, such as lines. Drawing objects automatically persist across bars until deleted by the runtime system or calls to the built-in `*.delete()` functions, even if a script does not assign their IDs to variables. However, using var variables to

directly store drawing IDs, or the data that the drawings require, often makes them simpler to manage across bars. Additionally, it helps promote runtime efficiency.

For example, the script below draws a line from the open to the close of each daily period on the chart. It uses ta.valuewhen() calls to calculate the opening time and price values for the current period, and assigns those values to variables. On historical bars, the script creates a new line using line.new() and initializes a `currLine` variable with the returned ID when a new period starts. On realtime bars where the current period is open, the script retrieves the last line ID saved by the variable (`currLine[1]`), deletes the referenced line with a call to line.delete(), and then creates a new line to follow the latest price.

This code is not the most efficient way to achieve the intended result, because it uses ta.valuewhen() to calculate values that the script does *not* require on every bar, and it deletes and redraws lines on the last bar rather than using `line.set*()` functions to *modify* the latest line:

//@version=6 indicator("Inefficient line management demo", overlay = true) //@variable Holds `true` on the first bar in a "1D" period, and `false` on all other bars. bool newPeriod = timeframe.change("1D") // Retrieve the `time` and `open` values from the last bar where the `newPeriod` value was `true`, // and assign the results to variables. int openTime = ta.valuewhen(newPeriod, time, 0) float openPrice = ta.valuewhen(newPeriod, open, 0) // Declare a variable to reference the latest line. line currLine = na if barstate.islast // Create a new `line` object and assign its ID to the `currLine` variable. currLine := line.new(openTime, openPrice, time, close, xloc.bar_time) // Delete the line drawn on the previous bar if the `newPeriod` value is `false`. if not newPeriod line.delete(currLine[1]) // On historical bars where a new period starts, draw a line connecting the period's final values. else if newPeriod currLine := line.new(openTime[1], openPrice[1], time[1], close[1], xloc.bar_time)

The following script version demonstrates a simpler and more efficient way to achieve the same result. It uses the var keyword in the `currLine` variable declaration to initialize the variable on only the first bar. On historical bars where a new period starts, the script calls line.set_xy2() to update the end coordinates of the current line referenced by the `currLine` variable, and then creates a new line for the current bar and reassigns the variable to store that line's ID. On the latest bar where the current period is open, the script passes the variable to a line.set_xy2() call to update the current line instead of deleting that line and creating a new one:

//@version=6 indicator("Efficient line management demo", overlay = true) //@variable Holds `true` on the first bar in a "1D" period, and `false` on all other bars. bool newPeriod = timeframe.change("1D") // Declare a variable that persistently stores a `line` ID or `na` across bars until reassigned. var line currLine = na if barstate.islast // At the start of a new period, create a new `line` object with coordinates for the current bar, and reassign // the `currLine` variable. The variable stores the new `line` ID until the `newPeriod` value is `true` again. if newPeriod currLine := line.new(time, open, time, close, xloc.bar_time, color = color.purple) // Set the `x2` and `y2` (end) coordinates of the current line to the current bar's `time` and `close` values // while the period is open. currLine.set_xy2(time, close) else if newPeriod // Update the end coordinates of the latest line on historical bars to the final value of the previous period. currLine.set_xy2(time[1], close[1]) // Create a new `line` object and assign its ID to the `currLine` variable. On the next historical bar where // a new period starts, the script modifies the new line. currLine := line.new(time, open, time, close, xloc.bar_time, color = color.purple)

Note that:

- Programmers can observe the performance difference between these scripts by analyzing them with the Pine Profiler on the historical and realtime bars of an intraday chart.

- It is possible to search the built-in line.all array to access the last drawn line instead of using a persistent variable. However, that approach requires checking the array's size with the array.size() function and using the array.get() or array.last() function to retrieve the latest line ID. These extra steps require *more* resources than maintaining a persistent ID with a var variable and updating the referenced line's data across specific bars.

Scripts can use the var keyword to declare variables in global or local scopes. If a var variable declaration is in a local scope, such as within a conditional structure, that variable persists across each execution of the scope and preserves changes that occur on a bar's closing tick. The variable does not reset to its initial state on bars where the scope does not execute.

For example, the script below declares a persistent local variable named `localVar` with an initial value of -1 inside the scope of an if structure. The structure's local code executes once every 10 bars. After initializing the persistent variable, the structure uses the *= operator to multiply the variable's current value by -1 and reassign it. The script assigns the local variable's value to a variable in the global scope named `globalVar` and plots the result on the chart. Because the `localVar` variable persists without resetting to its initial value, the plotted results on each 10th bar alternate between -1 and 1:

//@version=6 indicator("Persistent local variable demo") //@variable Stores the value of the `localVar` variable on each 10th bar, and `na` on other bars. int globalVar = if bar_index % 10 == 0 //@variable A persistent local variable initialized to -1 and then modified across bars. // This variable persists across bars after initialization, even when the scope does not execute. var int localVar = -1 // Multiply the local variable's value by -1 and reassign it using the result. // If the current value is -1, it changes to 1. If 1, it changes to -1. // The structure returns the result of this operation only on bars where the scope executes. localVar *= -1 // Plot the value of the `globalVar` variable. plot(globalVar, "Alternating value from local scope", color.teal, 3)

Note that:

- The if structure returns na on each bar where the local scope does not execute. Therefore, the `globalVar` variable stores a value other than na only on each 10th bar.
- If we remove var from the `localVar` variable declaration, causing it to use the default declaration mode, the script plots a consistent value of 1 on each 10th bar, and na on other bars. The result changes because each execution of the scope reinitializes the value to -1, and multiplying that value by -1 results in a value of 1.

It's important to note that local variables declared using var inside loops behave very differently from those declared using the default declaration mode. If a local variable in a loop uses the default mode, the script reinitializes it on *every iteration*. By contrast, if the local declaration uses var, the variable remains initialized after the *first* loop iteration on a bar's closing tick. From that point onward, it persists and preserves changes to its data from each iteration on the closing ticks of subsequent bars.

For example, the following script declares two local variables inside the body of a for loop that performs five iterations. The `local1` variable declaration uses the default declaration mode, and the `local2` declaration uses the var keyword. The loop reassigns a value of 0 to the `local2` variable on the first iteration, then increments the values of both variables by one on every iteration. The loop *returns* a tuple containing both values when it ends, and the script uses a tuple declaration to create global variables that store the returned values for its plots:

//@version=6 indicator("Persistent loop variable demo") // Declare two global variables to store the values of the `local1` and `local2` variables returned // after the loop's final iteration. [global1, global2] = for i = 1 to 5

//@variable A local variable declared using the default mode. // The script reinitializes this variable to 0 on *every* loop iteration. int local1 = 0 //@variable A local variable declared using `varip`. // This variable remains initialized after the first loop iteration. var int local2 = 0 // On the first iteration, reset the `local2` variable's value to 0. Without this statement, the value would // continue to increase across bars. if i == 1 local2 := 0 // Because the `local1` variable consistently resets to 0 before this operation, // its final value is 1 on each iteration. local1 += 1 // By contrast, the `local2` variable does not reset to its previous state on each iteration. The operation on the // first iteration changes the value to 1, the operation on the second changes the value to 2, and so on. local2 += 1 // Return both variables' values in a tuple for plotting. [local1, local2] // Plot the values saved to `global1` and `global2`, which equal those of `local1` and `local2`, on the chart. plot(global1, "Non-persistent", color.blue, 3) plot(global2, "Persistent", color.orange, 3)

As shown below, the final value of the `local1` variable on each bar is 1, whereas the value of the `local2` variable is 5. This difference occurs because the script reinitializes the `local1` variable to hold 0 on every loop iteration, so the += operation on that variable consistently sets the value to 1. In contrast, the `local2` variable remains initialized after the first loop iteration. Therefore, the += operation on that variable consistently increases the assigned value. The variable stores a value of 1 on the first iteration, 2 on the second iteration, and so on until it reaches the final value of 5 on the last iteration:

Note that:

- As demonstrated by the previous example, even local variables declared with var persist across bars. Therefore, if we remove the if statement that reassigns 0 to the `local2` variable, that variable's value consistently increases by five on each bar.

`varip`

A variable declaration that includes the varip keyword creates a variable that persists across *every tick*. The variable becomes permanently initialized after the *first* execution of its scope, even if that execution occurs *before* a bar's closing tick. From that point onward, all changes to the variable's data persist, even those that occur during script executions on an *open* bar. The "ip" in the keyword stands for *"intrabar persist"*, as the value or reference stored by the variable persists across every update within each bar until the script explicitly reassigns the variable.

The varip keyword is compatible with variables that store only specific types of data, including the following:

- Values of any fundamental type ("int", "float", "bool", "color", or "string").
- Members of enum types.
- IDs of the chart.point, footprint, or volume_row type.
- The IDs for objects of user-defined types (UDTs).

The keyword is also compatible with variables that store the IDs of collections, but only if those collections store the following types of data:

- Values of a fundamental type.
- IDs of the chart.point, footprint, or volume_row type.
- IDs for objects of a user-defined type with fields for storing data of only the above types or the IDs of other collections that contain elements of only these types.

A variable declared with varip typically behaves the same as a variable declared with var on *historical bars* (where the value of the barstate.ishistory variable is `true`), because by default, all scripts execute *once per*

bar on that part of the dataset. However, on realtime bars, which form over time as new ticks become available from the data feed, indicator and library scripts execute *once per tick* instead of once per bar. Variables declared with var and varip typically behave differently on these bars.

As noted in the previous section, if a script modifies a var variable while executing on an open bar, those modifications **do not** persist. Pine's *rollback* process *reverts* the variable to its last confirmed state as of the previous bar's close before the script executes on the bar again. This process ensures that the variable stores only *confirmed* data at the start of each execution, and not any *temporary* data from ticks that arrive before the bar closes.

By contrast, a variable declared with varip is *not* affected by rollback. If a script modifies a variable declared with varip while executing on an open bar, the variable preserves its new value or reference without reverting to a previous state after the execution ends. The variable's new data persists across every subsequent execution on that bar and the bars that follow until the script explicitly changes it again.

> TipFor *advanced* details about the rollback process, refer to the Executions on realtime bars section of the Execution model page.

The following indicator script demonstrates how varip variables behave differently from var variables on realtime bars. The script declares two global variables named `counter1` and `counter2`. The first declaration uses the var keyword, and the second uses varip. On each execution, the script uses the += operator to increment the values of both variables by one, and then plots the resulting values on the chart. The script also colors the background when the value of barstate.isrealtime is `true` to emphasize realtime bars:

//@version=6 indicator("Persistence across ticks demo") //@variable A persistent variable whose value increases by one on each bar. var int counter1 = 0 //@variable A persistent variable whose value increases by one on each execution. varip int counter2 = 0 // Increase the `counter1` variable's value by one on each execution. If the current bar is open, the // system resets the variable to its previous state before the next execution. // Regardless of how many times the script executes on a realtime bar, the variable's final value for that bar is // only one greater than the value on the previous bar. counter1 += 1 // Increase the `counter2` variable's value by one on each execution. Unlike the `counter1` variable, the `counter2` // variable does not reset. If the bar is open, the new value persists into the next execution. // Therefore, if five executions occur on a realtime bar, the variable's final value for that bar is five greater // than the value on the previous bar. counter2 += 1 // Plot the values of the two variables on the chart. Both plots show the same value on historical bars, // but they can differ on realtime bars when the script executes more than once per bar. plot(counter2, "`varip` counter", color.purple, 5) plot(counter1, "`var` counter", color.teal, 2) // Highlight the background of realtime bars for visual reference. bgcolor(barstate.isrealtime ? color.new(color.orange, 80) : na, title = "Realtime bar highlight")

While running on historical bars, the script executes once on each bar's closing tick. Therefore, the values of both variables consistently increase by one on each bar in that part of the dataset, and the plots for the two variables show the same results. Then, when the script reaches realtime bars, the two plots begin to diverge.

The script executes *multiple times* on each realtime bar – once for each new tick – to calculate the bar's results using the latest available data. The += operations on each execution increase the values of both variables by one. However, while the current bar is open, the change to the `counter1` variable *resets* before

each new execution. The variable preserves only the change that occurs on the bar's *closing tick*. Therefore, the variable's final value increases by only one on each realtime bar, just like it does on historical bars.

By contrast, the `counter2` variable, declared using varip, does *not* revert to a previous state on any execution. With each new tick in an open realtime bar, the += operation increases the variable's value by one, and the new value for the variable persists into the execution on the next tick. Therefore, the variable's final value for each realtime bar increases by the number of ticks that are available for that bar:

When using the varip keyword to declare variables that access *objects* of built-in reference types, including chart points or collections of value types, changes to the values stored by those objects also persist across each tick without resetting to a previous state.

For example, the script below uses varip to declare a variable named `testPoint` that stores a persistent reference to a chart.point object. Then, it uses the += operator to increase the value of the object's `price` field by one on each execution and plots the field's final value for each bar. The plot increments by one across all historical bars, where the script executes only once per bar. On realtime bars, the plot increments by the number of ticks available for each bar, because the chart point's `price` field does *not* reset to a previous state after each += operation while a bar is open:

//@version=6 indicator("Persistent built-in object demo") //@variable Stores a persistent reference to a `chart.point` object. The object's fields persist across ticks. varip chart.point testPoint = chart.point.now(0) // Increment the `price` field of the persistent chart point. testPoint.price += 1 // Plot the field's value on the chart. plot(testPoint.price, "Persistent `price` field value", linewidth = 3) // Highlight the background of realtime bars for visual reference. bgcolor(barstate.isrealtime ? color.new(color.orange, 80) : na, title = "Realtime bar highlight")

Note that:

- The same persistent behavior applies to built-in objects whose IDs are stored in collections referenced by varip variables. For example, if a script declares a varip variable that references an array of chart.point IDs, changes to the chart points referenced by the array, and their fields, persist across ticks.

> NoteIn contrast to objects of *built-in* reference types, objects of user-defined types **do not** automatically apply varip behaviors to their *fields* when referenced by variables declared using the varip keyword. To enable these behaviors for the fields of a UDT, prefix the identifier of each field with the varip keyword in the UDT declaration. See the Objects page for an example.

It's crucial to note that strategies execute *differently* from indicators. By default, a strategy executes strictly *once per bar*, even on realtime bars. Therefore, varip variables in a strategy behave the same as var variables by default. However, users can change a strategy's calculation behavior to enable additional executions on each realtime tick, each historical tick, or after order fills. These settings can cause a strategy's varip variables to behave differently on both realtime and historical bars.

For example, the simple strategy below alternates between creating a long and short market order on each execution. It also declares two persistent variables named `counter1` and `counter2` and increments their values by one with the += operator. The first declaration uses var, and the second uses varip. The script also colors the background of all realtime bars for visual reference:

```
//@version=6 strategy("`varip` vs. `var` in strategies demo") // This logic creates a new market order on each execution for demonstration purposes. if strategy.position_size <= 0 strategy.entry("Long", strategy.long) else strategy.entry("Short", strategy.short) //@variable A persistent variable whose value increases by one on each bar. var int counter1 = 0 //@variable A persistent variable whose value increases by one on each execution. varip int counter2 = 0 // The result of this operation does not vary with the strategy's calculation behavior. // The variable's value consistently increases by one on each bar. counter1 += 1 // By contrast, this operation's result does depend on the specified calculation behavior: // - If the default behavior is used, the value increases by one on each bar, just like the value of `counter1`. // - If recalculation on each tick is enabled, the value can increase by more than one on each realtime bar. // - If recalculation after order fills is enabled, the value increases by four on each historical bar by default, // and by the number of new ticks on each realtime bar. counter2 += 1 // Plot the values of the `counter1` and `counter2` variables for comparison. plot(counter2, "`varip` counter", color.purple, 5) plot(counter1, "`var` counter", color.teal, 2) // Highlight the background of realtime bars. bgcolor(barstate.isrealtime ? color.new(color.orange, 80) : na, title = "Realtime bar highlight")
```

If we run the script with the default calculation behavior, the strategy executes only once on every closed bar. On realtime bars, it waits for each bar to close before performing a new execution. As such, the values of both variables consistently increment by the same amount across all bars and do not diverge:

If we select the “On realtime bar tick” checkbox in the strategy’s Script execution settings, the script executes on *each new tick* in a realtime bar, similar to an indicator. With this change, the plot for the `counter2` variable diverges from that of the `counter1` variable on realtime bars:

If we select the “On order fill” checkbox, the script executes again on *any* bar where the broker emulator fills an order. By default, the emulator assumes that the open, high, low, and close of historical bars are all valid ticks for filling orders, and our script creates a new order on every available tick. With this change, in addition to incrementing by the number of ticks on each realtime bar, the value of the `counter2` variable increments by *four* instead of one on each *historical bar* after the first:

For more detailed information about this historical behavior, see the Executions on historical bars section of the Execution model page.

> Notice
>
> Saving data to varip variables is often helpful for various types of calculations that require information from realtime intrabar updates, such as tracking the temporary states of a metric, analyzing value fluctuations within a bar, counting script executions, and more. However, we recommend exercising caution and inspecting your scripts carefully when using the varip keyword in variable declarations, because it can easily cause *repainting*.
>
> After a script reloads across a dataset, all *elapsed realtime bars* from the former script run become *historical bars*, which **do not** contain any of the *temporary* data from past realtime ticks. Therefore, a script that uses a variable declared using varip might yield different results after reloading, especially if the variable’s associated data changes while a realtime bar is open. Depending on how the script uses the variable, this behavior can impact the script’s alerts, strategy reports, visuals, or overall logic. To see this behavior in action, reload any of the above scripts in this section after running them on realtime bars.

> For advanced details about this behavior, as well as the events that cause a script to reload, refer to the Events that trigger script executions section of the Execution model page. For general information about the different types of repainting behaviors in Pine and their causes, refer to the Repainting page.

Overview

The migration guides catalogue all changes introduced between Pine Script® versions and explain how to rewrite code to convert the indicator successfully. Each guide details sequential changes; for example, the To Pine Script version 6 page assumes that the script is already written in Pine v5.

Pine converter

Scripts written in every Pine Script version starting from v3 can be converted to the next version automatically using the converter available in the “Manage Scripts” menu:

A script can be converted only if its code compiles successfully. In rare cases, converting a valid script automatically can result in a script with compilation errors. In that case, resolve the errors using the information in the appropriate article.

To Pine Script® version 2

Pine Script version 2 is fully backwards compatible with version 1. As a result, all v1 scripts can be converted to v2 by adding the `//@version=2` annotation to them.

An example v1 script:

study("Simple Moving Average", shorttitle="SMA") src = close length = input(10) plot(sma(src, length))

The converted v2 script:

//@version=2 study("Simple Moving Average", shorttitle="SMA") src = close length = input(10) plot(sma(src, length))

To Pine Script® version 3

This document helps to migrate Pine Script code from `@version=2` to `@version=3`.

Default behaviour of security function has changed

Let's look at the simple `security` function use case. Add this indicator on an intraday chart:

// Add this indicator on an intraday (e.g., 30 minutes) chart //@version=2 study("My Script", overlay=true) s = security(tickerid, 'D', high, false) plot(s)

This indicator is calculated based on historical data and looks somewhat *into the future*. At the first bar of every session an indicator plots the high price of the entire day. This could be useful in some cases for analysis, but doesn't work for backtesting strategies.

We worked on this and made changes in Pine Script version 3. If this indicator is compiled with `//@version=3` directive, we get a completely different picture:

The old behaviour is still available though. We added a parameter to the `security` function (the fifth one) called `lookahead`.

It can take on the form of two different values: `barmerge.lookahead_off` (and this is the default for Pine Script version 3) or `barmerge.lookahead_on` (which is the default for Pine Script version 2).

Self-referenced variables are removed

Pine Script version 2 pieces of code, containing a self-referencing variable:

//@version=2 //... s = nz(s[1]) + close

Compiling this piece of code with Pine Script version 3 will give you an `Undeclared identifier 's'` error. It should be rewritten as:

//@version=3 //... s = 0.0 s := nz(s[1]) + close

`s` is now a *mutable variable* that is initialized at line 3. At line 3 the initial value gives the Pine Script compiler the information about the variable type. It's a float in this example.

In some cases you may initialize that mutable variable (like `s`) with a `na` value. But in complex cases that won't work.

Forward-referenced variables are removed

//@version=2 //... d = nz(f[1]) e = d + 1 f = e + close

In this example `f` is a forward-referencing variable, because it's referenced at line 3 before it was declared and initialized. In Pine Script version 3 this will give you an error `Undeclared identifier 'f'`. This example should be rewritten in Pine Script version 3 as follows:

//@version=3 //... f = 0.0 d = nz(f[1]) e = d + 1 f := e + close

Resolving a problem with a mutable variable in a security expression

When you migrate script to version 3 it's possible that after removing self-referencing and forward-referencing variables the Pine Script compiler will give you an error:

//@version=3 //... s = 0.0 s := nz(s[1]) + close t = security(tickerid, period, s)

```
Cannot use mutable variable as an argument for security function!
```

This limitation exists since mutable variables were introduced in Pine Script, i.e., in version 2. It can be resolved as before: wrap the code with a mutable variable in a function:

//@version=3 //... calcS() => s = 0.0 s := nz(s[1]) + close t = security(tickerid, period, calcS())

Math operations with booleans are forbidden

In Pine Script v2 there were rules of implicit conversion of booleans into numeric types. In v3 this is forbidden. There is a conversion of numeric types into booleans instead (0 and `na` values are `false`, all the other numbers are `true`). Example (In v2 this code compiles fine):

//@version=2 study("My Script") s = close >= open s1 = close[1] >= open[1] s2 = close[2] >= open[2] sum = s + s1 + s2 col = sum == 1 ? white : sum == 2 ? blue : sum == 3 ? red : na bgcolor(col)

Variables `s`, `s1` and `s2` are of *bool* type. But at line 6 we add three of them and store the result in a variable `sum`. `sum` is a number, since we cannot add booleans. Booleans were implicitly converted to numbers (`true` values to `1.0` and `false` to `0.0`) and then they were added.

This approach leads to unintentional errors in more complicated scripts. That's why we no longer allow implicit conversion of booleans to numbers.

If you try to compile this example as a Pine Script v3 code, you'll get an error: `Cannot call `operator +` with arguments (series__bool, series__bool); <...>` It means that you cannot use the addition operator with boolean values. To make this example work in Pine Script v3 you can do the following:

//@version=3 study("My Script") bton(b) => b ? 1 : 0 s = close >= open s1 = close[1] >= open[1] s2 = close[2] >= open[2] sum = bton(s) + bton(s1) + bton(s2) col = sum == 1 ? white : sum == 2 ? blue : sum == 3 ? red : na bgcolor(col)

Function `bton` (abbreviation of boolean-to-number) explicitly converts any boolean value to a number if you really need this.

To Pine Script® version 4

This is a guide to converting Pine Script code from `@version=3` to `@version=4`.

Converter

The Pine Editor can automatically convert v3 indicators and strategies to v4. The Pine converter is described in the Overview page.

Not all scripts can be automatically converted from v3 to v4. If you want to convert the script manually or if your indicator returns a compilation error after conversion, consult the guide below for more information.

Renaming of built-in constants, variables, and functions

In Pine Script v4 the following built-in constants, variables, and functions were renamed:

- Color constants (e.g `red`) are moved to the `color.*` namespace (e.g. `color.red`).
- The `color` function has been renamed to `color.new`.
- Constants for `input()` types (e.g. `integer`) are moved to the `input.*` namespace (e.g. `input.integer`).
- The plot style constants (e.g. `histogram` style) are moved to the `plot.style_*` namespace (e.g. `plot.style_histogram`).
- Style constants for the `hline` function (e.g. the `dotted` style) are moved to the `hline.style_*` namespace (e.g. `hline.style_dotted`).
- Constants of days of the week (e.g. `sunday`) are moved to the `dayofweek.*` namespace (e.g. `dayofweek.sunday`).
- The variables of the current chart timeframe (e.g. `period`, `isintraday`) are moved to the `timeframe.*` namespace (e.g. `timeframe.period`, `timeframe.isintraday`).
- The `interval` variable was renamed to `timeframe.multiplier`.
- The `ticker` and `tickerid` variables are renamed to `syminfo.ticker` and `syminfo.tickerid` respectively.
- The `n` variable that contains the bar index value has been renamed to `bar_index`.

The reason behind renaming all of the above was to structure the standard language tools and make working with code easier. New names are grouped according to assignments under common prefixes. For example, you will see a list with all available color constants if you type 'color' in the editor and press Ctrl + Space.

Explicit variable type declaration

In Pine Script v4 it's no longer possible to create variables with an unknown data type at the time of their declaration. This was done to avoid a number of issues that arise when the variable type changes after its initialization with the na value. From now on, you need to explicitly specify their type using keywords or type functions (for example, `float`) when declaring variables with the na value:

//@version=4 study("Green Candle Close") // We expect `src` to hold float values, so we declare in with the `float` keyword float src = na if close > open src := close plot(src)

To Pine Script® version 5

Introduction

This guide documents the **changes** made to Pine Script from v4 to v5. It will guide you in the adaptation of existing Pine scripts to Pine Script v5. See our Release notes for a list of the **new** features in Pine Script v5.

The most frequent adaptations required to convert older scripts to v5 are:

- Changing study() for indicator() (the function's signature has not changed).
- Renaming built-in function calls to include their new namespace (e.g., highest() in v4 becomes ta.highest() in v5).
- Restructuring inputs to use the more specialized `input.*()` functions.
- Eliminating uses of the deprecated `transp` parameter by using color.new() to simultaneously define color and transparency for use with the `color` parameter.
- If you used the `resolution` and `resolution_gaps` parameters in v4's study(), they will require changing to `timeframe` and `timeframe_gaps` in v5's indicator().

v4 to v5 converter

The Pine Editor can automatically convert v4 indicators and strategies to v5. The Pine converter is described in the Overview page.

Not all scripts can be automatically converted from v4 to v5. If you want to convert the script manually or if your indicator returns a compilation error after conversion, use the following sections to determine how to complete the conversion. A list of some errors you can encounter during the automatic conversion and how to fix them can be found in the Common script conversion errors section of this guide.

Renamed functions and variables

For clarity and consistency, many built-in functions and variables were renamed in v5. The inclusion of v4 function names in a new namespace is the cause of most changes. For example, the sma() function in v4 is moved to the `ta.` namespace in v5: ta.sma(). Remembering the new namespaces is not necessary; if you type the older name of a function without its namespace in the Editor and press the 'Auto-complete' hotkey (`Ctrl` + `Space`, or `Cmd` on MacOS), a popup showing matching suggestions appears:

Not counting functions moved to new namespaces, only two functions have been renamed:

- `study()` is now indicator().
- `tickerid()` is now ticker.new().

The full list of renamed functions and variables can be found in the All variable, function, and parameter name changes section of this guide.

Renamed function parameters

The parameter names of some built-in functions were changed to improve the nomenclature. This has no bearing on most scripts, but if you used these parameter names when calling functions, they will require adaptation. For example, we have standardized all mentions:

// Valid in v4. Not valid in v5. timev4 = time(resolution = "1D") // Valid in v5. timev5 = time(timeframe = "1D") // Valid in v4 and v5. timeBoth = time("1D")

The full list of renamed function parameters can be found in the All variable, function, and parameter name changes section of this guide.

Removed an `rsi()` overload

In v4, the rsi() function had two different overloads:

- `rsi(series float, simple int)` for the normal RSI calculation, and
- `rsi(series float, series float)` for an overload used in the MFI indicator, which did a calculation equivalent to `100.0 - (100.0 / (1.0 + arg1 / arg2))`.

This caused a single built-in function to behave in two very different ways, and it was difficult to distinguish which one applied because it depended on the type of the second argument. As a result, a number of indicators misused the function and were displaying incorrect results. To avoid this, the second overload was removed in v5.

The ta.rsi() function in v5 only accepts a “simple int” argument for its `length` parameter. If your v4 code used the now deprecated overload of the function with a `float` second argument, you can replace the whole `rsi()` call with the following formula, which is equivalent:

100.0 - (100.0 / (1.0 + arg1 / arg2))

Note that when your v4 code used a “series int” value as the second argument to rsi(), it was automatically cast to “series float” and the second overload of the function was used. While this was syntactically correct, it most probably did **not** yield the result you expected. In v5, ta.rsi() requires a “simple int” for the argument to `length`, which precludes dynamic (or “series”) lengths. The reason for this is that RSI calculations use the ta.rma() moving average, which is similar to ta.ema() in that it relies on a length-dependent recursive process using the values of previous bars. This makes it impossible to achieve correct results with a “series” length that could vary bar to bar.

If your v4 code used a length that was “const int”, “input int” or “simple int”, no changes are required.

Reserved keywords

A number of words are reserved and cannot be used for variable or function names. They are: `catch`, `class`, `do`, `ellipse`, `in`, `is`, `polygon`, `range`, `return`, `struct`, `text`, `throw`, `try`. If your v4 indicator uses any of these, rename your variable or function for the script to work in v5.

Removed `iff()` and `offset()`

The iff() and offset() functions have been removed. Code using the iff() function can be rewritten using the ternary operator:

// iff(<condition>, <return_when_true>, <return_when_false>) // Valid in v4, not valid in v5 barColorIff = iff(close >= open, color.green, color.red) // <condition> ? <return_when_true> : <return_when_false> // Valid in v4 and v5 barColorTernary = close >= open ? color.green : color.red

Note that the ternary operator is evaluated “lazily”; only the required value is calculated (depending on the condition's evaluation to `true` or `false`). This is different from iff(), which always evaluated both values but returned only the relevant one.

Some functions require evaluation on every bar to correctly calculate, so you will need to make special provisions for these by pre-evaluating them before the ternary:

// `iff()` in v4: `highest()` and `lowest()` are calculated on every bar v1 = iff(close > open, highest(10), lowest(10)) plot(v1) // In v5: forced evaluation on every bar prior to the ternary statement. h1 = ta.highest(10) l1 = ta.lowest(10) v1 = close > open ? h1 : l1 plot(v1)

The offset() function was deprecated because the more readable [] operator is equivalent:

// Valid in v4. Not valid in v5. prevClosev4 = offset(close, 1) // Valid in v4 and v5. prevClosev5 = close[1]

Split of `input()` into several functions

The v4 input() function was becoming crowded with a plethora of overloads and parameters. We split its functionality into different functions to clear that space and provide a more robust structure to accommodate the additions planned for inputs. Each new function uses the name of the `input.*` type of the v4 `input()` call it replaces. E.g., there is now a specialized input.float() function replacing the v4 `input(1.0, type = input.float)` call. Note that you can still use `input(1.0)` in v5, but because only input.float() allows for parameters such as `minval`, `maxval`, etc., it is more powerful. Also note that input.int() is the only specialized input function that does not use its equivalent v4 `input.integer` name. The `input.*` constants have been removed because they were used as arguments for the `type` parameter, which was deprecated.

To convert, for example, a v4 script using an input of type `input.symbol`, the input.symbol() function must be used in v5:

// Valid in v4. Not valid in v5. aaplTicker = input("AAPL", type = input.symbol) // Valid in v5 aaplTicker = input.symbol("AAPL")

The input() function persists in v5, but in a simpler form, with less parameters. It has the advantage of automatically detecting input types “bool/color/int/float/string/source” from the argument used for `defval`:

// Valid in v4 and v5. // While "AAPL" is a valid symbol, it is only a string here because `input.symbol()` is not used. tickerString = input("AAPL", title = "Ticker string")

Some function parameters now require built-in arguments

In v4, built-in constants such as `plot.style_area` used as arguments when calling Pine Script functions corresponded to pre-defined values of a specific type. For example, the value of `barmerge.lookahead_on` was `true`, so you could use `true` instead of the named constant when supplying an argument to the `lookahead` parameter in a security() function call. We found this to be a common source of confusion, which caused unsuspecting programmers to produce code yielding unintended results.

In v5, the use of correct built-in named constants as arguments to function parameters requiring them is mandatory:

// Not valid in v5: `true` is used as an argument for `lookahead`. request.security(syminfo.tickerid, "1D", close, lookahead = true) // Valid in v5: uses a named constant instead of `true`.

request.security(syminfo.tickerid, "1D", close, lookahead = barmerge.lookahead_on) // Would compile in v4 because `plot.style_columns` was equal to 5. // Won't compile in v5. a = 2 * plot.style_columns plot(a)

To convert your script from v4 to v5, make sure you use the correct named built-in constants as function arguments.

Deprecated the `transp` parameter

The `transp=` parameter used in the signature of many v4 plotting functions was deprecated because it interfered with RGB functionality. Transparency must now be specified along with the color as an argument to parameters such as `color`, `textcolor`, etc. The color.new() or color.rgb() functions will be needed in those cases to join a color and its transparency.

Note that in v4, the bgcolor() and fill() functions had an optional `transp` parameter that used a default value of 90. This meant that the code below could display Bollinger Bands with a semi-transparent fill between two bands and a semi-transparent backround color where bands cross price, even though no argument is used for the `transp` parameter in its bgcolor() and fill() calls:

//@version=4 study("Bollinger Bands", overlay = true) [middle, upper, lower] = bb(close, 5, 4) plot(middle, color=color.blue) p1PlotID = plot(upper, color=color.green) p2PlotID = plot(lower, color=color.green) crossUp = crossover(high, upper) crossDn = crossunder(low, lower) // Both `fill()` and `bgcolor()` have a default `transp` of 90 fill(p1PlotID, p2PlotID, color = color.green) bgcolor(crossUp ? color.green : crossDn ? color.red : na)

In v5 we need to explictly mention the 90 transparency with the color, yielding:

//@version=5 indicator("Bollinger Bands", overlay = true) [middle, upper, lower] = ta.bb(close, 5, 4) plot(middle, color=color.blue) p1PlotID = plot(upper, color=color.green) p2PlotID = plot(lower, color=color.green) crossUp = ta.crossover(high, upper) crossDn = ta.crossunder(low, lower) var TRANSP = 90 // We use `color.new()` to explicitly pass transparency to both functions fill(p1PlotID, p2PlotID, color = color.new(color.green, TRANSP)) bgcolor(crossUp ? color.new(color.green, TRANSP) : crossDn ? color.new(color.red, TRANSP) : na)

Changed the default session days for `time()` and `time_close()`

The default set of days for `session` strings used in the time() and time_close() functions, and returned by input.session(), has changed from `"23456"` (Monday to Friday) to `"1234567"` (Sunday to Saturday):

// On symbols that are traded during weekends, this will behave differently in v4 and v5. t0 = time("1D", "1000-1200") // v5 equivalent of the behavior of `t0` in v4. t1 = time("1D", "1000-1200:23456") // v5 equivalent of the behavior of `t0` in v5. t2 = time("1D", "1000-1200:1234567")

This change in behavior should not have much impact on scripts running on conventional markets that are closed during weekends. If it is important for you to ensure your session definitions preserve their v4 behavior in v5 code, add `":23456"` to your session strings. See this manual's page on Sessions for more information.

`strategy.exit()` now must do something

Gone are the days when the strategy.exit() function was allowed to loiter. Now it must actually have an effect on the strategy by using at least one of the following parameters: `profit`, `limit`, `loss`, `stop`, or one of the following pairs: `trail_offset` combined with either `trail_price` or `trail_points`. When uses of strategy.exit() not meeting these criteria trigger an error while converting a strategy to v5, you can safely eliminate these lines, as they didn't do anything in your code anyway.

Common script conversion errors

Invalid argument 'style'/'linestyle' in 'plot'/'hline' call

To make this work, you need to change the “int” arguments used for the `style` and `linestyle` arguments in plot() and hline() for built-in constants:

// Will cause an error during conversion plotStyle = input(1) hlineStyle = input(1) plot(close, style = plotStyle) hline(100, linestyle = hlineStyle) // Will work in v5 //@version=5 indicator("") plotStyleInput = input.string("Line", options = ["Line", "Stepline", "Histogram", "Cross", "Area", "Columns", "Circles"]) hlineStyleInput = input.string("Solid", options = ["Solid", "Dashed", "Dotted"]) plotStyle = plotStyleInput == "Line" ? plot.style_line : plotStyleInput == "Stepline" ? plot.style_stepline : plotStyleInput == "Histogram" ? plot.style_histogram : plotStyleInput == "Cross" ? plot.style_cross : plotStyleInput == "Area" ? plot.style_area : plotStyleInput == "Columns" ? plot.style_columns : plot.style_circles hlineStyle = hlineStyleInput == "Solid" ? hline.style_solid : hlineStyleInput == "Dashed" ? hline.style_dashed : hline.style_dotted plot(close, style = plotStyle) hline(100, linestyle = hlineStyle)

See the Some function parameters now require built-in arguments section of this guide for more information.

Undeclared identifier 'input.%input_name%'

To fix this issue, remove the `input.*` constants from your code:

// Will cause an error during conversion _integer = input.integer _bool = input.bool i1 = input(1, "Integer", _integer) i2 = input(true, "Boolean", _bool) // Will work in v5 i1 = input.int(1, "Integer") i2 = input.bool(true, "Boolean")

See the User Manual's page on Inputs, and the Some function parameters now require built-in arguments section of this guide for more information.

Invalid argument 'when' in 'strategy.close' call

This is caused by a confusion between strategy.entry() and strategy.close().

The second parameter of strategy.close() is `when`, which expects a “bool” argument. In v4, it was allowed to use `strategy.long` an argument because it was a “bool”. With v5, however, named built-in constants must be used as arguments, so `strategy.long` is no longer allowed as an argument to the `when` parameter.

The `strategy.close("Short", strategy.long)` call in this code is equivalent to `strategy.close("Short")`, which is what must be used in v5:

// Will cause an error during conversion if (longCondition) strategy.close("Short", strategy.long) strategy.entry("Long", strategy.long) // Will work in v5: if (longCondition) strategy.close("Short") strategy.entry("Long", strategy.long)

See the Some function parameters now require built-in arguments section of this guide for more information.

Cannot call 'input.int' with argument 'minval'='%value%'. An argument of 'literal float' type was used but a 'const int' is expected

In v4, it was possible to pass a “float” argument to `minval` when an “int” value was being input. This is no longer possible in v5; “int” values are required for “int” inputs:

// Works in v4, will break on conversion because minval is a 'float' value int_input = input(1, "Integer", input.integer, minval = 1.0) // Works in v5 int_input = input.int(1, "Integer", minval = 1)

See the User Manual's page on Inputs, and the Some function parameters now require built-in arguments section of this guide for more information.

All variable, function, and parameter name changes

Removed functions and variables

v4	v5
`input.bool` input	Replaced by `input.bool()`
`input.color` input	Replaced by `input.color()`
`input.float` input	Replaced by `input.float()`
`input.integer` input	Replaced by `input.int()`
`input.resolution` input	Replaced by `input.timeframe()`
`input.session` input	Replaced by `input.session()`
`input.source` input	Replaced by `input.source()`
`input.string` input	Replaced by `input.string()`
`input.symbol` input	Replaced by `input.symbol()`
`input.time` input	Replaced by `input.time()`
`iff()`	Use the `?:` operator instead
`offset()`	Use the `[]` operator instead

Renamed functions and parameters

No namespace change

v4	v5
`study(<...>, resolution, resolution_gaps, <...>)`	`indicator(<...>, timeframe, timeframe_gaps, <...>)`
`strategy.entry(long)`	`strategy.entry(direction)`
`strategy.order(long)`	`strategy.order(direction)`
`time(resolution)`	`time(timeframe)`
`time_close(resolution)`	`time_close(timeframe)`
`nz(x, y)`	`nz(source, replacement)`

“ta” namespace for technical analysis functions and variables

Indicator functions and variables

v4	v5
accdist	ta.accdist
alma()	ta.alma()
atr()	ta.atr()
bb()	ta.bb()
bbw()	ta.bbw()
cci()	ta.cci()
cmo()	ta.cmo()
cog()	ta.cog()
dmi()	ta.dmi()
ema()	ta.ema()
hma()	ta.hma()
iii	ta.iii
kc()	ta.kc()
kcw()	ta.kcw()
linreg()	ta.linreg()
macd()	ta.macd()
mfi()	ta.mfi()
mom()	ta.mom()
nvi	ta.nvi
obv	ta.obv
pvi	ta.pvi
pvt	ta.pvt
rma()	ta.rma()
roc()	ta.roc()
rsi(x, y)	ta.rsi(source, length)
sar()	ta.sar()
sma()	ta.sma()
stoch()	ta.stoch()
supertrend()	ta.supertrend()
swma(x)	ta.swma(source)
tr	ta.tr
tr()	ta.tr()
tsi()	ta.tsi()

v4	v5
vwap	ta.vwap
vwap(x)	ta.vwap(source)
vwma()	ta.vwma()
wad	ta.wad
wma()	ta.wma()
wpr()	ta.wpr()
wvad	ta.wvad

Supporting functions

v4	v5
barsince()	ta.barsince()
change()	ta.change()
correlation(source_a, source_b, length)	ta.correlation(source1, source2, length)
cross(x, y)	ta.cross(source1, source2)
crossover(x, y)	ta.crossover(source1, source2)
crossunder(x, y)	ta.crossunder(source1, source2)
cum(x)	ta.cum(source)
dev()	ta.dev()
falling()	ta.falling()
highest()	ta.highest()
highestbars()	ta.highestbars()
lowest()	ta.lowest()
lowestbars()	ta.lowestbars()
median()	ta.median()
mode()	ta.mode()
percentile_linear_interpolation()	ta.percentile_linear_interpolation()
percentile_nearest_rank()	ta.percentile_nearest_rank()
percentrank()	ta.percentrank()
pivothigh()	ta.pivothigh()
pivotlow()	ta.pivotlow()
range()	ta.range()
rising()	ta.rising()
stdev()	ta.stdev()
valuewhen()	ta.valuewhen()
variance()	ta.variance()

“math” namespace for math-related functions and variables

v4	v5
abs(x)	math.abs(number)
acos(x)	math.acos(number)
asin(x)	math.asin(number)
atan(x)	math.atan(number)
avg()	math.avg()
ceil(x)	math.ceil(number)
cos(x)	math.cos(angle)
exp(x)	math.exp(number)
floor(x)	math.floor(number)
log(x)	math.log(number)
log10(x)	math.log10(number)
max()	math.max()
min()	math.min()
pow()	math.pow()
random()	math.random()
round(x, precision)	math.round(number, precision)
round_to_mintick(x)	math.round_to_mintick(number)
sign(x)	math.sign(number)
sin(x)	math.sin(angle)
sqrt(x)	math.sqrt(number)
sum()	math.sum()
tan(x)	math.tan(angle)
todegrees()	math.todegrees()
toradians()	math.toradians()

“request” namespace for functions that request external data

v4	v5
financial()	request.financial()
quandl()	request.quandl()
security(<...>, resolution, <...>)	request.security(<...>, timeframe, <...>)
splits()	request.splits()
dividends()	request.dividends()
earnings()	request.earnings()

“ticker” namespace for functions that help create tickers

v4	v5
`heikinashi()`	`ticker.heikinashi()`
`kagi()`	`ticker.kagi()`
`linebreak()`	`ticker.linebreak()`
`pointfigure()`	`ticker.pointfigure()`
`renko()`	`ticker.renko()`
`tickerid()`	`ticker.new()`

“str” namespace for functions that manipulate strings

v4	v5
`tostring(x, y)`	`str.tostring(value, format)`
`tonumber(x)`	`str.tonumber(string)`

To Pine Script® version 6

Introduction

Pine Script v6 introduces a number of changes and new features. See the Release Notes for a list of all new features.

Some changes are not compatible with v5 scripts. This guide explains how to update your script from v5 to v6. If you want to convert a script from v4 or earlier to v6, refer to the migration guides for previous versions and update the script one version at a time.

The Pine Editor converter can handle many of these changes *automatically*, while other changes might require *manual* fixes.

Here are the changes that affect v5 scripts:

- Values of the “int” and “float” types are no longer implicitly cast to “bool”.
- Boolean values can no longer be na, and the na(), nz(), and fixnan() functions no longer accept “bool” arguments.
- The and and or operators now evaluate conditions lazily.
- All `request.*()` functions can now execute dynamically.
- Division of two “const int” values can now return a fractional value.
- The `when` parameter is removed from all applicable `strategy.*()` functions.
- The default long and short margin percentage for strategies is now 100.
- Strategies now trim the oldest orders in their results instead of raising an error when they exceed the 9000 trade limit.
- The strategy.exit() command no longer ignores relative parameters defining take-profit and stop-loss prices or trailing stop activation levels when the call also includes arguments for the related absolute parameters.
- The history-referencing operator [] can no longer reference the history of literal values or fields of user-defined types directly.
- Function calls can no longer include more than one argument for the same parameter.
- The `offset` parameter of plot() and other functions no longer accepts “series” values.
- na values are no longer allowed in place of built-in constants of unique types.
- The value of timeframe.period now always includes a multiplier (e.g., `"1D"` instead of `"D"`).
- Some `array.*()` functions now accept negative index arguments.
- Some mutable variables are no longer erroneously marked as “const”.
- The `transp` parameter is removed from all applicable functions.
- Some default colors and color constants have updated values.
- The for loop statement now evaluates its end boundary dynamically before every iteration.

Converting v5 to v6 using the Pine Editor

The Pine Editor can automatically convert a v5 script to v6. The Pine Editor highlights the `//@version=5` annotation of a v5 script in yellow.

To convert the script, click the editor’s “Manage script” dropdown menu and select “Convert code to v6”:

A script can be converted only if its v5 code compiles successfully. In rare cases, converting the script automatically can result in a v6 script with compilation errors. In that case, the errors are highlighted in the Editor, and you need to resolve them by using the information in the following sections.

Dynamic requests

In Pine v6, scripts can call all `request.*()` functions *dynamically by default*, allowing any single `request.*()` call instance in the code to request data from different datasets and work within local scopes.

When a script does *not* use dynamic requests, the *context* of a data request (ticker ID and timeframe) must be known on the first script execution and remain *unchanged* across all subsequent executions. Therefore, `symbol`, `timeframe`, and other parameters specifying a non-dynamic `request.*()` call's context require arguments with the "simple" qualifier. Additionally, non-dynamic requests must execute globally, meaning they are not allowed inside the local scopes of loops and other structures.

In contrast, when a script allows dynamic requests, it can call `request.*()` functions with *"series" arguments* for the parameters that define the context of the data requests. With this qualifier change, scripts can:

- Retrieve data from new datasets on *any* historical bar, even *after* the first available bar, with a single `request.*()` call instance.
- Store symbol and timeframe strings in collections or objects, then use the collected values to define a `request.*()` call's context.
- Call `request.*()` functions within the local scopes of loops and conditional structures.
- Export library functions containing `request.*()` calls.

Non-dynamic requests are the default in Pine v5. Scripts coded in v5 can execute dynamic requests, but only if programmers specify `dynamic_requests = true` in the indicator(), strategy(), or library() declaration statement. If not specified, the default argument is `false`.

In Pine v6, dynamic requests are *always* available by default. When a script includes `request.*()` calls, the compiler analyzes the script to determine whether dynamic requests are necessary. If unnecessary for the script, the compiler automatically turns the feature off to optimize performance.

The following v6 example uses a single request.security() instance in a loop to retrieve data for multiple symbols stored in an array. On each iteration, the script dynamically retrieves the close price for one of the stored symbols from the "1D" timeframe and pushes the retrieved value into an array with array.push(). After the loop terminates, the script calculates that array's average using array.avg() and plots the result. In Pine v5, this script would cause a *compilation error* unless we included `dynamic_requests = true` in the declaration statement:

//@version=6 indicator("Dynamic `request` demo") //For v5: must add `dynamic_requests = true` to `indicator()` for this code to work. //@variable User-input toggle to display each symbol's `close` price on chart alongside average `close`. bool showSymbols = input.bool(false, "Plot symbol closes") //@variable Persistent array of "string" symbol ticker IDs to request for our custom index. var array<string> symbols = array.from("NASDAQ:MSFT", "NASDAQ:AAPL", "NASDAQ:GOOGL", "NASDAQ:NVDA") //@variable Array storing the `close` prices for the `symbols` on each bar. array<float> symCloses = array.new<float>() // Loop through `symbols` and request daily `close` prices. for [i, sym] in symbols float reqClose = request.security(sym, "1D", close) symCloses.push(reqClose) // Calculate and plot the average `close` for the `symbols` to create our custom index plot. float avgClose = symCloses.avg() plot(avgClose, "Avg close", avgClose >= avgClose[1] ? color.green : color.red, 3) // Plot each symbol's `close` for reference if

`showSymbols` is `true`. plot(showSymbols ? symCloses.get(0) : na, "MSFT", color.blue) plot(showSymbols ? symCloses.get(1) : na, "AAPL", color.navy) plot(showSymbols ? symCloses.get(2) : na, "GOOGL", color.aqua) plot(showSymbols ? symCloses.get(3) : na, "NVDA", color.teal)

There are minor differences between dynamic and non-dynamic requests in some obscure cases, such as when using the result of one request.security() call in the `expression` argument of another call. As a result, a valid v5 script without dynamic requests can behave differently after conversion to v6, even if nothing related to the requests was changed in the code.

Fix: In Pine v6, the `dynamic_requests` parameter of the indicator(), strategy(), and library() functions is `true` by default. If you find differences in a `request.*()` call's behavior after converting a script to v6, you can include `dynamic_requests = false` in the declaration statement to force dynamic requests off and replicate most of the previous v5 behavior.

It's important to note that, in Pine v5, it is possible to call user-defined functions or methods containing `request.*()` calls inside the local blocks of loops and conditional structures *without* enabling dynamic requests. However, such calls still require *"simple"* arguments for all parameters defining the requested context, which limits their utility.

In Pine v6, calling a `request.*()` function from the scope of a loop or conditional structure is **not allowed** if `dynamic_requests` is set to `false` in the script's declaration statement, even if the call is within a user-defined function. A v6 script that attempts to use wrapped `request.*()` calls in local scopes without dynamic requests enabled causes a *compilation error*.

Fix: If a v5 script specifies `dynamic_requests` is `false` in its declaration statement and uses functions containing `request.*()` calls inside local blocks, remove the explicit `dynamic_requests` argument when converting the script to v6. The converted script will use dynamic requests automatically, allowing the functions containing `request.*()` calls to work correctly inside local scopes.

Types

The following changes have been made to how Pine handles types.

Explicit "bool" casting

In Pine v6, "int" and "float" values are no longer implicitly cast to "bool".

In Pine v5, values of "int" and "float" types can be implicitly cast to "bool" when an expression or function requires a boolean value. In such cases, `na`, `0`, or `0.0` are considered `false`, and *any other value* is considered `true`.

For example, take a look at this conditional expression:

color expr = bar_index ? color.green : color.red

It assigns `color.red` to `expr` on the *first* bar of the chart, because that bar has a `bar_index` of 0, and then assigns `color.green` on *every* following bar, because any *nonzero* value is `true`. The ternary operator ?: expects a "bool" expression for its condition, but in v5 it can also accept a numeric value as its conditional expression, which it automatically converts (implicitly casts) to a "bool".

In v6, scripts must *explicitly* cast a numeric value to "bool" to use it where a "bool" type is required.

Fix: Wrap the numeric value with the bool() function to cast it explicitly.

color expr = bool(bar_index) ? color.green : color.red

Boolean values cannot be `na`

In v6, “bool” values can no longer be `na`. Consequently, the na(), nz(), and fixnan() functions no longer accept “bool” types.

In v5, “bool” variables have *three* possible values: they can be `true`, `false`, or `na`. The boolean `na` value behaves differently from both `true` and `false`:

- When implicitly cast to “bool”, `na` is evaluated as `false`.
- The boolean `na` value is **not** considered equal to `false` when compared using the `==` operator.
- When the boolean `na` value is passed to the na() function, it returns `true`, whereas `na(true)` and `na(false)` both return `false`.

To manage the boolean `na` value, the na(), nz(), and fixnan() functions in v5 have overloads that accept “bool” type arguments. This third boolean state leads to occasional confusion in v5 scripts.

In v6, this is no longer the case: a “bool” must be *either* `true` or `false`, with **no** third state. This means that in v6 scripts:

- A variable declared as “bool” can **no longer** be assigned `na` as its default value.
- In conditional expressions like if and switch, if the return type of the expression is “bool”, any *unspecified* condition returns `false` instead of `na`.
- Expressions that returned a boolean `na` value in v5 now return `false`. For example, using the history-referencing operator `[]` on the very first bar of the dataset to request a historical value of a “bool” variable returned `na` in v5, because no past bars exist, but in Pine v6 it returns `false`.
- Functions that explicitly check whether a value is `na` – specifically, na(), nz(), and fixnan() – do **not** accept “bool” arguments in v6.

This example v5 script creates a simple strategy that switches between long and short positions when two moving averages cross. An if-statement assigns `true` or `false` to a “bool” variable `isLong` to track the trade’s long or short direction, using the strategy’s positive (> 0) or negative (< 0) position size. However, when the position size is zero, *neither* of these conditions are valid. In v5, the undefined condition (== 0) assigns `na` to the variable `isLong`.

Therefore, a boolean `na` value occurs on the first few bars in the dataset before the strategy enters any positions. We can visualize the three “bool” states by setting the background color based on the value of `isLong`:

//@version=5 strategy("Bool `na` demo v5", overlay=true, margin_long=100, margin_short=100) // Strategy's long and short trades are based on moving average cross over/under. longCondition = ta.crossover(ta.sma(close, 14), ta.sma(close, 28)) if (longCondition) strategy.entry("My Long Entry Id", strategy.long) shortCondition = ta.crossunder(ta.sma(close, 14), ta.sma(close, 28)) if (shortCondition) strategy.entry("My Short Entry Id", strategy.short) //@variable Boolean variable that tracks the current

direction of the trade. // Is `true` when `position_size` is greater than 0 (long), and `false` when `position_size` is less than 0 (short). bool isLong = if strategy.position_size > 0 true else if strategy.position_size < 0 false // When `position_size` is equal to 0, neither condition is met. In v5, an undefined condition sets `isLong` to `na`. //@variable Background color, set depending on the state of `isLong` (`true`/`false`/`na`). color stateColor = switch isLong == true => color.new(color.blue, 90) // Blue color if long position. isLong == false => color.new(color.orange, 90) // Orange color if short position. na(isLong) => color.new(color.red, 40) // Red color if no position. Note this line is invalid in v6. bgcolor(stateColor) // On the first bar, display the raw value of `isLong` in a table. if barstate.isfirst var table t = table.new(position. bottom_right, 2, 4, color.yellow, frame_color = color.black, frame_width = 1) t.cell(0, 0, "On first bar") t.cell(0, 1, "`isLong` raw value:", bgcolor = color.new(color.red, 40)) t.cell(1, 1, str.tostring(isLong), bgcolor = color.new(color.red, 40)) // Compare `isLong` value to Boolean `true` and `false` values. t.cell(0, 2, "`isLong` == `true`?") t.cell(1, 2, str.tostring(isLong == true)) t.cell(0, 3, "`isLong` == `false`?") t.cell(1, 3, str.tostring(isLong == false))

Fix: Remove any na(), nz(), and fixnan() functions that run on “bool” values. Ensure that all “bool” values are correctly interpreted as `true` or `false` states **only**. If your code logic requires a third na state to execute as intended, rewrite the code using a different type or structure to achieve the previous three-state behavior.

To adapt our code to Pine v6, we must first remove the following line to resolve the initial compilation error:

na(isLong) => color.new(color.red, 40)

In v6, the undefined condition (`strategy.position_size == 0`) now returns `false` instead of `na`. Consequently, the script *incorrectly* highlights the bars where there are *no* trade positions the same color as those where there are *short* positions, since `isLong` has the same `false` result for both conditions:

We want to distinguish between *three* unique states: long positions, short positions, and no entered positions. Therefore, using a two-state Boolean variable in v6 is no longer suitable. Instead, to maintain our desired behavior, we must *rewrite* the v6 code to replace the “bool” variable with a different type. For example, we can use an “int” variable to represent our three different `position_size` states using -1, 0, and 1:

//@version=6 strategy("Bool `na` demo v6", overlay=true, margin_long=100, margin_short=100) // Strategy's long and short trades are based on moving average cross over/under. longCondition = ta.crossover(ta.sma(close, 14), ta.sma(close, 28)) if (longCondition) strategy.entry("My Long Entry Id", strategy.long) shortCondition = ta.crossunder(ta.sma(close, 14), ta.sma(close, 28)) if (shortCondition) strategy.entry("My Short Entry Id", strategy.short) //@variable Integer variable that tracks the current direction of the trade. // Is `-1` when `position_size` is less than 0 (short), `+1` when `position_size` is greater than 0 (long), // and `0` when `position_size` is equal to 0 (no trades). int tradeDirection = if strategy.position_size < 0 -1 else if strategy.position_size > 0 1 else //strategy.position_size == 0 0 //@variable Background color, set depending on the `tradeDirection`. color directionColor = switch tradeDirection == 1 => color.new(color.blue, 90) // Blue color if long position. tradeDirection == -1 => color.new(color.orange, 90) // Orange color if short position. tradeDirection == 0 => na // No color if no position. bgcolor(directionColor) // On the first bar, display the value of `tradeDirection` in a table for reference. var table t = table.new(position.bottom_right, 2, 3, color.yellow, frame_color = color.black, frame_width = 1) if barstate.isfirst t.cell(0, 0, "On first bar") t.cell(0, 1, "`tradeDirection` value:", bgcolor = color.new(color.green, 60)) t.cell(1, 1, str.tostring(tradeDirection), bgcolor = color.new(color.green, 60)) //@variable A "string" representation of `tradeDirection` value on current bar. string directionString = tradeDirection == 1 ? "Long" : tradeDirection == -1 ? "Short" : "No entered positions" t.cell(0, 2, "State: ")

t.cell(1, 2, directionString) if barstate.islastconfirmedhistory //@variable A "string" representation of `tradeDirection` value on current bar. string directionString = tradeDirection == 1 ? "Long" : tradeDirection == -1 ? "Short" : "No entered positions" label.new(bar_index, high, "On last bar \n `tradeDirection` value: " + str.tostring(tradeDirection) + "\n State: " + directionString)

Unique parameters cannot be `na`

Some Pine Script function parameters expect values of *unique* types. For example, the `style` parameter of the plot() function expects a value of the “input plot_style” qualified type, which must be one of the constants in the `plot.style_*` group.

In v5, passing na to the plot() function’s `style` parameter simply plots a line using the default style `plot.style_line`, without raising an error.

In v6, parameters that expect unique types **no longer** accept na values. Additionally, conditional expressions that return these unique types must be used in a form that **cannot** result in an na value. For example, a switch-statement must have a `default` block, and an if-statement must have an `else`-block, because these conditional expressions can return na otherwise.

The following example script shows two code structures that work in v5 but raise errors in v6.

//@version=5 indicator("`na` and unique types demo v5") //@variable User-selected "string" to determine type of plot used for `plot()` function's `style` argument. string inputStyle = input.string("Area", "Plot style", options = ["Area", "Columns", "Histogram", "Stepline-diamond"]) // Initialize an `input plot_style` type variable based on user's selected `inputStyle`. selectedPlotStyle = switch inputStyle "Area" => plot.style_area "Columns" => plot.style_columns "Histogram" => plot.style_histogram "Stepline-diamond" => plot.style_stepline_diamond // `switch` statement covers all `inputStyle` options, but does not include `default` block. // Valid in v5. Invalid in v6 - `switch` statement must include a `default` block, otherwise raises error. plot(close, "Source plot", color.blue, 2, style = selectedPlotStyle) //@variable Toggle for the style of the line plotted at price "100". inputHundredStyle = input.bool(true, " Use crosses style for '100-line'") hundredLineStyle = if inputHundredStyle plot.style_cross // Since there is no `else` block, setting `inputHundredStyle` to `false` makes this variable `na`. // In v5, passing `na` to the `style` parameter makes the `plot()` function use its default style `plot.style_line`. // In v6, this raises a compilation error because `style` cannot be `na`. // Plot the "100-line" using the `hundredLineStyle` style constant. plot(100, "100-line", color.orange, 4, style = hundredLineStyle)

Fix: Ensure that no na value is passed to parameters that expect unique types, and that all conditional statements return a suitable non-na value.

//@version=6 indicator("`na` and unique types demo v6") //@variable User-selected "string" to determine type of plot used for `plot()` function's `style` argument. string inputStyle = input.string("Area", "Plot style", options = ["Area", "Columns", "Histogram", "Stepline-diamond"]) // Initialize an `input plot_style` type variable based on user's selected `inputStyle`. selectedPlotStyle = switch inputStyle "Area" => plot.style_area "Columns" => plot.style_columns "Histogram" => plot.style_histogram "Stepline-diamond" => plot.style_stepline_diamond // A default block must be included in v6. => plot.style_line plot(close, "Source plot", color.blue, 2, style = selectedPlotStyle) //@variable Toggle for the style of the line plotted at price "100". inputHundredStyle = input.bool(true, " Use crosses style for '100-line'") hundredLineStyle = if inputHundredStyle plot.style_cross else //`else` block must be included in v6. Sets "line" style if `inputHundredStyle` is `false`. plot.style_line // Plot the "100-line" using the `hundredLineStyle` style constant. plot(100, "100-line", color.orange, 4, style = hundredLineStyle)

Constants

The following changes have been made to how Pine handles constant values.

Fractional division of constants

Dividing two integer “const” values can return a fractional value.

In v5, the result of the division of two “int” values is inconsistent. If *both* values are qualified as “const”, the script performs what is known as *integer division*, and discards any fractional remainder in the result, e.g., `5/2 = 2`. However, if *at least one* of the integers is qualified as “input”, “simple”, or “series”, the script *preserves* the fractional remainder in the division result: `5/2 = 2.5`.

//@version=5 indicator("`int` division demo") // `float` division produces fractional remainder in both v5 and v6. plot(5.0 / 2.0, "`float` values", color.blue) // `const int` division produces rounded-down result in v5. In v6, it produces a fractional remainder. plot(5 / 2, "`const int` values", color.orange) plot(int(5) / int(2), "values wrapped `int()`", color.red) // Wrapped `int()` division produces rounded down result in both v5 and v6. plot(int(5 / 2), "result wrapped `int()`", color.green) // Using `input int` type in division preserves the fractional remainder in both v5 and v6. inputNum = input.int(2, "Division int", minval = 1) plot(5 / inputNum, "`input int` value", color.purple)

In v6, dividing two “int” values that are not evenly divisible *always* results in a number with a *fractional value*, regardless of the type and qualifier of the two arguments used. Therefore, the v6 division result is `5/2 = 2.5`, even if both values involved are “const int”.

Fix: If you need an “int” division result *without* a fractional value, wrap the division with the int() function to cast the *result* to “int”, which discards the fractional remainder. Alternatively, use math.round(), math.floor(), or math.ceil() to *round* the division result in a specific direction.

Mutable variables are always “series”

In Pine v5, some mutable variables are qualified as “series” values but are *erroneously* qualified as “const”. This behavior is incorrect and allows a programmer to pass them where “series” variables are usually not accepted.

For example, the ta.ema() function expects its `length` argument to be an integer qualified as “simple” or weaker (see the Qualifiers hierarchy). In the example script below the `seriesLen` variable is effectively a “series” type because its value changes between bars. In v5, `seriesLen` *can* be passed to ta.ema(). Although this does not raise an error, it does not work as expected, because only its *first* recorded value `1` is used as the `length` in the script:

//@version=5 indicator("`const` mutable variables demo") // Variable is effectively of `series int` type. var seriesLen = 0 seriesLen += 1 // `ta.ema()` only uses `length = 1` throughout execution, even as `seriesLen` changes. plot(ta.ema(close, seriesLen))

In v6, `seriesLen` is correctly parsed as a “series int” type, and raises a compilation error if passed in place of the expected “simple int” argument for `length`.

Fix: Pass values of the expected qualified type to built-in functions. In our example, set the `length` argument to a “const int” value.

//@version=6 indicator("`const` mutable variables demo") // Variable is now of `const int` type. var seriesLen = 1 // `ta.ema()` uses `length = 1` throughout execution. plot(ta.ema(close, seriesLen))

Color changes

The color values behind some of the `color.*` constants have changed in Pine v6 to better reflect the TradingView palette:

Constant name	Pine v5 color	Pine v6 color
color.red	#FF5252	#F23645
color.teal	#00897B	#089981
color.yellow	#FFEB3B	#FDD835

Additionally, the default text color for label.new() is now `color.white` in v6 (previously `color.black` in v5) to ensure that the text is more visible against the default `color.blue` label.

//@version=6 indicator("Default colors v6") color defaultColor = switch bar_index == last_bar_index => color.yellow bar_index == last_bar_index - 1 => color.green bar_index == last_bar_index - 2 => color.red => na bgcolor(defaultColor) if barstate.islastconfirmedhistory label.new(bar_index + 2, 0, "Default text color")

Strategies

Removal of `when` parameter

The `when` parameter for order creation functions was deprecated in v5 and is removed in v6. An order is created only if the `when` condition is `true`, which is its default value. This parameter affects the following functions: strategy.entry(), strategy.order(), strategy.exit(), strategy.close(), strategy.close_all(), strategy.cancel(), and strategy.cancel_all().

The following example strategy shows the use of the `when` parameter, and works in v5 but not v6.

//@version=5 strategy("Conditional strategy", overlay=true) longCondition = ta.crossover(ta.sma(close, 14), ta.sma(close, 28)) strategy.entry("My Long Entry Id", strategy.long, when = longCondition) shortCondition = ta.crossunder(ta.sma(close, 14), ta.sma(close, 28)) strategy.entry("My Short Entry Id", strategy.short, when = shortCondition)

Fix: To trigger the order creation conditionally, use if statements instead.

//@version=6 strategy("Conditional strategy", overlay=true) longCondition = ta.crossover(ta.sma(close, 14), ta.sma(close, 28)) if longCondition strategy.entry("My Long Entry Id", strategy.long) shortCondition = ta.crossunder(ta.sma(close, 14), ta.sma(close, 28)) if shortCondition strategy.entry("My Short Entry Id", strategy.short)

Default margin percentage

The default margin percentage for strategies is now 100.

In v5, the default value of the `margin_long` and `margin_short` parameters is 0, which means that the strategy **does not check** its available funds before creating or managing orders. It can create orders that require

more money than is available, and will **not** close short orders even when they lose more money than available to the strategy.

In *Pine v6*, the default margin percentage is 100. The strategy **does not open** entries that require more money than is available, and short orders are *margin called* if too much money is lost.

For example, we can see the difference in strategy behavior by running this simple strategy on the “ARM” symbol’s 4h chart using the v5 and v6 default margin values. When using Pine v5, there are no margin calls:

//@version=5 strategy("My strategy", overlay=true, default_qty_type = strategy.percent_of_equity, default_qty_value=100) // v6 defaults: margin_long=100, margin_short=100 // v5 defaults: margin_long=0, margin_short=0 longCondition = ta.crossover(ta.sma(close, 14), ta.sma(close, 28)) if (longCondition) strategy.entry("My Long Entry Id", strategy.long) shortCondition = ta.crossunder(ta.sma(close, 14), ta.sma(close, 28)) if (shortCondition) strategy.entry("My Short Entry Id", strategy.short)

However, if we adjust this script to `//@version=6` on the same chart, we see that it triggers 14 margin calls because of the new margin percentages:

Fix: To replicate the previous v5 behavior, set the strategy() function’s `margin_short` and `margin_long` arguments to 0.

Excess orders are trimmed

Strategy orders above the 9000 limit are trimmed (removed) in v6.

In v5, outside of Deep Backtesting, when a strategy creates more than 9000 orders, it raises a runtime error and halts any further calculations.

For example, this strategy script places several orders on each bar in the dataset. As a result, it can quickly surpass the 9000 order limit and trigger an error in Pine v5:

//@version=5 strategy("Strategy order limit demo", overlay=true, pyramiding=5) // Place several long orders on every even bar. This reaches the maximum orders limit in v5 and raises a runtime error. if bar_index % 2 == 0 for i = 1 to 5 strategy.entry("Entry " + str.tostring(i), strategy.long, qty = 5) // Place short orders on every odd bar. else strategy.entry("Short", strategy.short, qty = 25)

In v6, when the total number of orders exceeds 9000, the strategy does *not* halt. Instead, the orders are *trimmed* from the beginning until the limit is reached, meaning that the strategy only stores the information for the most recent orders.

Trimmed orders no longer show in the strategy report, and referencing them using the `strategy.closedtrades.*` functions returns na. Use strategy.closedtrades.first_index to get the index of the first *non-trimmed* trade:

//@version=6 strategy("Strategy order limit demo", overlay=true, pyramiding=5) //@variable Count of total orders placed. var int totalOrders = 0 // Place several long orders on every even bar. if bar_index % 2 == 0 for i = 1 to 5 strategy.entry("Entry " + str.tostring(i), strategy.long, qty = 5) totalOrders += 1 // Place short orders on every odd bar. else strategy.entry("Short", strategy.short, qty = 25) totalOrders += 1 // Display total orders and index of first non-trimmed trade in a table cell on last bar. if barstate.islastconfirmedhistory var table t = table.new(position.bottom_right, 1, 3, color.yellow, color.black, 1) // Display total orders and closed trades counts. string ordersText = "Total orders: " + str.tostring(totalOrders, "#,###") + "\n Closed trades: " +

str.tostring(strategy.closedtrades, "#,###") t.cell(0, 0, ordersText, text_halign = text.align_right, text_size = size.large) // Display the first non-trimmed trade index and its entry price. string firstTradeIndex = str.tostring(strategy.closedtrades.first_index, "#,###") string firstTradePrice = str.tostring(strategy.closedtrades.entry_price(strategy.closedtrades.first_index), "$##.##") string firstTradeText = str.format("Index of first non-trimmed trade: {0}\nEntry price of trade #{0}: {1}", firstTradeIndex, firstTradePrice) t.cell(0, 1, firstTradeText, text_halign = text.align_right, text_size = size.large, bgcolor = #61dd5165) // Trying to reference the trimmed trades (e.g., first closed trade) returns `na`. if totalOrders > 9000 string trimmedTradePrice = "Entry price of trade #0: " + str.tostring(strategy.closedtrades.entry_price(0)) t.cell(0, 2, trimmedTradePrice, text_size = size.large, bgcolor = #dd51c665)

`strategy.exit()` evaluates parameter pairs

The strategy.exit() function has three sets of *relative* and *absolute* parameters that define price levels for exit order calculations. The relative parameters `profit`, `loss`, and `trail_points` specify the take-profit and stop-loss levels and trailing stop activation level as *tick distances* from the entry price. In contrast, the absolute parameters `limit`, `stop`, and `trail_price` specify the exit and trail activation *prices* directly.

In Pine v5, a strategy.exit() call containing arguments for both the relative and absolute parameters that define a price level for the same exit order always prioritizes the *absolute* parameter and ignores the relative one. For instance, a call that includes a `limit` and `profit` argument consistently places take-profit orders at the `limit` value. It never places an exit order using the `profit` distance.

In Pine v6, if a strategy.exit() call contains arguments for related absolute and relative parameters, it evaluates *both* specified levels and uses the one that the market price is expected to *trigger first*.

The example below demonstrates how the behavior of this command differs for v5 and v6 scripts. This v5 script creates a long market order and an exit order bracket on each 28th bar. The strategy.exit() call contains arguments for the relative parameters that determine take-profit and stop-loss levels (`profit` and `loss`), and it includes arguments for the absolute parameters (`limit` and `stop`). The `profit` and `loss` arguments are both 0, which would result in consistent exits at the entry price if the command used them. However, the command never uses these values to determine the exit order levels because the `limit` and `stop` parameters *always* take precedence when they have specified values:

//@version=5 strategy("`strategy.exit()` with parameter pairs demo", overlay = true, margin_long = 100, margin_short = 100) //@variable The 14-bar Average True Range. float atr = ta.atr(14) if bar_index % 28 == 0 strategy.entry("Buy", strategy.long) strategy.exit("Exit", "Buy", profit = 0, limit = close + 2.0 * atr, loss = 0, stop = close - 2.0 * atr)

If we convert the script to Pine v6, its behavior changes. Instead of prioritizing the absolute `limit` and `stop` parameters exclusively, the strategy.exit() command always prioritizes the price levels that will trigger exits *first*. In this example, the market price reaches the `limit` or `stop` value *after* the `profit` and `loss` distance of 0 ticks. Consequently, the command ignores the `limit` and `stop` values and places its exit orders at the entry price, which causes the strategy to exit each trade immediately after opening it:

//@version=6 strategy("`strategy.exit()` with parameter pairs demo", overlay = true, margin_long = 100, margin_short = 100) //@variable The 14-bar Average True Range. float atr = ta.atr(14) if bar_index % 28 == 0 strategy.entry("Buy", strategy.long) strategy.exit("Exit", "Buy", profit = 0, limit = close + 2.0 * atr, loss = 0, stop = close - 2.0 * atr)

History-referencing operator

Pine v6 contains several changes to referencing the history of values.

No history for literal values

The history-referencing operator `[]` can no longer be used with literal values or built-in constants.

In v5, the history-referencing operator `[]` can be used with built-in constants, such as `true` and `color.red`, and with *literal*s, which are raw values used directly in a script that are not stored as variables, such as `6` or `"myString"`, etc.

However, referencing the history of a literal is usually redundant, because by definition every literal represents a fixed value. The only exception where the returned historic value may vary is if the historical offset points to a *non-existent* bar, in which case referencing the historic literal value returns na.

//@version=5 indicator("History-referencing on literals demo") // These lines all use history-referencing on literals, which works in v5, but is not really useful to do here. plot(6[1], "6[1]", linewidth = 3) bgcolor(true[10] ? color.orange[3] : na) if barstate.islastconfirmedhistory // Since "string literal" is only defined in the last bar scope, history-referencing here returns `na`. labelText = "string literal"[20] label.new(bar_index - 3, 3, labelText + ", more text", textcolor = color.white, size = size.large) // Label output will only show ", more text" in v5, since `labelText` is `na`. // In v6, using any history-referencing on literals or built-in constants causes an error.

In *Pine v6*, you can **no longer** use the history-referencing operator `[]` on literals or built-in constants. Trying to do so triggers a compilation error.

Fix: Remove any `[]` operators used with literals or constants.

//@version=6 indicator("History-referencing on literals demo") // We no longer use history-referencing on literals in v6. plot(6, "6", linewidth = 3) bgcolor(true ? color.orange : na) if barstate.islastconfirmedhistory labelText = "string literal" label.new(bar_index - 3, 3, labelText + ", more text", textcolor = color.white, size = size.large) // Label output shows "string literal, more text" in v6, since `labelText` is defined without history-referencing anymore.

History of UDT fields

The history-referencing operator `[]` can no longer be used directly on fields of user-defined types.

In v5, you can use the history-referencing operator `[]` on the *fields* from objects of user-defined types. While this does not cause any compilation errors, the behavior itself is erroneous.

For example, the script below draws an arrow label on each bar and displays its percentage increase/decrease. The label style, color, and text are set based on a bar's direction (`close > open`). The script defines a UDT `LblSettings` to initialize an object on each bar that stores these settings. On the last bar, it draws a table cell that displays the arrow direction and percentage difference from 10 bars back. In v5, we could use the history-referencing operator `[]` on the required `LblSettings` fields directly:

//@version=5 indicator("UDT history-referencing demo", overlay = true) //@type A custom type to hold bar's `label` settings based on bar's direction. //Includes bar direction, label style and color, and "string" percentage difference between bar's `open` and `close`. type LblSettings bool isUp = false string lblStyle

color lblColor string diff //@variable A `LblSettings` instance declared on every bar. LblSettings infoObject = LblSettings.new() // Set the `LblSettings` object fields based on current bar's direction and price information. infoObject.isUp := close > open infoObject.lblStyle := infoObject.isUp ? label.style_arrowup : label.style_arrowdown infoObject.lblColor := infoObject.isUp ? color.green : color.red infoObject.diff := str.tostring((close - open) / open * 100, "#.##") + "%" // Display a new `label` on each bar using its `infoObject` settings. label.new(bar_index, high, infoObject.diff, style = infoObject.lblStyle, color = infoObject.lblColor, textcolor = infoObject.lblColor) // Highlight the bar that is 10 bars back from the last bar. bgcolor(bar_index == last_bar_index - 10 ? color.yellow : na) // On last bar, output table cell to display `LblSettings` object's `lblStyle` and `diff` fields from 10 bars back. if barstate.islast var table t = table.new(position.bottom_right, 1, 1, color.yellow) // In v5, you could use history-referencing operator `[]` on UDT fields directly. //@variable Text displayed in table cell. Set based on the `lblStyle` and `diff` fields from 10 bars back. string txt = "10 bars back: Arrow was " + (infoObject.lblStyle[10] == label.style_arrowdown ? "DOWN" : "UP") + " by " + infoObject.diff[10] t.cell(0, 0, txt, text_size = size.large)

In *Pine v6*, you can **no longer** use the history-referencing operator `[]` on the field of a user-defined type directly.

Fix: Use the history-referencing operator on the UDT *object* instead, then retrieve the field of the historic object. To do so, use the syntax `(myObject[10]).field` - ensure the object's historical reference is wrapped in *parentheses*, otherwise it is invalid. Alternatively, assign the UDT *field* to a *variable* first, and then use the history-referencing operator `[]` on the variable to access its historic value.

// Reference history of object, then retrieve field of historic object. [fieldType] historicFieldValue = (myObject[10]).field // Alternative: Assign field to variable, then reference history of variable to get historic field value. [fieldType] newVariable = myObject.field [fieldType] historicFieldValue = newVariable[10]

Therefore, we can adjust the v5 code to access a historic instance of our `infoObject` on the last bar, wrapped in parentheses. Then, we retrieve our desired field values from the historic object `(infoObject[10])` to display the arrow direction and percentage difference from 10 bars back:

//@version=6 indicator("UDT history-referencing demo", overlay = true) //@type A custom type to hold bar's `label` settings based on bar's direction. //Includes bar direction, label style and color, and "string" percentage difference between bar's `open` and `close`. type LblSettings bool isUp = false string lblStyle color lblColor string diff //@variable A `LblSettings` instance declared on every bar. LblSettings infoObject = LblSettings.new() // Set the `LblSettings` object fields based on current bar's direction and price information. infoObject.isUp := close > open infoObject.lblStyle := infoObject.isUp ? label.style_arrowup : label.style_arrowdown infoObject.lblColor := infoObject.isUp ? color.green : color.red infoObject.diff := str.tostring((close - open) / open * 100, "#.##") + "%" // Display a new `label` on each bar using its `infoObject` settings. label.new(bar_index, high, infoObject.diff, style = infoObject.lblStyle, color = infoObject.lblColor, textcolor = infoObject.lblColor) // Highlight the bar that is 10 bars back from the last bar. bgcolor(bar_index == last_bar_index - 10 ? color.yellow : na) // On last bar, output table cell to display `LblSettings` object's `lblStyle` and `diff` fields from 10 bars back. if barstate.islast var table t = table.new(position.bottom_right, 1, 1, color.yellow) // In v6, cannot use `[]` on UDT fields (e.g., `infoObject.lblStyle[10]` is invalid). // Instead, Use `[]` to reference UDT object's history, wrapped in parentheses, then retrieve its fields. //@variable The `lblStyle` field value from 10 bars back. Is either `label.style_arrowdown` or `label.style_arrowup`. string historicArrowStyle = (infoObject[10]).lblStyle //@variable The `diff` field value (percentage difference between `close` and `open`) from 10 bars back. string historicPercentDifference = (infoObject[10]).diff //@variable Text displayed in table cell. Set based on the `lblStyle` and `diff` fields from 10 bars back. string txt = "10 bars back: Arrow was " + (historicArrowStyle

== label.style_arrowdown ? "DOWN" : "UP") + " by " + historicPercentDifference t.cell(0, 0, txt, text_size = size.large)

Timeframes must include a multiplier

The timeframe.period variable holds a “string” that represents the chart’s timeframe, typically consisting of a *quantity* (multiplier) and *unit*.

In v5, the timeframe.period variable does *not* include a quantity when the chart timeframe has a multiplier of `1`. Instead, the string consists of only the timeframe unit, e.g., `"D"`, `"W"`, `"M"`. This is inconsistent with the timeframe strings for these same units at higher intervals, e.g., `"2D"`, `"3M"`.

To simplify the timeframe format in v6, the timeframe.period variable now *always* includes a multiplier with its timeframe unit. So, `"D"` becomes `"1D"`, `"W"` becomes `"1W"`, and `"M"` becomes `"1M"`.

This change might affect the behavior of older scripts that used `==` to compare the value of timeframe.period with the “string” representation of a timeframe directly (e.g., `timeframe.period == "D"`).

To show the difference between the v5 and v6 timeframe.period variables, we ran the script below on a daily chart (1D) for each Pine version. The script displays the timeframe.period string in a table, and compares the variable’s value with the “string” literals `"D"` and `"1D"`:

//@version=6 indicator("`timeframe.period` multiplier - v6") //@function Compares `timeframe.period` to passed `timeframeString` and outputs result in selected table cell. compareTF(string timeframeString, table t, int row) => bool tfComparison = timeframe.period == timeframeString // Format table cell text and determine cell color based on comparison result. string displayText = "`timeframe.period` == '" + timeframeString + "'? " + str.tostring(tfComparison) color cellColor = tfComparison ? color.rgb(76, 175, 79, 40) : color.rgb(255, 82, 82, 40) // Display `tfComparison` result. t.cell(0, row, displayText, bgcolor = cellColor, text_halign = text.align_left, text_size = size.large) // Display the chart's timeframe information in a table. if barstate.islastconfirmedhistory //@variable Table displaying chart timeframe information. var table t = table.new(position.middle_center, 1, 3, #FFEB3B99, border_color = color.black, border_width = 1) //@variable The text to display in the table, consisting of the chart timeframe and multiplier. string tfInfo = "Chart timeframe: " + timeframe.period + "\n Chart multiplier: " + str.tostring(timeframe.multiplier) t.cell(0, 0, tfInfo, text_halign = text.align_left, text_size = size.large) // Compare the current chart timeframe (daily chart) to timeframe strings with and without a multiplier. compareTF("D", t, 1) compareTF("1D", t, 2)

Fix: In general, ensure that all timeframe strings include a multiplier. In this example, change the timeframe comparison “string” (`timeframe.period == "D"`) to ensure the “string” literal includes a multiplier (`timeframe.period == "1D"`).

Lazy evaluation of conditions

The `and` and `or` conditions are now evaluated *lazily* rather than *strictly*.

An `and` condition is `true` if *all* of its arguments are `true`, which means that if the *first* argument is `false`, we can deduce that the whole condition is `false`, regardless of the value of the second argument. Conversely, an `or` condition is `true` when *at least one* of the arguments is `true`, so if the *first* argument is already `true`, then the whole condition is `true`, regardless of the second argument’s state.

Pine v5 evaluates all bool expressions except for the `?:` ternary operator *strictly*, meaning the *second* part of a conditional expression is *always* evaluated, regardless of the value of the first argument.

Lazy evaluation can have consequences for script calculation. In the example below, we assign a value of `true` to the `signal` variable *only* when `close > open` *and* `ta.rsi(close, 14) > 50`. The ta.rsi() function must be executed on every bar in order to calculate its result correctly. In v5, the function *is* called on every bar, even when `close > open` is *not* `true`, due to the strict bool evaluation, and therefore the function calculates correctly.

//@version=5 indicator("Evaluation test v5") //@variable A signal flag. Is `true` if two conditions `close > open` and `ta.rsi(close, 14) > 50` are both `true`. bool signal = false if close > open and ta.rsi(close, 14) > 50 signal := true // Highlight background on bars where `signal` is `true`. bgcolor(signal ? color.new(color.green, 90) : na)

In v6, bool expressions are evaluated *lazily*, which means the expression *stops evaluating* once it determines the overall condition's result, even if there are other arguments remaining in the expression.

If we convert the script above to v6, we see that the plotted signals *differ* between the two scripts. This variation occurs because of the lazy bool evaluation – since an `and` condition is only `true` if *all* its arguments are `true`, when `close > open` is `false`, the `and` condition is *definitely* `false` regardless of the second argument `ta.rsi(close, 14) > 50`. Consequently, the ta.rsi() call is *not* evaluated on every bar, which interferes with the internal history that the RSI function stores for its calculation and results in incorrect values:

Fix: Ensure that the script evaluates all functions that rely on previous values on each bar. For example, extract calls that rely on historical context to the *global scope* and assign them to a variable. Then, reference that *variable* in the `and` and `or` conditions.

Note that you can and should take advantage of the lazy bool evaluation to create smarter, more concise code.

For example, the script below calls array.first() on an array that is occasionally empty (on bars where `close > open` is `false`). In *Pine v5*, calling array.first() on an empty array results in a runtime error, so you must keep the two if-conditions that check the array size and first element separated in *different scopes* to avoid the error. However, in *Pine v6*, you can have the two conditions in the *same scope* without error because the `and` condition's lazy evaluation ensures that array.first() will only be called if `array.size() != 0` is `true` first:

//@version=5 indicator("Lazy evaluation error showcase") array<bool> myArray = array.new<bool>() if close > open myArray.push(true) // Causes a runtime error in v5 when trying to call `array.first()` on an empty array. // Works in v6 because `array.first()` is only called if the array is not empty. if myArray.size() != 0 and myArray.first() label.new(bar_index, high, "Test") // A correct approach for v5: `array.first()` is only called when we're sure the array is not empty. if myArray.size() != 0 if myArray.first() label.new(bar_index, high, "Test")

Cannot repeat parameters

In v5, you can specify the same parameter in a function more than once. However, doing so raises a *compiler warning*, and only the *first* value will be used.

// In v5, compiles but raises warning. Only uses first value, so plot color will be `blue`. plot(close, "Close", color = color.blue, linewidth = 2, color = color.red)

In v6, you can specify a parameter only *once*, and doing otherwise will result in a *compilation error*.

Fix: Remove the duplicate parameters.

// In v6, script will not compile if parameter is specified more than once. plot(close, "Close", color = color.blue, linewidth = 2)

No series `offset` values

The `offset` parameter can no longer accept “series” values

In Pine v5, the `offset` parameter in plot() and similar functions can accept “*series* int” arguments. However, passing a “series” argument raises a compiler warning, and the behavior is *incorrect*: only the *last* calculated offset is used on the whole chart, regardless of its previous values.

For example, this script uses `bar_index / 2` as a “series” `offset` argument while plotting the high points of each bar’s body. Because the plot() function uses only the *last* `offset` value, the plot appears offset by 10 bars here for the *entire* “GOOGL” 12M chart (since the chart’s last `bar_index` is 20 here):

//@version=5 indicator("`offset` parameter demo", overlay = true) //@variable `series int` value. Used as `offset` parameter value in `plot()`. int seriesOffset = bar_index / 2 // In v5, a `series` type `offset` value is valid, but only the last calculated value is used. plot(math.max(close, open),"", color.orange, 4, plot.style_stepline, offset = seriesOffset)

In v6, the `offset` parameter accepts an argument qualified as “*simple*” or weaker. The value used must be the same on every bar.

Remember that the Pine Script qualifiers hierarchy means that a parameter expecting a “simple” value can also accept values qualified as “input” or “const”. However, passing a “series” argument triggers a compilation error.

Fix: Change any “series” values passed to `offset` to “simple” values.

Minimum `linewidth` is 1

In v5, the `linewidth` parameter of the plot() and hline() functions can accept a value smaller than 1, although the width on the chart will still appear as 1 for these drawings:

//@version=5 indicator("Linewidth demo") //@variable User-input width for a line. Default value set to 0, with no minimum limit. int userWidth = input.int(0, "Linewidth") // Valid in v5, but line widths on chart all appear as `linewidth=1`. Not valid in v6. plot(close, "LW 1", linewidth = 1) plot(close + 5, "LW 0", linewidth = userWidth) plot(close + 10, "LW -5", linewidth = -5) hline(240, "hline", color.maroon, linewidth = -3)

In v6, the `linewidth` argument **must** be 1 or greater. Passing a smaller value causes a compilation error.

Fix: Replace any `linewidth` argument that is smaller than 1 to ensure all width values are *at least* 1.

//@version=6 indicator("Linewidth demo") //@variable User-input width for a line. Default value set to 2, with minimum value set to 1. int userWidth = input.int(2, "Linewidth", minval = 1) // In v6, all line widths must be at

least 1 or greater. plot(close, "LW 1", linewidth = 1) plot(close + 5, "LW 2", linewidth = userWidth) plot(close + 10, "LW 5", linewidth = 5) hline(240, "hline", color.maroon, linewidth = 3)

Negative indices in arrays

Some array functions now accept negative indices.

In v5, array functions that require an element's *index* always expect a value *greater than or equal to 0*. Therefore, functions like array.get(), array.insert(), array.set(), and array.remove() raise a runtime error if a negative index is passed.

In v6, array.get(), array.insert(), array.set(), and array.remove() allow you to pass a *negative index* to request items from the *end* of the array. For example, `-1` refers to the last item in the array, `-2` refers to the second to last, and so forth.

//@variable Array of "int" numbers from 1 to 5. array<int> countingArray = array.from(1, 2, 3, 4, 5) // Array indexing starts from 0 to retrieve first element in both v5 and v6. int firstValue = countingArray.get(0) //Returns "1" // In v6, can retrieve last array element using negative index. This index is invalid in v5. int lastValue = countingArray.get(-1) //Returns "5" // Other `array.*()` functions also accept negative indexing in v6. These lines raise runtime errors in v5. countingArray.set(-2, 10) // Updated array: [1, 2, 3, 10, 5] countingArray.remove(-5) // Updated array: [2, 3, 10, 5] countingArray.insert(-1, 20) // Updated array: [2, 3, 10, 20, 5]

As a result, scripts that return a runtime error for using negative indices in v5 can be executed without error in v6.

However, if you create or update a script in v6, you must be aware of this new behavior to ensure that the script does not behave unexpectedly.

Keep in mind that negative indexing is still bound by the size of the array. Therefore, an array of 5 elements only accepts indexing from 0 to 4 (first to last element) or -1 to -5 (last to first element). Any other indices are out of bounds and raise a runtime error:

//@variable Array of "int" numbers from 1 to 5. array<int> countingArray = array.from(1, 2, 3, 4, 5) // Trying to index negatively beyond the size of the array causes a runtime error. countingArray.remove(-6)

The `transp` parameter is removed

In Pine v4 and earlier, plot() and similar functions had a `transp` parameter that specified the transparency of the resulting plot.

Pine v5 deprecated and hid the `transp` parameter, because it is not fully compatible with the color system that Pine currently uses. Using both transparency settings together can result in unexpected behavior, as the `transp` parameter can get overwritten by the transparency of the color passed to the function. In v5, using the color.new() function and not the `transp` parameter avoids any such conflicts.

Pine v6 removes the `transp` parameter completely from the following functions: bgcolor(), fill(), plot(), plotarrow(), plotchar(), and plotshape(). Whenever the converter encounters a `transp` argument, it removes the argument from the converted v6 script.

Fix: To set the transparency of a drawn plot, use the color.new() function. Pass the color value as the first argument, and the desired transparency value as the second.

For example, this v5 code uses the hidden `transp` parameter to set the color of the plot to `80` transparency:

//@version=5 indicator("Transparency demo v5") color myColor = close > open ? color.green : color.red plot(close, color = myColor, transp = 80)

In Pine v6, the same result can be achieved using color.new():

//@version=6 indicator("Transparency demo v6") color myColor = close > open ? color.green : color.red plot(close, color = color.new(myColor, 80))

If you need to preserve the color inputs in the “Settings/Style” menu, you must ensure that every color that gets passed to every color.new() call is qualified as either “const” or “input”. If at least one of these color values is calculated dynamically (like the code above), the color selector does not appear in the settings:

You can learn more about why this happens and how to avoid it here.

Dynamic `for` loop boundaries

A for loop is a *count-controlled* loop that executes successive iterations of its local block based on a counter variable. The counter starts with an *initial value* (`from_num`) and increases or decreases by a fixed amount after every iteration until it reaches the specified *final value* (`to_num`).

In Pine v5, a for loop statement establishes its final counter value strictly *before* starting its iterations. If the script changes the value of a variable or expression used as the loop’s `to_num` argument during the loop’s iterations, those changes do **not** affect the counter’s boundaries. This behavior differs from while and for…in loops, which have control criteria that can change across iterations.

In Pine v6, all for loops *dynamically* evaluate their stopping criteria before **each iteration**. Variables and expressions used as the `to_num` argument that depend on values or objects modified in the loop’s scope can *update* the counter variable’s boundaries across iterations. This behavior enables scripts to use a for loop for iterative tasks where the exact iteration boundaries are unknown before the loop starts.

Because for loop boundaries can be *dynamic* in Pine v6, a v5 script using this loop structure with a mutated variable or dynamic expression such as `array.size(id) - 1` as the `to_num` argument can behave differently after conversion to v6.

Fix: If a for loop requires only **one** evaluation of an expression used as the `to_num` argument across iterations, assign the expression to a variable *outside* the loop’s scope, then use that variable as the `to_num` argument instead.

The following v5 example uses two user-defined methods to manage the elements in an array. The script calls the `dequeue()` method before a for loop to remove the first element from the `data` array. Then, it calls the `queue()` method inside the loop statement to add the current bar’s close into the array and return the array’s size for the loop’s end boundary. Within the loop’s scope, the script increments the `belowCount` variable by one for each element with a value below the current bar’s ohlc4 value:

//@version=5 indicator("v5 vs v6 `to_num` demo") //@variable The size of the `data` array. int sizeInput = input.int(20, "Size", 1) //@function Appends a new `value` to `this` array and returns the array's size. method

queue(array<float> this, float value) => this.push(value) this.size() //@function Removes the first value from `this` array if the size equals the `sizeInput`. method dequeue(array<float> this) => if this.size() == sizeInput this.shift() //@variable An array that holds `sizeInput` recent `close` prices. var array<float> data = array.new<float>() //@variable The number of elements in the `data` array that are below the current bar's `ohlc4`. int belowCount = 0 // Remove the oldest element from the `data` array when its size reaches the `sizeInput`. data.dequeue() // Push the bar's `close` into `data` and loop from zero to one less than the array's size. // In v5, the loop evaluates the `data.queue()` call *once*, meaning the final counter value is fixed across iterations. // In v6, it evaluates the call before *every* iteration, causing the array and loop boundary to expand indefinitely. for i = 0 to data.queue(close) - 1 // Add 1 to `belowCount` when the `data` element at index `i` is less than the `ohlc4` value. if data.get(i) < ohlc4 belowCount += 1 // Plot the `belowCount` in a separate pane. plot(belowCount, "Closes below OHLC4", color.blue, 3)

In v5, the above loop statement evaluates `data.queue(close) - 1` only **once**, before it starts the first iteration. It does *not* execute that expression again across iterations. As such, each script execution queues exactly one new value into the `data` array, and the number of times the loop executes its local code *does not change* while the loop runs.

However, the script does not work after conversion to v6, because the for loop evaluates `data.queue(close) - 1` before **every iteration**. Each evaluation of the expression adds a *new element* to the `data` array and *increases* the `to_num` boundary, causing the loop to iterate indefinitely until the script raises a runtime error:

We can fix the script's behavior by assigning the expression's initial result to a variable outside the loop's scope and using that variable in the for loop statement. This change prevents the expression from modifying the array's size or altering the loop's end boundary between iterations:

//@version=6 indicator("v5 to v6 fixed `to_num` demo") //@variable The size of the `data` array. int sizeInput = input.int(20, "Size", 1) //@function Appends a new `value` to `this` array and returns the array's size. method queue(array<float> this, float value) => this.push(value) this.size() //@function Removes the first value from `this` array if the size equals the `sizeInput`. method dequeue(array<float> this) => if this.size() == sizeInput this.shift() //@variable An array that holds `sizeInput` recent `close` prices. var array<float> data = array.new<float>() //@variable The number of elements in the `data` array that are below the current bar's `ohlc4`. int belowCount = 0 // Remove the oldest element from the `data` array when its size reaches the `sizeInput`. data.dequeue() // Push the `close` into the `data` array and assign one less than the array's size to a variable, // ensuring only one evaluation per script execution. int lastCount = data.queue(close) - 1 // Use `lastCount` as the `to_num` in the `for` loop statement. // This change prevents the array and loop from expanding indefinitely, // because the value is calculated *outside* the loop's scope. for i = 0 to lastCount // Add 1 to `belowCount` when the `data` element at index `i` is less than the `ohlc4` value. if data.get(i) < ohlc4 belowCount += 1 // Plot the `belowCount` in a separate pane. plot(belowCount, "Closes below OHLC4", color.blue, 3)

First indicator

The Pine Editor

The Pine Editor is where you will be working on your scripts. While you can use any text editor you want to write your Pine scripts, using the Pine Editor has many advantages:

- It highlights your code following Pine Script® syntax.
- It pops up syntax reminders when you hover over language constructs.
- It provides quick access to the Pine Script Reference Manual popup when you select `Ctrl` or `Cmd` and a built-in Pine Script construct, and opens the library publication page when doing the same with code imported from libraries.
- It provides an auto-complete feature that you can activate by selecting `Ctrl+Space` or `Cmd+I`, depending on your operating system.
- It makes the write/compile/run cycle more efficient because saving a new version of a script already loaded on the chart automatically compiles and executes it.

To open the Pine Editor, select the “Pine Editor” tab at the bottom of the TradingView chart.

First version

Let’s create our first working Pine script, an implementation of the MACD indicator:

1. Open the Pine Editor’s dropdown menu (the arrow at the top-left corner of the Pine Editor pane, beside the script name) and select “Create new/Indicator”.
2. Copy the example script code below by clicking the button on the top-right of the code widget.
3. Select all the code already in the editor and replace it with the example code.
4. Save the script by selecting the script name or using the keyboard shortcut `Ctrl+S`. Choose a name for the script (e.g., “MACD #1”). The script is saved in TradingView’s cloud servers, and is local to your account, meaning only you can see and use this version.
5. Select “Add to chart” in the Pine Editor’s menu bar. The MACD indicator appears in a *separate pane* under the chart.

//@version=6 indicator("MACD #1") fast = 12 slow = 26 fastMA = ta.ema(close, fast) slowMA = ta.ema(close, slow) macd = fastMA - slowMA signal = ta.ema(macd, 9) plot(macd, color = color.blue) plot(signal, color = color.orange)

Our first Pine script is now running on the chart, which should look like this:

Let’s look at our script’s code, line by line:

Line 1: `//@version=6`

This is a compiler annotation telling the compiler the script uses version 6 of Pine Script.

Line 2: `indicator("MACD #1")`

Declares this script as an indicator, and defines the title of the script that appears on the chart as “MACD #1”.

Line 3: `fast = 12`

Defines an integer variable `fast` as the length of the fast moving average.

Line 4: `slow = 26`

Defines an integer variable `slow` as the length of the slow moving average.

Line 5: `fastMA = ta.ema(close, fast)`

Defines the variable `fastMA`, which holds the result of the *EMA* (Exponential Moving Average) calculated on the close series, i.e., the closing price of bars, with a length equal to `fast` (12).

Line 6: `slowMA = ta.ema(close, slow)`

Defines the variable `slowMA`, which holds the result of the EMA calculated on the close series with a length equal to `slow` (26).

Line 7: `macd = fastMA - slowMA`

Defines the variable `macd` as the difference between the two EMAs.

Line 8: `signal = ta.ema(macd, 9)`

Defines the variable `signal` as a smoothed value of `macd` using the EMA algorithm with a length of 9.

Line 9: `plot(macd, color = color.blue)`

Calls the plot() function to output the variable `macd` using a blue line.

Line 10: `plot(signal, color = color.orange)`

Calls the plot() function to output the variable `signal` using an orange line.

Second version

The first version of our script calculated the MACD using multiple steps, but because Pine Script is specially designed to write indicators and strategies, built-in functions exist for many common indicators, including one for MACD: ta.macd().

Therefore, we can write a second version of our script that takes advantage of Pine's available built-in functions:

//@version=6 indicator("MACD #2") fastInput = input(12, "Fast length") slowInput = input(26, "Slow length") [macdLine, signalLine, histLine] = ta.macd(close, fastInput, slowInput, 9) plot(macdLine, color = color.blue) plot(signalLine, color = color.orange)

Note that:

- We add inputs so we can change the lengths of the moving averages from the script's settings.
- We now use the ta.macd() built-in function to calculate our MACD directly, which replaces three lines of calculations and makes our code easier to read.

Let's repeat the same process as before to create our new indicator:

1. Open the Pine Editor's dropdown menu (the arrow at the top-left corner of the Pine Editor pane, beside the script name) and select "Create new/Indicator".
2. Copy the example script code above. The button on the top-right of the code widget allows you to copy it with a single click.
3. Select all the code already in the editor and replace it with the example code.
4. Save the script by selecting the script name or using the keyboard shortcut `Ctrl+S`. Choose a name for your script that is different from the previous one (e.g., "MACD #2").
5. Select "Add to chart" in the Pine Editor's menu bar. The "MACD #2" indicator appears in a *separate pane* under the "MACD #1" indicator.

Our second Pine script is now running on the chart. If we double-click on the indicator's name on the chart, it displays the script's "Settings/Inputs" tab, where we can now change the fast and slow lengths used in the MACD calculation:

Let's look at the lines that have changed in the second version of our script:

Line 2: `indicator("MACD #2")`

We have changed `#1` to `#2` so the second version of our indicator displays a different name on the chart.

Line 3: `fastInput = input(12, "Fast length")`

Instead of assigning a constant value to the variable, we used the input() function so we can change the length value from the script's "Settings/Inputs" tab. The default value is `12` and the input field's label is `"Fast length"`. When we change the value in the "Inputs" tab, the `fastInput` variable updates to contain the new length and the script re-executes on the chart with that new value. Note that, as our Pine Script Style guide recommends, we add `Input` to the end of the variable's name to remind us, later in the script, that its value comes from a user input.

Line 4: `slowInput = input(26, "Slow length")`

As with `fastInput` in the previous line, we do the same for the slow length, taking care to use a different variable name, default value, and title string for the input field's label.

Line 5: `[macdLine, signalLine, histLine] = ta.macd(close, fastInput, slowInput, 9)`

This is where we call the ta.macd() built-in function to perform all the first version's calculations in only one line. The function requires four *parameters* (the values enclosed in parentheses after the function name). It returns *three* values, unlike the other functions we've used so far that only returned one. For this reason, we need to enclose the list of three variables receiving the function's result in square brackets (to form a tuple) to the left of the `=` sign. Note that two of the values we pass to the function are the "input" variables containing the fast and slow lengths (`fastInput` and `slowInput`).

Lines 6 and 7: `plot(macdLine, color = color.blue)` and `plot(signalLine, color = color.orange)`

The variable names we are plotting here have changed, but the lines still behave the same as in our first version.

Our second version of the script performs the same calculations as our first, but we've made the indicator more efficient as it now leverages Pine's built-in capabilities and easily supports variable lengths for the MACD calculation. Therefore, we have successfully improved our Pine script.

Next

We now recommend you go to the Next Steps page.

First steps

Introduction

Welcome to the Pine Script® v6 User Manual, which will accompany you in your journey to learn to program your own trading tools in Pine Script. Welcome also to the very active community of Pine Script programmers on TradingView.

On this page, we present a step-by-step approach that you can follow to gradually become more familiar with indicators and strategies (also called *scripts*) written in the Pine Script programming language on TradingView. We will get you started on your journey to:

1. **Use** some of the tens of thousands of existing scripts on the platform.
2. **Read** the Pine Script code of existing scripts.
3. **Write** Pine scripts.

If you are already familiar with the use of Pine scripts on TradingView and are now ready to learn how to write your own, then jump to the Writing scripts section of this page.

If you are new to our platform, then please read on!

Using scripts

If you are interested in using technical indicators or strategies on TradingView, you can first start exploring the thousands of indicators already available on our platform. You can access existing indicators on the platform in two different ways:

- By using the chart's "Indicators, metrics, and strategies" button.
- By browsing TradingView's Community scripts, the largest repository of trading scripts in the world, with more than 150,000 scripts, half of which are free and *open-source*, which means you can see their Pine Script code.

If you can find the tools you need already written for you, it can be a good way to get started and gradually become proficient as a script user, until you are ready to start your programming journey in Pine Script.

Loading scripts from the chart

To explore and load scripts from your chart, click the "Indicators, metrics, and strategies" button, or use the forward slash `/` keyboard shortcut:

The dialog box that appears presents different categories of scripts in its left pane:

- **"Favorites"** lists the scripts you have "favorited" by clicking on the star that appears to the left of the script name when you hover over it.
- **"Personal"** displays the scripts you have written and saved in the Pine Editor. They are saved on TradingView's servers.
- **"Technicals"** groups most TradingView built-in scripts, organized in four categories: "Indicators", "Strategies", "Profiles", and "Patterns". Most are written in Pine Script and available for free.
- **"Financials"** contains all built-in indicators that display financial metrics. The contents of that tab and the subcategories they are grouped into depend on the symbol currently open on the chart.

- **"Community"** is where you can search from the more than 150,000 published scripts written by TradingView users. The scripts can be sorted by one of the three different filters – "Editors' picks" only shows open-source scripts hand-picked by our script moderators, "Top" shows the most popular scripts of all time, and "Trending" displays the most popular scripts that were published recently.
- **"Invite-only"** contains the list of the invite-only scripts you have been granted access to by their authors.

Here, we selected the "Technicals" tab to see the TradingView built-in indicators:

Clicking on one of the listed indicators or strategies loads the script on your chart. Strategy scripts are distinguished from indicators by a special symbol that appears to the right of the script name.

Browsing community scripts

To access the Community scripts feed from TradingView's homepage, select "Indicators and strategies" from the "Community" menu:

You can also search for scripts using the homepage's "Search" field, and filter scripts using different criteria. See this Help Center page explaining the different types of scripts that are available.

The scripts feed generates *script widgets*, which show the title and author of each publication with a preview of the published chart and description. Clicking on a widget opens the *script page*, which shows the publication's complete description, an enlarged chart, and any additional release notes. Users can boost, favorite, share, and comment on publications. If it is an open-source script, the source code is also available on the script page.

When you find an interesting script in the Community scripts, follow the instructions in the Help Center to load it on your chart.

Changing script settings

Once a script is loaded on the chart, you can double-click on its name or hover over the name and press the "Settings" button to bring up its "Settings/Inputs" tab:

The "Inputs" tab allows you to change the settings which the script's author has decided to make editable. You can configure some of the script's visuals using the "Style" tab of the same dialog box, and which timeframes the script should appear on using the "Visibility" tab.

Other settings are available to all scripts from the buttons that appear to the right of its name when you mouse over it, and from the "More" menu (the three dots):

Reading scripts

Reading code written by **good** programmers is the best way to develop your understanding of the language. This is as true for Pine Script as it is for all other programming languages. Finding good open-source Pine Script code is relatively easy. These are reliable sources of code written by good programmers on TradingView:

- The TradingView built-in indicators
- Scripts selected as Editors' Picks
- Scripts by the authors the PineCoders account follows

- Many scripts by authors with high reputations and open-source publications

Reading code from Community scripts is easy; if there is no grey or red “lock” icon in the upper-right corner of the script widget, then the script is open-source. By opening the script page, you can read its full source code.

To see the code of a TradingView built-in indicator, load the indicator on your chart, then hover over its name and select the “Source code” curly braces icon (if you don’t see it, it’s because the indicator’s source is unavailable). When you click on the `{}` icon, the Pine Editor opens below the chart and displays the script’s code. If you want to edit the script, you must first select the “Create a working copy” button. You will then be able to modify and save the code. Because the working copy is a different version of the script, you need to use the Editor’s “Add to chart” button to add that new copy to the chart.

For example, this image shows the Pine Editor, where we selected to view the source code from the “Bollinger Bands” indicator on our chart. Initially, the script is *read-only*, as indicated by the orange warning text:

You can also open editable versions of the TradingView built-in scripts from the Pine Editor by using the “Create new” > “Built-in…” menu selection:

Writing scripts

We have built Pine Script to empower both new and experienced traders to create their own trading tools. Although learning a first programming language, like trading, is rarely **very** easy for anyone, we have designed Pine Script so it is relatively easy to learn for first-time programmers, yet powerful enough for knowledgeable programmers to build tools of moderate complexity.

Pine Script allows you to write three types of scripts:

- **Indicators**, like RSI, MACD, etc.
- **Strategies**, which include logic to issue trading orders and can be backtested and forward-tested.
- **Libraries**, which are used by more advanced programmers to package often-used functions that can be reused by other scripts.

The next step we recommend is to write your first indicator.

Next steps

After your first steps and your first indicator, let us explore a bit more of the Pine Script® landscape by sharing some pointers to guide you in your journey to learn Pine Script.

“indicators” vs “strategies”

Pine Script strategies are used to backtest on historical data and forward test on open markets. In addition to indicator calculations, they contain `strategy.*()` calls to send trade orders to the broker emulator, which can then simulate their execution. Strategies display trade markers on the chart and simulated backtest results in a strategy report within the chart’s bottom panel.

Pine Script *indicators* contain calculations, but cannot be used in backtesting. They perform calculations across a dataset to generate outputs like visuals, alerts, or Pine Logs. Because they do not require the broker emulator, they often use fewer resources and run faster than strategies. It is thus advantageous to use indicators whenever you can.

Both indicators and strategies can run in either overlay mode (displaying over the main chart’s bars) or pane mode (displaying in a separate section below or above the main chart). In either mode, both script types can also plot information, display drawing objects, and generate alert events.

For more information about the unique characteristics of the different script types in Pine and how to declare them, refer to the Declaration statements page.

How scripts are executed

A Pine script is **not** like programs in many programming languages that execute once and then stop. In the Pine Script *runtime* environment, a script runs in the equivalent of an invisible loop where it is executed once on each bar of whatever chart you are on, from left to right. Chart bars that have already closed when the script executes on them are called *historical bars*. When execution reaches the chart's last bar and the market is open, it is on the *realtime bar*. The script then executes once every time a price or volume change is detected, and one last time for that realtime bar when it closes. That realtime bar then becomes an *elapsed realtime bar*. Note that when the script executes in realtime, it does not recalculate on all the chart's historical bars on every price/volume update. It has already calculated once on those bars, so it does not need to recalculate them on every chart tick. See the Execution model page for more information.

When a script executes on a historical bar, the close built-in variable holds the value of that bar's close. When a script executes on the realtime bar, close returns the **current** price of the symbol until the bar closes.

Contrary to indicators, strategies normally execute only once on realtime bars, when they close. They can also be configured to execute on each price/volume update if that is what you need. See the page on Strategies for more information, and to understand how strategies calculate differently than indicators.

Time series

The main data structure used in Pine Script is called a time series. Time series contain one value for each bar the script executes on, so they continuously expand as the script executes on more bars. Past values of

the time series can be referenced using the history-referencing operator: []. `close[1]`, for example, refers to the value of close on the bar preceding the one where the script is executing.

While this indexing mechanism may remind many programmers of arrays, a time series is different and thinking in terms of arrays will be detrimental to understanding this key Pine Script concept. A good comprehension of both the execution model and time series is essential in understanding how Pine scripts work. If you have never worked with data organized in time series before, you will need practice to put them to work for you. Once you familiarize yourself with these key concepts, you will discover that by combining the use of time series with our built-in functions specifically designed to handle them efficiently, much can be accomplished in very few lines of code.

Publishing scripts

TradingView is home to a large community of Pine Script programmers and millions of traders from all around the world. Once you become proficient enough in Pine Script, you can choose to share your scripts with other traders. Before doing so, please take the time to learn Pine Script well-enough to supply traders with an original and reliable tool. All publicly published scripts are analyzed by our team of moderators and must comply with our Script Publishing Rules, which require them to be original and well-documented.

If you want to use Pine scripts for your own use, simply write them in the Pine Editor and add them to your chart from there; you don't have to publish them to use them. If you want to share your scripts with just a few friends, you can publish them privately and send your friends the browser's link to your private publication. See the page on Publishing for more information.

Getting around the Pine Script documentation

While reading code from published scripts is no doubt useful, spending time in our documentation will be necessary to attain any degree of proficiency in Pine Script. Our two main sources of documentation on Pine Script are:

- This Pine Script v6 User Manual
- Our Pine Script v6 Reference Manual

The Pine Script v6 User Manual, which is located on its separate page and in English only.

The Pine Script v6 Reference Manual documents what each language construct does. It is an essential tool for all Pine Script programmers; your life will be miserable if you try to write scripts of any reasonable complexity without consulting it. It exists in two formats: a separate page linked above, and the popup version. Access the popup version from the Pine Editor by either selecting `Ctrl` or `Cmd` and selecting a keyword, or by using the Editor's "More/Reference Manual…" menu. The Reference Manual is translated into multiple languages.

There are six different versions of Pine Script released. Ensure the documentation you use corresponds to the Pine Script version you are coding with.

Where to go from here?

This Pine Script v6 User Manual contains numerous examples of code used to illustrate the concepts we discuss. By going through it, you will be able to both learn the foundations of Pine Script and study the example scripts. Reading about key concepts and trying them out right away with real code is a productive

way to learn any programming language. As you hopefully have already done in the First indicator page, copy this documentation's examples in the Editor and play with them. Explore! You won't break anything.

This is how the Pine Script v6 User Manual you are reading is organized:

- The Language section explains the main components of the Pine Script language and how scripts execute.
- The Concepts section is more task-oriented. It explains how to do things in Pine Script.
- The Writing section explores tools and tricks that will help you write and publish scripts.
- The FAQ section answers common questions from Pine Script programmers.
- The Error messages page documents causes and fixes for the most common runtime and compiler errors.
- The Release Notes page is where you can follow the frequent updates to Pine Script.
- The Migration guides section explains how to port between different versions of Pine Script.
- The Where can I get more information page lists other useful Pine Script-related content, including where to ask questions when you are stuck on code.

We wish you a successful journey with Pine Script... and trading!

Release notes

This page contains release notes describing notable changes to the Pine Script® experience.

2026

October 2026

Input alignment improvements

We've improved the horizontal alignment behavior of inline inputs in the "Settings/Inputs" tab. The new system allows coders to create big input groups with many inlined inputs that are still aligned with each other, without adding any manual aligning logic or spacing workarounds.

Previously, any `input*()` calls that used a non-empty `inline` argument had different alignment behavior from those that did not, affecting the readability and aesthetics of inputs in the tab. The typical workaround was to apply Unicode spaces to inline input titles to force horizontal alignment, but that workaround produced inconsistent results between different operating systems and devices.

Now, inline inputs follow different rules to align consistently with non-inline inputs, without requiring workarounds such as character padding in titles. However, these rules apply only if the inline inputs meet the following criteria:

- The **first** `input*()` call with specific `inline` and `group` arguments includes a non-empty `title` argument.
- All other inputs with the same `inline` and `group` arguments use an empty string ("") as the `title` argument.

Here's an example of how this system changes properly formatted inputs to ensure all rows align:

//@version=6 indicator("Improved input alignment demo", overlay = true) // ————— Pivots ————— const string GRP_PIVOTS = "Pivots" bool showPivotsInput = input.bool(true, "Show Pivots", inline = "pivots", group = GRP_PIVOTS) string pivotsStyleInput = input.string("Solid", "", options = ["Solid", "Dashed", "Dotted"], inline = "pivots", group = GRP_PIVOTS) int pivotsWidthInput = input.int(1, "", minval = 1, maxval = 4, inline = "pivots", group = GRP_PIVOTS) bool showMidpointsInput = input.bool(false, "Midpoints", inline = "mid", group = GRP_PIVOTS) string midpointsStyleInput = input.string("Dotted", "", options = ["Solid", "Dashed", "Dotted"], inline = "mid", group = GRP_PIVOTS) int midpointsWidthInput = input.int(1, "", minval = 1, maxval = 4, inline = "mid", group = GRP_PIVOTS) bool showLabelsInput = input.bool(true, "Pivot Labels", inline = "labels", group = GRP_PIVOTS) string labelsSideInput = input.string("Right", "", options = ["Left", "Right"], inline = "labels", group = GRP_PIVOTS, tooltip = "Shows the name of each pivot level on the chosen side of the line.") bool alertBrokenInput = input.bool(true, "Alert Broken Pivots", group = GRP_PIVOTS, tooltip = "Triggers an alert when price breaks through a pivot level.") string extendPivotsInput = input.string("Until Mitigated", "Extend Pivots...", options = ["Until Mitigated", "To Session End", "Indefinitely"], group = GRP_PIVOTS) string fromSessionsInput = input.string("Most Recent", "...From Which Sessions", options = ["Most Recent", "All"], group = GRP_PIVOTS, tooltip = "Selects which sessions' pivots are extended.") // ————— Range Data & Levels ————— const string GRP_RANGE = "Range Data & Levels" bool showTableInput = input.bool(true, "Show Data Table", inline = "table", group = GRP_RANGE) string tablePositionInput = input.string("Top Right", "", options = ["Top Left", "Top Center",

"Top Right", "Middle Left", "Middle Center", "Middle Right", "Bottom Left", "Bottom Center", "Bottom Right"], inline = "table", group = GRP_RANGE) string tableSizeInput = input.string("Small", "", options = ["Tiny", "Small", "Normal", "Large"], inline = "table", group = GRP_RANGE, tooltip = "Position and text size of the data table.") bool pivotStatsInput = input.bool(true, "Pivot Stats", inline = "stats", group = GRP_RANGE) bool rangeLevelStatsInput = input.bool(true, "Range Level Stats", inline = "stats", group = GRP_RANGE) bool dwmStatsInput = input.bool(true, "D/W/M Stats", inline = "stats", group = GRP_RANGE, tooltip = "Selects which sections are shown in the data table.") int dataLookbackInput = input.int(20, "Data Lookback", minval = 1, group = GRP_RANGE, tooltip = "Number of past sessions used to calculate range statistics.") string rangeMeasureInput = input.string("Average", "Range Measurement", options = ["Average", "Median"], group = GRP_RANGE, tooltip = "How the typical session range is calculated.") plot(na)

September 2026

Tables in the bottom panel

Users can now move tables drawn by a script into a *separate tab* in the chart's *bottom panel*. To move a script's tables to the panel, open the "More" menu in the script's status line and select the *"Move tables to bottom"* option. To move the tables back to the chart, select *"Move tables to chart"* from the "More" menu or from the *context menu* opened at the top of the panel's tab.

Tables in the bottom panel have different positioning and cell-sizing rules from those on the chart. All tables stack *vertically* in order from first to last, regardless of the `position` and `force_overlay` arguments of each table.new() call. The cells of each table also stretch as necessary to fit the allocated width and height within the pane, without truncating the displayed text. Additionally, users can select and copy the text from tables in the bottom panel, unlike tables displayed on the chart.

See the Displaying tables in the bottom panel section of the Tables page to learn more about this feature and how it works.

Improved line wrapping in square brackets

We've improved the line wrapping behaviors for code within *square brackets* (`[ ]`). Previously, all line-wrapped history-referencing operations or tuples required indenting each line after the first by any number of spaces that was *not* a multiple of four. The only exception was for square brackets enclosed in a set of parentheses, because code within parentheses can contain any amount of indentation, whether zero or a multiple of four spaces.

We've extended the indentation flexibility for code in parentheses to all code between square brackets. Now, wrapped code within square brackets can use any amount of indentation without restriction. For example:

//@version=6 indicator("Four-space indentation in square brackets demo") // This function definition line wraps history-referencing operations and the tuple return using indentation // in multiples of four spaces. ohlcChange(int length) => float oChange = open - open[length] // Indented by eight spaces. float hChange = high - high[length // Indented by eight spaces.] // Indented by four spaces. float lChange = low - low[length // Indented by 12 spaces.] // Indented by eight spaces. float cChange = close - close[length // Indented by four spaces.] // Indented by 12 spaces. [oChange, hChange, // No indentation. lChange, cChange // Indented by 16 spaces.] // Indented by four spaces. // This tuple declaration lists all identifiers on separate lines, using multiples of

four spaces for indentation. // Previously, such a declaration caused a compilation error. [oc, // Indented by four spaces. hc, // Indented by eight spaces. lc, // Indented by 12 spaces. cc // Indented by 16 spaces.] = ohlcChange(10) // No indentation. plotcandle(oc, hc, lc, cc, "Change candles")

August 2026

Pine Screener improvements

We've made two improvements to the Pine Screener: adding indices as a symbol source option, and making the full "Indicators" dialog box available when selecting an indicator script for the screener.

Previously, the Pine Screener could scan symbols only from a user's watchlist and use indicators only from the user's "Favorites" list.

Now, users can scan the symbols of any *index* using the Pine Screener, without first creating a watchlist of those symbols. Select the "Index" tab in the "Symbol source" dropdown menu to choose an index and scan up to 4,000 of its symbols. Clicking the "Add indicator" button now opens the "Indicators" dialog box. Users can select personal, invite-only, purchased, built-in, or community scripts to use with the Pine Screener. Note that the Pine Screener supports only *indicator* scripts that contain *at least one* `plot*()` or alertcondition() call; if a script is not compatible with the screener, it is grayed out in the "Indicators" dialog box and cannot be selected.

Refer to the Pine Screener key features and requirements article in our Help Center to learn more about the Pine Screener.

`once` conditional structure

We've added a new once keyword. It defines a conditional structure that executes a block of code when its condition evaluates to `true`. After the local block executes for the first time on a closed bar, it does not execute again on any subsequent bars, regardless of the condition. For more information, see the `once` structure section of the Conditional structures page.

Binary search in UDT arrays

The array.binary_search(), array.binary_search_leftmost(), and array.binary_search_rightmost() functions can now search arrays that store IDs of user-defined types (UDTs). Like the sorting functions for UDT collections, these functions include a `sort_field` parameter, which specifies the object field that a call compares while searching. The parameter accepts a "const int" *field index* (0 by default, referring to the first field in the type declaration) or a "const string" *field name*. Note that the array must be *sorted* by the same field, in ascending order, for the search to return correct results.

July 2026

Strategy improvements

We've added a parameter to the strategy() declaration statement for defining a new calculation behavior: `calc_on_every_history_tick`. An argument for this parameter must include the parameter's name (e.g., `calc_on_every_history_tick = true`). If the value is `true`, the script executes once for *each available tick* in

every historical bar. As the script executes on a historical bar, the values of built-in price and volume variables, such as high, low, close, and volume, *update* on each tick to approximate the data that was available when the bar was still developing. This behavior allows for more granular calculations and order fills across a chart's history, while also reducing the risk of lookahead bias in the simulation. This new feature is available on *standard charts* for users with Premium and Ultimate plans. See the Altering calculation behavior section of the Strategies page to learn more.

We've also made multiple changes to the "Settings/Properties" tab and the strategy report interface (formerly known as the *Strategy Tester*), including the following:

- The strategy report and the "Properties" tab now include a Script execution menu, where users can customize the strategy's allowed executions. The `calc_on_every_tick`, `calc_on_every_history_tick`, and `calc_on_order_fills` parameters of the strategy() statement define the default settings enabled in this menu.
- The former "Using bar magnifier" checkbox has been changed to a new Bar detalization menu, which is available in the "Properties" tab and at the top of the strategy report. This menu replaces the *Bar Magnifier* feature; it controls the level of intrabar detail available for filling orders and updating calculations on historical bars. The `use_bar_magnifier` parameter defines the strategy's default "Bar detalization" setting. See the Adjusting historical bar detail section of the Strategies page to learn more.
- The former "Fill orders using standard OHLC" checkbox for specifying whether a strategy uses real prices on Heikin Ashi charts has been replaced by a *"Heikin Ashi mode"* dropdown input. The input appears in the "Properties" tab only while the strategy runs on a Heikin Ashi chart. The `fill_orders_on_standard_ohlc` parameter defines the default state of this input.
- The former "Verify price for limit orders" input in the "Properties" tab has been replaced by a *"Limit order execution"* dropdown input, with a fixed set of options for defining assumptions about unfilled limit orders. Programmers can still specify an alternative default assumption by including a `backtest_fill_limits_assumption` argument in the declaration statement.
- The checkbox for filling orders immediately after creating them on a bar's closing tick has been replaced by an *"Order execution delay"* input. The default state of the input is still defined by the declaration statement's `process_orders_on_close` parameter.
- The "Margin for long positions" and "Margin for short positions" inputs in the "Properties" tab have been replaced by *"Long leverage"* and *"Short leverage"* inputs, allowing users to define a specific level of *leverage* for the strategy rather than calculating margin percentages. However, for backward compatibility, the strategy() function still includes `margin_long` and `margin_short` parameters, which define the strategy's default margin percentages directly. The strategy converts the arguments to leverage quantities and displays the converted values in these new inputs. See the Margin and leverage section of the Strategies page for more information.

Refer to the Strategies page to learn more about the current features of strategy scripts and the updated user interface.

Automatic parentheses

We've improved line wrapping functionality in the Pine Editor. When a programmer presses the `Enter` key to break part of a single-line expression and wrap it across lines, the editor now *automatically* encloses the expression in a pair of parentheses. This improvement reduces the need to insert indentation or parentheses manually when wrapping a single-line expression after writing it. For example:

//@version=6 indicator("Auto parentheses demo") // Press `Enter` to break any part of the expression following the `=` operator. // The Pine Editor automatically encloses the entire expression in a pair of parentheses. // Programmers no longer need to add indentation or parentheses manually for the code to compile. float someValue = time > chart.left_visible_bar_time and last_bar_time <= chart.right_visible_bar_time ? close > open ? high : low : na plot(someValue)

April 2026

Multiline strings

We've added support for multiline strings. A multiline string is a literal "string" value enclosed by *three* pairs of quotation marks (e.g., `"""..."""`) or apostrophes (e.g., `'''...'''`). Unlike single-line string syntax (e.g., `"..."`), which typically defines a literal string on a single line of code, the multiline syntax can define a literal string across *multiple* visible code lines.

All code between the `"""` or `'''` delimiters in a multiline string definition represents *literal text*. Each code line between the delimiters defines a separate *text line* for the string. The resulting string automatically includes the *newline* control character between each separate line; it does *not* require the `\n` escape sequence to insert the character at those points. For example:

//@version=6 indicator("Multiline string demo") //@variable A multiline string enclosed by `"""` delimiters. string multilineStr = """This is a multiline string. Each of these code lines literally represents a separate line of text. The newline character is automatically included before each new line. We do not have to manually add the `\\n` escape sequence to separate the lines.""" // Log the string's text in the Pine Logs pane on the first bar. if barstate.isfirst log.info(multilineStr)

Likewise, multiline strings automatically include *all spaces* used for indentation in the code, regardless of the code blocks that include their definitions. For example:

//@version=6 indicator("Indentation in multiline strings demo") //@variable A multiline string with indentation defined in the global scope. string globalIndentedStr = """No indentation. Also no indentation. Indented by one space. Indented by four spaces. Indented by 12 spaces. """ if barstate.islastconfirmedhistory //@variable A multiline string with indentation defined in a local block. // Although the block requires four spaces of indentation for its statements, the string itself does not. // Any indentation in the definition is still included literally in the string. string localIndentedStr = """--- No indentation. Indented by four spaces. """ // Concatenate both strings and display the result in a label. label.new(bar_index, 0, globalIndentedStr + localIndentedStr, textalign = text.align_left)

Expressions can use multiline strings as operands and arguments, just like single-line strings. Therefore, programmers can use the multiline string syntax to create unique line wrapping formats in their code. For example:

//@version=6 indicator("Line wrapping expressions with multiline strings demo") //@variable A string formed by concatenating three multiline strings. string concatenated = """String 1 """ + """String 2 """ + '''String 3 ''' // Log the resulting string's text in the Pine Logs pane on the first bar. if barstate.isfirst log.info(concatenated)

See the Multiline strings section of the Strings page to learn more about multiline strings and how they differ from single-line strings.

Updated editor settings

The Pine Editor's settings include a new "Use word wrap by default" checkbox. If selected, the Pine Editor automatically applies word wrapping when the user creates a new script, opens an existing script, or reopens the editor. The user can deactivate or reactivate word wrap for the current editor session at any time by using the `Alt + Z`/`Option + Z` hotkey or the "Toggle Word Wrap" option in the command palette.

Sorting UDT collections

The array.sort(), array.sort_indices(), and matrix.sort() functions can now sort arrays and matrices that store IDs of user-defined types (UDTs). These functions sort UDT collections by comparing values from one of the "int", "float", or "string" *fields* in the objects referenced by their elements.

The new `sort_field` *parameter* specifies *which* object field a call to these functions compares to sort a UDT collection. It accepts either a *"const int"* or *"const string"* argument:

- A "const int" argument specifies a field by its *field index*, where a value of 0 (the default) refers to the *first* field in the type declaration.
- A "const string" argument specifies a field by its assigned *name*.

For example:

//@version=6 indicator("Sorting UDT collections demo") //@type A custom type for creating objects that store "float", "int", and "string" data. type Data float price // Field index 0. int timestamp // Field index 1. string note // Field index 2. //@function Create a formatted string representation of an array of `Data` IDs. repr(array<Data> this) => string result = "\n[\n" for data in this result += str.format("(price: {0,number,0.000}, timestamp: {1,number,0}, note: {2}),\n", data.price, data.timestamp, data.note) result := str.replace(str.substring(result, 0, str.length(result) - 2), "[", "[") + "\n]" if barstate.islastconfirmedhistory //@variable References an array of `Data` objects representing data from a specific timeframe. array<Data> reqData = array.new<Data>(1, Data.new(hl2, time, timeframe.period)) //@variable The typical number of seconds in the chart's timeframe. int tfSeconds = timeframe.in_seconds() //@variable References an array of timeframe strings. array<string> timeframes = array.from(timeframe.from_seconds(tfSeconds * 2), timeframe.from_seconds(tfSeconds * 8), timeframe.from_seconds(tfSeconds * 4)) // Request a `Data` object for each timeframe and push the object's ID into the `reqData` array. for tf in timeframes reqData.push(request.security("", tf, Data.new(hl2, time, timeframe.period))) // Log a message showing the unsorted array's structure. log.info("Unsorted" + repr(reqData)) //#region Display the structure of the array after sorting it using each field. // First, let's sort the `reqData` array using the default `sort_field` argument (0) and log the result. // The default value refers to the *first field* listed in the `Data` type declaration (`price`). array.sort(reqData) log.info("Sorted using field 0 ('price')" + repr(reqData)) // Next, let's sort the array using the field named `timestamp` (at index 1) and log the result. reqData.sort(sort_field = "timestamp") log.info("Sorted using field named 'timestamp' (index 1)" + repr(reqData)) // Lastly, let's sort the array using the field at index 2 (`note`) and log the result. reqData.sort(sort_field = 2) log.info("Sorted using field 2 ('note')" + repr(reqData)) //#endregion

Refer to the Sorting arrays of user-defined types section of the Arrays page and the Sorting matrices of user-defined types section of the Matrices page to learn more about sorting UDT collections and the `sort_field` parameter.

January 2026

Footprint requests

We've added a new request.footprint() function and two new *data types*, footprint and volume_row. These features enable scripts to retrieve and work with volume footprint data for a chart's dataset:

- The request.footprint() function requests volume footprint information for the current bar. It returns either the *reference (ID)* of a footprint *object*, or na if no footprint data is available for the bar.
- A footprint object contains the available volume footprint data retrieved for a specific bar. Scripts can use IDs of this type with the new `footprint.*()` functions to retrieve a bar's overall footprint information, such as its total "buy" or "sell" volume and overall volume delta, or to retrieve volume_row IDs for *individual rows* within the footprint, including those for the bar's Point of Control (POC) and Value Area (VA) boundaries.
- A volume_row object contains data for a specific footprint row. Scripts can use IDs of this type with the new `volume_row.*()` functions to retrieve a footprint row's information, including its price levels, volume values, volume delta, and imbalances.

Programmers who have a Premium or Ultimate plan can use these features to create scripts that analyze volume footprint information across bars or perform custom footprint-based calculations. For example:

//@version=6 indicator("Footprint requests demo", overlay = true, behind_chart = false, max_labels_count = 50) //@variable The number of ticks to use as the price interval for each footprint row. int numTicksInput = input.int(100, "Ticks per footprint row", minval = 1) //@variable The percentage of each footprint's total volume to use for calculating the Value Area (VA). int vaInput = input.int(70, "Value Area percentage", minval = 1) //@variable References a `footprint` object for the current bar, or holds `na` if no footprint data is available. footprint reqFootprint = request.footprint(numTicksInput, vaInput) // If footprint data is available for the bar, retrieve overall and row-wise information for the footprint. [vaUpper, vaLower, pocUpper, pocLower] = if not na(reqFootprint) // Retrieve bar's total buy volume, sell volume, and volume delta from `footprint` object referenced by `reqFootprint`. // These `footprint.*()` functions return "float" volume values. float buyVolume = reqFootprint.buy_volume() float sellVolume = reqFootprint.sell_volume() float deltaVolume = reqFootprint.delta() // Get Value Area High (VAH), Value Area Low (VAL), and Point of Control (POC) row IDs from the `footprint` object. // These `footprint.*()` functions return IDs of `volume_row` objects containing data for the specific rows. volume_row vahRow = reqFootprint.vah() volume_row valRow = reqFootprint.val() volume_row pocRow = reqFootprint.poc() // Retrieve upper and lower price boundaries of VAH, VAL, and POC rows from `volume_row` objects. // These `volume_row.*()` functions return "float" price values. float vahUpperPrice = vahRow.up_price() float valLowerPrice = valRow.down_price() float pocUpperPrice = pocRow.up_price() float pocLowerPrice = pocRow.down_price() // Draw a label on each bar to show the footprint's volume and price levels as formatted text. string footprintInfo = str.format("Total buy volume: {0}\nTotal sell volume: {1}\nVolume delta: {2}\n---\nPOC range: {3}–{4}\nVA range: {5}–{6}", buyVolume, sellVolume, deltaVolume, pocLowerPrice, pocUpperPrice, valLowerPrice, vahUpperPrice) label.new(bar_index, high, text = footprintInfo, yloc = yloc.abovebar, size = 10) // Return VA and POC price boundaries to the variables in the tuple declaration. [vahUpperPrice, valLowerPrice, pocUpperPrice, pocLowerPrice] // Plot footprint row price boundaries to visualize VA and POC range of each bar. Hidden if requested footprint is `na`. plot(vaUpper, "VAH upper", color.navy, 3, plot.style_stepline, linestyle = plot.linestyle_dotted) plot(vaLower, "VAL lower", color.blue, 3, plot.style_stepline, linestyle = plot.linestyle_dotted) plot(pocUpper, "POC upper", color.purple, 4, plot.style_stepline) plot(pocLower, "POC lower", color.fuchsia, 4, plot.style_stepline)

See the `request.footprint()` section of the Other timeframes and data page to learn more about footprint requests. For more information about the footprint and volume_row types and the functions in their namespaces, refer to the footprint and volume_row section of the Type system page.

2025

December 2025

Updated line wrapping

Scripts now have improved line wrapping behavior. Previously, all multiline text representing a *single line* of code required indenting each line after the first by any number of spaces that was *not* a multiple of four, because Pine reserved four-space indentation for local code blocks.

We've removed the indentation restriction for all parts of an expression or statement enclosed in *parentheses*, including operations, function calls, and function parameter declarations. Scripts can now indent wrapped lines enclosed in parentheses by *zero or more* spaces, including multiples of four. For example:

//@version=6 // Before the update, wrapped lines in this code that start at multiples of four spaces caused compilation errors. indicator("Line wrapping between parentheses demo", // Indented by four spaces. overlay = true // Indented by eight spaces.) // No indentation. float median = 0.5 * (ta.highest(20) + ta.lowest(20) // Indented by four spaces.) // No indentation. plot(median, // No indentation. "Median", // Indented by two spaces. chart.fg_color, // Indented by three spaces. 3 // Indented by four spaces.) // No indentation.

However, if a line-wrapped expression is not enclosed in parentheses, all subsequent lines still require an indentation that is *not* a multiple of four spaces. For example:

//@version=6 indicator("Invalid line wrap demo", overlay = true) // The second line that starts with `*` in this wrapped expression causes a compilation error. // For the script to compile successfully, do any of the following: // - Move that part of the expression to line 9. // - Add another leading space to line 10 so that it doesn't start after a multiple of four spaces. // - Enclose the entire expression in another set of parentheses. float median = 0.5 * (ta.highest(20) + ta.lowest(20)) plot(median)

November 2025

We've added a new variable, syminfo.isin, which holds a string containing the 12-character International Securities Identification Number (ISIN) for the security represented by the symbol, or an empty string if no ISIN is available. An ISIN uniquely identifies a security *globally* and does not vary across exchanges, unlike ticker symbols. As such, programmers can use this variable to identify a symbol's underlying stock or other instrument, regardless of the name listed by an exchange. For example:

//@version=6 indicator("ISIN demo") // Define inputs for two symbols to compare. string symbol1Input = input.symbol("NASDAQ:AAPL", "Symbol 1") string symbol2Input = input.symbol("GETTEX:APC", "Symbol 2") if barstate.islastconfirmedhistory // Retrieve ISIN strings for `symbol1Input` and `symbol2Input`. var string isin1 = request.security(symbol1Input, "", syminfo.isin) var string isin2 = request.security(symbol2Input, "", syminfo.isin) // Log the retrieved ISIN codes. log.info("Symbol 1 ISIN: " + isin1) log.info("Symbol 2 ISIN: " + isin2) // Log an error message if one of the symbols does not have ISIN information. if isin1 == "" or isin2 == "" log.error("ISIN information is not available for both symbols.") // If both symbols do have ISIN information, log a message to confirm whether both refer to the same security. else if isin1 == isin2 log.info("Both symbols refer to the same security.") else log.info("The two symbols refer to different securities.")

October 2025

The time() and time_close() functions feature a new parameter: `timeframe_bars_back`. In contrast to the `bars_back` parameter, which determines the bar offset on the script's *main timeframe* for the timestamp calculation, `timeframe_bars_back` determines the bar offset on the *separate timeframe* specified by the `timeframe` argument. If the `timeframe_bars_back` value is positive, the function calculates the timestamp of the *past* bar that is N bars *back* on the specified timeframe. If negative, it calculates the *expected* timestamp of the bar that is N bars *forward* on that timeframe.

If a call to time() or time_close() includes arguments for *both* the `bars_back` and `timeframe_bars_back` parameters, it determines the timestamp corresponding to the `bars_back` offset *first*. Then, it applies the `timeframe_bars_back` offset to that time to calculate the final timestamp. For example:

//@version=6 indicator("`bars_back` and `timeframe_bars_back` demo") //@variable The number of bars back on the script's main timeframe (chart timeframe). int barsBackInput = input.int(10, "Chart bar offset") //@variable The number of bars back on the "1M" timeframe. int tfBarsBackInput = input.int(3, "'1M' bar offset") //@variable The opening UNIX timestamp of the current "1M" bar. int monthTime = time("1M") //@variable The opening time of the "1M" bar that contains the bar from `barsBackInput` bars back on the main timeframe. int offsetTime1 = time("1M", bars_back = barsBackInput) //@variable The "1M" opening time that is `tfBarsBackInput` monthly bars back, relative to the "1M" bar that opens at `offsetTime1`. // This `time()` call first determines the "1M" bar time corresponding to `barsBackInput` bars back on the // main timeframe, just like the previous call. Then, it calculates and returns the "1M" opening time that is // `tfBarsBackInput` *monthly* bars back relative to that time. int offsetTime2 = time("1M", bars_back = barsBackInput, timeframe_bars_back = tfBarsBackInput) // Plot the values for visual comparison. plot(monthTime, "No offset") plot(offsetTime1, "`bars_back`", color.red) plot(offsetTime2, "`bars_back` + `timeframe_bars_back`", color.purple) // Log formatted timestamps in the Pine Logs pane. log.info("\n{0}\n{1}\n{2}", str.format_time(monthTime), str.format_time(offsetTime1), str.format_time(offsetTime2))

September 2025

The plot() function can now draw dotted and dashed lines via the new `linestyle` parameter, which takes one of the following arguments: plot.linestyle_solid, plot.linestyle_dashed, or plot.linestyle_dotted. The `linestyle` parameter setting takes effect only for `style` arguments that plot lines.

August 2025

We've updated the maximum length for strings. Previously, a "string" value could not exceed 4,096 characters. Now, strings can contain up to 40,960 encoded characters.

Pine Editor changes

The Pine Editor is moving from the bottom panel to the *side panel*. This change will happen in phases over the following weeks.

By default, the new editor view overlays on the right side of the screen. For wider screens, a *split-view* mode is available, which automatically adjusts the chart's width to keep it visible alongside the editor. With this new vertical orientation, users can easily edit code and view other tabs such as the "Strategy Tester" or "Replay Trading" tabs at the same time.

The vertical editor view includes a *word wrap* feature, which enables users to read or modify long lines of code without scrolling horizontally. Note that word wrapping is only a *visual* feature; it does *not* change the

source code's structure or line numbering. Users can activate or deactivate word wrapping with the `Alt + Z`/`Option + Z` hotkey.

July 2025

All `input*()` functions feature a new parameter: `active`. This parameter specifies whether users can change the value of the input in the "Settings/Inputs" tab. If `true`, users can change the input's value. If `false`, the input is *grayed out*, and users *cannot* change the value. Programmers can use this parameter to define inputs whose states depend on the values of *other* inputs. For example:

//@version=6 indicator("Active input demo") //@variable The length for the RSI calculation. int rsiLengthInput = input.int(14, "RSI length") string GRP1 = "Smoothing" //@variable If `true`, the script applies smoothing based on the two inputs below. // If `false`, it does not apply smoothing, and those inputs are grayed out. bool enableSmoothingInput = input.bool(false, "Enable", group = GRP1) //@variable The length of the EMA for smoothing the RSI. int smoothLengthInput = input.int(9, "Length", 1, inline = "01", group = GRP1, active = enableSmoothingInput) //@variable The strength of the smoothing. If 1, the result is the EMA of the RSI. // If less than 1, the result is a mix between the EMA and the original RSI. float mixInput = input.float(1.0, "Mix", 0, 1, 0.01, inline = "01", group = GRP1, active = enableSmoothingInput) //@variable The RSI of `close`. float rsi = ta.rsi(close, rsiLengthInput) //@variable The smoothed RSI. float smoothed = ta.ema(rsi, enableSmoothingInput ? smoothLengthInput : 1) //@variable The mixture between `rsi` and `smoothed`, based on the inputs. float osc = enableSmoothingInput ? (1.0 - mixInput) * rsi + mixInput * smoothed : rsi // Make horizontal lines, and fill the space between `obLine` and `osLine`. obLine = hline(70) hline(50) osLine = hline(30) fill(obLine, osLine, color.new(color.purple, 90)) // Plot the `osc` series. plot(osc, "Custom RSI")

We've added a new `syminfo.*` variable:

- syminfo.current_contract – The ticker identifier of the underlying contract, if the current symbol is a continuous futures contract; na otherwise.

June 2025

Libraries can now export user-defined constant variables. Exported variables must be of the "int", "float", "bool", "color", or "string" type and include the const keyword in their declaration. For example:

//@version=6 library("MyConstants") export const float SILVER_RATIO = 1.0 + math.sqrt(2)

May 2025

The time_close variable and the time_close() function have improved behavior on tick charts and price-based charts (Renko, line break, Kagi, point & figure, and range). On chart types that are not time-based, the closing time of the open realtime bar is knowable only **after** the bar closes. Therefore, the value of time_close and time_close() is always na for that bar.

Previously, it was impossible to use expressions such as `time_close[1]` or `time_close("", 1)` to retrieve the closing timestamp of an *elapsed realtime* bar on these chart types. These expressions always returned na when they referenced a realtime bar, because the bar's timestamp was *not saved* after the closing tick.

With this new update, the closing timestamp of a realtime bar on tick charts or price-based charts is always available immediately after the bar closes. Now, scripts can use time_close with the `[]` history-referencing operator or call time_close() with a positive `bars_back` argument to retrieve the closing times of elapsed realtime bars on *any* chart type. For example:

//@version=6 indicator("Previous closing time") // Plot the `time_close[1]` value, representing the UNIX timestamp of the past bar's closing time. // This plot used to show `na` on all realtime bars of tick charts and price-based charts. plot(time_close[1], "Previous bar's closing timestamp")

April 2025

The `style` parameter of the ticker.renko(), ticker.pointfigure(), and ticker.kagi() functions accepts a new argument for box sizing: `"PercentageLTP"`. When a call to these functions uses this `style` argument, the returned ticker ID refers to a non-standard chart dataset with box sizes based on a user-defined percentage of the last trading price.

March 2025

We've added a setter function for boxes: box.set_xloc(). It is similar to the `*.set_xloc()` functions for lines and labels. The function sets the left and right coordinates of the box borders, and defines whether their values represent bar indices or UNIX timestamps.

For loop updates

The for loop structure has updated boundary-checking behavior. Previously, any for statement established the loop counter's end boundary (`to_num`) *before* starting the first iteration, and the final possible counter value *could not change* during the loop's execution. Changing the result of an expression used as a for loop's `to_num` argument inside the local scope *did not* affect the loop's iteration range.

Now, a for loop evaluates the `to_num` boundary *dynamically*, before *every iteration*. With this update, the loop statement can modify its stopping condition after any change to the `to_num` argument's result across iterations.

To learn more about this new behavior, refer to the `for` loops section of the Loops page and the Dynamic `for` loop boundaries section of the v6 migration guide.

February 2025

We've removed the scope count limit. Previously, any script's total number of scopes, including the global scope and all local scopes from user-defined functions and methods, loops, conditional structures, user-defined types, and enums, was limited to 550. Now, scripts can contain an indefinite number of local scopes from these structures.

We've introduced two new built-in variables, `bid` and `ask`, providing access to real-time market prices:

- bid - represents the highest price an active buyer is willing to pay for the instrument at its current value.
- ask - represents the lowest price an active seller will accept for the instrument at its current value.

These variables are only available on the `"1T"` timeframe. On other timeframes, their values are `na`.

2024

December 2024

The strategy.exit() function has updated calculation behaviors. Previously, calls to this command with arguments for the absolute and relative parameters defining a price level for the same exit order always prioritized the *absolute* parameter and *ignored* the relative one. For example, a call with specified `limit` and `profit` values always ignored the `profit` value. Now, the command evaluates *both* related parameters and uses the level that the market price is expected to *activate first*. See this section of the v6 migration guide for more information.

November 2024

Introducing Pine Script v6

Pine Script has graduated to v6! Starting today, future Pine updates will apply exclusively to this version. Therefore, we recommend converting existing v5 scripts to access new features as we roll them out. See our migration guide to understand the changes to existing Pine behaviors and learn how to convert scripts to v6.

Several new features and behaviors come with this version's release:

- Scripts can now call `request.*()` functions with *"series string"* arguments for the parameters that define the requested context, meaning a single `request.*()` call can change its requested data feed on *any* historical bar. Additionally, it is now possible to call `request.*()` functions inside the local scopes of loops, conditional structures, and exported library functions. See the Dynamic requests section of the Other timeframes and data page to learn more.
- Values of the "bool" type are now strictly `true` or `false`. They are never na in v6. Additionally, the or and and operators now feature *short-circuit ("lazy")* evaluation. If the first expression of an or operation is `true`, or the first expression of an and operation is `false`, the script does **not** evaluate the second expression because it is not necessary to determine the result. These improvements help boost the runtime efficiency of scripts that rely on "bool" values and conditional expressions.
- The `size` property of labels and the `text_size` property of boxes and tables now support "int" values in addition to the `size.*` constants. These "int" values represent sizes in *typographic points*, offering a more granular and wide range of text size possibilities.
- The new `text_formatting` parameter of the label.new(), box.new(), and table.cell() functions determines whether the object's displayed text is **bold**, *italicized*, or ***both***. It accepts one of these three new `text.*` constants: text.format_bold, text.format_italic, text.format_none. To modify a drawing object's `text_formatting` property, use the corresponding `*set_text_formatting()` functions.
- Strategies no longer stop calculating and raise an error when they reach the 9000 trade limit while not using Deep Backtesting mode. Instead, they *trim* the oldest orders to make space for new ones. The trimmed orders are *not* visible in the strategy report, but that does not change the strategy's simulation. To retrieve the trade index of the earliest *non-trimmed* order, use the strategy.closedtrades.first_index variable.
- The array.get(), array.set(), array.insert(), and array.remove() functions now support *negative* `index` arguments to reference elements starting from the *end* of an array. For instance, the call `array.get(myArray, -2)` retrieves the second to last element in `myArray`, which is equivalent to `array.get(myArray, array.size(myArray) - 2)`.
- The new syminfo.mincontract variable holds a value representing the smallest number of contracts/shares/lots/units required to trade the current symbol, as set by the exchange.
- Two new variables, syminfo.main_tickerid and timeframe.main_period, reference the ticker ID and timeframe from the script's *main context*, even if the script uses them in the `expression` argument of a

`request.*()` call. Here, “main context” refers to the current chart’s symbol and timeframe, unless the script is an indicator() that includes `symbol` or `timeframe` arguments in its declaration statement.

October 2024

We’ve added an optional `behind_chart` parameter to the indicator() and strategy() functions. This parameter specifies where plots and drawings appear relative to the main chart display when the `overlay` parameter is `true`. If `behind_chart` is `true`, the script’s visuals appear behind the chart display. If `false`, they appear in front of the chart display. The default is `true`.

August 2024

The ticker.new() and ticker.modify() functions feature two new parameters: `settlement_as_close` and `backadjustment`. Users can specify whether these parameters are on, off, or set to inherit the symbol’s default settings. These settings only affect the data from futures symbols with these options available on their charts. They have no effect on other symbols.

- The `backadjustment` parameter specifies whether past contract data on continuous futures symbols is back-adjusted. Its possible values are: backadjustment.on, backadjustment.off, or backadjustment.inherit.
- The `settlement_as_close` parameter specifies whether a futures symbol’s close value represents the actual closing price or the settlement price on “1D” and higher timeframes. Its possible values are: settlement_as_close.on, settlement_as_close.off, or settlement_as_close.inherit.

The Sharpe and Sortino ratios in the Strategy Tester module have updated calculations. Previously, the ratios used strategy returns over monthly periods if the trading range was three or more months and daily periods if the range was three or more days but less than three months. Both ratios now always use monthly periods for consistency.

June 2024

We've added a new parameter to the box.new(), label.new(), line.new(), polyline.new(), and table.new() functions:

- `force_overlay` - If true, the drawing will display on the main chart pane, even when the script occupies a separate pane. Optional. The default is false.

Pine Script Enums

Enums, also known as *enumerations*, *enumerated types*, or enum types, are unique data types with all possible values declared by the programmer. They can help programmers maintain more strict control over the values allowed by variables, conditional expressions, and collections, and they enable convenient dropdown input creation with the new input.enum() function. See our User Manual's Enums page to learn more about these new types and how to use them.

May 2024

We've added an optional `calc_bars_count` parameter to the indicator(), strategy(), request.security(), request.security_lower_tf(), and request.seed() functions that allows users to limit the number of recent historical bars a script or data request can execute across. When a script's indicator() or strategy()

declaration statement includes a `calc_bars_count` argument, its “Settings/Inputs” tab will include a “Calculated bars” input in the “Calculation” section. The default value in all these functions is 0, which signifies that the script or request executes across all the available data.

The `strategy.*` namespace features several new built-in variables:

- strategy.avg_trade - Returns the average amount of money gained or lost per trade. Calculated as the sum of all profits and losses divided by the number of closed trades.
- strategy.avg_trade_percent - Returns the average percentage gain or loss per trade. Calculated as the sum of all profit and loss percentages divided by the number of closed trades.
- strategy.avg_winning_trade - Returns the average amount of money gained per winning trade. Calculated as the sum of profits divided by the number of winning trades.
- strategy.avg_winning_trade_percent - Returns the average percentage gain per winning trade. Calculated as the sum of profit percentages divided by the number of winning trades.
- strategy.avg_losing_trade - Returns the average amount of money lost per losing trade. Calculated as the sum of losses divided by the number of losing trades.
- strategy.avg_losing_trade_percent - Returns the average percentage loss per losing trade. Calculated as the sum of loss percentages divided by the number of losing trades.

Pine Profiler

Our new Pine Profiler is a powerful utility that analyzes the executions of all significant code in a script and displays helpful performance information next to the code lines *inside* the Pine Editor. The Profiler's information provides insight into a script's runtime, the distribution of runtime across significant code regions, and the number of times each code region executes. With these insights, programmers can effectively pinpoint performance *bottlenecks* and ensure they focus on optimizing their code where it truly matters when they need to improve execution times.

See the new Profiling and optimization page to learn more about the Profiler, how it works, and how to use it to analyze a script's performance and identify optimization opportunities.

Pine Editor improvements

When opening the detached Pine Editor from a tab with a chart, it now links directly to that tab, as indicated by the “Linked” status and green icon in the bottom-right corner. While linked, the “Add to chart”, “Update on chart”, and “Apply to entire layout” buttons affect the charts on the main tab.

The detached Pine Editor now includes the Pine console.

April 2024

We've added a new parameter to the plot(), plotchar(), plotcandle(), plotbar(), plotarrow(), plotshape(), and bgcolor() functions:

- `force_overlay` - If true, the output will display on the main chart pane, even when the script occupies a separate pane.

March 2024

The `syminfo.*` namespace features a new built-in variable:

- syminfo.expiration_date - On non-continuous futures symbols, returns a UNIX timestamp representing the start of the last day of the current contract.

The time() and time_close() functions have a new parameter:

- `bars_back` - If specified, the function will calculate the timestamp from the bar N bars back relative to the current bar on its timeframe. It can also calculate the expected time of a future bar up to 500 bars away if the argument is a negative value. Optional. The default is 0.

February 2024

We've added two new functions for working with strings:

- str.repeat() - Constructs a new string containing the source string repeated a specified number of times with a separator injected between each repeated instance.
- str.trim() - Constructs a new string with all consecutive whitespaces and other control characters removed from the left and right of the source string.

The request.financial() function now accepts “D” as a `period` argument, allowing scripts to request available daily financial data.

For example:

//@version=5 indicator("Daily financial data demo") //@variable The daily Premium/Discount to Net Asset Value for "AMEX:SPY" float f1 = request.financial("AMEX:SPY", "NAV", "D") plot(f1)

The `strategy.*` namespace features a new variable for monitoring available capital in a strategy's simulation:

- strategy.opentrades.capital_held - Returns the capital amount currently held by open trades.

January 2024

The `syminfo.*` namespace features new built-in variables:

Syminfo:

- syminfo.employees - The number of employees the company has.
- syminfo.shareholders - The number of shareholders the company has.
- syminfo.shares_outstanding_float - The total number of shares outstanding a company has available, excluding any of its restricted shares.
- syminfo.shares_outstanding_total - The total number of shares outstanding a company has available, including restricted shares held by insiders, major shareholders, and employees.

Target price:

- syminfo.target_price_average - The average of the last yearly price targets for the symbol predicted by analysts.
- syminfo.target_price_date - The starting date of the last price target prediction for the current symbol.
- syminfo.target_price_estimates - The latest total number of price target predictions for the current symbol.
- syminfo.target_price_high - The last highest yearly price target for the symbol predicted by analysts.

- syminfo.target_price_low - The last lowest yearly price target for the symbol predicted by analysts.
- syminfo.target_price_median - The median of the last yearly price targets for the symbol predicted by analysts.

Recommendations:

- syminfo.recommendations_buy - The number of analysts who gave the current symbol a “Buy” rating.
- syminfo.recommendations_buy_strong - The number of analysts who gave the current symbol a “Strong Buy” rating.
- syminfo.recommendations_date - The starting date of the last set of recommendations for the current symbol.
- syminfo.recommendations_hold - The number of analysts who gave the current symbol a “Hold” rating.
- syminfo.recommendations_total - The total number of recommendations for the current symbol.
- syminfo.recommendations_sell - The number of analysts who gave the current symbol a “Sell” rating.
- syminfo.recommendations_sell_strong - The number of analysts who gave the current symbol a “Strong Sell” rating.

2023

December 2023

We've added `format` and `precision` parameters to all `plot*()` functions, allowing indicators and strategies to selectively apply formatting and decimal precision settings to plotted results in the chart pane's y-axis, the script's status line, and the Data Window. The arguments passed to these parameters supersede the values in the indicator() and strategy() functions. Both are optional. The defaults for these parameters are the same as the values specified in the script's declaration statement.

For example:

//@version=5 indicator("My script", format = format.percent, precision = 4) plot(close, format = format.price) // Price format with 4-digit precision. plot(100 * bar_index / close, precision = 2) // Percent format with 2-digit precision.

November 2023

We've added the following variables and functions to the `strategy.*` namespace:

- strategy.grossloss_percent - The total gross loss value of all completed losing trades, expressed as a percentage of the initial capital.
- strategy.grossprofit_percent - The total gross profit value of all completed winning trades, expressed as a percentage of the initial capital.
- strategy.max_runup_percent - The maximum rise from a trough in the equity curve, expressed as a percentage of the trough value.
- strategy.max_drawdown_percent - The maximum drop from a peak in the equity curve, expressed as a percentage of the peak value.
- strategy.netprofit_percent - The total value of all completed trades, expressed as a percentage of the initial capital.
- strategy.openprofit_percent - The current unrealized profit or loss for all open positions, expressed as a percentage of realized equity.

- strategy.closedtrades.max_drawdown_percent() - Returns the maximum drawdown of the closed trade, i.e., the maximum possible loss during the trade, expressed as a percentage.
- strategy.closedtrades.max_runup_percent() - Returns the maximum run-up of the closed trade, i.e., the maximum possible profit during the trade, expressed as a percentage.
- strategy.closedtrades.profit_percent() - Returns the profit/loss value of the closed trade, expressed as a percentage. Losses are expressed as negative values.
- strategy.opentrades.max_drawdown_percent() - Returns the maximum drawdown of the open trade, i.e., the maximum possible loss during the trade, expressed as a percentage.
- strategy.opentrades.max_runup_percent() - Returns the maximum run-up of the open trade, i.e., the maximum possible profit during the trade, expressed as a percentage.
- strategy.opentrades.profit_percent() - Returns the profit/loss of the open trade, expressed as a percentage. Losses are expressed as negative values.

October 2023

Pine Script Polylines

Polylines are drawings that sequentially connect the coordinates from an array of up to 10,000 chart points using straight or *curved* line segments, allowing scripts to draw custom formations that are difficult or impossible to achieve using line or box objects. To learn more about this new drawing type, see the Polylines section of our User Manual's page on Lines and boxes.

September 2023

New functions were added:

- strategy.default_entry_qty() - Calculates the default quantity, in units, of an entry order from strategy.entry() or strategy.order() if it were to fill at the specified `fill_price` value.
- chart.point.new() - Creates a new chart.point object with the specified `time`, `index`, and `price`.
- request.seed() - Requests data from a user-maintained GitHub repository and returns it as a series. An in-depth tutorial on how to add new data can be found here.
- ticker.inherit() - Constructs a ticker ID for the specified `symbol` with additional parameters inherited from the ticker ID passed into the function call, allowing the script to request a symbol's data using the same modifiers that the `from_tickerid` has, including extended session, dividend adjustment, currency conversion, non-standard chart types, back-adjustment, settlement-as-close, etc.
- timeframe.from_seconds() - Converts a specified number of `seconds` into a valid timeframe string based on our timeframe specification format.

The `dividends.*` namespace now includes variables for retrieving future dividend information:

- dividends.future_amount - Returns the payment amount of the upcoming dividend in the currency of the current instrument, or `na` if this data isn't available.
- dividends.future_ex_date - Returns the Ex-dividend date (Ex-date) of the current instrument's next dividend payment, or `na` if this data isn't available.
- dividends.future_pay_date - Returns the Payment date (Pay date) of the current instrument's next dividend payment, or `na` if this data isn't available.

The request.security_lower_tf() function has a new parameter:

- `ignore_invalid_timeframe` - Determines how the function behaves when the chart's timeframe is smaller than the `timeframe` value in the function call. If `false`, the function will raise a runtime error and halt the script's execution. If `true`, the function will return `na` without raising an error.

Users can now explicitly declare variables with the `const`, `simple`, and `series` type qualifiers, allowing more precise control over the types of variables in their scripts. For example:

//@version=5 indicator("My script") //@variable A constant `string` used as the `title` in the `plot()` function. const string plotTitle = "My plot" //@variable An `int` variable whose value is consistent after the first chart bar. simple int a = 10 //@variable An `int` variable whose value can change on every bar. series int b = bar_index plot(b % a, title = plotTitle)

August 2023

Added the following alert placeholders:

- `{{syminfo.currency}}` - Returns the currency code of the current symbol (“EUR”, “USD”, etc.).
- `{{syminfo.basecurrency}}` - Returns the base currency code of the current symbol if the symbol refers to a currency pair. Otherwise, it returns `na`. For example, it returns “EUR” when the symbol is “EURUSD”.

Pine Script Maps

Maps are collections that hold elements in the form of *key-value pairs*. They associate unique keys of a *fundamental type* with values of a *built-in* or user-defined type. Unlike arrays, these collections are *unordered* and do not utilize an internal lookup index. Instead, scripts access the values of maps by referencing the *keys* from the key-value pairs put into them. For more information on these new collections, see our User Manual’s page on Maps.

July 2023

Fixed an issue that caused strategies to occasionally calculate the sizes of limit orders incorrectly due to improper tick rounding of the `limit` price.

Added a new built-in variable to the `strategy.*` namespace:

- strategy.margin_liquidation_price - When a strategy uses margin, returns the price value after which a margin call will occur.

June 2023

New `syminfo.*` built-in variables were added:

- syminfo.sector - Returns the sector of the symbol.
- syminfo.industry - Returns the industry of the symbol.
- syminfo.country - Returns the two-letter code of the country where the symbol is traded.

A new display parameter for all `input.*()` functions was added. It provides you with more control over the display of input values next to a script's name. Four arguments can be used: display.status_line, display.data_window, display.all, and display.none. Combinations of these arguments using plus or minus

signs are allowed, and regardless of the argument used, input values will always continue to appear in the `Inputs` tab of the script's settings.

May 2023

New parameter added to the strategy.entry(), strategy.order(), strategy.close(), strategy.close_all(), and strategy.exit() functions:

- `disable_alert` - Disables order fill alerts for any orders placed by the function.

Our "Indicator on indicator" feature, which allows a script to pass another indicator's plot as a source value via the input.source() function, now supports multiple external inputs. Scripts can use a multitude of external inputs originating from up to 10 different indicators.

We've added the following array functions:

- array.every() - Returns `true` if all elements of the `id` array are `true`, `false` otherwise.
- array.some() - Returns `true` if at least one element of the `id` array is `true`, `false` otherwise. These functions also work with arrays of int and float types, in which case zero values are considered `false`, and all others `true`.

April 2023

Fixed an issue with trailing stops in strategy.exit() being filled on high/low prices rather than on intrabar prices.

Fixed behavior of array.mode(), matrix.mode() and ta.mode(). Now these functions will return the smallest value when the data has no most frequent value.

March 2023

It is now possible to use seconds-based timeframe strings for the `timeframe` parameter in request.security() and request.security_lower_tf().

A new function was added:

- request.currency_rate() - provides a daily rate to convert a value expressed in the `from` currency to another in the `to` currency.

February 2023

Pine Script Methods

Pine Script methods are specialized functions associated with specific instances of built-in or user-defined types. They offer a more convenient syntax than standard functions, as users can access methods in the same way as object fields using the handy dot notation syntax. Pine Script includes built-in methods for array, matrix, line, linefill, label, box, and table types and facilitates user-defined methods with the new method keyword. For more details on this new feature, see our User Manual's page on methods.

January 2023

New array functions were added:

- array.first() - Returns the array's first element.
- array.last() - Returns the array's last element.

2022

December 2022

Pine Objects

Pine objects are instantiations of the new user-defined composite types (UDTs) declared using the type keyword. Experienced programmers can think of UDTs as method-less classes. They allow users to create custom types that organize different values under one logical entity. A detailed rundown of the new functionality can be found in our User Manual's page on objects.

A new function was added:

- ticker.standard() - Creates a ticker to request data from a standard chart that is unaffected by modifiers like extended session, dividend adjustment, currency conversion, and the calculations of non-standard chart types: Heikin Ashi, Renko, etc.

New `strategy.*` functions were added:

- strategy.opentrades.entry_comment() - The function returns the comment message of the open trade's entry.
- strategy.closedtrades.entry_comment() - The function returns the comment message of the closed trade's entry.
- strategy.closedtrades.exit_comment() - The function returns the comment message of the closed trade's exit.

November 2022

Fixed behaviour of math.round_to_mintick() function. For 'na' values it returns 'na'.

October 2022

Pine Script now has a new, more powerful and better-integrated editor. Read our blog to find out everything to know about all the new features and upgrades.

New overload for the fill() function was added. Now it can create vertical gradients. More info about it in the blog post.

A new function was added:

- str.format_time() - Converts a timestamp to a formatted string using the specified format and time zone.

September 2022

The `text_font_family` parameter now allows the selection of a monospace font in label.new(), box.new() and table.cell() function calls, which makes it easier to align text vertically. Its arguments can be:

- font.family_default - Specifies the default font.
- font.family_monospace - Specifies a monospace font.

The accompanying setter functions are:

- label.set_text_font_family() - The function sets the font family of the text inside the label.
- box.set_text_font_family() - The function sets the font family of the text inside the box.
- table.cell_set_text_font_family() - The function sets the font family of the text inside the cell.

August 2022

A new label style label.style_text_outline was added.

A new parameter for the ta.pivot_point_levels() function was added:

- `developing` - If `false`, the values are those calculated the last time the anchor condition was true. They remain constant until the anchor condition becomes true again. If `true`, the pivots are developing, i.e., they constantly recalculate on the data developing between the point of the last anchor (or bar zero if the anchor condition was never true) and the current bar. Cannot be `true` when `type` is set to `"Woodie"`.

A new parameter for the box.new() function was added:

- `text_wrap` - It defines whether the text is presented in a single line, extending past the width of the box if necessary, or wrapped so every line is no wider than the box itself.

This parameter supports two arguments:

- text.wrap_none - Disabled wrapping mode for box.new and box.set_text_wrap functions.
- text.wrap_auto - Automatic wrapping mode for box.new and box.set_text_wrap functions.

New built-in functions were added:

- ta.min() - Returns the all-time low value of `source` from the beginning of the chart up to the current bar.
- ta.max() - Returns the all-time high value of `source` from the beginning of the chart up to the current bar.

A new annotation `//@strategy_alert_message` was added. If the annotation is added to the strategy, the text written after it will be automatically set as the default alert message in the [Create Alert] window.

//@version=5 // @strategy_alert_message My Default Alert Message strategy("My Strategy") plot(close)

July 2022

It is now possible to fine-tune where a script's plot values are displayed through the introduction of new arguments for the `display` parameter of the plot(), plotchar(), plotshape(), plotarrow(), plotcandle(), and plotbar() functions.

Four new arguments were added, complementing the previously available display.all and display.none:

- display.data_window displays the plot values in the Data Window, one of the items available from the chart's right sidebar.

- display.pane displays the plot in the pane where the script resides, as defined in with the `overlay` parameter of the script's indicator(), strategy(), or library() declaration statement.
- display.price_scale controls the display of the plot's label and price in the price scale, if the chart's settings allow them.
- display.status_line displays the plot values in the script's status line, next to the script's name on the chart, if the chart's settings allow them.

The `display` parameter supports the addition and subtraction of its arguments:

- `display.all - display.status_line` will display the plot's information everywhere except in the script's status line.
- `display.price_scale + display.status_line` will display the plot in the price scale and status line only.

June 2022

The behavior of the argument used with the `qty_percent` parameter of strategy.exit() has changed. Previously, the percentages used on successive exit orders of the same position were calculated from the remaining position at any given time. Instead, the percentages now always apply to the initial position size. When executing the following strategy, for example:

//@version=5 strategy("strategy.exit() example", overlay = true) strategy.entry("Long", strategy.long, qty = 100) strategy.exit("Exit Long1", "Long", trail_points = 50, trail_offset = 0, qty_percent = 20) strategy.exit("Exit Long2", "Long", trail_points = 100, trail_offset = 0, qty_percent = 20)

20% of the initial position will be closed on each strategy.exit() call. Before, the first call would exit 20% of the initial position, and the second would exit 20% of the remaining 80% of the position, so only 16% of the initial position.

Two new parameters for the built-in ta.vwap() function were added:

- `anchor` - Specifies the condition that triggers the reset of VWAP calculations. When `true`, calculations reset; when `false`, calculations proceed using the values accumulated since the previous reset.
- `stdev_mult` - If specified, the ta.vwap() calculates the standard deviation bands based on the main VWAP series and returns a `[vwap, upper_band, lower_band]` tuple.

New overloaded versions of the strategy.close() and strategy.close_all() functions with the `immediately` parameter. When `immediately` is set to `true`, the closing order will be executed on the tick where it has been placed, ignoring the strategy parameters that restrict the order execution to the open of the next bar.

New built-in functions were added:

- timeframe.change() - Returns `true` on the first bar of a new `timeframe`, `false` otherwise.
- ta.pivot_point_levels() - Returns a float array with numerical values representing 11 pivot point levels: `[P, R1, S1, R2, S2, R3, S3, R4, S4, R5, S5]`. Levels absent from the specified `type` return na values.

New built-in variables were added:

- session.isfirstbar - returns `true` if the current bar is the first bar of the day's session, `false` otherwise.
- session.islastbar - returns `true` if the current bar is the last bar of the day's session, `false` otherwise.

- session.isfirstbar_regular - returns `true` on the first regular session bar of the day, `false` otherwise.
- session.islastbar_regular - returns `true` on the last regular session bar of the day, `false` otherwise.
- chart.left_visible_bar_time - returns the `time` of the leftmost bar currently visible on the chart.
- chart.right_visible_bar_time - returns the `time` of the rightmost bar currently visible on the chart.

May 2022

Support for matrices has been added to the request.security() function.

The historical states of arrays and matrices can now be referenced with the [] operator. In the example below, we reference the historical state of a matrix 10 bars ago:

//@version=5 indicator("matrix.new<float> example") m = matrix.new<float>(1, 1, close) float x = na if bar_index > 10 x := matrix.get(m[10], 0, 0) plot(x) plot(close)

The ta.change() function now can take values of int and bool types as its `source` parameter and return the difference in the respective type.

New built-in variables were added:

- chart.bg_color - Returns the color of the chart's background from the "`Chart settings/Appearance/Background`" field.
- chart.fg_color - Returns a color providing optimal contrast with chart.bg_color.
- chart.is_standard - Returns true if the chart type is bars, candles, hollow candles, line, area or baseline, false otherwise.
- currency.USDT - A constant for the Tether currency code.

New functions were added:

- syminfo.prefix() - returns the exchange prefix of the `symbol` passed to it, e.g. “NASDAQ” for “NASDAQ:AAPL”.
- syminfo.ticker() - returns the ticker of the `symbol` passed to it without the exchange prefix, e.g. “AAPL” for “NASDAQ:AAPL”.
- request.security_lower_tf() - requests data from a lower timeframe than the chart's.

Added `use_bar_magnifier` parameter for the strategy() function. When `true`, the Broker Emulator uses lower timeframe data during history backtesting to achieve more realistic results.

Fixed behaviour of strategy.exit() function when stop loss triggered at prices outside the bars price range.

Added new `comment` and `alert` message parameters for the strategy.exit() function:

- `comment_profit` - additional notes on the order if the exit was triggered by crossing `profit` or `limit` specifically.
- `comment_loss` - additional notes on the order if the exit was triggered by crossing `stop` or `loss` specifically.
- `comment_trailing` - additional notes on the order if the exit was triggered by crossing `trail_offset` specifically.
- `alert_profit` - text that will replace the `'{{strategy.order.alert_message}}'` placeholder if the exit was triggered by crossing `profit` or `limit` specifically.

- `alert_loss` - text that will replace the `'{{strategy.order.alert_message}}'` placeholder if the exit was triggered by crossing `stop` or `loss` specifically.
- `alert_trailing` - text that will replace the `'{{strategy.order.alert_message}}'` placeholder if the exit was triggered by crossing `trail_offset` specifically.

April 2022

Added the `display` parameter to the following functions: barcolor, bgcolor, fill, hline.

A new function was added:

- request.economic() - Economic data includes information such as the state of a country's economy or of a particular industry.

New built-in variables were added:

- strategy.max_runup - Returns the maximum equity run-up value for the whole trading interval.
- syminfo.volumetype - Returns the volume type of the current symbol.
- chart.is_heikinashi - Returns true if the chart type is Heikin Ashi, false otherwise.
- chart.is_kagi - Returns true if the chart type is Kagi, false otherwise.
- chart.is_linebreak - Returns true if the chart type is Line break, false otherwise.
- chart.is_pnf - Returns true if the chart type is Point & figure, false otherwise.
- chart.is_range - Returns true if the chart type is Range, false otherwise.
- chart.is_renko - Returns true if the chart type is Renko, false otherwise.

New matrix functions were added:

- matrix.new<type>() - Creates a new matrix object. A matrix is a two-dimensional data structure containing rows and columns. All elements in the matrix must be of the type specified in the type template ("<type>").
- matrix.row() - Creates a one-dimensional array from the elements of a matrix row.
- matrix.col() - Creates a one-dimensional array from the elements of a matrix column.
- matrix.get() - Returns the element with the specified index of the matrix.
- matrix.set() - Assigns `value` to the element at the `column` and `row` index of the matrix.
- matrix.rows() - Returns the number of rows in the matrix.
- matrix.columns() - Returns the number of columns in the matrix.
- matrix.elements_count() - Returns the total number of matrix elements.
- matrix.add_row() - Adds a row to the matrix. The row can consist of `na` values, or an array can be used to provide values.
- matrix.add_col() - Adds a column to the matrix. The column can consist of `na` values, or an array can be used to provide values.
- matrix.remove_row() - Removes the row of the matrix and returns an array containing the removed row's values.
- matrix.remove_col() - Removes the column of the matrix and returns an array containing the removed column's values.
- matrix.swap_rows() - Swaps the rows in the matrix.
- matrix.swap_columns() - Swaps the columns in the matrix.
- matrix.fill() - Fills a rectangular area of the matrix defined by the indices `from_column` to `to_column`.
- matrix.copy() - Creates a new matrix which is a copy of the original.
- matrix.submatrix() - Extracts a submatrix within the specified indices.

- matrix.reverse() - Reverses the order of rows and columns in the matrix. The first row and first column become the last, and the last become the first.
- matrix.reshape() - Rebuilds the matrix to `rows` x `cols` dimensions.
- matrix.concat() - Append one matrix to another.
- matrix.sum() - Returns a new matrix resulting from the sum of two matrices, or of a matrix and a scalar (a numerical value).
- matrix.diff() - Returns a new matrix resulting from the subtraction between matrices, or of matrix and a scalar (a numerical value).
- matrix.mult() - Returns a new matrix resulting from the product between the matrices, or between a matrix and a scalar (a numerical value), or between a matrix and a vector (an array of values).
- matrix.sort() - Rearranges the rows in the `id` matrix following the sorted order of the values in the `column`.
- matrix.avg() - Calculates the average of all elements in the matrix.
- matrix.max() - Returns the largest value from the matrix elements.
- matrix.min() - Returns the smallest value from the matrix elements.
- matrix.median() - Calculates the median (“the middle” value) of matrix elements.
- matrix.mode() - Calculates the mode of the matrix, which is the most frequently occurring value from the matrix elements. When there are multiple values occurring equally frequently, the function returns the smallest of those values.
- matrix.pow() - Calculates the product of the matrix by itself `power` times.
- matrix.det() - Returns the determinant of a square matrix.
- matrix.transpose() - Creates a new, transposed version of the matrix by interchanging the row and column index of each element.
- matrix.pinv() - Returns the pseudoinverse of a matrix.
- matrix.inv() - Returns the inverse of a square matrix.
- matrix.rank() - Calculates the rank of the matrix.
- matrix.trace() - Calculates the trace of a matrix (the sum of the main diagonal's elements).
- matrix.eigenvalues() - Returns an array containing the eigenvalues of a square matrix.
- matrix.eigenvectors() - Returns a matrix of eigenvectors, in which each column is an eigenvector of the matrix.
- matrix.kron() - Returns the Kronecker product for the two matrices.
- matrix.is_zero() - Determines if all elements of the matrix are zero.
- matrix.is_identity() - Determines if a matrix is an identity matrix (elements with ones on the main diagonal and zeros elsewhere).
- matrix.is_binary() - Determines if the matrix is binary (when all elements of the matrix are 0 or 1).
- matrix.is_symmetric() - Determines if a square matrix is symmetric (elements are symmetric with respect to the main diagonal).
- matrix.is_antisymmetric() - Determines if a matrix is antisymmetric (its transpose equals its negative).
- matrix.is_diagonal() - Determines if the matrix is diagonal (all elements outside the main diagonal are zero).
- matrix.is_antidiagonal() - Determines if the matrix is anti-diagonal (all elements outside the secondary diagonal are zero).
- matrix.is_triangular() - Determines if the matrix is triangular (if all elements above or below the main diagonal are zero).
- matrix.is_stochastic() - Determines if the matrix is stochastic.
- matrix.is_square() - Determines if the matrix is square (it has the same number of rows and columns).

Added a new parameter for the strategy() function:

- `risk_free_rate` - The risk-free rate of return is the annual percentage change in the value of an investment with minimal or zero risk, used to calculate the Sharpe and Sortino ratios.

March 2022

New array functions were added:

- array.sort_indices() - returns an array of indices which, when used to index the original array, will access its elements in their sorted order.
- array.percentrank() - returns the percentile rank of a value in the array.
- array.percentile_nearest_rank() - returns the value for which the specified percentage of array values (percentile) are less than or equal to it, using the nearest-rank method.
- array.percentile_linear_interpolation() - returns the value for which the specified percentage of array values (percentile) are less than or equal to it, using linear interpolation.
- array.abs() - returns an array containing the absolute value of each element in the original array.
- array.binary_search() - returns the index of the value, or -1 if the value is not found.
- array.binary_search_leftmost() - returns the index of the value if it is found or the index of the next smallest element to the left of where the value would lie if it was in the array.
- array.binary_search_rightmost() - returns the index of the value if it is found or the index of the element to the right of where the value would lie if it was in the array.

Added a new optional `nth` parameter for the array.min() and array.max() functions.

Added `index` in for…in operator. It tracks the current iteration's index.

Table merging and cell tooltips

- It is now possible to merge several cells in a table. A merged cell doesn't have to be a header: you can merge cells in any direction, as long as the resulting cell doesn't affect any already merged cells and doesn't go outside of the table's bounds. Cells can be merged with the new table.merge_cells() function.
- Tables now support tooltips, floating labels that appear when you hover over a table's cell. To add a tooltip, pass a string to the `tooltip` argument of the table.cell() function or use the new table.cell_set_tooltip() function.

February 2022

Added templates and the ability to create arrays via templates. Instead of using one of the `array.new_*()` functions, a template function array.new<type>() can be used. In the example below, we use this functionality to create an array filled with `float` values:

//@version=5 indicator("array.new<float> example") length = 5 var a = array.new<float>(length, close) if array.size(a) == length array.remove(a, 0) array.push(a, close) plot(array.sum(a) / length, "SMA")

New functions were added:

- timeframe.in_seconds(timeframe) - converts the timeframe passed to the `timeframe` argument into seconds.
- input.text_area() - adds multiline text input area to the Script settings.
- strategy.closedtrades.entry_id() - returns the id of the closed trade's entry.

- strategy.closedtrades.exit_id() - returns the id of the closed trade's exit.
- strategy.opentrades.entry_id() - returns the id of the open trade's entry.

January 2022

Added new functions to clone drawings:

- line.copy()
- label.copy()
- box.copy()

2021

December 2021

Linefills

The space between lines drawn in Pine Script can now be filled! We've added a new `linefill` drawing type, along with a number of functions dedicated to manipulating it. Linefills are created by passing two lines and a color to the `linefill.new()` function, and their behavior is based on the lines they're tied to: they extend in the same direction as the lines, move when their lines move, and are deleted when one of the two lines is deleted.

New linefill-related functions:

- array.new_linefill()
- linefill()
- linefill.delete()
- linefill.get_line1()
- linefill.get_line2()
- linefill.new()
- linefill.set_color()
- linefill.all()

New functions for string manipulation

Added a number of new functions that provide more ways to process strings, and introduce regular expressions to Pine Script:

- str.contains(source, str) - Determines if the `source` string contains the `str` substring.
- str.pos(source, str) - Returns the position of the `str` string in the `source` string.
- str.substring(source, begin_pos, end_pos) - Extracts a substring from the `source` string.
- str.replace(source, target, replacement, occurrence) - Contrary to the existing str.replace_all() function, `str.replace()` allows the selective replacement of a matched substring with a replacement string.
- str.lower(source) and str.upper(source) - Convert all letters of the `source` string to lower or upper case:
- str.startswith(source, str) and str.endswith(source, str) - Determine if the `source` string starts or ends with the `str` substring.
- str.match(source, regex) - Extracts the substring matching the specified regular expression.

Textboxes

Box drawings now supports text. The box.new() function has five new parameters for text manipulation: `text`, `text_size`, `text_color`, `text_valign`, and `text_halign`. Additionally, five new functions to set the text properties of existing boxes were added:

- box.set_text()
- box.set_text_color()
- box.set_text_size()
- box.set_text_valign()
- box.set_text_halign()

New built-in variables

Added new built-in variables that return the `bar_index` and `time` values of the last bar in the dataset. Their values are known at the beginning of the script's calculation:

- last_bar_index - Bar index of the last chart bar.
- last_bar_time - UNIX time of the last chart bar.

New built-in `source` variable:

- hlcc4 - A shortcut for `(high + low + close + close)/4`. It averages the high and low values with the double-weighted close.

November 2021

for…in

Added a new for…in operator to iterate over all elements of an array:

//@version=5 indicator("My Script") int[] a1 = array.from(1, 3, 6, 3, 8, 0, -9, 5) highest(array) => var int highestNum = na for item in array if na(highestNum) or item > highestNum highestNum := item highestNum plot(highest(a1))

Function overloads

Added function overloads. Several functions in a script can now share the same name, as long one of the following conditions is true:

- Each overload has a different number of parameters:

//@version=5 indicator("Function overload") // Two parameters mult(x1, x2) => x1 * x2 // Three parameters mult(x1, x2, x3) => x1 * x2 * x3 plot(mult(7, 4)) plot(mult(7, 4, 2))

- When overloads have the same number of parameters, all parameters in each overload must be explicitly typified, and their type combinations must be unique:

//@version=5 indicator("Function overload") // Accepts both 'int' and 'float' values - any 'int' can be automatically cast to 'float' mult(float x1, float x2) => x1 * x2 // Returns a 'bool' value instead of a number mult(bool x1, bool x2) => x1 and x2 ? true : false mult(string x1, string x2) => str.tonumber(x1) *

str.tonumber(x2) // Has three parameters, so explicit types are not required mult(x1, x2, x3) => x1 * x2 * x3 plot(mult(7, 4)) plot(mult(7.5, 4.2)) plot(mult(true, false) ? 1 : 0) plot(mult("5", "6")) plot(mult(7, 4, 2))

Currency conversion

Added a new [currency] argument to most `request.*()` functions. If specified, price values returned by the function will be converted from the source currency to the target currency. The following functions are affected:

- request.dividends()
- request.earnings()
- request.financial()
- request.security()

October 2021

Pine Script v5 is here! This is a list of the **new** features added to the language, and a few of the **changes** made. See the Pine Script v5 Migration guide for a complete list of the **changes** in v5.

New features

Libraries are a new type of publication. They allow you to create custom functions for reuse in other scripts. See this manual's page on Libraries.

Pine Script now supports switch structures! They provide a more convenient and readable alternative to long ternary operators and if statements.

while loops are here! They allow you to create a loop that will only stop when its controlling condition is false, or a `break` command is used in the loop.

New built-in array variables are maintained by the Pine Script runtime to hold the IDs of all the active objects of the same type drawn by your script. They are label.all, line.all, box.all and table.all.

The runtime.error() function makes it possible to halt the execution of a script and display a runtime error with a custom message. You can use any condition in your script to trigger the call.

Parameter definitions in user-defined functions can now include a default value: a function defined as `f(x = 1) => x` will return 1 when called as `f()`, i.e., without providing an argument for its `x` parameter.

New variables and functions provide better script visibility on strategy information:

- strategy.closedtrades.entry_price() and strategy.opentrades.entry_price()
- strategy.closedtrades.entry_bar_index() and strategy.opentrades.entry_bar_index()
- strategy.closedtrades.entry_time() and strategy.opentrades.entry_time()
- strategy.closedtrades.size() and strategy.opentrades.size()
- strategy.closedtrades.profit() and strategy.opentrades.profit()
- strategy.closedtrades.commission() and strategy.opentrades.commission()
- strategy.closedtrades.max_runup() and strategy.opentrades.max_runup()
- strategy.closedtrades.max_drawdown() and strategy.opentrades.max_drawdown()
- strategy.closedtrades.exit_price()
- strategy.closedtrades.exit_bar_index()

- strategy.closedtrades.exit_time()
- strategy.convert_to_account()
- strategy.convert_to_symbol()
- strategy.account_currency

A new earnings.standardized constant for the request.earnings() function allows requesting standardized earnings data.

A v4 to v5 converter is now included in the Pine Script Editor. See the Pine Script v5 Migration guide for more information on converting your scripts to v5.

The Reference Manual now includes the systematic mention of the form and type (e.g., “simple int”) required for each function parameter.

The User Manual was reorganized and new content was added.

Changes

Many built-in variables, functions and function arguments were renamed or moved to new namespaces in v5. The venerable `study()`, for example, is now indicator(), and `security()` is now request.security(). New namespaces now group related functions and variables together. This consolidation implements a more rational nomenclature and provides an orderly space to accommodate the many additions planned for Pine Script.

See the Pine Script v5 Migration guide for a complete list of the **changes** made in v5.

September 2021

New parameter has been added for the `dividends()`, `earnings()`, `financial()`, `quandl()`, `security()`, and `splits()` functions:

- `ignore_invalid_symbol` - determines the behavior of the function if the specified symbol is not found: if `false`, the script will halt and return a runtime error; if `true`, the function will return `na` and execution will continue.

July 2021

`tostring` now accepts “bool” and “string” types.

New argument for `time` and `time_close` functions was added:

- `timezone` - timezone of the `session` argument, can only be used when a session is specified. Can be written out in GMT notation (e.g. “GMT-5”) or as an IANA time zone database name (e.g. “America/New_York”).

It is now possible to place a drawing object in the future with `xloc = xloc.bar_index`.

New argument for `study` and `strategy` functions was added:

- `explicit_plot_zorder` - specifies the order in which the indicator's plots, fills, and hlines are rendered. If true, the plots will be drawn based on the order in which they appear in the indicator's code, each newer plot being drawn above the previous ones.

June 2021

New variable was added:

- `barstate.islastconfirmedhistory` - returns `true` if script is executing on the dataset's last bar when market is closed, or script is executing on the bar immediately preceding the real-time bar, if market is open. Returns `false` otherwise.

New function was added:

- `round_to_mintick(x)` - returns the value rounded to the symbol's mintick, i.e. the nearest value that can be divided by `syminfo.mintick`, without the remainder, with ties rounding up.

Expanded `tostring()` functionality. The function now accepts three new formatting arguments:

- `format.mintick` to format to tick precision.
- `format.volume` to abbreviate large values.
- `format.percent` to format percentages.

May 2021

Improved backtesting functionality by adding the Leverage mechanism.

Added support for table drawings and functions for working with them. Tables are unique objects that are not anchored to specific bars; they float in a script's space, independently of the chart bars being viewed or the zoom factor used. For more information, see the Tables User Manual page.

New functions were added:

- `color.rgb(red, green, blue, transp)` - creates a new color with transparency using the RGB color model.
- `color.from_gradient(value, bottom_value, top_value, bottom_color, top_color)` - returns color calculated from the linear gradient between bottom_color to top_color.
- `color.r(color)`, `color.g(color)`, `color.b(color)`, `color.t(color)` - retrieves the value of one of the color components.
- `array.from()` - takes a variable number of arguments with one of the types: `int`, `float`, `bool`, `string`, `label`, `line`, `color`, `box`, `table` and returns an array of the corresponding type.

A new `box` drawing has been added to Pine Script, making it possible to draw rectangles on charts using the Pine Script syntax. For more details, see the Pine Script reference entry for box.new() and the Lines and boxes User Manual page.

The `color.new` function can now accept series and input arguments, in which case, the colors will be calculated at runtime. For more information about this, see our Colors User Manual page.

April 2021

New math constants were added:

- `math.pi` - is a named constant for Archimedes' constant. It is equal to 3.1415926535897932.
- `math.phi` - is a named constant for the golden ratio. It is equal to 1.6180339887498948.
- `math.rphi` - is a named constant for the golden ratio conjugate. It is equal to 0.6180339887498948.

- `math.e` - is a named constant for Euler's number. It is equal to 2.7182818284590452.

New math functions were added:

- `round(x, precision)` - returns the value of `x` rounded to the nearest integer, with ties rounding up. If the precision parameter is used, returns a float value rounded to that number of decimal places.
- `median(source, length)` - returns the median of the series.
- `mode(source, length)` - returns the mode of the series. If there are several values with the same frequency, it returns the smallest value.
- `range(source, length)` - returns the difference between the `min` and `max` values in a series.
- `todegrees(radians)` - returns an approximately equivalent angle in degrees from an angle measured in radians.
- `toradians(degrees)` - returns an approximately equivalent angle in radians from an angle measured in degrees.
- `random(min, max, seed)` - returns a pseudorandom value. The function will generate a different sequence of values for each script execution. Using the same value for the optional seed argument will produce a repeatable sequence.

New functions were added:

- `session.ismarket` - returns `true` if the current bar is a part of the regular trading hours (i.e. market hours), `false` otherwise.
- `session.ispremarket` - returns `true` if the current bar is a part of the pre-market, `false` otherwise.
- `session.ispostmarket` - returns `true` if the current bar is a part of the post-market, `false` otherwise.
- `str.format` - converts the values to strings based on the specified formats. Accepts certain `number` modifiers: `integer`, `currency`, `percent`.

March 2021

New assignment operators were added:

- `+=` - addition assignment
- `-=` - subtraction assignment
- `*=` - multiplication assignment
- `/=` - division assignment
- `%=` - modulus assignment

New parameters for inputs customization were added:

- `inline` - combines all the input calls with the same inline value in one line.
- `group` - creates a header above all inputs that use the same group string value. The string is also used as the header text.
- `tooltip` - adds a tooltip icon to the `Inputs` menu. The tooltip string is shown when hovering over the tooltip icon.

New argument for `fill` function was added:

- `fillgaps` - controls whether fills continue on gaps when one of the `plot` calls returns an `na` value.

A new keyword was added:

- `varip` - is similar to the `var` keyword, but variables declared with `varip` retain their values between the updates of a real-time bar.

New functions were added:

- `tonumber()` - converts a string value into a float.
- `time_close()` - returns the UNIX timestamp of the close of the current bar, based on the resolution and session that is passed to the function.
- `dividends()` - requests dividends data for the specified symbol.
- `earnings()` - requests earnings data for the specified symbol.
- `splits()` - requests splits data for the specified symbol.

New arguments for the study() function were added:

- `resolution_gaps` - fills the gaps between values fetched from higher timeframes when using `resolution`.
- `format.percent` - formats the script output values as a percentage.

February 2021

New variable was added:

- `time_tradingday` - the beginning time of the trading day the current bar belongs to.

January 2021

The following functions now accept a series length parameter:

- bb()
- bbw()
- cci()
- cmo()
- cog()
- correlation()
- dev()
- falling()
- mfi()
- percentile_linear_interpolation()
- percentile_nearest_rank()
- percentrank()
- rising()
- roc()
- stdev()
- stoch()
- variance()
- wpr()

A new type of alerts was added - script alerts. More information can be found in our Help Center.

2020

December 2020

New array types were added:

- `array.new_line()`
- `array.new_label()`
- `array.new_string()`

New functions were added:

- `str.length()` - returns number of chars in source string.
- `array.join()` - concatenates all of the elements in the array into a string and separates these elements with the specified separator.
- `str.split()` - splits a string at a given substring separator.

November 2020

- New `max_labels_count` and `max_lines_count` parameters were added to the study and strategy functions. Now you can manage the number of lines and labels by setting values for these parameters from 1 to 500.

New function was added:

- `array.range()` - return the difference between the min and max values in the array.

October 2020

The behavior of `rising()` and `falling()` functions have changed. For example, `rising(close,3)` is now calculated as following:

close[0] > close[1] and close[1] > close[2] and close[2] > close[3]

September 2020

Added support for `input.color` to the `input()` function. Now you can provide script users with color selection through the script's “Settings/Inputs” tab with the same color widget used throughout the TradingView user interface. Learn more about this feature in our blog

//@version=4 study("My Script", overlay = true) color c_labelColor = input(color.green, "Main Color", input.color) var l = label.new(bar_index, close, yloc = yloc.abovebar, text = "Colored label") label.set_x(l, bar_index) label.set_color(l, c_labelColor)

Added support for arrays and functions for working with them. You can now use the powerful new array feature to build custom datasets. See our User Manual page on arrays and our blog

//@version=4 study("My Script") a = array.new_float(0) for i = 0 to 5 array.push(a, close[i] - open[i]) plot(array.get(a, 4))

The following functions now accept a series length parameter. Learn more about this feature in our blog:

- alma()
- change()
- highest()

- highestbars()
- linreg()
- lowest()
- lowestbars()
- mom()
- sma()
- sum()
- vwma()
- wma()

//@version=4 study("My Script", overlay = true) length = input(10, "Length", input.integer, minval = 1, maxval = 100) avgBar = avg(highestbars(length), lowestbars(length)) float dynLen = nz(abs(avgBar) + 1, length) dynSma = sma(close, int(dynLen)) plot(dynSma)

August 2020

- Optimized script compilation time. Scripts now compile 1.5 to 2 times faster.

July 2020

- Minor bug fixes and improvements.

June 2020

- New `resolution` parameter was added to the `study` function. Now you can add MTF functionality to scripts and decide the timeframe you want the indicator to run on.

Please note that you need to reapply the indicator in order for the [resolution] parameter to appear.

- The `tooltip` argument was added to the `label.new` function along with the `label.set_tooltip` function:

//@version=4 study("My Script", overlay=true) var l=label.new(bar_index, close, yloc=yloc.abovebar, text="Label") label.set_x(l,bar_index) label.set_tooltip(l, "Label Tooltip")

- Added an ability to create alerts on strategies.
- A new function line.get_price() can be used to determine the price level at which the line is located on a certain bar.
- New label styles allow you to position the label pointer in any direction.

- Find and Replace was added to Pine Editor. To use this, press CTRL+F (find) or CTRL+H (find and replace).

- `timezone` argument was added for time functions. Now you can specify timezone for `second`, `minute`, `hour`, `year`, `month`, `dayofmonth`, `dayofweek` functions:

//@version=4 study("My Script") plot(hour(1591012800000, "GMT+1"))

- `syminfo.basecurrency` variable was added. Returns the base currency code of the current symbol. For EURUSD symbol returns EUR.

May 2020

- `else if` statement was added
- The behavior of `security()` function has changed: the `expression` parameter can be series or tuple.

April 2020

New function was added:

- `quandl()` - request quandl data for a symbol

March 2020

New function was added:

- `financial()` - request financial data for a symbol

New functions for common indicators were added:

- `cmo()` - Chande Momentum Oscillator
- `mfi()` - Money Flow Index
- `bb()` - Bollinger Bands
- `bbw()` - Bollinger Bands Width
- `kc()` - Keltner Channels
- `kcw()` - Keltner Channels Width
- `dmi()` - DMI/ADX
- `wpr()` - Williams % R
- `hma()` - Hull Moving Average
- `supertrend()` - SuperTrend

Added a detailed description of all the fields in the Strategy Tester Report.

February 2020

- New Pine Script indicator VWAP Anchored was added. Now you can specify the time period: Session, Month, Week, Year.
- Fixed a problem with calculating `percentrank` function. Now it can return a zero value, which did not happen before due to an incorrect calculation.
- The default `transparency` parameter for the `plot()`, `plotshape()`, and `plotchar()` functions is now 0%.
- For the functions `plot()`, `plotshape()`, `plotchar()`, `plotbar()`, `plotcandle()`, `plotarrow()`, you can set the `display` parameter, which controls the display of the plot. The following values can be assigned to it:
  - `display.none` - the plot is not displayed
  - `display.all` - the plot is displayed (Default)
- The `textalign` argument was added to the `label.new` function along with the `label.set_textalign` function. Using those, you can control the alignment of the label's text:

//@version=4 study("My Script", overlay = true) var l = label.new(bar_index, high, text="Right\n aligned\n text", textalign=text.align_right) label.set_xy(l, bar_index, high) .. image:: images/ReleaseNotes-Label_text_align.png

January 2020

New built-in variables were added:

- `iii` - Intraday Intensity Index
- `wvad` - Williams Variable Accumulation/Distribution
- `wad` - Williams Accumulation/Distribution
- `obv` - On Balance Volume
- `pvt` - Price-Volume Trend
- `nvi` - Negative Volume Index
- `pvi` - Positive Volume Index

New parameters were added for `strategy.close()`:

- `qty` - the number of contracts/shares/lots/units to exit a trade with
- `qty_percent` - defines the percentage of entered contracts/shares/lots/units to exit a trade with
- `comment` - addtional notes on the order

New parameter was added for `strategy.close_all`:

- `comment` - additional notes on the order

2019

December 2019

- Warning messages were added.

  For example, if you don't specify exit parameters for `strategy.exit` - `profit`, `limit`, `loss`, `stop` or one of the following pairs: `trail_offset` and `trail_price` / `trail_points` - you will see a warning message in the console in the Pine Script editor.

- Increased the maximum number of arguments in `max`, `min`, `avg` functions. Now you can use up to ten arguments in these functions.

October 2019

- `plotchar()` function now supports most of the Unicode symbols:

//@version=4 study("My Script", overlay=true) plotchar(open > close, char="🐻 ") .. image:: images/ReleaseNotes-Bears_in_plotchar.png

- New `bordercolor` argument of the `plotcandle()` function allows you to change the color of candles' borders:

//@version=4 study("My Script") plotcandle(open, high, low, close, title='Title', color = open < close ? color.green : color.red, wickcolor=color.black, bordercolor=color.orange)

- New variables added:
  - `syminfo.description` - returns a description of the current symbol
  - `syminfo.currency` - returns the currency code of the current symbol (EUR, USD, etc.)
  - `syminfo.type` - returns the type of the current symbol (stock, futures, index, etc.)

September 2019

New parameters to the `strategy` function were added:

- `process_orders_on_close` allows the broker emulator to try to execute orders after calculating the strategy at the bar's close
- `close_entries_rule` allows to define the sequence used for closing positions

Some fixes were made:

- `fill()` function now works correctly with `na` as the `color` parameter value
- `sign()` function now calculates correctly for literals and constants

`str.replace_all(source, target, replacement)` function was added. It replaces each occurrence of a `target` string in the `source` string with a `replacement` string

July-August 2019

New variables added:

- `timeframe.isseconds` returns true when current resolution is in seconds
- `timeframe.isminutes` returns true when current resolution is in minutes
- `time_close` returns the current bar's close time

The behavior of some functions, variables and operators has changed:

- The `time` variable returns the correct open time of the bar for more special cases than before
- An optional *seconds* parameter of the `timestamp()` function allows you to set the time to within seconds
- `security()` function:
    - Added the possibility of requesting resolutions in seconds:

      1, 5, 15, 30 seconds (chart resolution should be less than or equal to the requested resolution)
    - Reduced the maximum value that can be requested in some of the other resolutions:

      from 1 to 1440 minutes

      from 1 to 365 days

      from 1 to 52 weeks

      from 1 to 12 months
- Changes to the evaluation of ternary operator branches:

  In Pine Script v3, during the execution of a ternary operator, both its branches are calculated, so when this script is added to the chart, a long position is opened, even if the long() function is not called:

//@version=3 strategy(title = "My Strategy") long() => strategy.entry("long", true, 1, when = open > high[1]) 1 c = 0 c := true ? 1 : long() plot(c) Pine Script v4 contains built-in functions with side effects (``line.new`` and ``label.new``). If calls to these functions are present in both branches of a ternary operator, both function calls would be executed following v3 conventions. Thus, in Pine Script v4, only the branch corresponding to

the evaluated condition is calculated. While this provides a viable solution in some cases, it will modify the behavior of scripts which depended on the fact that both branches of a ternary were evaluated. The solution is to pre-evaluate expressions prior to the ternary operator. The conversion utility takes this requirement into account when converting scripts from v3 to v4, so that script behavior will be identical in v3 and v4.

June 2019

- Support for drawing objects. Added *label* and *line* drawings
- `var` keyword for one time variable initialization
- Type system improvements:
  - *series string* data type
  - functions for explicit type casting
  - syntax for explicit variable type declaration
  - new *input* type forms
- Renaming of built-ins and a version 3 to 4 converter utility
- `max_bars_back` function to control series variables internal history buffer sizes
- Pine Script documentation versioning

2018

October 2018

- To increase the number of indicators available to the whole community, Invite-Only scripts can now be published by Premium users only.

April 2018

- Improved the Strategy Tester by reworking the Maximum Drawdown calculation formula.

2017

August 2017

- With the new argument `show_last` in the plot-type functions, you can restrict the number of bars that the plot is displayed on.

June 2017

- A major script publishing improvement: it is now possible to update your script without publishing a new one via the Update button in the publishing dialog.

May 2017

- Expanded the type system by adding a new type of constants that can be calculated during compilation.

April 2017

- Expanded the keyword argument functionality: it is now possible to use keyword arguments in all built-in functions.

- A new `barstate.isconfirmed` variable has been added to the list of variables that return bar status. It lets you create indicators that are calculated based on the closed bars only.
- The `options` argument for the `input()` function creates an input with a set of options defined by the script's author.

March 2017

- Pine Script v3 is here! Some important changes:
    - Changes to the default behavior of the `security()` function: it can no longer access the future data by default. This can be changes with the `lookahead` parameter.
    - An implicit conversion of boolean values to numeric values was replaced with an implicit conversion of numeric values (integer and float) to boolean values.
    - Self-referenced and forward-referenced variables were removed. Any PineScript code that used those language constructions can be equivalently rewritten using mutable variables.

February 2017

- Several improvements to the strategy tester and the strategy report:
    - New Buy & Hold equity graph – a new graph that lets you compare performance of your strategy versus a “buy and hold”, i.e if you just bought a security and held onto it without trading.
    - Added percentage values to the absolute currency values.
    - Added Buy & Hold Return to display the final value of Buy & Hold Equity based on last price.
    - Added Sharpe Ratio – it shows the relative effectiveness of the investment portfolio (security), a measure that indicates the average return minus the risk-free return divided by the standard deviation of return on an investment.
    - Slippage lets you simulate a situation when orders are filled at a worse price than expected. It can be set through the Properties dialog or through the `slippage` argument in the `strategy()` function.
    - Commission allows yot to add commission for placed orders in percent of order value, fixed price or per contract. The amount of commission paid is shown in the Commission Paid field. The commission size and its type can be set through the Properties dialog or through the `commission_type` and `commission_value` arguments in the `strategy()` function.

2016

December 2016

- Added invite-only scripts. The invite-only indicators are visible in the Community Scripts, but nobody can use them without explicit permission from the author, and only the author can see the source code.

October 2016

- Introduded indicator revisions. Each time an indicator is saved, it gets a new revision, and it is possible to easily switch to any past revision from the Pine Editor.

September 2016

- It is now possible to publish indicators with protected source code. These indicators are available in the public Script Library, and any user can use them, but only the author can see the source code.

July 2016

- Improved the behavior of the `fill()` function: one call can now support several different colors.

March 2016

- Color type variables now have an additional parameter to set default transparency. The transparency can be set with the `color.new()` function, or by adding an alpha-channel value to a hex color code.

February 2016

- Added `for` loops and keywords `break` and `continue`.
- Pine Script now supports mutable variables! Use the `:=` operator to assign a new value to a variable that has already been defined.
- Multiple improvements and bug fixes for strategies.

January 2016

- A new `alertcondition()` function allows for creating custom alert conditions in Pine Script-based indicators.

2015

October 2015

- Pine has graduated to v2! The new version of Pine Script added support for `if` statements, making it easier to write more readable and concise code.

September 2015

- Added backtesting functionality to Pine Script. It is now possible to create trading strategies, i.e. scripts that can send, modify and cancel orders to buy or sell. Strategies allow you to perform backtesting (emulation of strategy trading on historical data) and forward testing (emulation of strategy trading on real-time data) according to your algorithms. Detailed information about the strategy's calculations and the order fills can be seen in the newly added Strategy Tester tab.

July 2015

- A new `editable` parameter allows hiding the plot from the Style menu in the indicator settings so that it is not possible to edit its style. The parameter has been added to all the following functions: all plot-type functions, `barcolor()`, `bgcolor()`, `hline()`, and `fill()`.

June 2015

- Added two new functions to display custom barsets using PineScipt: `plotbar()` and `plotcandle()`.

April 2015

- Added two new shapes to the `plotshape()` function: shape.labelup and shape.labeldown.
- PineScipt Editor has been improved and moved to a new panel at the bottom of the page.
- Added a new `step` argument for the `input()` function, allowing to specify the step size for the indicator's inputs.

March 2015

- Added support for inputs with the `source` type to the `input()` function, allowing to select the data source for the indicator's calculations from its settings.

February 2015

- Added a new `text` argument to `plotshape()` and `plotchar()` functions.
- Added four new shapes to the `plotshape()` function: shape.arrowup, shape.arrowdown, shape.square, shape.diamond.

2014

August 2014

- Improved the script sharing capabilities, changed the layout of the Indicators menu and separated published scripts from ideas.

July 2014

- Added three new plotting functions, `plotshape()`, `plotchar()`, and `plotarrow()` for situations when you need to highlight specific bars on a chart without drawing a line.
- Integrated QUANDL data into Pine Script. The data can be accessed by passing the QUANDL ticker to the `security` function.

June 2014

- Added Pine Script sharing, enabling programmers and traders to share their scripts with the rest of the TradingView community.

April 2014

- Added line wrapping.

February 2014

- Added support for inputs, allowing users to edit the indicator inputs through the properties window, without needing to edit the Pine script.
- Added self-referencing variables.
- Added support for multiline functions.
- Implemented the type-casting mechanism, automatically casting constant and simple float and int values to series when it is required.
- Added several new functions and improved the existing ones:
    - `barssince()` and `valuewhen()` allow you to check conditions on historical data easier.

- The new `barcolor()` function lets you specify a color for a bar based on filling of a certain condition.
- Similar to the `barcolor()` function, the `bgcolor()` function changes the color of the background.
- Reworked the `security()` function, further expanding its functionality.
- Improved the `fill()` function, enabling it to be used more than once in one script.
- Added the `round()` function to round and convert float values to integers.

2013

- The first version of Pine Script is introduced to all TradingView users, initially as an open beta, on December 13th.

Backgrounds

The bgcolor() function changes the color of the script's background. If the script is running in `overlay = true` mode, then it will color the chart's background.

The function's signature is:

```
bgcolor(color, offset, editable, show_last, title, force_overlay) → void
```

Its `color` parameter allows a “series color” to be used for its argument, so it can be dynamically calculated in an expression.

If the correct transparency is not part of the color to be used, it can be be generated using the color.new() function.

Here is a script that colors the background of trading sessions (try it on 30min EURUSD, for example):

//@version=6 indicator("Session backgrounds", overlay = true) // Default color constants using tranparency of 25. BLUE_COLOR = #0050FF40 PURPLE_COLOR = #0000FF40 PINK_COLOR = #5000FF40 NO_COLOR = color(na) // Allow user to change the colors. preMarketColor = input.color(BLUE_COLOR, "Pre-market") regSessionColor = input.color(PURPLE_COLOR, "Pre-market") postMarketColor = input.color(PINK_COLOR, "Pre-market") // Function returns `true` when the bar's time is timeInRange(tf, session) => time(tf, session) != 0 // Function prints a message at the bottom-right of the chart. f_print(_text) => var table _t = table.new(position.bottom_right, 1, 1) table.cell(_t, 0, 0, _text, bgcolor = color.yellow) var chartIs30MinOrLess = timeframe.isseconds or (timeframe.isintraday and timeframe.multiplier <=30) sessionColor = if chartIs30MinOrLess switch timeInRange(timeframe.period, "0400-0930") => preMarketColor timeInRange(timeframe.period, "0930-1600") => regSessionColor timeInRange(timeframe.period, "1600-2000") => postMarketColor => NO_COLOR else f_print("No background is displayed.\nChart timeframe must be <= 30min.") NO_COLOR bgcolor(sessionColor)

Note that:

- The script only works on chart timeframes of 30min or less. It prints an error message when the chart's timeframe is higher than 30min.
- When the if structure's `else` branch is used because the chart's timeframe is incorrect, the local block returns the `NO_COLOR` color so that no background is displayed in that case.
- We first initialize constants using our base colors, which include the `40` transparency in hex notation at the end. 40 in the hexadecimal notation on the reversed 00-FF scale for transparency corresponds to 75 in Pine Script®'s 0-100 decimal scale for transparency.
- We provide color inputs allowing script users to change the default colors we propose.

In our next example, we generate a gradient for the background of a CCI line:

//@version=6 indicator("CCI Background") bullColor = input.color(color.lime, "□", inline = "1") bearColor = input.color(color.fuchsia, "□", inline = "1") // Calculate CCI. myCCI = ta.cci(hlc3, 20) // Get relative position of CCI in last 100 bars, on a 0-100% scale. myCCIPosition = ta.percentrank(myCCI, 100) // Generate a bull gradient when position is 50-100%, bear gradient when position is 0-50%. backgroundColor = if myCCIPosition >= 50 color.from_gradient(myCCIPosition, 50, 100, color.new(bullColor, 75), bullColor) else

color.from_gradient(myCCIPosition, 0, 50, bearColor, color.new(bearColor, 75)) // Wider white line background. plot(myCCI, "CCI", color.white, 3) // Think black line. plot(myCCI, "CCI", color.black, 1) // Zero level. hline(0) // Gradient background. bgcolor(backgroundColor)

Note that:

- We use the ta.cci() built-in function to calculate the indicator value.
- We use the ta.percentrank() built-in function to calculate `myCCIPosition`, i.e., the percentage of past `myCCI` values in the last 100 bars that are below the current value of `myCCI`.
- To calculate the gradient, we use two different calls of the color.from_gradient() built-in: one for the bull gradient when `myCCIPosition` is in the 50-100% range, which means that more past values are below its current value, and another for the bear gradient when `myCCIPosition` is in the 0-49.99% range, which means that more past values are above it.
- We provide inputs so the user can change the bull/bear colors, and we place both color input widgets on the same line using `inline = "1"` in both input.color() calls.
- We plot the CCI signal using two plot() calls to achieve the best contrast over the busy background: the first plot is a 3-pixel wide white background, the second plot() call plots the thin, 1-pixel wide black line.

See the Colors page for more examples of backgrounds.

Bar coloring

The barcolor() function colors bars on the main chart, regardless of whether the script is running in the main chart pane or a separate pane.

The function's signature is:

```
barcolor(color, offset, editable, show_last, title, display) → void
```

The coloring can be conditional because the `color` parameter accepts "series color" arguments.

The following script renders *inside* and *outside* bars in different colors:

//@version=6 indicator("barcolor example", overlay = true) isUp = close > open isDown = close <= open isOutsideUp = high > high[1] and low < low[1] and isUp isOutsideDown = high > high[1] and low < low[1] and isDown isInside = high < high[1] and low > low[1] barcolor(isInside ? color.yellow : isOutsideUp ? color.aqua : isOutsideDown ? color.purple : na)

Note that:

- The na value leaves bars as is.
- In the barcolor() call, we use embedded ?: ternary operator expressions to select the color.

Bar plotting

Introduction

The plotcandle() built-in function is used to plot candles. plotbar() is used to plot conventional bars.

Both functions require four arguments that will be used for the OHLC prices (open, high, low, close) of the bars they will be plotting. If one of those is na, no bar is plotted.

Plotting candles with `plotcandle()`

The signature of plotcandle() is:

```
plotcandle(open, high, low, close, title, color, wickcolor, editable, show_last
```

This plots simple candles, all in blue, using the habitual OHLC values, in a separate pane:

//@version=6 indicator("Single-color candles") plotcandle(open, high, low, close)

To color them green or red, we can use the following code:

//@version=6 indicator("Example 2") paletteColor = close >= open ? color.lime : color.red plotbar(open, high, low, close, color = paletteColor)

Note that the `color` parameter accepts “series color” arguments, so constant values such as `color.red`, `color.lime`, `"#FF9090"`, as well as expressions that calculate colors at runtime, as is done with the `paletteColor` variable here, will all work.

You can build bars or candles using values other than the actual OHLC values. For example you could calculate and plot smoothed candles using the following code, which also colors wicks depending on the position of close relative to the smoothed close (`c`) of our indicator:

//@version=6 indicator("Smoothed candles", overlay = true) lenInput = input.int(9) smooth(source, length) => ta.sma(source, length) o = smooth(open, lenInput) h = smooth(high, lenInput) l = smooth(low, lenInput) c = smooth(close, lenInput) ourWickColor = close > c ? color.green : color.red plotcandle(o, h, l, c, wickcolor = ourWickColor)

You may find it useful to plot OHLC values taken from a higher timeframe. You can, for example, plot daily bars on an intraday chart:

// NOTE: Use this script on an intraday chart. //@version=6 indicator("Daily bars", behind_chart = false, overlay = true) // Use gaps to return data only when the 1D timeframe completes, and to return `na` otherwise. [o, h, l, c] = request.security(syminfo.tickerid, "D", [open, high, low, close], gaps = barmerge.gaps_on) const color UP_COLOR = color.silver const color DN_COLOR = color.blue color wickColor = c >= o ? UP_COLOR : DN_COLOR color bodyColor = c >= o ? color.new(UP_COLOR, 70) : color.new(DN_COLOR, 70) // Plot candles on intraday timeframes, // and when non `na` values are returned by `request.security()` because a HTF bar has completed. plotcandle(timeframe.isintraday ? o : na, h, l, c, color = bodyColor, wickcolor = wickColor)

Note that:

- We set the `behind_chart` parameter of the indicator() declaration to `false`. This causes our script's candles to appear on top of the chart's candles. Selecting "Visual Order/Bring to Front" from the script's "More" menu achieves the same result.
- The script displays candles only when two conditions are met:
    - The chart is using an intraday timeframe (see the check on `timeframe.isintraday` in the plotcandle() call). We do this because it's not useful to show a daily value on timeframes higher or equal to 1D.
    - The request.security() function returns non na values (see `gaps = barmerge.gaps_on` in the function call).
- We use a tuple (`[open, high, low, close]`) with request.security() to fetch four values in one call.
- We create a lighter transparency for the body of our candles in the `bodyColor` variable initialization, so they don't obstruct the chart's candles.

Plotting bars with `plotbar()`

The signature of plotbar() is:

```
plotbar(open, high, low, close, title, color, editable, show_last, display, for
```

Note that plotbar() has no parameter for `bordercolor` or `wickcolor`, as there are no borders or wicks on conventional bars.

This plots conventional bars using the same coloring logic as in the second example of the previous section:

//@version=6 indicator("Dual-color bars") paletteColor = close >= open ? color.lime : color.red plotbar(open, high, low, close, color = paletteColor)

Colors

Introduction

Script visuals can play a critical role in the usability of the indicators we write in Pine Script®. Well-designed plots and drawings make indicators easier to use and understand. Good visual designs establish a visual hierarchy that allows the more important information to stand out, and the less important one to not get in the way.

Using colors in Pine can be as simple as you want, or as involved as your concept requires. The 4,294,967,296 possible assemblies of color and transparency available in Pine Script can be applied to:

- Any element you can plot or draw in an indicator's visual space, be it lines, fills, text or candles.
- The background of a script's visual space, whether the script is running in its own pane, or in overlay mode on the chart.
- The color of bars or the body of candles appearing on a chart.

A script can only color the elements it places in its own visual space. The only exception to this rule is that a pane indicator can color chart bars or candles.

Pine Script has built-in colors such as color.green, as well as functions like color.rgb() which allow you to dynamically generate any color in the RGBA color space.

Transparency

Each color in Pine Script is defined by four values:

- Its red, green and blue components (0-255), following the RGB color model.
- Its transparency (0-100), often referred to as the Alpha channel outside Pine, as defined in the RGBA color model. Even though transparency is expressed in the 0-100 range, its value can be a “float” when used in functions, which gives you access to the 256 underlying values of the alpha channel.

The transparency of a color defines how opaque it is: zero is fully opaque, 100 makes the color — whichever it is — invisible. Modulating transparency can be crucial in more involved color visuals or when using backgrounds, to control which colors dominate the others, and how they mix together when superimposed.

Constant colors

There are 17 built-in colors in Pine Script. This table lists their names, hexadecimal equivalent, and RGB values as arguments to color.rgb():

Name	Hex	RGB values
color.aqua	#00BCD4	color.rgb(0, 188, 212)
color.black	#363A45	color.rgb(54, 58, 69)
color.blue	#2196F3	color.rgb(33, 150, 243)
color.fuchsia	#E040FB	color.rgb(224, 64, 251)
color.gray	#787B86	color.rgb(120, 123, 134)

Name	Hex	RGB values
color.green	#4CAF50	color.rgb(76, 175, 80)
color.lime	#00E676	color.rgb(0, 230, 118)
color.maroon	#880E4F	color.rgb(136, 14, 79)
color.navy	#311B92	color.rgb(49, 27, 146)
color.olive	#808000	color.rgb(128, 128, 0)
color.orange	#FF9800	color.rgb(255, 152, 0)
color.purple	#9C27B0	color.rgb(156, 39, 176)
color.red	#F23645	color.rgb(242, 54, 69)
color.silver	#B2B5BE	color.rgb(178, 181, 190)
color.teal	#089981	color.rgb(8, 153, 129)
color.white	#FFFFFF	color.rgb(255, 255, 255)
color.yellow	#FDD835	color.rgb(253, 216, 53)

The following script shows three different ways to express color.olive with 40% transparency. All three methods are functionally equivalent:

//@version=6 indicator("Constant colors demo", overlay = true) // Create a plot using the hex color code equivalent for `color.olive` with `99` as the alpha value (60% opacity). plot(ta.sma(close, 10), "10-bar SMA", #80800099, 3) // Create a plot using `color.new()` to modify `color.olive` with 40% transparency. plot(ta.sma(close, 30), "30-bar SMA", color.new(color.olive, 40), 3) // Create a plot using `color.rgb()` with the `r`, `g`, and `b` components of `color.olive` and 40% transparency. plot(ta.sma(close, 50), "50-bar SMA", color.rgb(128, 128, 0, 40), 3)

Note that:

- An alpha value of `99` in a hexadecimal color code is equivalent to 60% opacity, meaning the resulting color is 40% transparent.
- Transparency does *not* affect plot outputs in the status line, price scale, or Data Window. All these locations show the color with 0% transparency.

The colors in the previous script do not vary as the script executes bar to bar. Sometimes, however, colors need to be created as the script executes on each bar because they depend on conditions that are unknown at compile time, or when the script begins execution on bar zero. For those cases, programmers have two options:

1. Use conditional statements to select colors from a few pre-determined base colors.
2. Build new colors dynamically, by calculating them as the script executes bar to bar, to implement color gradients, for example.

Conditional coloring

Let's say you want to color a moving average in different colors, depending on some conditions you define. To do so, you can use a conditional statement that will select a different color for each of your states. Let's start by coloring a moving average in a bull color when it's rising, and in a bear color when it's not:

//@version=6 indicator("Conditional colors", "", true) int lengthInput = input.int(20, "Length", minval = 2) color maBullColorInput = input.color(color.green, "Bull") color maBearColorInput = input.color(color.maroon, "Bear") float ma = ta.sma(close, lengthInput) // Define our states. bool maRising = ta.rising(ma, 1) // Build our color. color maColor = maRising ? maBullColorInput : maBearColorInput plot(ma, "MA", maColor, 2)

Note that:

- We provide users of our script a selection of colors for our bull/bear colors.
- We define an `maRising` boolean variable which will hold `true` when the moving average is higher on the current bar than it was on the last.
- We define an `maColor` variable that is assigned one of our two colors, depending on the value of the `maRising` variable. We use the ternary operator to define our conditional expression.

You can also use conditional colors to avoid plotting under certain conditions. Here, we plot high and low pivots using a line, but we do not want to plot anything when a new pivot comes in, to avoid the joints that would otherwise appear in pivot transitions. To do so, we test for pivot changes and use na as the color value when a change is detected, so that no line is plotted on that bar:

//@version=6 indicator("Conditional colors", "", true) int legsInput = input.int(5, "Pivot Legs", minval = 1) color pHiColorInput = input.color(color.olive, "High pivots") color pLoColorInput = input.color(color.orange, "Low pivots") // Intialize the pivot level variables. var float pHi = na var float pLo = na // When a new pivot is detected, save its value. pHi := nz(ta.pivothigh(legsInput, legsInput), pHi) pLo := nz(ta.pivotlow(legsInput, legsInput), pLo) // When a new pivot is detected, do not plot a color. plot(pHi, "High", ta.change(pHi) != 0 ? na : pHiColorInput, 2, plot.style_line) plot(pLo, "Low", ta.change(pLo) != 0 ? na : pLoColorInput, 2, plot.style_line)

To undertand how this code works, one must first know that ta.pivothigh() and ta.pivotlow(), used as they are here without an argument to the `source` parameter, will return a value when they find a high/low pivot, otherwise they return na.

When we test the value returned by the pivot function for na using the nz() function, we allow the value returned to be assigned to the `pHi` or `pLo` variables only when it is not na, otherwise the previous value of the variable is simply reassigned to it, which has no impact on its value. Keep in mind that previous values of `pHi` and `pLo` are preserved bar to bar because we use the var keyword when initializing them, which causes the initialization to only occur on the first bar.

All that's left to do next is, when we plot our lines, to insert a ternary conditional statement that will yield na for the color when the pivot value changes, or the color selected in the script's inputs when the pivot level does not change.

Calculated colors

Using functions like color.new(), color.rgb() and color.from_gradient(), one can build colors on the fly, as the script executes bar to bar.

color.new() is most useful when you need to generate different transparency levels from a base color.

color.rgb() is useful when you need to build colors dynamically from red, green, blue, or tranparency components. While color.rgb() creates a color, its sister functions color.r(), color.g(), color.b() and color.t()

can be used to extract the red, green, blue or transparency values from a color, which can in turn be used to generate a variant.

color.from_gradient() is useful to create linear gradients between two base colors. It determines which intermediary color to use by evaluating a source value against minimum and maximum values.

color.new()

Let's put color.new() to use to create different transparencies for volume columns using one of two bull/bear base colors:

//@version=6 indicator("Volume") // We name our color constants to make them more readable. var color GOLD_COLOR = #CCCC00ff var color VIOLET_COLOR = #AA00FFff color bullColorInput = input.color(GOLD_COLOR, "Bull") color bearColorInput = input.color(VIOLET_COLOR, "Bear") int levelsInput = input.int(10, "Gradient levels", minval = 1) // We initialize only once on bar zero with `var`, otherwise the count would reset to zero on each bar. var float riseFallCnt = 0 // Count the rises/falls, clamping the range to: 1 to `i_levels`. riseFallCnt := math.max(1, math.min(levelsInput, riseFallCnt + math.sign(volume - nz(volume[1])))) // Rescale the count on a scale of 80, reverse it and cap transparency to <80 so that colors remains visible. float transparency = 80 - math.abs(80 * riseFallCnt / levelsInput) // Build the correct transparency of either the bull or bear color. color volumeColor = color.new(close > open ? bullColorInput : bearColorInput, transparency) plot(volume, "Volume", volumeColor, 1, plot.style_columns)

Note that:

- In the next to last line of our script, we dynamically calculate the column color by varying both the base color used, depending on whether the bar is up or down, **and** the transparency level, which is calculated from the cumulative rises or falls of volume.
- We offer the script user control over not only the base bull/bear colors used, but also on the number of brightness levels we use. We use this value to determine the maximum number of rises or falls we will track. Giving users the possiblity to manage this value allows them to adapt the indicator's visuals to the timeframe or market they use.
- We take care to control the maximum level of transparency we use so that it never goes higher than 80. This ensures our colors always retain some visibility.
- We also set the minimum value for the number of levels to 1 in the inputs. When the user selects 1, the volume columns will be either in bull or bear color of maximum brightness — or transparency zero.

color.rgb()

In our next example we use color.rgb() to build colors from RGBA values. We use the result in a holiday season gift for our friends, so they can bring their TradingView charts to parties:

//@version=6 indicator("Holiday candles", "", true) float r = math.random(0, 255) float g = math.random(0, 255) float b = math.random(0, 255) float t = math.random(0, 100) color holidayColor = color.rgb(r, g, b, t) plotcandle(open, high, low, close, color = holidayColor, wickcolor = holidayColor, bordercolor = holidayColor)

Note that:

- We generate values in the zero to 255 range for the red, green and blue channels, and in the zero to 100 range for transparency. Also note that because math.random() returns float values, the float 0.0-100.0 range provides access to the full 0-255 transparency values of the underlying alpha channel.

- We use the math.random(min, max, seed) function to generate pseudo-random values. We do not use an argument for the third parameter of the function: `seed`. Using it is handy when you want to ensure the repeatability of the function's results. Called with the same seed, it will produce the same sequence of values.

color.from_gradient()

Our last examples of color calculations will use color.from_gradient(). Let's first use it in its simplest form, to color a CCI signal in a version of the indicator that otherwise looks like the built-in:

//@version=6 indicator(title="CCI line gradient", precision=2, timeframe="") var color GOLD_COLOR = #CCCC00 var color VIOLET_COLOR = #AA00FF var color BEIGE_COLOR = #9C6E1B float srcInput = input.source(close, title="Source") int lenInput = input.int(20, "Length", minval = 5) color bullColorInput = input.color(GOLD_COLOR, "Bull") color bearColorInput = input.color(BEIGE_COLOR, "Bear") float signal = ta.cci(srcInput, lenInput) color signalColor = color.from_gradient(signal, -200, 200, bearColorInput, bullColorInput) plot(signal, "CCI", signalColor) bandTopPlotID = hline(100, "Upper Band", color.silver, hline.style_dashed) bandBotPlotID = hline(-100, "Lower Band", color.silver, hline.style_dashed) fill(bandTopPlotID, bandBotPlotID, color.new(BEIGE_COLOR, 90), "Background")

Note that:

- To calculate the gradient, color.from_gradient() requires minimum and maximum values against which the argument used for the `value` parameter will be compared. The fact that we want a gradient for an unbounded signal like CCI (i.e., without fixed boundaries such as RSI, which always oscillates between 0-100), does not entail we cannot use color.from_gradient(). Here, we solve our conundrum by providing values of -200 and 200 as arguments. They do not represent the real minimum and maximum values for CCI, but they are at levels from which we do not mind the colors no longer changing, as whenever the series is outside the `bottom_value` and `top_value` limits, the colors used for `bottom_color` and `top_color` will apply.
- The color progression calculated by color.from_gradient() is linear. If the value of the series is halfway between the `bottom_value` and `top_value` arguments, the generated color's RGBA components will also be halfway between those of `bottom_color` and `top_color`.
- Many common indicator calculations are available in Pine Script as built-in functions. Here we use ta.cci() instead of calculating it the long way.

The argument used for `value` in color.from_gradient() does not necessarily have to be the value of the line we are calculating. Anything we want can be used, as long as arguments for `bottom_value` and `top_value` can be supplied. Here, we enhance our CCI indicator by coloring the band using the number of bars since the signal has been above/below the centerline:

//@version=6 indicator(title="CCI line gradient", precision=2, timeframe="") var color GOLD_COLOR = #CCCC00 var color VIOLET_COLOR = #AA00FF var color GREEN_BG_COLOR = color.new(color.green, 70) var color RED_BG_COLOR = color.new(color.maroon, 70) float srcInput = input.source(close, "Source") int lenInput = input.int(20, "Length", minval = 5) int stepsInput = input.int(50, "Gradient levels", minval = 1) color bullColorInput = input.color(GOLD_COLOR, "Line: Bull", inline = "11") color bearColorInput = input.color(VIOLET_COLOR, "Bear", inline = "11") color bullBgColorInput = input.color(GREEN_BG_COLOR, "Background: Bull", inline = "12") color bearBgColorInput = input.color(RED_BG_COLOR, "Bear", inline = "12") // Plot colored signal line. float signal = ta.cci(srcInput, lenInput) color signalColor = color.from_gradient(signal, -200, 200, color.new(bearColorInput, 0),

color.new(bullColorInput, 0)) plot(signal, "CCI", signalColor, 2) // Detect crosses of the centerline. bool signalX = ta.cross(signal, 0) // Count no of bars since cross. Capping it to the no of steps from inputs. int gradientStep = math.min(stepsInput, nz(ta.barssince(signalX))) // Choose bull/bear end color for the gradient. color endColor = signal > 0 ? bullBgColorInput : bearBgColorInput // Get color from gradient going from no color to `endColor` color bandColor = color.from_gradient(gradientStep, 0, stepsInput, na, endColor) bandTopPlotID = hline(100, "Upper Band", color.silver, hline.style_dashed) bandBotPlotID = hline(-100, "Lower Band", color.silver, hline.style_dashed) fill(bandTopPlotID, bandBotPlotID, bandColor, title = "Band")

Note that:

- The signal plot uses the same base colors and gradient as in our previous example. We have however increased the width of the line from the default 1 to 2. It is the most important component of our visuals; increasing its width is a way to give it more prominence, and ensure users are not distracted by the band, which has become busier than it was in its original, flat beige color.
- The fill must remain unobtrusive for two reasons. First, it is of secondary importance to the visuals, as it provides complementary information, i.e., the duration for which the signal has been in bull/bear territory. Second, since fills have a greater z-index than plots, the fill will cover the signal plot. For these reasons, we make the fill's base colors fairly transparent, at 70, so they do not mask the plots. The gradient used for the band starts with no color at all (see the na used as the argument to `bottom_color` in the color.from_gradient() call), and goes to the base bull/bear colors from the inputs, which the conditional `endColor` variable contains.
- We provide users with distinct bull/bear color selections for the line and the band.
- When we calculate the `gradientStep` variable, we use nz() on ta.barssince() because in early bars of the dataset, when the condition tested has not occurred yet, ta.barssince() will return na. Because we use nz(), the value returned is replaced with zero in those cases.

Mixing transparencies

In this example we take our CCI indicator in another direction. We will build dynamically adjusting extremes zone buffers using a Donchian Channel (historical highs/lows) calculated from the CCI. We build the top/bottom bands by making them 1/4 the height of the DC. We will use a dynamically adjusting lookback to calculate the DC. To modulate the lookback, we will calculate a simple measure of volatility by keeping a ratio of a short-period ATR to a long one. When that ratio is higher than 50 of its last 100 values, we consider the volatility high. When the volatility is high/low, we decrease/increase the lookback.

Our aim is to provide users of our indicator with:

- The CCI line colored using a bull/bear gradient, as we illustrated in our most recent examples.
- The top and bottom bands of the Donchian Channel, filled in such a way that their color darkens as a historical high/low becomes older and older.
- A way to appreciate the state of our volatility measure, which we will do by painting the background with one color whose intensity increases when volatility increases.

This is what our indicator looks like using the light theme:

And with the dark theme:

//@version=6 indicator("CCI DC", precision = 6) color GOLD_COLOR = #CCCC00ff color VIOLET_COLOR = #AA00FFff int lengthInput = input.int(20, "Length", minval = 5) color bullColorInput = input.color(GOLD_COLOR, "Bull") color bearColorInput = input.color(VIOLET_COLOR, "Bear") //

————— Function clamps `val` between `min` and `max`. clamp(val, min, max) => math.max(min, math.min(max, val)) // ————— Volatility expressed as 0-100 value. float v = ta.atr(lengthInput / 5) / ta.atr(lengthInput * 5) float vPct = ta.percentrank(v, lengthInput * 5) // ————— Calculate dynamic lookback for DC. It increases/decreases on low/high volatility. bool highVolatility = vPct > 50 var int lookBackMin = lengthInput * 2 var int lookBackMax = lengthInput * 10 var float lookBack = math.avg(lookBackMin, lookBackMax) lookBack += highVolatility ? -2 : 2 lookBack := clamp(lookBack, lookBackMin, lookBackMax) // ————— Dynamic lookback length Donchian channel of signal. float signal = ta.cci(close, lengthInput) // `lookBack` is a float; need to cast it to int to be used a length. float hiTop = ta.highest(signal, int(lookBack)) float loBot = ta.lowest(signal, int(lookBack)) // Get margin of 25% of the DC height to build high and low bands. float margin = (hiTop - loBot) / 4 float hiBot = hiTop - margin float loTop = loBot + margin // Center of DC. float center = math.avg(hiTop, loBot) // ————— Create colors. color signalColor = color.from_gradient(signal, -200, 200, bearColorInput, bullColorInput) // Bands: Calculate transparencies so the longer since the hi/lo has changed, // the darker the color becomes. Cap highest transparency to 90. float hiTransp = clamp(100 - (100 * math.max(1, nz(ta.barssince(ta.change(hiTop) != 0) + 1)) / 255), 60, 90) float loTransp = clamp(100 - (100 * math.max(1, nz(ta.barssince(ta.change(loBot) != 0) + 1)) / 255), 60, 90) color hiColor = color.new(bullColorInput, hiTransp) color loColor = color.new(bearColorInput, loTransp) // Background: Rescale the 0-100 range of `vPct` to 0-25 to create 75-100 transparencies. color bgColor = color.new(color.gray, 100 - (vPct / 4)) // ————— Plots // Invisible lines for band fills. hiTopPlotID = plot(hiTop, color = na) hiBotPlotID = plot(hiBot, color = na) loTopPlotID = plot(loTop, color = na) loBotPlotID = plot(loBot, color = na) // Plot signal and centerline. p_signal = plot(signal, "CCI", signalColor, 2) plot(center, "Centerline", color.silver, 1) // Fill the bands. fill(hiTopPlotID, hiBotPlotID, hiColor) fill(loTopPlotID, loBotPlotID, loColor) // ————— Background. bgcolor(bgColor)

Note that:

- We clamp the transparency of the background to a 100-75 range so that it doesn't overwhelm. We also use a neutral color that will not distract too much. The darker the background is, the higher our measure of volatility.
- We also clamp the transparency values for the band fills between 60 and 90. We use 90 so that when a new high/low is found and the gradient resets, the starting transparency makes the color somewhat visible. We do not use a transparency lower than 60 because we don't want those bands to hide the signal line.
- We use the very handy ta.percentrank() function to generate a 0-100 value from our ATR ratio measuring volatility. It is useful to convert values whose scale is unknown into known values that can be used to produce transparencies.
- Because we must clamp values three times in our script, we wrote an `f_clamp()` function, instead of explicitly coding the logic three times.

Tips

Maintaining automatic color selectors

Under certain conditions, Pine Script can automatically display all of the colors used in a script's plots in the "Settings/Style" menu. These plots are graphics created by all `plot*()` functions, barcolor(), and bgcolor(). The user can change the colors using a color picker. This feature allows colors in scripts to be customized without any extra code.

For example, this simple script has a plot() that is colored either teal or red, depending on the relationship between the bar's close and open. The script does not specify that these colors should be editable, nor does it create any color-related inputs. Nevertheless, Pine Script automatically displays the colors in the "Settings/Style" menu and allows the user to change them, along with the style of the plot:

//@version=6 indicator("Color picker showcase") plotColor = close > open ? color.teal : color.red plot(close, color = plotColor)

> TipTo prevent the user from changing the color or the type of a plot from a script's "Settings/Style" tab, include `editable = false` in the plot() call.

The colors in the above script can be automatically displayed in this way because they are *not dynamically calculated* and are known as soon as the script has finished compiling. All colors of the "const" type, and all colors of type "input" that are *not modified* via the color.new() or color.rgb() functions can be automatically displayed like this.

> NoteIf color.new() and color.rgb() functions use a "const color" and other "const" parameters, color modifications are calculated during the script's compilation, not at its runtime. As a result, they are available when the script finishes compiling and can be displayed in the "Settings/Style" tab.

However, if *even a single calculated color* is of type "simple color" or "series color", or if an "input color" is passed to color.new() or color.rgb(), *all* colors are calculated in the script's runtime, and no color pickers are available in the "Style" section.

In practice, the creation of "simple" or "series" colors is also most often due to using color.new() and color.rgb() functions. The qualifier of the color that these functions return is the strongest qualifier of the values passed to them. If each call to these functions passes only "const" values, the resulting colors are also "const", and the script *does* display them in the "Style" menu.

> NoticeThe color.from_gradient() function always returns a "series color" value, regardless of the parameters passed to it. If it's used in a script, all of the script's colors are calculated at runtime.

For example, let's try to make the plots in the script above semi-transparent by adding a transparency of `50` to its colors via color.new(). The easiest way to do this is to wrap the `plotColor` variable with color.new(), like in the example below:

//@version=6 indicator("Color picker showcase") plotColor = color.new(close > open ? color.teal : color.red, 50) plot(close, color = plotColor)

Unfortunately, with these changes the "Style" tab does not display a color picker any longer. This is because we use the "series bool" condition `close > open` to decide the color, and then pass the result of this expression to a single color.new() call. The qualified type of the calculated color that it returns is "series color".

To avoid this, we can ensure that every calculated color created by color.new() is a “const color”. Below, we wrap color.teal and color.red separately with color.new() — creating two constant calculated colors in the process — and then decide which one to assign to `plotColor` based on the condition. And while the `plotColor` variable is a “series color”, each color.new() call returns a constant color, so the script displays a color picker in the “Style” tab:

//@version=6 indicator("Color picker showcase") plotColor = close > open ? color.new(color.teal, 50) : color.new(color.red, 50) plot(close, color = plotColor)

To calculate the colors at runtime, create custom color inputs for all of the colors that are to be editable. This approach requires more effort, but allows significantly more control over what the user can affect. Learn more about creating color inputs on the Inputs page.

Designing usable colors schemes

If you write scripts intended for other traders, try to avoid colors that will not work well in some environments, whether it be for plots, labels, tables or fills. At a minimum, test your visuals to ensure they perform satisfactorily with both the light and dark TradingView themes; they are the most commonly used. Colors such as black and white, for example, should be avoided.

Build the appropriate inputs to provide script users the flexibility to adapt your script's visuals to their particular environments.

Take care to build a visual hierarchy of the colors you use that matches the relative importance of your script's visual components. Good designers understand how to achieve the optimal balance of color and weight so the eye is naturally drawn to the most important elements of the design. When you make everything stand out, nothing does. Make room for some elements to stand out by toning down the visuals surrounding it.

Providing a selection of color presets in your inputs — rather than a single color that can be changed — can help color-challenged users. Our Technical Ratings demonstrates one way of achieving this.

Plot crisp lines

It is best to use zero transparency to plot the important lines in your visuals, to keep them crisp. This way, they will show through fills more precisely. Keep in mind that fills have a higher z-index than plots, so they are placed on top of them. A slight increase of a line's width can also go a long way in making it stand out.

If you want a special plot to stand out, you can also give it more importance by using multiple plots for the same line. These are examples where we modulate the successive width and transparency of plots to achieve this:

//@version=6 indicator("") plot(high, "", color.new(color.orange, 80), 8) plot(high, "", color.new(color.orange, 60), 4) plot(high, "", color.new(color.orange, 00), 1) plot(hl2, "", color.new(color.orange, 60), 4) plot(hl2, "", color.new(color.orange, 00), 1) plot(low, "", color.new(color.orange, 0), 1)

Customize gradients

When building gradients, adapt them to the visuals they apply to. If you are using a gradient to color candles, for example, it is usually best to limit the number of steps in the gradient to ten or less, as it is more difficult for the eye to perceive intensity variations of discrete objects. As we did in our examples, cap

minimum and maximum transparency levels so your visual elements remain visible and do not overwhelm when it's not necessary.

Fills

Introduction

Some of Pine Script's visual outputs, including plots, hlines, lines, boxes, and polylines, allow one to fill the chart space they occupy with colors. Three different mechanisms facilitate filling the space between such outputs:

- The fill() function fills the space between two plots from plot() calls or two horizontal lines (hlines) from hline() calls with a specified color.
- Objects of the linefill type fill the space between line instances created with line.new().
- Other drawing types, namely boxes and polylines, have built-in properties that allow the drawings to fill the visual spaces they occupy.

`plot()` and `hline()` fills

The fill() function fills the space between two plots or horizontal lines. It has the following two signatures:

```
fill(plot1, plot2, color, title, editable, show_last, fillgaps) → void
fill(hline1, hline2, color, title, editable, fillgaps) → void
```

The `plot1`, `plot2`, `hline1`, and `hline2` parameters accept plot or hline IDs returned by plot() and hline() function calls. The fill() function is the only built-in that can use these IDs.

This simple example demonstrates how the fill() function works with plot and hline IDs. It calls plot() and hline() three times to display arbitrary values on the chart. Each of these calls returns an ID, which the script assigns to variables for use in the fill() function. The values of `p1`, `p2`, and `p3` are “plot” IDs, whereas `h1`, `h2`, and `h3` reference “hline” IDs:

//@version=6 indicator("Example 1") // Assign "plot" IDs to the `p1`, `p2`, and `p3` variables. p1 = plot(math.sin(high), "Sine of `high`") p2 = plot(math.cos(low), "Cosine of `low`") p3 = plot(math.sin(close), "Sine of `close`") // Fill the space between `p1` and `p2` with 90% transparent red. fill(p1, p3, color.new(color.red, 90), "`p1`-`p3` fill") // Fill the space between `p2` and `p3` with 90% transparent blue. fill(p2, p3, color.new(color.blue, 90), "`p2`-`p3` fill") // Assign "hline" IDs to the `h1`, `h2`, and `h3` variables. h1 = hline(0, "First level") h2 = hline(1.0, "Second level") h3 = hline(0.5, "Third level") h4 = hline(1.5, "Fourth level") // Fill the space between `h1` and `h2` with 90% transparent yellow. fill(h1, h2, color.new(color.yellow, 90), "`h1`-`h2` fill") // Fill the space between `h3` and `h4` with 90% transparent lime. fill(h3, h4, color.new(color.lime, 90), "`h3`-`h4` fill")

It's important to note that the fill() function requires *either* two “plot” IDs or two “hline” IDs. One *cannot* mix and match these types in the function call. Consequently, programmers will sometimes need to use plot() where they otherwise might have used hline() if they want to fill the space between a consistent level and a fluctuating series.

For example, this script calculates an `oscillator` based on the percentage distance between the chart's close price and the 10-bar moving average from a ta.sma() call, then plots it on the chart pane. In this case, we wanted to fill the area between the `oscillator` and zero. Although we can display the zero level with

hline() since its value does not change, we cannot pass a “plot” and “hline” ID to the fill() function. Therefore, we must use a plot() call for the level to allow the script to fill the space:

//@version=6 indicator("Example 2") //@variable The 10-bar moving average of `close` prices. float ma = ta.sma(close, 10) //@variable The distance from the `ma` to the `close` price, as a percentage of the `ma`. float oscillator = 100 * (ma - close) / ma //@variable The ID of the `oscillator` plot for use in the `fill()` function. oscPlotID = plot(oscillator, "Oscillator") //@variable The ID of the zero level plot for use in the `fill()` function. // Requires a "plot" ID since the `fill()` function can't use "plot" and "hline" IDs at the same time. zeroPlotID = plot(0, "Zero level", color.silver, 1, plot.style_circles) // Filll the space between the `oscPlotID` and `zeroPlotID` with 90% transparent blue. fill(oscPlotID, zeroPlotID, color.new(color.blue, 90), "Oscillator fill")

The `color` parameter of the fill() function accepts a “series color” argument, meaning the fill's color can change across chart bars. For example, this code fills the space between two moving average plots with 90% transparent green or red colors based on whether `ma1` is above `ma2`:

//@version=6 indicator("Example 3", overlay = true) //@variable The 5-bar moving average of `close` prices. float ma1 = ta.sma(close, 5) //@variable The 20-bar moving average of `close` prices. float ma2 = ta.sma(close, 20) //@variable The 90% transparent color of the space between MA plots. Green if `ma1 > ma2`, red otherwise. color fillColor = ma1 > ma2 ? color.new(color.green, 90) : color.new(color.red, 90) //@variable The ID of the `ma1` plot for use in the `fill()` function. ma1PlotID = plot(ma1, "5-bar SMA") //@variable The ID of the `ma2` plot for use in the `fill()` function. ma2PlotID = plot(ma2, "20-bar SMA") // Fill the space between the `ma1PlotID` and `ma2PlotID` using the `fillColor`. fill(ma1PlotID, ma2PlotID, fillColor, "SMA plot fill")

Line fills

While the fill() function allows a script to fill the space between two plots or hlines, it does not work with line objects. When a script needs to fill the space between lines, it requires a linefill object created by the linefill.new() function. The function has the following signature:

```
linefill.new(line1, line2, color) → series linefill
```

The `line1` and `line2` parameters accept line IDs. These IDs determine the chart region that the linefill object will fill with its specified `color`. A script can update the `color` property of a linefill ID returned by this function by calling linefill.set_color() with the ID as its `id` argument.

The behavior of linefills depends on the lines they reference. Scripts cannot move linefills directly, as the lines that a linefill uses determine the space it will fill. To retrieve the IDs of the lines referenced by a linefill object, use the linefill.get_line1() and linefill.get_line2() functions.

Any pair of line instances can only have *one* linefill between them. Successive calls to linefill.new() using the same `line1` and `line2` arguments will create a new linefill ID that *replaces* the previous one associated with them.

The example below demonstrates a simple use case for linefills. The script calculates a `pivotHigh` and `pivotLow` series using the built-in ta.pivothigh() and ta.pivotlow() functions with constant `leftbars` and `rightbars` arguments. On the last confirmed historical bar, the script draws two extended lines. The first line

connects the two most recent non-na `pivotHigh` values, and the second connects the most recent non-na `pivotLow` values.

To emphasize the “channel” formed by these lines, the script fills the space between them using linefill.new():

//@version=6 indicator("Linefill demo", "Channel", true) //@variable The number bars to the left of a detected pivot. int LEFT_BARS = 15 //@variable The number bars to the right for pivot confirmation. int RIGHT_BARS = 5 //@variable The price of the pivot high point. float pivotHigh = ta.pivothigh(LEFT_BARS, RIGHT_BARS) //@variable The price of the pivot low point. float pivotLow = ta.pivotlow(LEFT_BARS, RIGHT_BARS) // Initialize the chart points the lines will use. var firstHighPoint = chart.point.new(na, na, na) var secondHighPoint = chart.point.new(na, na, na) var firstLowPoint = chart.point.new(na, na, na) var secondLowPoint = chart.point.new(na, na, na) // Update the `firstHighPoint` and `secondHighPoint` when `pivotHigh` is not `na`. if not na(pivotHigh) firstHighPoint := secondHighPoint secondHighPoint := chart.point.from_index(bar_index - RIGHT_BARS, pivotHigh) // Update the `firstLowPoint` and `secondLowPoint` when `pivotlow` is not `na`. if not na(pivotLow) firstLowPoint := secondLowPoint secondLowPoint := chart.point.from_index(bar_index - RIGHT_BARS, pivotLow) if barstate.islastconfirmedhistory //@variable An extended line that passes through the `firstHighPoint` and `secondHighPoint`. line pivotHighLine = line.new(firstHighPoint, secondHighPoint, extend = extend.right) //@variable An extended line that passes through the `firstLowPoint` and `secondLowPoint`. line pivotLowLine = line.new(firstLowPoint, secondLowPoint, extend = extend.right) //@variable The color of the space between the lines. color fillColor = switch secondHighPoint.price > firstHighPoint.price and secondLowPoint.price > firstLowPoint.price => color.lime secondHighPoint.price < firstHighPoint.price and secondLowPoint.price < firstLowPoint.price => color.red => color.silver //@variable A linefill that colors the space between the `pivotHighLine` and `pivotLowLine`. linefill channelFill = linefill.new(pivotHighLine, pivotLowLine, color.new(fillColor, 90))

Box and polyline fills

The box and polyline types allow scripts to draw geometric shapes and other formations on the chart. Scripts create boxes and polylines with the box.new() and polyline.new() functions, which include parameters that allow the drawings to fill their visual spaces.

To fill the space inside the borders of a box with a specified color, include a `bgcolor` argument in the box.new() function. To fill a polyline's visual space, pass a `fill_color` argument to the polyline.new() function.

For example, this script draws an octagon with a polyline and an inscribed rectangle with a box on the last confirmed historical bar. It determines the size of the drawings using the value from the `radius` variable, which corresponds to approximately one-fourth of the number of bars visible on the chart. We included `fill_color = color.new(color.blue, 60)` in the polyline.new() call to fill the octagon with a translucent blue color, and we used `bgcolor = color.purple` in the box.new() call to fill the inscribed rectangle with opaque purple:

//@version=6 indicator("Box and polyline fills demo") //@variable The number of visible chart bars, excluding the leftmost and rightmost bars. var int barCount = 0 if time > chart.left_visible_bar_time and time < chart.right_visible_bar_time barCount += 1 //@variable The approximate radius used to calculate the octagon and rectangle coordinates. int radius = math.ceil(barCount / 4) if barstate.islastconfirmedhistory //@variable An array of chart points. The polyline uses all points in this array, but the box only needs two.

array<chart.point> points = array.new<chart.point>() //@variable The counterclockwise angle of each point, in radians. Updates on each loop iteration. float angle = 0.0 //@variable The radians to add to the `angle` on each loop iteration. float increment = 0.25 * math.pi // Loop 8 times to calculate octagonal points. for i = 0 to 7 //@variable The point's x-coordinate (bar offset). int x = int(math.round(math.cos(angle) * radius)) //@variable The point's y-coordinate. float y = math.round(math.sin(angle) * radius) // Push a new chart point into the `points` array and increase the `angle`. points.push(chart.point.from_index(bar_index - radius + x, y)) angle += increment // Create a closed polyline to draw the octagon and fill it with translucent blue. polyline.new(points, closed = true, fill_color = color.new(color.blue, 60)) // Create a box for the rectangle using index 3 and 7 for the top-left and bottom-right corners, // and fill it with opaque purple. box.new(points.get(3), points.get(7), bgcolor = color.purple)

See this manual's Lines and boxes page to learn more about working with these types.

Levels

`hline()` levels

Levels are lines plotted using the hline() function. It is designed to plot **horizontal** levels using a **single color**, i.e., it does not change on different bars. See the Levels section of the page on plot() for alternative ways to plot levels when hline() won't do what you need.

The function has the following signature:

```
hline(price, title, color, linestyle, linewidth, editable, display) → hline
```

hline() has a few constraints when compared to plot():

- Since the function's objective is to plot horizontal lines, its `price` parameter requires an “input int/float” argument, which means that “series float” values such as close or dynamically-calculated values cannot be used.
- Its `color` parameter requires an “input color” argument, which precludes the use of dynamic colors, i.e., colors calculated on each bar — or “series color” values.
- Three different line styles are supported through the `linestyle` parameter: hline.style_solid, hline.style_dotted and hline.style_dashed.

Let's see hline() in action in the “True Strength Index” indicator:

//@version=6 indicator("TSI") myTSI = 100 * ta.tsi(close, 25, 13) hline(50, "+50", color.lime) hline(25, "+25", color.green) hline(0, "Zero", color.gray, linestyle = hline.style_dotted) hline(-25, "-25", color.maroon) hline(-50, "-50", color.red) plot(myTSI)

Note that:

- We display 5 levels, each of a different color.
- We use a different line style for the zero centerline.
- We choose colors that will work well on both light and dark themes.
- The usual range for the indicator's values is +100 to -100. Since the ta.tsi() built-in returns values in the +1 to -1 range, we make the adjustment in our code.

Fills between levels

The space between two levels plotted with hline() can be colored using fill(). Keep in mind that **both** plots must have been plotted with hline().

Let's put some background colors in our TSI indicator:

//@version=6 indicator("TSI") myTSI = 100 * ta.tsi(close, 25, 13) plus50Hline = hline(50, "+50", color.lime) plus25Hline = hline(25, "+25", color.green) zeroHline = hline(0, "Zero", color.gray, linestyle = hline.style_dotted) minus25Hline = hline(-25, "-25", color.maroon) minus50Hline = hline(-50, "-50", color.red) // ————— Function returns a color in a light shade for use as a background. fillColor(color col) => color.new(col, 90) fill(plus50Hline, plus25Hline, fillColor(color.lime)) fill(plus25Hline, zeroHline,

fillColor(color.teal)) fill(zeroHline, minus25Hline, fillColor(color.maroon)) fill(minus25Hline, minus50Hline, fillColor(color.red)) plot(myTSI)

Note that:

- We have now used the return value of our hline() function calls, which is of the hline special type. We use the `plus50Hline`, `plus25Hline`, `zeroHline`, `minus25Hline` and `minus50Hline` variables to store those "hline" IDs because we will need them in our fill() calls later.
- To generate lighter color shades for the background colors, we declare a `fillColor()` function that accepts a color and returns its 90 transparency. We use calls to that function for the `color` arguments in our fill() calls.
- We make our fill() calls for each of the four different fills we want, between four different pairs of levels.
- We use color.teal in our second fill because it produces a green that fits the color scheme better than the color.green used for the 25 level.

Lines and boxes

Introduction

Pine Script® facilitates drawing lines, boxes, and other geometric formations from code using the line, box, and polyline types. These types provide utility for programmatically drawing support and resistance levels, trend lines, price ranges, and other custom formations on a chart.

Unlike plots, the flexibility of these types makes them particularly well-suited for visualizing current calculated data at virtually any available point on the chart, irrespective of the chart bar the script executes on.

Lines, boxes, and polylines are *objects*, like labels, tables, and other *special types*. Scripts reference objects of these types using IDs, which act like *pointers*. As with other objects, line, box, and polyline IDs are qualified as “series” values, and all functions that manage these objects accept “series” arguments.

> TipUsing the types we discuss on this page often involves working with *arrays*, especially when creating polylines or managing active drawings on the chart. Therefore, we recommend reading and understanding the Arrays page to make the most of these drawing types in your scripts.

Lines drawn by a script may be vertical, horizontal, or angled. Boxes are always rectangular. Polylines sequentially connect multiple vertical, horizontal, angled, or curved line segments. Although all of these drawing types have different characteristics, they do have some things in common:

- Lines, boxes, and polylines can have coordinates at any available location on the chart, including ones at future times beyond the last chart bar.
- Objects of these types can use chart.point instances to set their coordinates.
- The x-coordinates of each object can be bar index or time values, depending on their specified `xloc` property.
- Each object can have one of multiple predefined line styles.
- Scripts can call the functions that manage these objects from within the scopes of loops and conditional structures, allowing iterative and conditional control of their drawings.
- There are limits on the number of these objects that a script can reference and display on the chart. A single script instance can display up to 500 lines, 500 boxes, and 100 polylines. Users can specify the maximum number allowed for each type via the `max_lines_count`, `max_boxes_count`, and `max_polylines_count` parameters of the script's indicator() or strategy() declaration statement. If unspecified, the default is ~50. As with label and table types, lines, boxes, and polylines utilize a *garbage collection* mechanism that deletes the oldest objects on the chart when the total number of drawings exceeds the script's limit.

> NoteThe Supercharts interface features a set of *drawing tools* that enable users to draw on the chart using mouse actions. Although some of those drawings might resemble the outputs of a script’s drawing objects, it’s crucial to understand that they are **unrelated** entities. Scripts cannot interact with the chart’s drawing tools. Additionally, mouse actions do not directly affect a script’s drawing objects.

Lines

The built-ins in the `line.*` namespace control the creation and management of line objects:

- The line.new() function creates a new line.
- The `line.set_*()` functions modify line properties.
- The `line.get_*()` functions retrieve values from a line instance.
- The line.copy() function clones a line instance.
- The line.delete() function deletes an existing line instance.
- The line.all variable references a read-only array containing the IDs of all lines displayed by the script. The array's size depends on the `max_lines_count` of the indicator() or strategy() declaration statement and the number of lines the script has drawn.

Scripts can call `line.set_*()`, `line.get_*()`, line.copy(), and line.delete() built-ins as functions or methods.

Creating lines

The line.new() function creates a new line instance to display on the chart. It has the following signatures:

```
line.new(first_point, second_point, xloc, extend, color, style, width, force_ov

line.new(x1, y1, x2, y2, xloc, extend, color, style, width, force_overlay) → se
```

The first overload of this function contains the `first_point` and `second_point` parameters. The `first_point` is a chart.point representing the start of the line, and the `second_point` is a chart.point representing the line's end. The function copies the information from these chart points to determine the line's coordinates. Whether it uses the `index` or `time` fields from the `first_point` and `second_point` as x-coordinates depends on the function's `xloc` value.

The second overload specifies `x1`, `y1`, `x2`, and `y2` values independently, where `x1` and `x2` are int values representing the starting and ending x-coordinates of the line, and `y1` and `y2` are float values representing the y-coordinates. Whether the line considers the `x` values as bar indices or timestamps depends on the `xloc` value in the function call.

Both overloads share the same additional parameters:

`xloc`

Controls whether the x-coordinates of the new line use bar index or time values. Its default value is xloc.bar_index.

When calling the first overload, using an `xloc` value of xloc.bar_index tells the function to use the `index` fields of the `first_point` and `second_point`, and a value of xloc.bar_time tells the function to use the `time` fields of the points.

When calling the second overload, an `xloc` value of xloc.bar_index prompts the function to treat the `x1` and `x2` arguments as bar index values. When using xloc.bar_time, the function will treat `x1` and `x2` as time values.

When the specified x-coordinates represent *bar index* values, it's important to note that the minimum x-coordinate allowed is `bar_index - 10000`. For larger offsets, one can use xloc.bar_time.

`extend`

Determines whether the drawn line will infinitely extend beyond its defined start and end coordinates. It accepts one of the following values: extend.left, extend.right, extend.both, or extend.none (default).

`color`

Specifies the color of the line drawing. The default is color.blue.

`style`

Specifies the line's style, which can be any of the options listed in this page's Line styles section. The default value is line.style_solid.

`width`

Controls the width of the line, in pixels. The default value is 1.

`force_overlay`

If `true`, the drawing will display on the main chart pane, even when the script occupies a separate pane. Optional. The default is `false`.

The example below demonstrates how one can draw lines in their simplest form. This script draws a new vertical line connecting the open and close prices at the horizontal center of each chart bar:

//@version=6 indicator("Creating lines demo", overlay = true) //@variable The `chart.point` for the start of the line. Contains `index` and `time` information. firstPoint = chart.point.now(open) //@variable The `chart.point` for the end of the line. Contains `index` and `time` information. secondPoint = chart.point.now(close) // Draw a basic line with a `width` of 5 connecting the `firstPoint` to the `secondPoint`. // This line uses the `index` field from each point for its x-coordinates. line.new(firstPoint, secondPoint, width = 5) // Color the background on the unconfirmed bar. bgcolor(barstate.isconfirmed ? na : color.new(color.orange, 70), title = "Unconfirmed bar highlight")

Note that:

- If the `firstPoint` and `secondPoint` reference identical coordinates, the script will *not* display a line since there is no distance between them to draw. However, the line ID will still exist.
- The script will only display approximately the last 50 lines on the chart, as it does not have a specified `max_lines_count` in the indicator() function call. Line drawings persist on the chart until deleted using line.delete() or removed by the garbage collector.
- The script *redraws* the line on the open chart bar (i.e., the bar with an orange background highlight) until it closes. After the bar closes, it will no longer update the drawing.

Let's look at a more involved example. This script uses the previous bar's hl2 price and the current bar's high and low prices to draw a fan with a user-specified number of lines projecting a range of hypothetical price values for the following chart bar. It calls line.new() within a for loop to create `linesPerBar` lines on each bar:

//@version=6 indicator("Creating lines demo", "Simple projection fan", true, max_lines_count = 500) //@variable The number of fan lines drawn on each chart bar. int linesPerBar = input.int(20, "Line drawings

per bar", 2, 100) //@variable The distance between each y point on the current bar. float step = (high - low) / (linesPerBar - 1) //@variable The `chart.point` for the start of each line. Does not contain `time` information. firstPoint = chart.point.from_index(bar_index - 1, hl2[1]) //@variable The `chart.point` for the end of each line. Does not contain `time` information. secondPoint = chart.point.from_index(bar_index + 1, float(na)) //@variable The stepped y value on the current bar for `secondPoint.price` calculation, starting from the `low`. float barValue = low // Loop to draw the fan. for i = 1 to linesPerBar // Update the `price` of the `secondPoint` using the difference between the `barValue` and `firstPoint.price`. secondPoint.price := 2.0 * barValue - firstPoint.price //@variable Is `color.aqua` when the line's slope is positive, `color.fuchsia` otherwise. color lineColor = secondPoint.price > firstPoint.price ? color.aqua : color.fuchsia // Draw a new `lineColor` line connecting the `firstPoint` and `secondPoint` coordinates. // This line uses the `index` field from each point for its x-coordinates. line.new(firstPoint, secondPoint, color = lineColor) // Add the `step` to the `barValue`. barValue += step // Color the background on the unconfirmed bar. bgcolor(barstate.isconfirmed ? na : color.new(color.orange, 70), title = "Unconfirmed bar highlight")

Note that:

- We've included `max_lines_count = 500` in the indicator() function call, meaning the script preserves up to 500 lines on the chart.
- Each line.new() call *copies* the information from the chart.point referenced by the `firstPoint` and `secondPoint` variables. As such, the script can change the `price` field of the `secondPoint` on each loop iteration without affecting the y-coordinates in other lines.

Modifying lines

The `line.*` namespace contains multiple *setter* functions that modify the properties of line instances:

- line.set_first_point() and line.set_second_point() respectively update the start and end points of the `id` line using information from the specified `point`.
- line.set_x1() and line.set_x2() set one of the x-coordinates of the `id` line to a new `x` value, which can represent a bar index or time value depending on the line's `xloc` property.
- line.set_y1() and line.set_y2() set one of the y-coordinates of the `id` line to a new `y` value.
- line.set_xy1() and line.set_xy2() update one of the `id` line's points with new `x` and `y` values.
- line.set_xloc() sets the `xloc` of the `id` line and updates both of its x-coordinates with new `x1` and `x2` values.
- line.set_extend() sets the `extend` property of the `id` line.
- line.set_color() updates the `id` line's `color` value.
- line.set_style() changes the `style` of the `id` line.
- line.set_width() sets the `width` of the `id` line.

All setter functions directly modify the `id` line passed into the call and do not return any value. Each setter function accepts “series” arguments, as a script can change a line's properties throughout its execution.

The following example draws lines connecting the opening price of a `timeframe` to its closing price. The script uses the var keyword to declare `periodLine` and the variables that reference chart.point objects (`openPoint` and `closePoint`) only on the *first* chart bar, and it assigns new references to these variables over its execution. After detecting a new bar on the specified timeframe with timeframe.change(), the script uses line.set_color() to set the `color` property of the current line referenced by `periodLine`, creates new chart points for `openPoint` and `closePoint` using chart.point.now(), calls line.new() to create another line anchored to those points, then assigns the new line’s reference to `periodLine`.

On other bars where the `periodLine` reference is not na, the script assigns a new chart.point reference to the `closePoint` variable, then uses line.set_second_point() and line.set_color() as methods to update the end coordinate and color of the latest line:

```
//@version=6 indicator("Modifying lines demo", overlay = true) //@variable The size of each period. string timeframe = input.timeframe("D", "Timeframe") //@variable A line connecting the period's opening and closing prices. var line periodLine = na //@variable The first point of the line. Contains `time` and `index` information. var chart.point openPoint = chart.point.now(open) //@variable The closing point of the line. Contains `time` and `index` information. var chart.point closePoint = chart.point.now(close) if timeframe.change(timeframe) //@variable The final color of the `periodLine`. color finalColor = switch closePoint.price > openPoint.price => color.green closePoint.price < openPoint.price => color.red => color.gray // Update the color of the current `periodLine` to the `finalColor`. line.set_color(periodLine, finalColor) // Assign new points to the `openPoint` and `closePoint`. openPoint := chart.point.now(open) closePoint := chart.point.now(close) // Assign a new line to the `periodLine`. Uses `time` fields from the `openPoint` and `closePoint` as x-coordinates. periodLine := line.new(openPoint, closePoint, xloc.bar_time, style = line.style_arrow_right, width = 3) else if not na(periodLine) // Assign a new point to the `closePoint`. closePoint := chart.point.now(close) //@variable The color of the developing `periodLine`. color developingColor = switch closePoint.price > openPoint.price => color.aqua closePoint.price < openPoint.price => color.fuchsia => color.gray // Update the coordinates of the line's second point using the new `closePoint`. // It uses the `time` field from the point for its new x-coordinate. periodLine.set_second_point(closePoint) // Update the color of the line using the `developingColor`. periodLine.set_color(developingColor)
```

Note that:

- Each line drawing in this example uses the line.style_arrow_right style. See the Line styles section below for an overview of all available style settings.

Line styles

Users can control the style of their scripts' line drawings by passing one of the following variables as the `style` argument in their line.new() or line.set_style() function calls:

Argument	Line
`line.style_solid`	
`line.style_dotted`	
`line.style_dashed`	
`line.style_arrow_left`	
`line.style_arrow_right`	
`line.style_arrow_both`	

Note that:

- *Polylines* can also use any of these variables as their `line_style` value. See the Creating polylines section of this page.

Reading line values

The `line.*` namespace includes *getter* functions, which allow a script to retrieve values from a line object for further use:

- line.get_x1() and line.get_x2() respectively get the first and second x-coordinate from the `id` line. Whether the value returned represents a bar index or time value depends on the line's `xloc` property.
- line.get_y1() and line.get_y2() respectively get the `id` line's first and second y-coordinate.
- line.get_price() retrieves the price (y-coordinate) from a line `id` at a specified `x` value, including at bar indices outside the line's start and end points. This function is only compatible with lines that use xloc.bar_index as the `xloc` value.

The script below draws a new line upon the onset of a rising or falling price pattern forming over `length` bars. It uses the var keyword to declare the `directionLine` variable on the first chart bar. The line reference assigned to `directionLine` persists over subsequent bars until the `newDirection` condition occurs, in which case the script assigns a creates a new line with line.new() and assigns that line's reference to the variable.

On every bar, the script calls the line.get_y2(), line.get_y1(), line.get_x2(), and line.get_x1() getters as methods to retrieve values from the current line referenced by `directionLine` and calculate its slope, then uses the result to determine the color of each drawing and plot. The script retrieves an extended value of the current line from *beyond* its second point using line.get_price() and plots the returned value on the chart:

//@version=6 indicator("Reading line values demo", overlay = true) //@variable The number of bars for rising and falling calculations. int length = input.int(2, "Length", 2) //@variable A line that's drawn whenever `hlc3` starts rising or falling over `length` bars. var line directionLine = na //@variable Is `true` when `hlc3` is rising over `length` bars, `false` otherwise. bool rising = ta.rising(hlc3, length) //@variable Is `true` when `hlc3` is falling over `length` bars, `false` otherwise. bool falling = ta.falling(hlc3, length) //@variable Is `true` when a rising or falling pattern begins, `false` otherwise. bool newDirection = (rising and not rising[1]) or (falling and not falling[1]) // Update the `directionLine` when `newDirection` is `true`. The line uses the default `xloc.bar_index`. if newDirection directionLine := line.new(bar_index - length, hlc3[length], bar_index, hlc3, width = 3) //@variable The slope of the `directionLine`. float slope = (directionLine.get_y2() - directionLine.get_y1()) / (directionLine.get_x2() - directionLine.get_x1()) //@variable The value extrapolated from the `directionLine` at the `bar_index`. float lineValue = line.get_price(directionLine, bar_index) //@variable Is `color.green` when the `slope` is positive, `color.red` otherwise. color slopeColor = slope > 0 ? color.green : color.red // Update the color of the `directionLine`. directionLine.set_color(slopeColor) // Plot the `lineValue`. plot(lineValue, "Extrapolated value", slopeColor, 3, plot.style_circles)

Note that:

- This example calls the second overload of the line.new() function, which uses `x1`, `y1`, `x2`, and `y2` parameters to define the start and end points of the line. The `x1` value is `length` bars behind the current bar_index, and the `y1` value is the hlc3 value at that index. The `x2` and `y2` in the function call use the current bar's bar_index and hlc3 values.
- The line.get_price() function call treats the `directionLine` as though it extends infinitely, regardless of its `extend` property.
- The script only displays approximately the last 50 lines on the chart, but the plot of extrapolated values spans throughout the chart's history.

Cloning lines

Scripts can clone a line `id` and all its properties with the line.copy() function. Any changes to the copied line instance do not affect the original.

For example, this script creates a horizontal line at the the bar's open price once every `length` bars, which it assigns to a `mainLine` variable. On all other bars, it creates a `copiedLine` using line.copy() and calls `line.set_*()` functions to modify its properties. As we see below, altering the `copiedLine` does not affect the `mainLine` in any way:

//@version=6 indicator("Cloning lines demo", overlay = true, max_lines_count = 500) //@variable The number of bars between each new mainLine assignment. int length = input.int(20, "Length", 2, 500) //@variable The first `chart.point` used by the `mainLine`. Contains `index` and `time` information. firstPoint = chart.point.now(open) //@variable The second `chart.point` used by the `mainLine`. Does not contain `time` information. secondPoint = chart.point.from_index(bar_index + length, open) //@variable A horizontal line drawn at the `open` price once every `length` bars. var line mainLine = na if bar_index % length == 0 // Assign a new line to the `mainLine` that connects the `firstPoint` to the `secondPoint`. // This line uses the `index` fields from both points as x-coordinates. mainLine := line.new(firstPoint, secondPoint, color = color.purple, width = 2) //@variable A copy of the `mainLine`. Changes to this line do not affect the original. line copiedLine = line.copy(mainLine) // Update the color, style, and second point of the `copiedLine`. line.set_color(copiedLine, color.orange) line.set_style(copiedLine, line.style_dotted) line.set_second_point(copiedLine, chart.point.now(close))

Note that:

- The `index` field of the `secondPoint` is `length` bars beyond the current bar_index. Since the maximum x-coordinate allowed with xloc.bar_index is `bar_index + 500`, we've set the `maxval` of the `length` input to 500.

Deleting lines

To delete a line drawn by a script, use the line.delete() function. This function removes the line instance from the script and its drawing on the chart.

Deleting line instances is often handy when one wants to only keep a specific number of lines on the chart at any given time or conditionally remove drawings as a chart progresses.

For example, this script creates a horizontal line with the `extend` property set to extend.right whenever an RSI crosses its EMA.

The script stores all line IDs in a `lines` array that it uses as a queue to display only a specified number of lines on the chart. When the size of the array exceeds the specified `numberOfLines` value, the script removes the array's oldest line ID using array.shift() and deletes it with line.delete():

//@version=6 //@variable The maximum number of lines allowed on the chart. const int MAX_LINES_COUNT = 500 indicator("Deleting lines demo", "RSI cross levels", max_lines_count = MAX_LINES_COUNT) //@variable The length of the RSI. int rsiLength = input.int(14, "RSI length", 2) //@variable The length of the RSI's EMA. int emaLength = input.int(28, "RSI average length", 2) //@variable The maximum number of lines to keep on the chart. int numberOfLines = input.int(20, "Lines on the chart", 0, MAX_LINES_COUNT) //@variable An array containing the IDs of lines on the chart. var array<line> lines = array.new<line>() //@variable An `rsiLength` RSI of `close`. float rsi = ta.rsi(close, rsiLength) //@variable A `maLength` EMA of the `rsi`. float rsiMA = ta.ema(rsi, emaLength) if ta.cross(rsi, rsiMA) //@variable The color of the horizontal line. color lineColor = rsi > rsiMA ? color.green : color.red // Draw a new horizontal line. Uses the default `xloc.bar_index`. newLine = line.new(bar_index, rsiMA, bar_index + 1, rsiMA, extend = extend.right, color = lineColor, width = 2) // Push the `newLine` into the `lines` array. lines.push(newLine) //

Delete the oldest line when the size of the array exceeds the specified `numberOfLines`. if array.size(lines) > numberOfLines line.delete(lines.shift()) // Plot the `rsi` and `rsiMA`. plot(rsi, "RSI", color.new(color.blue, 40)) plot(rsiMA, "EMA of RSI", color.new(color.gray, 30))

Note that:

- We declared a `MAX_LINES_COUNT` variable with the “const int” *qualified type*, which the script uses as the `max_lines_count` in the indicator() function and the `maxval` of the input.int() assigned to the `numberOfLines` variable.
- This example uses the second overload of the line.new() function, which specifies `x1`, `y1`, `x2`, and `y2` coordinates independently.

Filling the space between lines

Scripts can *fill* the space between two line drawings by creating a linefill object that references them with the linefill.new() function. Linefills automatically determine their fill boundaries using the properties from the `line1` and `line2` IDs that they reference.

For example, this script calculates a simple linear regression channel. On the first chart bar, the script declares the `basisLine`, `upperLine`, and `lowerLine` variables to reference the channel's line IDs, then it makes two linefill.new() calls to create linefill objects that fill the upper and lower portions of the channel. The first linefill fills the space between the `basisLine` and the `upperLine`, and the second fills the space between the `basisLine` and `lowerLine`.

The script updates the coordinates of the lines across subsequent bars. However, notice that the script never needs to update the linefills declared on the first bar. They automatically update their fill regions based on the coordinates of their assigned lines:

//@version=6 indicator("Filling the space between lines demo", "Simple linreg channel", true) //@variable The number of bars in the linear regression calculation. int lengthInput = input.int(100) //@variable The basis line of the regression channel. var line basisLine = line.new(na, na, na, na, extend = extend.right, color = chart.fg_color, width = 2) //@variable The channel's upper line. var line upperLine = line.new(na, na, na, na, extend = extend.right, color = color.teal, width = 2) //@variable The channel's lower line. var line lowerLine = line.new(na, na, na, na, extend = extend.right, color = color.maroon, width = 2) //@variable A linefill instance that fills the space between the `basisLine` and `upperLine`. var linefill upperFill = linefill.new(basisLine, upperLine, color.new(color.teal, 80)) //@variable A linefill instance that fills the space between the `basisLine` and `lowerLine`. var linefill lowerFill = linefill.new(basisLine, lowerLine, color.new(color.maroon, 80)) // Update the `basisLine` coordinates with current linear regression values. basisLine.set_xy1(bar_index + 1 - lengthInput, ta.linreg(close, lengthInput, lengthInput - 1)) basisLine.set_xy2(bar_index, ta.linreg(close, lengthInput, 0)) //@variable The channel's standard deviation. float stDev = 0.0 for i = 0 to lengthInput - 1 stDev += math.pow(close[i] - line.get_price(basisLine, bar_index - i), 2) stDev := math.sqrt(stDev / lengthInput) * 2.0 // Update the `upperLine` and `lowerLine` using the values from the `basisLine` and the `stDev`. upperLine.set_xy1(basisLine.get_x1(), basisLine.get_y1() + stDev) upperLine.set_xy2(basisLine.get_x2(), basisLine.get_y2() + stDev) lowerLine.set_xy1(basisLine.get_x1(), basisLine.get_y1() - stDev) lowerLine.set_xy2(basisLine.get_x2(), basisLine.get_y2() - stDev)

To learn more about the linefill type, see this section of the Fills page.

Boxes

The built-ins in the `box.*` namespace create and manage box objects:

- The box.new() function creates a new box.
- The `box.set_*()` functions modify box properties.
- The `box.get_*()` functions retrieve values from a box instance.
- The box.copy() function clones a box instance.
- The box.delete() function deletes a box instance.
- The box.all variable references a read-only array containing the IDs of all boxes displayed by the script. The array's size depends on the `max_boxes_count` of the indicator() or strategy() declaration statement and the number of boxes the script has drawn.

As with lines, users can call `box.set_*()`, `box.get_*()`, box.copy(), and box.delete() built-ins as functions or methods.

Creating boxes

The box.new() function creates a new box object to display on the chart. It has the following signatures:

```
box.new(top_left, bottom_right, border_color, border_width, border_style, exten

box.new(left, top, right, bottom, border_color, border_width, border_style, ext
```

This function's first overload includes the `top_left` and `bottom_right` parameters, which accept chart.point objects representing the top-left and bottom-right corners of the box, respectively. The function copies the information from these chart points to set the coordinates of the box's corners. Whether it uses the `index` or `time` fields of the `top_left` and `bottom_right` points as x-coordinates depends on the function's `xloc` value.

The second overload specifies `left`, `top`, `right`, and `bottom` edges of the box. The `left` and `right` parameters accept int values specifying the box's left and right x-coordinates, which can be bar index or time values depending on the `xloc` value in the function call. The `top` and `bottom` parameters accept float values representing the box's top and bottom y-coordinates.

The function's additional parameters are identical in both overloads:

`border_color`

Specifies the color of all four of the box's borders. The default is color.blue.

`border_width`

Specifies the width of the borders, in pixels. Its default value is 1.

`border_style`

Specifies the style of the borders, which can be any of the options in the Box styles section of this page.

`extend`

Determines whether the box's borders extend infinitely beyond the left or right x-coordinates. It accepts one of the following values: extend.left, extend.right, extend.both, or extend.none (default).

`xloc`

Determines whether the left and right edges of the box use bar index or time values as x-coordinates. The default is xloc.bar_index.

In the first overload, an `xloc` value of xloc.bar_index means that the function will use the `index` fields of the `top_left` and `bottom_right` chart points, and an `xloc` value of xloc.bar_time means that it will use their `time` fields.

In the second overload, using an `xloc` value of xloc.bar_index means the function treats the `left` and `right` values as bar indices, and xloc.bar_time means it will treat them as timestamps.

When the specified x-coordinates represent *bar index* values, it's important to note that the minimum x-coordinate allowed is `bar_index - 10000`. For larger offsets, one can use xloc.bar_time.

`bgcolor`

Specifies the background color of the space inside the box. The default value is color.blue.

`text`

The text to display inside the box. By default, its value is an empty string.

`text_size`

Specifies the size of the text within the box. It accepts both “int” size values and “string” `size.*` constants. The “int” size can be any positive integer. The `size.*` constants and their equivalent “int” sizes are: size.auto (0), size.tiny (8), size.small (10), size.normal (14), size.large (20), and size.huge (36). The default value is size.auto.

`text_color`

Controls the color of the text. Its default is color.black.

`text_halign`

Specifies the horizontal alignment of the text within the box's boundaries. It accepts one of the following: text.align_left, text.align_right, or text.align_center (default).

`text_valign`

Specifies the vertical alignment of the text within the box's boundaries. It accepts one of the following: text.align_top, text.align_bottom, or text.align_center (default).

`text_wrap`

Determines whether the box will wrap the text within it. If its value is text.wrap_auto, the box wraps the text to ensure it does not span past its vertical borders. It also clips the wrapped text when it extends past the bottom. If the value is text.wrap_none, the box displays the text on a single line that can extend beyond its borders. The default is text.wrap_none.

`text_font_family`

Defines the font family of the box's text. Using font.family_default displays the box's text with the system's default font. The font.family_monospace displays the text in a monospace format. The default value is font.family_default.

`force_overlay`

If `true`, the drawing will display on the main chart pane, even when the script occupies a separate pane. Optional. The default is `false`.

`text_formatting`

Specifies the formatting of the box's text. Using text.format_none displays the text with no special formatting. This parameter also accepts the arguments text.format_bold or text.format_italic. Using `text.format_bold + text.format_italic` applies both formats together to display text that is both bold and italicized. The default value is text.format_none.

Let's write a simple script to display boxes on a chart. The example below draws a box projecting each bar's high and low values from the horizontal center of the current bar to the center of the next available bar.

On each bar, the script creates `topLeft` and `bottomRight` points via chart.point.now() and chart.point.from_index(), then calls box.new() to construct a new box and display it on the chart. It also highlights the background on the unconfirmed chart bar using bgcolor() to indicate that it redraws that box until the bar's last update:

//@version=6 indicator("Creating boxes demo", overlay = true) //@variable The `chart.point` for the top-left corner of the box. Contains `index` and `time` information. topLeft = chart.point.now(high) //@variable The `chart.point` for the bottom-right corner of the box. Does not contain `time` information. bottomRight = chart.point.from_index(bar_index + 1, low) // Draw a box using the `topLeft` and `bottomRight` corner points. Uses the `index` fields as x-coordinates. box.new(topLeft, bottomRight, color.purple, 2, bgcolor = color.new(color.gray, 70)) // Color the background on the unconfirmed bar. bgcolor(barstate.isconfirmed ? na : color.new(color.orange, 70), title = "Unconfirmed bar highlight")

Note that:

- The `bottomRight` point's `index` field is one bar greater than the `index` in the `topLeft`. If the x-coordinates of the corners were equal, the script would draw a vertical line at the horizontal center of each bar, resembling the example in this page's Creating lines section.
- Similar to lines, if the `topLeft` and `bottomRight` contained identical coordinates, the box wouldn't display on the chart since there would be no space between them to draw. However, its ID would still exist.
- This script only displays approximately the last 50 boxes on the chart, as we have not specified a `max_boxes_count` in the indicator() function call.

Modifying boxes

Multiple *setter* functions exist in the `box.*` namespace, allowing scripts to modify the properties of box objects:

- box.set_top_left_point() and box.set_bottom_right_point() respectively update the top-left and bottom-right coordinates of the `id` box using information from the specified `point`.

- box.set_left() and box.set_right() set the left or right x-coordinate of the `id` box to a new `left/right` value, which can be a bar index or time value depending on the box's `xloc` property.
- box.set_top() and box.set_bottom() set the top or bottom y-coordinate of the `id` box to a new `top/bottom` value.
- box.set_lefttop() sets the `left` and `top` coordinates of the `id` box, and box.set_rightbottom() sets its `right` and `bottom` coordinates.
- box.set_xloc() sets the `xloc` property of the `id` box and updates its x-coordinates to new `left` and `right` values, which represent bar index or time values accordingly.
- box.set_border_color(), box.set_border_width() and box.set_border_style() respectively update the `color`, `width`, and `style` of the `id` box's border.
- box.set_extend() sets the horizontal `extend` property of the `id` box.
- box.set_bgcolor() sets the color of the space inside the `id` box to a new `color`.
- box.set_text(), box.set_text_size(), box.set_text_color(), box.set_text_halign(), box.set_text_valign(), box.set_text_wrap(), box.set_text_font_family(), and box.set_text_formatting() update the `id` box's text-related properties.

As with setter functions in the `line.*` namespace, all box setters modify the `id` box directly without returning a value, and each setter function accepts “series” arguments.

This example uses boxes to visualize the ranges of upward and downward bars with the highest volume over a user-defined `timeframe`. When the script detects a new bar on the specified timeframe with timeframe.change(), it assigns new boxes to the `upBox` and `downBox` variables, resets the `upVolume` and `downVolume` values, and highlights the chart background.

When an upward or downward bar's volume exceeds the `upVolume` or `downVolume`, the script updates the volume-tracking variables and calls box.set_top_left_point() and box.set_bottom_right_point() to update the `upBox` or `downBox` coordinates. The setters use the information from the chart.points created with chart.point.now() and chart.point.from_time() to project that bar's high and low values from the current time to the closing time of the specified timeframe:

//@version=6 indicator("Modifying boxes demo", "High volume boxes", true, max_boxes_count = 100) //@variable The timeframe of the calculation. string timeframe = input.timeframe("D", "Timeframe") //@variable A box projecting the range of the upward bar with the highest `volume` over the `timeframe`. var box upBox = na //@variable A box projecting the range of the downward bar with the lowest `volume` over the `timeframe`. var box downBox = na //@variable The highest volume of upward bars over the `timeframe`. var float upVolume = na //@variable The highest volume of downward bars over the `timeframe`. var float downVolume = na // Color variables. var color upBorder = color.teal var color upFill = color.new(color.teal, 90) var color downBorder = color.maroon var color downFill = color.new(color.maroon, 90) //@variable The closing time of the `timeframe`. int closeTime = time_close(timeframe) //@variable Is `true` when a new bar starts on the `timeframe`. bool changeTF = timeframe.change(timeframe) //@variable The `chart.point` for the top-left corner of the boxes. Contains `index` and `time` information. topLeft = chart.point.now(high) //@variable The `chart.point` for the bottom-right corner of the boxes. Does not contain `index` information. bottomRight = chart.point.from_time(closeTime, low) if changeTF and not na(volume) if close > open // Update `upVolume` and `downVolume` values. upVolume := volume downVolume := 0.0 // Draw a new `upBox` using `time` and `price` info from the `topLeft` and `bottomRight` points. upBox := box.new(topLeft, bottomRight, upBorder, 3, xloc = xloc.bar_time, bgcolor = upFill) // Draw a new `downBox` with `na` coordinates. downBox := box.new(na, na, na, na, downBorder, 3, xloc = xloc.bar_time, bgcolor = downFill) else // Update `upVolume` and `downVolume` values. upVolume := 0.0 downVolume := volume // Draw a new `upBox` with `na` coordinates. upBox := box.new(na, na, na, na, upBorder, 3, xloc =

xloc.bar_time, bgcolor = upFill) // Draw a new `downBox` using `time` and `price` info from the `topLeft` and `bottomRight` points. downBox := box.new(topLeft, bottomRight, downBorder, 3, xloc = xloc.bar_time, bgcolor = downFill) // Update the ``upVolume`` and change the ``upBox`` coordinates when volume increases on an upward bar. else if close > open and volume > upVolume upVolume := volume box.set_top_left_point(upBox, topLeft) box.set_bottom_right_point(upBox, bottomRight) // Update the ``downVolume`` and change the ``downBox`` coordinates when volume increases on a downward bar. else if close <= open and volume > downVolume downVolume := volume box.set_top_left_point(downBox, topLeft) box.set_bottom_right_point(downBox, bottomRight) // Highlight the background when a new `timeframe` bar starts. bgcolor(changeTF ? color.new(color.orange, 70) : na, title = "Timeframe change highlight")

Note that:

- The indicator() function call contains `max_boxes_count = 100`, meaning the script will preserve the last 100 boxes on the chart.
- We utilized *both overloads* of box.new() in this example. On the first bar of the `timeframe`, the script calls the first overload for the `upBox` when the bar is rising, and it calls that overload for the `downBox` when the bar is falling. It uses the second overload to assign a new box with na values to the other box variable on that bar.

Box styles

Users can include one of the following `line.style_*` variables in their box.new() or box.set_border_style() function calls to set the border styles of boxes drawn by their scripts:

Argument	Box
`line.style_solid`	
`line.style_dotted`	
`line.style_dashed`	

Reading box values

The `box.*` namespace features *getter* functions that allow scripts to retrieve coordinate values from a box instance:

- box.get_left() and box.get_right() respectively get the x-coordinates of the left and right edges of the `id` box. Whether the value returned represents a bar index or time value depends on the box's `xloc` property.
- box.get_top() and box.get_bottom() respectively get the top and bottom y-coordinates of the `id` box.

The example below draws boxes to visualize hypothetical price ranges over a period of `length` bars. At the start of each new period, it uses the average candle range multiplied by the `scaleFactor` input to calculate the corner points of a box centered at the hl2 price with an `initialRange` height. After drawing the first box, it creates `numberOfBoxes - 1` new boxes inside a for loop.

Within each loop iteration, the script gets the `lastBoxDrawn` reference by retrieving the last element from the read-only box.all array, then calls box.get_top() and box.get_bottom() to get the y-coordinates of the referenced box. It uses these values to calculate the coordinates for a new box that's `scaleFactor` times taller than the previous:

//@version=6 indicator("Reading box values demo", "Nested boxes", overlay = true, max_boxes_count = 500) //@variable The number of bars in the range calculation. int length = input.int(10, "Length", 2, 500) //@variable The number of nested boxes drawn on each period. int numberOfBoxes = input.int(5, "Nested box count", 1) //@variable The scale factor applied to each box. float scaleFactor = input.float(1.6, "Scale factor", 1) //@variable The initial box range. float initialRange = scaleFactor * ta.sma(high - low, length) if bar_index % length == 0 //@variable The top-left `chart.point` for the initial box. Does not contain `time` information. topLeft = chart.point.from_index(bar_index, hl2 + initialRange / 2) //@variable The bottom-right `chart.point` for the initial box. Does not contain `time` information. bottomRight = chart.point.from_index(bar_index + length, hl2 - initialRange / 2) // Calculate border and fill colors of the boxes. borderColor = color.rgb(math.random(100, 255), math.random(0, 100), math.random(100, 255)) bgColor = color.new(borderColor, math.max(100 * (1 - 1/numberOfBoxes), 90)) // Draw a new box using the `topLeft` and `bottomRight` points. Uses their `index` fields as x-coordinates. box.new(topLeft, bottomRight, borderColor, 2, bgcolor = bgColor) if numberOfBoxes > 1 // Loop to create additional boxes. for i = 1 to numberOfBoxes - 1 //@variable The last box drawn by the script. box lastBoxDrawn = box.all.last() //@variable The top price of the last box. float top = box.get_top(lastBoxDrawn) //@variable The bottom price of the last box. float bottom = box.get_bottom(lastBoxDrawn) //@variable The scaled range of the new box. float newRange = scaleFactor * (top - bottom) * 0.5 // Update the `price` fields of the `topLeft` and `bottomRight` points. // This does not affect the coordinates of previous boxes. topLeft.price := hl2 + newRange bottomRight.price := hl2 - newRange // Draw a new box using the updated `topLeft` and `bottomRight` points. box.new(topLeft, bottomRight, borderColor, 2, bgcolor = bgColor)

Note that:

- The indicator() function call uses `max_boxes_count = 500`, meaning the script can display up to 500 boxes on the chart.
- Each drawing has a `right` index `length` bars beyond the `left` index. Since the x-coordinates of these drawings can be up to 500 bars into the future, we've set the `maxval` of the `length` input to 500.
- On each new period, the script uses randomized color.rgb() values for the `border_color` and `bgcolor` of the boxes.
- Each box.new() call copies the coordinates from the chart.point objects assigned to the `topLeft` and `bottomRight` variables, which is why the script can modify their `price` fields on each loop iteration without affecting the other boxes.

Cloning boxes

To clone a specific box `id`, use box.copy(). This function copies the box and its properties. Any changes to the copied box do not affect the original.

For example, this script declares an `originalBox` variable on the first bar and assigns a new box reference to it once every `length` bars. On other bars, it uses box.copy() to create a copy of the box, assigns that copy to the `copiedBox` variable, then calls `box.set_*()` functions to modify the copy's properties. As shown on the chart below, changes to the copied box do not modify the box referenced by `originalBox`:

//@version=6 indicator("Cloning boxes demo", overlay = true, max_boxes_count = 500) //@variable The number of bars between each new mainLine assignment. int length = input.int(20, "Length", 2) //@variable The `chart.point` for the top-left of the `originalBox`. Contains `time` and `index` information. topLeft = chart.point.now(high) //@variable The `chart.point` for the bottom-right of the `originalBox`. Does not contain `time` information. bottomRight = chart.point.from_index(bar_index + 1, low) //@variable A new box with `topLeft` and `bottomRight` corners on every `length` bars. var box originalBox = na //@variable Is teal when

the bar is rising, maroon when it's falling. color originalColor = close > open ? color.teal : color.maroon if bar_index % length == 0 // Assign a new box using the `topLeft` and `bottomRight` info to the `originalBox`. // This box uses the `index` fields from the points as x-coordinates. originalBox := box.new(topLeft, bottomRight, originalColor, 2, bgcolor = color.new(originalColor, 60)) else //@variable A clone of the `originalBox`. box copiedBox = box.copy(originalBox) // Modify the `copiedBox`. These changes do not affect the `originalBox`. box.set_top(copiedBox, high) box.set_bottom_right_point(copiedBox, bottomRight) box.set_border_color(copiedBox, color.gray) box.set_border_width(copiedBox, 1) box.set_bgcolor(copiedBox, na)

Deleting boxes

To delete boxes drawn by a script, use box.delete(). As with `*.delete()` functions in other drawing namespaces, this function is handy for conditionally removing boxes or maintaining a specific number of boxes on the chart.

This example displays boxes representing periodic cumulative volume values. The script creates a new box ID and stores it in a `boxes` array once every `length` bars. If the array's size exceeds the `numberOfBoxes` value, the script removes the oldest box from the array using array.shift() and deletes it using box.delete().

On other bars, it accumulates volume over each period by modifying the `top` property of the last box in the `boxes` array. The script then uses `for` loops to find the `highestTop` of all the array's boxes and set the `bgcolor` of each box with a gradient color created using color.from_gradient() based on its box.get_top() value relative to the `highestTop`:

//@version=6 //@variable The maximum number of boxes to show on the chart. const int MAX_BOXES_COUNT = 500 indicator("Deleting boxes demo", "Cumulative volume boxes", format = format.volume, max_boxes_count = MAX_BOXES_COUNT) //@variable The number of bars in each period. int length = input.int(20, "Length", 1) //@variable The maximum number of volume boxes in the calculation. int numberOfBoxes = input.int(10, "Number of boxes", 1, MAX_BOXES_COUNT) //@variable An array containing the ID of each box displayed by the script. var boxes = array.new<box>() if bar_index % length == 0 // Push a new box into the `boxes` array. The box has the default `xloc.bar_index` property. boxes.push(box.new(bar_index, 0, bar_index + 1, 0, #000000, 2, text_color = #000000)) // Shift the oldest box out of the array and delete it when the array's size exceeds the `numberOfBoxes`. if boxes.size() > numberOfBoxes box.delete(boxes.shift()) //@variable The last box drawn by the script as of the current chart bar. box lastBox = boxes.last() // Add the current bar's volume to the top of the `lastBox` and update the `right` index. lastBox.set_top(lastBox.get_top() + volume) lastBox.set_right(bar_index + 1) // Display the top of the `lastBox` as volume-formatted text. lastBox.set_text(str.tostring(lastBox.get_top(), format.volume)) //@variable The highest `top` of all boxes in the `boxes` array. float highestTop = 0.0 for id in boxes highestTop := math.max(id.get_top(), highestTop) // Set the `bgcolor` of each `id` in `boxes` with a gradient based on the ratio of its `top` to the `highestTop`. for id in boxes id.set_bgcolor(color.from_gradient(id.get_top() / highestTop, 0, 1, color.purple, color.orange))

Note that:

- At the top of the code, we've declared a `MAX_BOXES_COUNT` variable with the "const int" *qualified type*. We use this value as the `max_boxes_count` in the indicator() function and the maximum possible value of the `numberOfBoxes` input.
- This script uses the second overload of the box.new() function, which specifies the box's `left`, `top`, `right`, and `bottom` coordinates separately.

- We've included format.volume as the `format` argument in the indicator() call, which tells the script that the y-axis of the chart pane represents *volume* values. Each box also displays its `top` coordinate as volume-formatted text.

Polylines

Pine Script polylines are **advanced** drawings that sequentially connect the coordinates from an array of chart.point instances using straight or *curved* line segments.

These powerful drawings can connect up to 10,000 points at any available location on the chart, allowing scripts to draw custom series, polygons, and other complex geometric formations that are otherwise difficult or impossible to draw using line or box objects.

The `polyline.*` namespace features the following built-ins for creating and managing polyline objects:

- The polyline.new() function creates a new polyline instance.
- The polyline.delete() function deletes an existing polyline instance.
- The polyline.all variable references a read-only array containing the IDs of all polylines displayed by the script. The array's size depends on the `max_polylines_count` of the indicator() or strategy() declaration statement and the number of polylines drawn by the script.

Unlike lines or boxes, polylines do not have functions for modification or reading their properties. To redraw a polyline on the chart, one can *delete* the existing instance and *create* a new polyline with the desired changes.

Creating polylines

The polyline.new() function creates a new polyline instance to display on the chart. It has the following signature:

```
polyline.new(points, curved, closed, xloc, line_color, fill_color, line_style,
```

The following eight parameters affect the behavior of a polyline drawing:

`points`

Accepts an array of chart.point objects that determine the coordinates of each point in the polyline. The drawing connects the coordinates from each element in the array sequentially, starting from the *first*. Whether the polyline uses the `index` or `time` field from each chart point for its x-coordinates depends on the `xloc` value in the function call.

`curved`

Specifies whether the drawing uses curved line segments to connect each chart.point in the `points` array. The default value is `false`, meaning it uses straight line segments.

`closed`

Controls whether the polyline will connect the last chart.point in the `points` array to the first, forming a *closed polyline*. The default value is `false`.

`xloc`

Specifies which field from each chart.point in the `points` array the polyline uses for its x-coordinates. When its value is xloc.bar_index, the function uses the `index` fields to create the polyline. When its value is xloc.bar_time, the function uses the `time` fields. The default value is xloc.bar_index.

`line_color`

Specifies the color of all line segments in the polyline drawing. The default is `color.blue`.

`fill_color`

Controls the color of the closed space filled by the polyline drawing. Its default value is na.

`line_style`

Specifies the style of the polyline, which can be any of the available options in the Line styles section of this page. The default is line.style_solid.

`line_width`

Specifies the width of the polyline, in pixels. The default value is 1.

`force_overlay`

If `true`, the drawing will display on the main chart pane, even when the script occupies a separate pane. Optional. The default is `false`.

This script demonstrates a simple example of drawing a polyline on the chart. It uses array.push() to push the reference of a new chart.point object with an alternating `price` value into a `points` array and colors the background with bgcolor() once every `length` bars.

On the last confirmed historical bar, where barstate.islastconfirmedhistory is `true`, the script creates a new polyline with polyline.new(). The polyline drawing passes through the coordinates of each chart point in the `points` array in order, starting from the first point:

//@version=6 indicator("Creating polylines demo", "Oscillating polyline") //@variable The number of bars between each point in the drawing. int length = input.int(20, "Length between points", 2) //@variable An array of `chart.point` objects to sequentially connect with a polyline. var points = array.new<chart.point>() //@variable The y-coordinate of each point in the `points`. Alternates between 1 and -1 on each `newPoint`. var int yValue = 1 //@variable Is `true` once every `length` bars, `false` otherwise. bool newPoint = bar_index % length == 0 if newPoint // Push a new `chart.point` into the `points`. The new point contains `time` and `index` info. points.push(chart.point.now(yValue)) // Change the sign of the `yValue`. yValue *= -1 // Draw a new `polyline` on the last confirmed historical chart bar. // The polyline uses the `time` field from each `chart.point` in the `points` array as x-coordinates. if barstate.islastconfirmedhistory polyline.new(points, xloc = xloc.bar_time, line_color = #9151A6, line_width = 3) // Highlight the chart background on every `newPoint` condition. bgcolor(newPoint ? color.new(color.gray, 70) : na, title = "New point highlight")

Note that:

- This script uses only *one* polyline to connect each chart point from the array with straight line segments, and this drawing spans throughout the available chart data, starting from the first bar.
- While one can achieve a similar effect using lines, doing so would require a new line instance on each occurrence of the `newPoint` condition, and such a drawing would be limited to a maximum of 500 line segments. This single unclosed polyline drawing, on the other hand, can contain up to 9,999 line segments.

Curved drawings

Polylines can draw *curves* that are otherwise impossible to produce with lines or boxes. When enabling the `curved` parameter of the polyline.new() function, the resulting polyline interpolates *nonlinear* values between the coordinates from each chart.point in its array of `points` to generate a curvy effect.

For instance, the “Oscillating polyline” script in our previous example uses *straight* line segments to produce a drawing resembling a triangle wave, meaning a waveform that zig-zags between its peaks and valleys. If we set the `curved` parameter in the polyline.new() call from that example to `true`, the resulting drawing would connect the points using *curved* segments, producing a smooth, nonlinear shape similar to a sine wave:

//@version=6 indicator("Curved drawings demo", "Smooth oscillating polyline") //@variable The number of bars between each point in the drawing. int length = input.int(20, "Length between points", 2) //@variable An array of `chart.point` objects to sequentially connect with a polyline. var points = array.new<chart.point>() //@variable The y-coordinate of each point in the `points`. Alternates between 1 and -1 on each `newPoint`. var int yValue = 1 //@variable Is `true` once every `length` bars, `false` otherwise. bool newPoint = bar_index % length == 0 if newPoint // Push a new `chart.point` into the `points`. The new point contains `time` and `index` info. points.push(chart.point.now(yValue)) // Change the sign of the `yValue`. yValue *= -1 // Draw a new curved `polyline` on the last confirmed historical chart bar. // The polyline uses the `time` field from each `chart.point` in the `points` array as x-coordinates. if barstate.islastconfirmedhistory polyline.new(points, curved = true, xloc = xloc.bar_time, line_color = #9151A6, line_width = 3) // Highlight the chart background on every `newPoint` condition. bgcolor(newPoint ? color.new(color.gray, 70) : na, title = "New point highlight")

Notice that in this example, the smooth curves have relatively consistent behavior, and no portion of the drawing extends past its defined coordinates, which is not always the case when drawing curved polylines. The data used to construct a polyline heavily impacts the smooth, piecewise function it interpolates between its points. In some cases, the interpolated curve *can* reach beyond its actual coordinates.

Let's add some variation to the chart points in our example's `points` array to demonstrate this behavior. In the version below, the script multiplies `yValue` by a pseudorandom value in each chart.point.now() call.

To visualize the behavior, this script also creates a horizontal line at the `price` value from each chart.point instance in the `points` array, and it displays another polyline connecting the same points with straight line segments. As we see on the chart, both polylines pass through all coordinates from the `points` array. However, the curvy polyline occasionally reaches *beyond* the vertical boundaries indicated by the horizontal lines, whereas the polyline drawn using straight segments does not:

//@version=6 indicator("Curved drawings demo", "Random oscillating polylines") //@variable The number of bars between each point in the drawing. int length = input.int(20, "Length between points", 2) //@variable An array of `chart.point` objects to sequentially connect with a polyline. var points = array.new<chart.point>() //@variable The sign of each `price` in the `points`. Alternates between 1 and -1 on each `newPoint`. var int

yValue = 1 //@variable Is `true` once every `length` bars. bool newPoint = bar_index % length == 0 if newPoint // Push a new `chart.point` with a randomized `price` into the `points`. // The new point contains `time` and `index` info. points.push(chart.point.now(yValue * math.random())) // Change the sign of the `yValue`. yValue *= -1 //@variable The newest `chart.point`. lastPoint = points.last() // Draw a horizontal line at the `lastPoint.price`. This line uses the default `xloc.bar_index`. line.new(lastPoint.index - length, lastPoint.price, lastPoint.index + length, lastPoint.price, color = color.red) // Draw two `polyline` instances on the last confirmed chart bar. // Both polylines use the `time` field from each `chart.point` in the `points` array as x-coordinates. if barstate.islastconfirmedhistory polyline.new(points, curved = false, xloc = xloc.bar_time, line_color = #EB8A3B, line_width = 2) polyline.new(points, curved = true, xloc = xloc.bar_time, line_color = #9151A6, line_width = 3) // Highlight the chart background on every `newPoint` condition. bgcolor(newPoint ? color.new(color.gray, 70) : na, title = "New point highlight")

Closed shapes

Since a single polyline can contain numerous straight or curved line segments, and the `closed` parameter allows the drawing to connect the coordinates from the first and last chart.point in its array of `points`, programmers can use polylines to draw many different types of closed polygonal shapes.

Let's draw some polygons in Pine. The following script periodically draws randomized polygons centered at hl2 price values.

On each occurrence of the `newPolygon` condition, the script clears the `points` array with array.clear(), calculates `numberOfSides` and `rotationOffset` values based on values from math.random() calls, then uses a `for` loop to push `numberOfSides` new chart points into the array. The chart points contain stepped coordinates from an elliptical path with `xScale` and `yScale` semi-axes. The script draws the polygon by connecting each point from the `points` array using a *closed polyline* with straight line segments:

//@version=6 indicator("Closed shapes demo", "N-sided polygons", true) //@variable The size of the horizontal semi-axis. float xScale = input.float(3.0, "X scale", 1.0) //@variable The size of the vertical semi-axis. float yScale = input.float(1.0, "Y scale") * ta.atr(2) //@variable An array of `chart.point` objects containing vertex coordinates. var points = array.new<chart.point>() //@variable The condition that triggers a new polygon drawing. Based on the horizontal axis to prevent overlaps. bool newPolygon = bar_index % int(math.round(2 * xScale)) == 0 and barstate.isconfirmed if newPolygon // Clear the `points` array. points.clear() //@variable The number of sides and vertices in the new polygon. int numberOfSides = int(math.random(3, 7)) //@variable A random rotation offset applied to the new polygon, in radians. float rotationOffset = math.random(0.0, 2.0) * math.pi //@variable The size of the angle between each vertex, in radians. float step = 2 * math.pi / numberOfSides //@variable The counter-clockwise rotation angle of each vertex. float angle = rotationOffset for i = 1 to numberOfSides //@variable The approximate x-coordinate from an ellipse at the `angle`, rounded to the nearest integer. int xValue = int(math.round(xScale * math.cos(angle))) + bar_index //@variable The y-coordinate from an ellipse at the `angle`. float yValue = yScale * math.sin(angle) + hl2 // Push a new `chart.point` containing the `xValue` and `yValue` into the `points` array. // The new point does not contain `time` information. points.push(chart.point.from_index(xValue, yValue)) // Add the `step` to the `angle`. angle += step // Draw a closed polyline connecting the `points`. // The polyline uses the `index` field from each `chart.point` in the `points` array. polyline.new(points, closed = true, line_color = color.navy, fill_color = color.new(color.orange, 50), line_width = 3)

Note that:

- This example shows the last ~50 polylines on the chart, as we have not specified a `max_polylines_count` value in the indicator() function call.
- The `yScale` calculation multiplies an input.float() by `ta.atr(2)` to adapt the vertical scale of the drawings to recent price ranges.
- The resulting polygons have a maximum width of twice the horizontal semi-axis (`2 * xScale`), rounded to the nearest integer. The `newPolygon` condition uses this value to prevent the polygon drawings from overlapping.
- The script rounds the `xValue` calculation to the nearest integer because the `index` field of a chart.point only accepts an int value, as the x-axis of the chart does not include fractional bar indices.

Deleting polylines

To delete a specific polyline, use polyline.delete(). This function removes the polyline object from the script and its drawing on the chart.

As with other drawing objects, we can use polyline.delete() to maintain a specific number of polyline drawings or conditionally remove drawings from a chart.

For example, the script below periodically draws approximate arithmetic spirals and stores their polyline references in an array, which it uses as a queue to manage the number of drawings it displays.

When the `newSpiral` condition occurs, the script creates a `points` array and adds chart points within a `for` loop. On each loop iteration, it calls the `spiralPoint()` user-defined function to create a new chart.point containing stepped values from an elliptical path that grows with respect to the `angle` value. The script then creates a randomly colored *curved polyline* connecting the coordinates from the `points` and pushes its reference into the `polylines` array.

When the array's size exceeds the `numberOfSpirals` value, the script removes the oldest polyline reference using array.shift() and deletes the object using polyline.delete():

//@version=6 //@variable The maximum number of polylines allowed on the chart. const int MAX_POLYLINES_COUNT = 100 indicator("Deleting polylines example", "Spirals", true, max_polylines_count = MAX_POLYLINES_COUNT) //@variable The number of spiral drawings on the chart. int numberOfSpirals = input.int(10, "Spirals shown", 1, MAX_POLYLINES_COUNT) //@variable The number of full spiral rotations to draw. int rotations = input.int(5, "Rotations", 1) //@variable The scale of the horizontal semi-axis. float xScale = input.float(1.0, "X scale") //@variable The scale of the vertical semi-axis. float yScale = input.float(0.2, "Y scale") * ta.atr(2) //@function Calculates an approximate point from an elliptically-scaled arithmetic spiral. //@returns A `chart.point` with `index` and `price` information. spiralPoint(float angle, int xOffset, float yOffset) => result = chart.point.from_index(int(math.round(angle * xScale * math.cos(angle))) + xOffset, angle * yScale * math.sin(angle) + yOffset) //@variable An array of polylines. var polylines = array.new<polyline>() //@variable The condition to create a new spiral. bool newSpiral = bar_index % int(math.round(4 * math.pi * rotations * xScale)) == 0 if newSpiral //@variable An array of `chart.point` objects for the `spiral` drawing. points = array.new<chart.point>() //@variable The counter-clockwise angle between calculated points, in radians. float step = math.pi / 2 //@variable The rotation angle of each calculated point on the spiral, in radians. float theta = 0.0 // Loop to create the spiral's points. Creates 4 points per full rotation. for i = 0 to rotations * 4 //@variable A new point on the calculated spiral. chart.point newPoint = spiralPoint(theta, bar_index, ohlc4) // Add the `newPoint` to the `points` array. points.push(newPoint) // Add the `step` to the `theta` angle. theta += step //@variable A random color for the new `spiral` drawing. color spiralColor = color.rgb(math.random(150, 255), math.random(0, 100),

math.random(150, 255)) //@variable A new polyline connecting the spiral points. Uses the `index` field from each point as x-coordinates. polyline spiral = polyline.new(points, true, line_color = spiralColor, line_width = 3) // Push the new `spiral` into the `polylines` array. polylines.push(spiral) // Shift the first polyline out of the array and delete it when the array's size exceeds the `numberOfSpirals`. if polylines.size() > numberOfSpirals polyline.delete(polylines.shift()) // Highlight the background when `newSpiral` is `true`. bgcolor(newSpiral ? color.new(color.blue, 70) : na, title = "New drawing highlight")

Note that:

- We declared a `MAX_POLYLINES_COUNT` global variable with a constant value of 100. The script uses this constant as the `max_polylines_count` value in the indicator() function and the `maxval` of the `numberOfSpirals` input.
- As with our “N-sided polygons” example in the previous section, we round the calculation of x-coordinates to the nearest integer since the `index` field of a chart.point can only accept an int value.
- Despite the smooth appearance of the drawings, each polyline's `points` array only contains *four* chart.point objects per spiral rotation. Since the polyline.new() call includes `curved = true`, each polyline uses *smooth curves* to connect their `points`, producing a visual approximation of the spiral's actual curvature.
- The width of each spiral is approximately `4 * math.pi * rotations * xScale`, rounded to the nearest integer. We use this value in the `newSpiral` condition to space each drawing and prevent overlaps.

Redrawing polylines

It may be desirable in some cases to change a polyline drawing throughout a script's execution. While the `polyline.*` namespace does not contain built-in setter functions, we can *redraw* polylines referenced by variables or collections by *deleting* the existing polylines and assigning *new instances* with the desired changes.

The following example uses polyline.delete() and polyline.new() calls to update the value of a polyline variable.

This script draws closed polylines that connect the open, high, low, and close points of periods containing `length` bars. It creates a `currentDrawing` variable on the first bar and assigns a polyline reference to it on every chart bar. It uses the `openPoint`, `highPoint`, `lowPoint`, and `closePoint` variables to reference chart points that track the period's developing OHLC values. As new values emerge, the script assigns new chart.point objects to the variables, collects them in an array using array.from(), then creates a new polyline connecting the coordinates from the array’s points with polyline.new().

When the `newPeriod` condition is `false` (i.e., the current period is not complete), the script deletes the polyline referenced by the `currentDrawing` variable before creating a new one, resulting in a dynamic drawing that changes over the developing period:

//@version=6 indicator("Redrawing polylines demo", "OHLC polygons", true, max_polylines_count = 100) //@variable The length of the period. int length = input.int(100, "Length", 1) //@variable A `chart.point` representing the start of each period. var chart.point openPoint = na //@variable A `chart.point` representing the highest point of each period. var chart.point highPoint = na //@variable A `chart.point` representing the lowest point of each period. var chart.point lowPoint = na //@variable A `chart.point` representing the current bar's closing point. closePoint = chart.point.now(close) //@variable The current period's polyline drawing. var polyline currentDrawing = na //@variable Is `true` once every `length` bars. bool newPeriod = bar_index % length == 0 if newPeriod // Assign new chart points to the `openPoint`, `highPoint`, and `closePoint`.

openPoint := chart.point.now(open) highPoint := chart.point.now(high) lowPoint := chart.point.now(low) else // Assign a new `chart.point` to the `highPoint` when the `high` is greater than its `price`. if high > highPoint.price highPoint := chart.point.now(high) // Assign a new `chart.point` to the `lowPoint` when the `low` is less than its `price`. if low < lowPoint.price lowPoint := chart.point.now(low) //@variable Is teal when the `closePoint.price` is greater than the `openPoint.price`, maroon otherwise. color drawingColor = closePoint.price > openPoint.price ? color.teal : color.maroon // Delete the polyline assigned to the `currentDrawing` if it's not a `newPeriod`. if not newPeriod polyline.delete(currentDrawing) // Assign a new polyline to the `currentDrawing`. // Uses the `index` field from each `chart.point` in its array as x-coordinates. currentDrawing := polyline.new(array.from(openPoint, highPoint, closePoint, lowPoint), closed = true, line_color = drawingColor, fill_color = color.new(drawingColor, 60))

Realtime behavior

Lines, boxes, and polylines are subject to both *commit* and *rollback* actions, which affect the behavior of a script when it executes on a realtime bar. See the page on Pine Script's Execution model.

This script demonstrates the effect of rollback when it executes on the realtime, *unconfirmed* chart bar:

//@version=6 indicator("Realtime behavior demo", overlay = true) //@variable Is orange when the `line` is subject to rollback and gray after the `line` is committed. color lineColor = barstate.isconfirmed ? color.gray : color.orange line.new(bar_index, hl2, bar_index + 1, hl2, color = lineColor, width = 4)

The line.new() call in this example creates a new line ID on each iteration when values change on the unconfirmed bar. The script automatically deletes the objects created on each change in that bar because of the *rollback* before each iteration. It only *commits* the last line created before the bar closes, and that line instance is the one that persists on the confirmed bar.

Limitations

Total number of objects

Lines, boxes, and polylines consume server resources, which is why there are limits on the total number of drawings per script. When a script creates more drawing objects than the allowed limit, the Pine Script runtime automatically deletes the oldest ones in a process referred to as *garbage collection*.

A single script can contain up to 500 lines, 500 boxes, and 100 polylines. Users can control the garbage collection limits by specifying the `max_lines_count`, `max_boxes_count`, and `max_polylines_count` values in their script's indicator() or strategy() declaration statement.

This script demonstrates how garbage collection works in Pine. It creates a new line, box, and polyline on each chart bar. We haven't specified values for the `max_lines_count`, `max_boxes_count`, or `max_polylines_count` parameters in the indicator() function call, so the script will maintain the most recent ~50 lines, boxes, and polylines on the chart, as this is the default setting for each parameter:

//@version=6 indicator("Garbage collection demo", overlay = true) //@variable A new `chart.point` at the current `bar_index` and `high`. firstPoint = chart.point.now(high) //@variable A new `chart.point` one bar into the future at the current `low`. secondPoint = chart.point.from_index(bar_index + 1, low) //@variable A new `chart.point` one bar into the future at the current `high`. thirdPoint = chart.point.from_index(bar_index + 1, high) // Draw a new `line` connecting the `firstPoint` to the `secondPoint`. line.new(firstPoint, secondPoint, color = color.red, width = 2) // Draw a new `box` with the `firstPoint` top-left corner and `secondPoint`

bottom-right corner. box.new(firstPoint, secondPoint, color.purple, 2, bgcolor = na) // Draw a new `polyline` connecting the `firstPoint`, `secondPoint`, and `thirdPoint` sequentially. polyline.new(array.from(firstPoint, secondPoint, thirdPoint), true, line_width = 2)

Note that:

- We've used TradingView's “Measure” drawing tool to measure the number of bars covered by the script's drawing objects.

Past and future references with `xloc.bar_index`

Objects positioned using xloc.bar_index can contain x-coordinates no further than 500 bars into the future or 10,000 bars into the past.

Other contexts

Scripts cannot use lines, boxes, or polylines in `request.*()` functions. Instances of these types can use the values from `request.*()` calls, but scripts can only create and draw them in the chart's context.

This limitation is also why drawing objects will not work when using the `timeframe` parameter in the indicator() declaration statement.

Historical buffer and `max_bars_back`

Using barstate.isrealtime in combination with drawings may sometimes produce unexpected results. For example, the intention of this script is to ignore all historical bars and draw horizontal lines spanning 300 bars back on *realtime* bars:

//@version=6 indicator("Historical buffer demo", overlay = true) //@variable A `chart.point` at the `bar_index` from 300 bars ago and current `close`. firstPoint = chart.point.from_index(bar_index[300], close) //@variable The current bar's `chart.point` containing the current `close`. secondPoint = chart.point.now(close) // Draw a new line on realtime bars. if barstate.isrealtime line.new(firstPoint, secondPoint)

However, it will fail at runtime and raise an error. The script fails because it cannot determine the buffer size for historical values of the underlying time series. Although the code doesn't contain the built-in time variable, the built-in bar_index uses the time series in its inner workings. Therefore, accessing the value of the bar_index from 300 bars back requires the history buffer of the time series to be at least 300 bars.

Pine Script includes a mechanism that detects the required historical buffer size automatically in most cases. It works by letting the script access historical values any number of bars back for a limited duration. In this script's case, using barstate.isrealtime to control the drawing of lines prevents it from accessing the historical series, so it cannot infer the required historical buffer size, and the script fails.

The simple solution to this issue is to use the max_bars_back() function to *explicitly define* the historical buffer of the time series before evaluating the conditional structure:

//@version=6 indicator("Historical buffer demo", overlay = true) //@variable A `chart.point` at the `bar_index` from 300 bars ago and current `close. firstPoint = chart.point.from_index(bar_index[300], close) //@variable The current bar's `chart.point` containing the current `close`. secondPoint = chart.point.now(close) // Explicitly set the historical buffer of the `time` series to 300 bars. max_bars_back(time, 300) // Draw a new line on realtime bars. if barstate.isrealtime line.new(firstPoint, secondPoint)

Such issues can be confusing, but they're quite rare. The Pine Script team hopes to eliminate them over time.

Overview

Introduction

Well-designed visuals make indicators and strategies easier to use and less cluttered. Each visual element presents data differently:

- Plot visuals include all `plot*()` functions, horizontal levels, background and bar coloring, and fills.
- Drawing visuals include lines, polylines, linefills, boxes, labels, and tables.

Scripts can configure where and how the visual elements appear by using the script-wide visual settings.

By understanding when to use each tool effectively, programmers can pick the best visual for the task to make the most of the power of Pine Script®.

> NoteUsers can draw directly on TradingView charts using the Drawing Tools. While such drawings might sometimes resemble visuals created by Pine scripts, they are **unrelated** entities. Pine scripts cannot interact with drawing tools from the chart interface, and cursor actions do not modify Pine drawing objects.

This page describes plots and drawings, and what their differences are. It includes all the available visual constructs and examples of their use in built-in indicators (for more details about a specific visual element, refer to its User Manual page).

Script-wide visual settings

Some visual settings control how all of the script's outputs *collectively* appear on the chart, regardless of their individual properties. These script-wide visual settings are parameters in the indicator() or strategy() declaration statement.

`overlay`

The `overlay` parameter controls whether the script's outputs appear in the main pane or a separate pane. By default, its value is `false`, so adding a script to the chart displays its visual outputs in a *separate* pane to the main chart series.

Whereas the `overlay` parameter affects the script as a whole, the `force_overlay` parameter controls the pane location for *individual* elements. Using `force_overlay = true` displays the specified element in the main pane, even if the script's `overlay` argument is `false`. This allows a script occupying a separate pane to overlay only some visuals on the main chart. The `force_overlay` parameter is available for all `plot*()` functions, bgcolor(), and all drawing `*.new()` constructor functions (box.new(), label.new(), etc.).

For example, the built-in Seasonality indicator uses `overlay = false` to display in a separate pane, where it displays its primary visual of a table, but draws boxes on the main chart with `force_overlay = true`:

`scale`

A script's `scale` parameter specifies the y-axis scale that its pane visuals use. By default, scripts overlayed in the main pane use the existing chart scale (scale.none). Specifying a scale.right or scale.left argument in overlayed scripts generates a *new scale* distinct from the main chart's price scale. Scripts displaying in a separate pane generate their own scale by default, which they can also set to the left or right position. For instance, this image shows an overlayed indicator using a distinct right-side scale, and a separate pane indicator using a left-side scale:

`behind_chart`

The `behind_chart` parameter specifies whether a script's visuals appear behind or in front of the main chart series. By default, its value is `true`, so visuals overlayed in the main pane appear behind the chart bars. When `behind_chart` is `false`, visuals appear *in front of* the bars, which may obscure bars, depending on the type of visual and its color transparency:

Changing settings

To adjust the visual settings of a script on the chart, click the "More" menu (three dots icon) in the script's status line. Options are available to adjust the script's visual order, move it to another pane, and change its y-axis scale:

> NoticeScripts evaluate the visual settings in the indicator() or strategy() declaration statement *only once*, when the script first loads on the chart. Updating parameters like `overlay` or `scale` in the code of a script instance that is already active on the chart does **not** update its existing display. Add a new script instance to the chart to apply updated visual settings.

Plot visuals

The outputs of the following functions are classified as plot visuals:

- All `plot*()` functions:
  - Data series plots using plot()
  - Shape plots using plotshape()
  - Character plots using plotchar()
  - Arrow plots using plotarrow()
  - Bar plots using plotbar()
  - Candle plots using plotcandle()
- Bar coloring using barcolor()
- Background coloring using bgcolor()
- Horizontal levels using hline()
- Fills for plots and horizontal levels using fill()

Plots are *serial* visuals that always return a result on each bar – although the result can be na. One plot therefore forms a *series*. By contrast, drawing visuals instantiate *individual objects*. A single plot visual function call can display results on all the bars in the main series, no matter how many bars display in the series, while drawings adhere to a drawing limit of approximately ~500 objects.

A script creates plot visuals sequentially as it executes across the chart bars, so it cannot draw them into the past or future all at once like drawings. For example, `plot(close)` plots the *current* close on the *current* bar.

Pine's execution model then repeats this for every bar in the dataset.

Scripts create plots with offsets in exactly the same way. They appear to end at past or future bars because the script executes the same plot call on each bar and simply displays each result the same *fixed* number of bars forwards or backwards.

Display in other locations

Plots can display results in locations other than the chart pane, unlike drawings. The last numeric value of a plot can display in the price scale. The script's status line and the Data Window can display plot values for specific bars, and the values update as the user hovers over different bars:

In one script, plots can display their results in different places by using different arguments for the `display` parameter for each plot function. For example, a script can display one plot's results in all locations, display another plot everywhere but the status line, and create a third plot with no visible display.

The `plot*()` functions accept multiple `display.*` arguments and support addition and subtraction to combine arguments for further customization. Other, numerically simpler plot visuals like horizontal levels, fills, and coloring functions have only two display states: they either display a pane visual (display.all) or are hidden (display.none).

This simple demonstration script uses various plot visuals and `display` locations to plot the open and close prices, plot the difference between them (`barCO`), and to signal when that difference is greater than 5:

//@version=6 indicator("Plot visuals display demo", overlay = false) //@variable The difference between the bar's `open` and `close` prices. float barCO = close - open // Horizontal lines and fills have only two possible `display` states. h1 = hline(125, "Level 125", linewidth = 2, display = display.none) h2 = hline(100, "Level 100", linewidth = 2, display = display.all) fill(h1, h2, color = color.new(color.blue, 90), display = display.all) // `plot*()` visuals accept multiple `display` options and support addition and subtraction. plot(close, "Close", color.blue, 3, display = display.all) plot(open, "Open", color.orange, 3, display = display.all - display.pane) plotarrow(barCO, "Bar CO", color.green, color.red, display = display.status_line + display.data_window) plotshape(barCO > 5, "Large CO", shape.circle, location.abovebar, color.fuchsia, display = display.pane)

Note that:

- Although there are no arrows visible in the script pane, the plotarrow() call still calculates and plots the `barCO` values on every bar, as indicated by the "Bar CO" result in the Data Window and the matching green result in the status line.
- Since `plotshape(barCO > 5)` uses a "bool" series, the plot's numeric results can only be 1 or 0 on any bar. We set it to display only in the chart pane because that's our most useful visual signal for this plot. Being selective with display options can help to keep results in any one location free from clutter.

> TipScripts can also use `display.*` arguments to show or hide input values next to the script title in the status line and Data Window.

The `format` and `precision` parameters of `plot*()` functions can further customize how numeric results appear in the status line, price scale, and Data Window. The `format` parameter specifies whether to format plot values as prices, percentages, or volume. The `precision` parameter specifies the number of decimal

digits that plot values include for non-volume formats. See the `plot()` parameters section of the Plots page to learn more.

Additionally, users can manage whether numeric plot results are visible for a given indicator or chart by using settings at both the indicator and chart level, without editing any source code (see the Help Center article on how to hide values of indicators for more). An indicator's settings control whether any plot values appear in *that indicator's* status line and price scale. A chart's settings control whether status line and price scale values appear *at all* in any indicators on that chart. Disabling the indicator settings overrides the script's per-plot `display` properties, while the chart settings override both.

Users can also customize the visibility, color, and style of plot visuals without needing to create new inputs or edit the script. Settings are automatically generated in the indicator's "Style" tab for every plot visual in the script, regardless of their `display` state:

Note that if the script generates any *dynamic colors*, the color pickers in the "Style" settings do not display. See the Maintaining automatic color selectors section of the Colors page to learn more.

The `display.*` arguments represent the *default state* of the script's plot visuals. Disabling a plot from the indicator's "Style" settings and then reactivating it causes the plot to revert to display.all, unless the indicator is reset to its default settings.

> TipTo prevent users from changing a plot's properties from the script's "Settings/Style" tab, set the `plot*()` call's `editable` argument to `false`.

The ability to display outputs in several locations and to visually track a series across the chart bars makes plot visuals useful debugging tools. See the Plots and chart colors section of the Debugging page for more information.

External uses: exports, alerts, and more

Unlike drawings, plots have uses outside the script: exporting data, creating alerts, setting another indicator's source input, and scanning watchlists using the Pine Screener.

These uses for plot results function regardless of a plot's `display.*` state on the chart and do not require special code for the outputs. Indeed, when creating plots for use in alerts or data exports, using display.none can keep a script's visuals clutter-free and avoid distorting the chart scale.

Users can export plots using the "Export chart data" feature, which generates a comma-separated values (CSV) file of the chart data (see the section on exporting indicator data to a file in the Indicators FAQ page). The exported data includes the symbol's OHLC (open, high, low, and close) values and any numeric plot results generated by active scripts on the chart, including those displayed only in the Data Window or status line. Drawings and hidden scripts are excluded from exports.

An alert can use any `plot*()` call executing on the chart as its trigger condition. Users can create alerts based on plots even if the script does not include any alert-specific code such as alert() or alertcondition(). See the Help Center article on how to create alerts from the user interface. Users can also include the dynamic results from up to 20 `plot*()` series in an alert's message using placeholders, as explained in the Help Center article on using variable values in alerts.

A script can use plots that are output by other indicators on the chart as a *source input*. The input.source() function creates a "Source" dropdown in the script's "Inputs" settings, from which users can then select any plots displayed on the chart as the input source. Any calculated plots can act as source inputs even if they are hidden from the current chart display (e.g., the unseen `plotarrow(barCO)` plot from the example in the display in other locations section above, or any hidden indicators). Using a source input links both scripts, so changes to the original plot subsequently alter the input plot, and removing the source indicator from the chart removes the dependent script.

The Pine Screener uses an indicator's plots to scan a watchlist of symbols. It generates columns showing the results of the indicator's plot() and alertcondition() calls for each symbol. Users can also choose to filter screener results based on plot conditions. See this Help Center article on the Pine Screener to learn more.

Limitations

Scripts can plot visuals only in the *global* scope. Unlike drawings, plots cannot be included in the local scopes of loops, conditional structures, or user-defined functions and methods, and plot calls can only accept variables and literals that are declared globally. However, a script can still create visuals that plot conditionally by using na values for a plot's `series` or `color` arguments, thus hiding the plot on certain bars.

While plot visuals are well suited for displaying dynamically-calculated series, those that support text, like plotshape() and plotchar(), cannot display dynamic text. The parameters of these functions accept "const string" arguments, so the same text displays on all the bars, and it cannot change during execution or be an input value, unlike the text supported in drawing visuals.

Plots can be offset into the past or future, but only by a fixed number of bars. This makes plotted shapes, for example, suitable for marking Williams fractals, which confirm after a known number of bars, but unsuitable for marking more complex types of events that confirm after an arbitrary number of bars.

Each script instance can create a maximum of 64 plots. Depending on the complexity of the plot and its arguments, one function call can count *more than once* towards the plot count limit. See the plot limits section of the Limitations page for more information.

Drawing visuals

Pine drawings display in a script's pane, and provide the flexibility to represent graphical data beyond plotting series. The following elements are classified as drawing visuals:

- Lines
- Polylines
- Linefills
- Boxes
- Labels
- Tables

Drawings are *objects*, unlike plots, which are *serial* visuals, so calling a drawing function **does not** create a visual that always returns a persistent result on every bar in the dataset. Instead, a drawing function references *one instance* of a drawing object, which can be at an arbitrary location relative to the bar on which the script called the function.

Since drawings are not serialized, scripts can call the same drawing function several times on one bar to create multiple drawings at different locations on the chart at once.

Each drawing visual has its own namespace with built-in functions for creating and managing the drawing objects. Most drawing parameters accept “series” types, which allows the visuals to use dynamic positions, colors, styles, etc. Drawing parameters support input values and complex expressions as arguments, and can update these arguments as the script executes from bar to bar. Drawings like labels, boxes, and tables can also display dynamic text.

Scripts can create and manage drawing visuals from *local* scopes, so programmers can include drawing calls in conditional structures, loops, and user-defined functions or methods, unlike plot calls. While scripts *can* call drawing functions globally, it’s rarely necessary to execute drawings on every bar. Further, because scripts that create drawing objects on each bar are likely to reach the limit for that drawing type, it’s more usual to create drawings in local scopes.

The ability of drawing functions to display dynamic data at any available chart location and to run in local scopes makes them useful debugging tools. See the Pine drawings section of the Debugging page for more information.

Display and customization

Unlike plots, drawings do not display in other locations – they display a visual only in the chart pane. Therefore, they cannot show any numeric results in the script’s status line, price scale, or Data Window, or by hovering over the drawing. Likewise, using drawings in a script does not automatically generate color/style customization options in the indicator’s “Style” tab.

Instead, the “Style” settings generate a checkbox for each drawing *type* used by a script, which toggles the visibility of **all** objects of that type in that indicator:

However, since drawings accept “series” arguments, scripts can use inputs to create fully customizable drawing visuals. For example, this script uses string inputs, color inputs, and integer inputs to allow users to easily customize the appearance of the table and label visuals from the indicator’s “Inputs” tab:

//@version=6 indicator("Customizable drawings demo", overlay = true) // Input `group` headers to distinguish the table style inputs and the label style inputs. const string G1 = "Table Style" const string G2 = "Label Style" // Create user inputs for customizing `table` style (position, colors, text size). string tbVerticalInput = input.string("Top", "Position", ["Top", "Middle", "Bottom"], inline = "Pos", group = G1) string tbHorizontalInput = input.string("Right", "Center", ["Left", "Center", "Right"], inline = "Pos", group = G1) color tbBackgroundInput = input.color(#ffeb3bb3, "Background color", inline = "Col", group = G1) color tbBorderInput = input.color(color.white,"Border color", inline = "Col", group = G1) string tbTextSizeInput = input.string(size.large, "Text size", inline = "Txt", group = G1, options = [size.tiny, size.small, size.normal, size.large, size.huge, size.auto]) color tbTextColorInput = input.color(color.black, "Text color", inline = "Txt", group = G1) // Create user inputs for customizing `label` style (size, colors). int lblSizeInput = input.int(16, "Label size", minval = 0, inline = "Lbl", group = G2) color lblColorInput = input.color(color.orange, "Label color", inline = "Lbl", group = G2) color lblTextColorInput = input.color(color.white, "Text color", group = G2) // On last confirmed bar, draw a table to show the `open` and `close` prices, and a label to show their difference. if barstate.islastconfirmedhistory //@variable The table's `position` argument based on the values of `tbVerticalInput` and `tbHorizontalInput`. string tbPos = switch tbVerticalInput == "Top" and tbHorizontalInput == "Left" => position.top_left tbVerticalInput == "Top" and tbHorizontalInput == "Center" => position.top_center tbVerticalInput == "Top" and tbHorizontalInput == "Right" => position.top_right

tbVerticalInput == "Middle" and tbHorizontalInput == "Left" => position.middle_left tbVerticalInput == "Middle" and tbHorizontalInput == "Center" => position.middle_center tbVerticalInput == "Middle" and tbHorizontalInput == "Right" => position.middle_right tbVerticalInput == "Bottom" and tbHorizontalInput == "Left" => position.bottom_left tbVerticalInput == "Bottom" and tbHorizontalInput == "Center" => position.bottom_center tbVerticalInput == "Bottom" and tbHorizontalInput == "Right" => position.bottom_right //@variable A table showing the last confirmed bar's `open` and `close` prices. Inputs customize the table's style. var table displayTable = table.new(tbPos, 2, 2, tbBackgroundInput, border_color = tbBorderInput, border_width = 1) displayTable.cell(0, 0, "Open", text_color = tbTextColorInput, text_size = tbTextSizeInput) displayTable.cell(1, 0, str.tostring(open), text_color = tbTextColorInput, text_size = tbTextSizeInput) displayTable.cell(0, 1, "Close", text_color = tbTextColorInput, text_size = tbTextSizeInput) displayTable.cell(1, 1, str.tostring(close), text_color = tbTextColorInput, text_size = tbTextSizeInput) //@variable The label text, containing the difference between the bar's `open` and `close` prices. string lblText = "Bar body = " + str.tostring(close - open) label.new(bar_index, high, lblText, color = lblColorInput, textcolor = lblTextColorInput, size = lblSizeInput)

Limitations

There are limits to the total number of drawing visuals a script can display on the chart. A single script instance can draw a maximum of approximately 500 lines, boxes, and labels, and a maximum of 100 polylines. If the number of drawings exceeds the limit, a garbage collection mechanism deletes the oldest drawings to keep only the most recent visuals on the chart.

The `max_lines_count`, `max_boxes_count`, `max_labels_count`, and `max_polylines_count` parameters in the indicator() or strategy() declaration statement control the total number of drawings the script can display for each object type. The default value for each `max_*_count` parameter is 50, so if a script does not specify this parameter, it displays the 50 most recent drawings of each type.

Most drawing types have x and y coordinates, so drawing objects move as the user scrolls the chart or zooms in or out. The only exception is *tables*, which are anchored to one of nine fixed positions in the pane itself. See the Tables section below for more details about their unique characteristics.

The leftmost (earliest) x coordinate of a drawing object can be no more than approximately 9999 bars before or 500 bars after the bar on which the script draws it. See this entry in the Techniques FAQ to learn how to work around this issue.

Unlike plots, Pine drawings do not have external uses like creating alerts or exporting data.

Z-index

All visual elements on the chart occupy a position along the z-axis, meaning that some elements appear on top of others. The *z-index* is a value that represents the relative position of elements on the z-axis. Elements with a higher z-index appear on top of elements with a lower z-index.

Pine elements are divided into z-index groups based on their visual type. Each group has its own position in the z-space, and **within the same group**, elements created *last* in the script's logic appear on top of other elements from the same group.

This list orders the visual element groups by *ascending* z-index, i.e., background colors are always at the bottom of z-space, and tables always appear on top of all other elements:

1. Background colors
2. Fills
3. Plots
4. Horizontal levels
5. Linefills
6. Lines
7. Boxes
8. Labels
9. Tables

An element cannot be placed outside the region of z-space that its group occupies – for example, a plot can never appear on top of a table, because tables have the highest z-index. The sole exception to this rule is that programmers can choose to arrange `plot*()`, hline(), and fill() visuals (and only these types of visuals) in z-space in the order in which they appear in the script, by using `explicit_plot_zorder = true` in indicator() or strategy() declaration statements.

When to use

Knowing the strengths of each type of visual element, and how they compare to each other, helps programmers develop efficient scripts that look good. The sections below describe some useful features of each visual element and spotlight a few built-in use cases. For more details about a specific visual element, refer to its User Manual page.

`plot()`

The plot() function displays a data series across the chart. A single plot() visual registers one value for every bar in the main series.

Unlike line and polyline drawings, which connect two or more chart points independent of the bar series, each data point in a plot() series relates to a specific chart bar, and only one point can exist per bar within the same plot series. Plotted “int” and “float” series can represent a variety of constant values, inputs, built-in series like close, and dynamically-calculated results like ta.sma().

The function offers multiple plot styles, including lines, step lines, histograms, areas, crosses, and circles (see the `plot()` parameters section of the Plots page for all available `style` options). Like other plot visuals, plot() outputs can display numeric results in locations other than the main chart pane, such as the status line, price scale, and Data Window.

Most built-in indicators generate plots in their outputs, e.g., RSI, EMA, and Bollinger Bands. Indicators can use several plot styles in the same script to display different kinds of data simultaneously, like the MACD indicator does with its line and histogram plots:

Scripts can also use plot() to create horizontal levels in cases where the dedicated hline() function is not suitable, for example, to display a dynamically-calculated level, or to create a fill between a horizontal line and a fluctuating series.

Unlike `plotshape()` and `plotchar()`, the plot() function cannot display text and doesn’t support “bool” series. However, it can create conditional plots by setting the plot’s series values or colors to na on certain bars.

`plotshape()` and `plotchar()`

The plotshape() and plotchar() functions plot a series across the chart, like plot(), but using a wide range of shapes and characters.

The plotshape() function displays specific `shape.*` styles like crosses, circles, and triangles, while plotchar() displays any single alphanumeric or symbol Unicode character. See the table in the `plotshape()` section of the Text and shapes page for all available `shape.*` styles.

Like other plot visuals, these plots are connected to the main series. They produce one plot value per bar, which can also appear in the status line and Data Window. Both functions accept “int” and “float” series, like plot(), and additionally support “bool” series to display conditional plots.

For instance, the built-in Moon phases indicator uses plotshape() to conditionally draw circles above or below the chart bars, which represent when a new or full moon occurs:

Both plotshape() and plotchar() have several `location` options, which can use either *relative* or *absolute* chart positions:

- They can plot graphics at *absolute* price positions, corresponding to each `series` value.
- They can position graphics *near each bar* in the main series, either above or below the bars.
- They can anchor graphics to the *pane* itself, either at the top or bottom of the pane.

The Moon Phases indicator above uses location.abovebar and location.belowbar arguments to position the circle plots *near* each bar at an automatic, consistent distance, regardless of the bar’s price fluctuation or the plotshape() `series` value.

Relative positioning also makes plotchar() and plotshape() useful for debugging numeric values or conditions. These functions can plot `series` values at a different scale than the chart bars without interfering with the chart scale, unlike plot() series. Hovering over a bar can verify its numeric `series` value in the status line or Data Window – these locations show `0` as the numeric result if there is no visual marker on this particular bar. The functions do not display a visual marker when the `series` value is `false` or na, and they also hide the marker for a `0` value in “int”/“float” series when using relative positioning.

For example, suppose we have a script overlayed in the main pane, and part of its logic generates an “int” series of `0` or `1` values based on some `testCondition`. Using plotchar() with a relative `location` argument quickly verifies that the condition occurs where expected as the function plots a visual marker *only* when the `series` value is `1`. Otherwise, plotting with the absolute `series` locations would distort the main price scale to accommodate a marker appearing on *every* bar at the low price levels `0.00` and `1.00`:

//@variable An "int" series where the value is either `0` or `1`. int mySeries = testCondition ? 1 : 0 // To verify `mySeries`, plot a "!" character at the bottom of the pane only if `mySeries` is `1`. plotchar(mySeries, "Debugging series", "!", location = location.bottom)

The plotshape() and plotchar() functions can also display text alongside their shapes. Unlike for labels, the string must be of type “const”, so the value cannot be dynamic and cannot represent series: the *same* text appears for all the points in the plot.

`plotarrow()`

Similar to `plotshape()` and `plotchar()`, the plotarrow() function plots a series across the chart that presents graphic information using an arrow shape.

A single plotarrow() call plots an arrow on every bar, setting each arrow's direction, position, and length based on the bar's value in the plot `series`. Like other plot visuals, an arrow's numeric value can also display in the script's status line and Data Window.

The plotarrow() function is useful for visualizing changes in the directionality and magnitude of "int" or "float" series values across the chart. The underlying `series` can be at a different scale than the chart bars without visually distorting the main chart scale.

Unlike plotchar() or plotshape(), the plotarrow() function cannot display text and doesn't accept "bool" series. However, the function can still achieve a conditional arrow plot by using na values for its `series` on certain bars.

This simple example indicator uses plotarrow() to visualize a `barGap` series, where each arrow represents the price difference between the current bar's open and the previous bar's close. The function call automatically sets the locations of all the arrows, plotting positive-value arrows below bars and negative-value arrows above bars, and adjusts their lengths relative to the other values in the `barGap` series:

//@version=6 indicator("`plotarrow()` demo", overlay = true) //@variable The difference between the current `open` and previous `close`. float barGap = open - close[1] plotarrow(barGap, "Bar gap", color.rgb(0, 187, 212, 40), color.rgb(223, 64, 251, 40))

`plotbar()` and `plotcandle()`

The plotbar() and plotcandle() functions create custom bar or candle sets on the chart. One call to either function registers four values – the bar or candle's `open`, `high`, `low`, and `close` values – on every bar of the main chart series. As a result, a single plotbar() or plotcandle() call generates *at least four* plots counting towards a script's total plot limit.

Indicators can use these functions to plot a new series separate from the main series, or to build new visuals for the main series itself, like the built-in Bollinger Bars indicator does to create candles with thicker wicks:

As with other plot visuals, the plotbar() and plotcandle() outputs can display in other locations: their numeric results in the script's status line and Data Window (four values per plot) and their latest `close` value on the price scale.

See the Bar plotting page for more information about these functions.

Horizontal levels

The hline() function creates a horizontal level across the script pane at a defined price. The horizontal level extends fully across the visible space of the chart in both directions.

Unlike other plot visuals, a horizontal level's only output is the line drawn in the script pane; it does not display values in the status line, price scale, or Data Window.

This visual element is useful for displaying minimum or maximum prices, thresholds, or support and resistance levels. Many built-in indicators like RSI, CCI, and Stochastic use horizontal levels to represent fixed boundaries for oscillator plots. For example, in the RSI indicator, the horizontal levels are upper and lower bands that represent the oversold and overbought boundaries:

Some built-in indicators also use horizontal levels with fills to create colored bands, which can help to visually distinguish the typical value ranges from outlier ranges, as seen above.

A horizontal level uses a *single, fixed* price value, so it cannot use a dynamically-calculated value or a "series" type like close. Instead, scripts can use plot() to produce similar horizontal lines for dynamically-calculated levels.

Because an hline() call plots only a fixed level in a single color, it is often more performant than similar plot() lines. Adding a horizontal level does not count towards a script's plot limit because the hline() function doesn't create a plot series internally or externally to generate its visual output.

Background and bar coloring

The bgcolor() function sets the background color of the chart space behind a bar, while the barcolor() function sets the body color of a candle.

The functions accept both constant colors and dynamically-calculated colors, so they can use conditional coloring for bars or backgrounds. For instance, the built-in Moon Phases indicator uses bgcolor() to conditionally set the background color of the bars to highlight waxing and waning moon phases:

A bgcolor() call, like most visuals, affects the *script pane* by default. It sets the background color behind the main bar series only when it's overlayed in the main pane — when `overlay = true` for the script or `force_overlay = true` for bgcolor() — otherwise it sets the background for the equivalent space in a separate pane.

By contrast, the barcolor() function **always** colors the main bar series in the *main* pane, even when called by a script executing in a separate pane.

As barcolor() only affects the main chart series, scripts cannot use it to alter the colors of new bars or candles created using plotbar() or plotcandle().

This simple example uses arbitrary bar_index and price conditions to set conditional background and bar colors:

//@version=6 indicator("`bgcolor()` and `barcolor()` demo") // Plot a new candle series for this script, separate to the main pane. Candles are set to main series OHLC values. plotcandle(open, high, low, close, color = color.silver) // Set the background color of the script pane. Color is set conditionally depending on divisibility of `bar_index` by 10 or 6. bgcolor(bar_index % 10 == 0 ? color.new(color.purple, 60) : bar_index % 6 == 0 ? color.new(color.teal, 60) : na) // Set the bar color for the main series. Although script executes in a separate pane, this call executes on the main pane. // Bar's body color is set conditionally to highlight bars with price move of 10 or more. barcolor(math.abs(close - open) >= 10 ? color.orange : color.white)

Note that:

- The script executes in a separate pane, but the barcolor() function colors the main series.
- The barcolor() call does *not* affect the new candles plotted in the script pane.

Fills

Scripts can use fills to set the background color of the space between a pair of plots or horizontal levels. The fill() function accepts both constant and dynamically-calculated colors. There is also a fill() function overload that can create color gradient fills.

Fills between plots are commonly used in built-in indicators to visualize calculated channels or bands, like those used in the Bollinger Bands indicator, which signify the upper and lower standard deviations from its

SMA line:

Fills between horizontal levels are often used in built-in oscillators to highlight chart regions of interest or to differentiate between typical and outlier ranges. For example, the Stochastic Momentum Index (SMI) indicator fills the background between horizontal levels that signify overbought and oversold boundaries, which can help easily identify signs of bullish or bearish trends beyond the filled regions:

The SMI indicator also uses the fill() function's color gradient overload to gradually color the space within the plot lines green or red as they enter the overbought or oversold zones respectively.

Other Pine visuals have their own dedicated fills, like linefills for setting the fill color between two lines, and built-in fill color arguments for drawing objects like boxes and polylines. See the Fills page for more information about the different fill mechanisms available.

> NoteA single fill() function call cannot color the space between a plot and a horizontal level. It can only color the region between either a pair of plots or a pair of horizontal levels.

Lines and polylines

Scripts can draw lines to visually connect any two points on the chart horizontally, vertically, or diagonally.

Like other drawing visuals, lines are independent from the main series, so scripts can draw them at any available chart locations from any bar.

Programmers can specify a line's start and end coordinates using any of the following:

- A bar_index x-coordinate and price y-coordinate.
- A UNIX timestamp x-coordinate and price y-coordinate.
- A chart.point object, where the x-coordinate is a bar index or time value.

Lines can also extend to the left or right of the chart, like those used in the built-in Auto Fib Extension indicator to visualize projected price levels:

Scripts can specify line coordinates at dynamic offsets from the bars on which they're calculated, to draw lines at varying lengths and distances. For instance, the built-in Zig Zag indicator draws straight, angled lines to connect calculated high and low pivots alternatingly across the chart, connecting the last leg to the last available bar. The indicator confirms a point as a high/low pivot only when the price reverses by a specified percentage over time. Therefore, it always draws its lines *into the past* from a different bar than that of the pivot point, and the number of bars between two sequential pivots is not predictable or consistent:

While a line object can connect only two points with a straight line, a polyline can connect *multiple* points on the chart consecutively to create a straight or *curved* line drawing. A polyline uses an array of chart.points to set the coordinates of its sequential line segments, which can contain up to 10,000 chart points.

Polylines can create more complex graphic formations than lines or boxes. A script can connect chart points together with closed polylines to draw polygons, or leave them open-ended to draw geometric series across the chart. Scripts can also use open-ended, curved polylines to draw chart patterns like the Cup and Handle pattern, which identifies a U-shape price trend that is difficult to produce with other drawing visuals:

A script can replicate the visuals made by drawing several sequential line objects with just one polyline object instead. Using polylines can thus help a script to stay under the limits for the total number of lines.

For example, we can use a simplified version of the Zig Zag indicator's logic to illustrate this. Here, we use one polyline drawing to connect pivot points across the chart. The script stores the high and low pivots together in one chart.point array, and creates the polyline object only on the last confirmed historical bar, using barstate.islastconfirmedhistory, drawing it retrospectively across the chart:

//@version=6 indicator("Polyline drawing demo", overlay = true) //@variable The left and right strength of the pivot. int pivotLegsInput = input.int(5, "Pivot leg length", minval = 1) //@variable Switches the `polyline` drawing to a straight or curved line drawing. bool isCurvedPolyline = input.bool(false, "Use curved polyline") //@variable A persistent array that stores high and low pivots for the polyline. var array<chart.point> pointsArray = array.new<chart.point>() // Calculate the high and low pivot prices using `ta.pivot*()` functions. float pivotHigh = ta.pivothigh(pivotLegsInput, pivotLegsInput) float pivotLow = ta.pivotlow(pivotLegsInput, pivotLegsInput) // Add all high and low pivot points sequentially to `pointsArray`, and draw labels at pivots to show prices. if not na(pivotHigh) chart.point highPoint = chart.point.from_index(bar_index - pivotLegsInput, pivotHigh) pointsArray.push(highPoint) label.new(highPoint, "Pivot: " + str.tostring(pivotHigh, "##.##")) if not na(pivotLow) chart.point lowPoint = chart.point.from_index(bar_index - pivotLegsInput, pivotLow) pointsArray.push(lowPoint) label.new(lowPoint, "Pivot: " + str.tostring(pivotLow, "##.##"), style = label.style_label_up) // On the last confirmed bar, draw a polyline across the chart to connect all pivots in `pointsArray`. if barstate.islastconfirmedhistory // First, remove chart points that are too far from the current bar, to prevent errors. // Iterate backwards to avoid index shifting issues when removing items. for i = (pointsArray.size() - 1) to 0 chart.point point = pointsArray.get(i) if (bar_index - point.index) > 9999 pointsArray.remove(i) polyline.new(pointsArray, curved = isCurvedPolyline, line_color = color.purple, line_width = 4) // For reference, display the total number of polyline drawings created by the script in a table cell on the chart. table displayTable = table.new(position.bottom_right, 1, 1, color.purple) displayTable.cell(0, 0, "Total polyline drawings: " + str.tostring(array.size(polyline.all)), text_color = color.white, text_size = size.large)

Note that:

- To avoid runtime errors due to the polyline trying to draw points more than approximately 9999 bars back from the current bar, one alternative is to use chart.point.from_time() to set x-coordinates with UNIX timestamps. Here, we instead use a loop to remove chart.point objects that are too far from the current bar, before drawing the polyline. Note that to accurately remove more than one element from an array using a loop, scripts must iterate *backwards* through the array.
- A polyline's `curved` parameter accepts a "series" argument, so scripts can use Boolean inputs like `isCurvedPolyline` in our example to easily switch between straight or curved line drawings from an indicator's settings.

Scripts can fill the closed space of a polyline drawing using the polyline.new() function's `fill_color` parameter. To fill the space between two lines with a specified color, use linefill objects, which are described in the next section.

Linefills

A linefill is a drawing object, unlike the fills for plots and horizontal levels. Calling the linefill.new() function instantiates an *object* of type "linefill". Scripts can store linefill objects and manipulate them with functions, e.g., to set the associated fill color or retrieve the pair of lines.

Similar to plot fills, linefills are useful for highlighting regions of interest, like calculated channels or trend zones, between two lines on the chart. For example, the built-in Linear Regression indicator uses two linefills between its baseline and its support and resistance lines, which signify the expected price movement ranges. Highlighting the upper and lower channels can make it easier to visually register the price reversal signals:

The exact dimensions occupied by a linefill object are defined by the pair of lines it's attached to. Moving one line farther away, for example, automatically widens the attached linefill. Only one linefill instance can exist between a pair of lines, and it covers only the common space between them. If a pair of lines both extend in the same direction, the linefill can also extend infinitely, as seen in the Auto Pitchfork indicator:

Linefills can fill the space only between two "line" objects. For polylines, the polyline.new() function has a `fill_color` parameter to fill the polyline drawing's closed space.

Boxes

Scripts can use boxes to create custom rectangle drawings on the chart. Like other drawing visuals, a box is a flexible object type, not a series visual, so a script can draw multiple boxes on the same bar, and can set box coordinates at any allowed chart locations ahead or behind the current bar.

Programmers can specify box coordinates using either two diagonal corner points or all four edges of the box, and can define the x-coordinates using bar_index or UNIX timestamp values.

Boxes can be useful for highlighting chart areas of interest, showing price ranges, or visually grouping bars. For example, the built-in Multi-time period charts indicator overlays boxes on the current chart to visualize the corresponding higher timeframe candles:

Boxes can also display text as part of their drawings, as shown in the Seasonality indicator below. Scripts can customize a box's text formatting, alignment, and wrapping, with auto-scaling and auto-wrapping options available to design boxes that are responsive to a user's chart adjustments:

Labels

Labels are drawing objects that can display dynamic text on the chart. They accept "series string" arguments, so they can use changeable text values that aren't known at the start of execution, like inputs or conditionally-calculated expressions, unlike the text displayed by `plotshape()` and `plotchar()`.

Scripts can manage labels in local scopes and draw them at historical or future positions, like other drawing visuals. Each label's position is anchored to the chart's x and y scales at a specific price and bar/time value. However, this position is flexible, as a script can modify a label's coordinates any number of times.

In the built-in Zig Zag indicator, text labels display the calculated pivot prices and, depending on the selected inputs, can also display the reversal price and cumulative volume data within these same labels. The indicator takes advantage of several dynamic label features when building the concatenated label text and setting each label's high/low position, color, and variable offset:

Many `label.style_*` options are available to customize a label's visual appearance, including standard pointing labels and shape-based labels like crosses, triangles, arrows, or flags. The indicator above uses the label.style_none style to display the text on the chart without a visible label shape or outline. See the table in the positioning labels section of the Text and shapes page for all available label styles.

The versatility of labels also makes them particularly useful for debugging scripts. A label can easily show calculated numeric values, strings, or test conditions directly on the chart with little extra code. Scripts can even display empty labels without text to create quick visual markers, for example, to verify that conditions occur on their expected bars.

Tables

Tables are special drawing objects useful for displaying customized, organized information that isn't connected to the chart's price or bar scales.

Tables are anchored to the *pane space* itself, not to any x or y chart coordinates. As such, they remain fixed in size and position when zooming into or scrolling across the chart, even if they are overlayed in the main pane. Like other drawing visuals, tables do not change the data they display when the user hovers over different bars.

Scripts can draw tables in one of *nine* fixed pane positions, specified by the `top`, `middle`, or `bottom` vertical region of the pane and the corresponding `left`, `center`, or `right` horizontal region:

If a script displays more than one table in the same location, the table that is drawn latest in the code replaces any previous tables.

Similar to other drawings, tables have various features that scripts can modify during execution using setter functions. These include *table-specific* features like the frame, border, and height/width in the pane, as well as *cell-specific* features like background color, alignment, and text formatting.

A customization feature unique to tables is that, within the same table object, *each cell* can have different visual properties.

For example, the built-in Performance indicator shows the price percentage change at multiple timeframes for a group of symbols. It uses a variable color intensity for the cell background colors to represent each value's absolute strength. The tabular format and dynamic cell colors make it easy to compare values across symbols and timeframes at a glance:

Unlike for lines, boxes, and labels, scripts **cannot** use getter functions to retrieve properties for tables drawn on the chart. To refer to an attribute of a table later in a script, first store the value in a separate variable.

The Performance indicator above draws its table only *once* during initial execution, on the last bar. This improves script performance and is recommended because a table only displays its *last state*. Tables are thus useful for displaying annotations or general information that won't change during execution, like selected settings, release notes, misconfigurations, etc.

The following example script displays labels for the start and end of each daily trading session. As such, it supports only intraday data and does not display any labels on a "1D" timeframe or higher. The script displays a single-cell table if timeframe.isdwm is true, to notify users of this information:

//@version=6 indicator("Timeframe warning table demo", overlay = true, max_labels_count = 500, behind_chart = false) // Input for the trading session (e.g., "0930-1600" for US stocks regular hours) sessionInput = input.session("0930-1600", "Trading Session") // Display `warningTable` on last bar if the chart timeframe is not intraday. if barstate.islastconfirmedhistory and timeframe.isdwm //@variable A single-cell `table` that inform users about the unsupported timeframe. var table warningTable = table.new(position.middle_center, 1, 1, color.yellow) warningTable.cell(0, 0, "Warning: This indicator supports

only intraday timeframes.\n Switch to a lower timeframe to see output.", text_size = size.large) // Plot a label at the opening price when the session starts if timeframe.isintraday and na(time("", sessionInput)[1]) and not na(time("", sessionInput)) label.new(bar_index, open, "Session Open: " + str.tostring(open, "#.##"), yloc = yloc.abovebar, color = color.green, style = label.style_label_down) // Plot a label at the closing price on the last bar of the session if timeframe.isintraday and not na(time("", sessionInput)[1]) and na(time("", sessionInput)) label.new(bar_index[1], close[1], "Session Close: " + str.tostring(close[1], "#.##"), yloc = yloc.belowbar, color = color.red, style = label.style_label_up)

Note that:

- Using a table in this case ensures that users clearly see the warning, because it appears directly in the chart pane regardless of how their chart is scaled.

Lastly, a table's organized format and fixed pane positions also makes it useful for debugging scripts. See the Tables section of the Debugging page for more details.

Plots

Introduction

The plot() function is the most frequently used function used to display information calculated using Pine scripts. It is versatile and can plot different styles of lines, histograms, areas, columns (like volume columns), fills, circles or crosses.

The use of plot() to create fills is explained in the page on Fills.

This script showcases a few different uses of plot() in an overlay script:

//@version=6 indicator("`plot()`", "", true) plot(high, "Dashed blue `high` line", linestyle = plot.linestyle_dashed) plot(math.avg(close, open), "Crosses in body center", close > open ? color.lime : color.purple, 6, plot.style_cross) plot(math.min(open, close), "Navy step line on body low point", color.navy, 3, plot.style_stepline) plot(low, "Gray dot on `low`", color.gray, 3, plot.style_circles) color VIOLET = #AA00FF color GOLD = #CCCC00 ma = ta.alma(hl2, 40, 0.85, 6) var almaColor = color.silver almaColor := ma > ma[2] ? GOLD : ma < ma[2] ? VIOLET : almaColor plot(ma, "Two-color ALMA", almaColor, 2)

Note that:

- The first plot() call plots a blue line across the bar highs with a dashed line style. The line uses the default width of 1 pixel.
- The second call plots crosses at the mid-point of candle bodies. The crosses are colored lime for up bars and purple for down bars. The argument `6` used for `linewidth` is a relative size, not a pixel value.
- The third call plots a 3-pixel-wide step line following the lowest point of candle bodies.
- The fourth call plots a gray circle at the bars' low.
- The last plot requires some preparation. We first define our bull/bear colors, calculate an Arnaud Legoux Moving Average, then make our color calculations. We initialize our color variable on bar zero only, using var. We initialize it to color.silver, so on the dataset's first bars, until the line is higher/lower than its value two bars ago, it is silver. Looking two bars back creates smoother color transitions than one bar back.

The next script shows other uses of plot() in a separate pane:

//@version=6 indicator("Volume change", format = format.volume) color GREEN = #008000 color GREEN_LIGHT = color.new(GREEN, 50) color GREEN_LIGHTER = color.new(GREEN, 85) color PINK = #FF0080 color PINK_LIGHT = color.new(PINK, 50) color PINK_LIGHTER = color.new(PINK, 90) bool barUp = ta.rising(close, 1) bool barDn = ta.falling(close, 1) float volumeChange = ta.change(volume) volumeColor = barUp ? GREEN_LIGHTER : barDn ? PINK_LIGHTER : color.gray plot(volume, "Volume columns", volumeColor, style = plot.style_columns) volumeChangeColor = barUp ? volumeChange > 0 ? GREEN : GREEN_LIGHT : volumeChange > 0 ? PINK : PINK_LIGHT plot(volumeChange, "Volume change columns", volumeChangeColor, 12, plot.style_histogram) plot(0, "Zero line", color.gray)

Note that:

- We are plotting normal volume values as wide columns above the zero line (see the `style = plot.style_columns` in our plot() call).

- Before plotting the columns we calculate our `volumeColor` by using the values of the `barUp` and `barDn` boolean variables. They become respectively `true` when the current bar's close is higher/lower than the previous one. Note that the "Volume" built-in does not use the same condition; it identifies an up bar with `close > open`. We use the `GREEN_LIGHTER` and `PINK_LIGHTER` colors for the volume columns.
- Because the first plot() call plots *columns*, we do not use the `linewidth` parameter, as it has no effect on columns.
- Our script's second plot is the **change** in volume, which we calculate using `ta.change(volume)`. This value is plotted as a *histogram*, for which the `linewidth` parameter controls the width of the column. We make this width `12` so that histogram elements are thinner than the columns of the first plot. Positive/negative `volumeChange` values plot above/below the zero line; no manipulation is required to achieve this effect.
- Before plotting the histogram of `volumeChange` values, we calculate its color value, which can be one of four different colors. We use the bright `GREEN` or `PINK` colors when the bar is up/down *and* the volume has increased since the last bar (`volumeChange > 0`). Because `volumeChange` is positive in this case, the histogram's element is plotted above the zero line. We use the bright `GREEN_LIGHT` or `PINK_LIGHT` colors when the bar is up/down *and* the volume has **not** increased since the last bar. Because `volumeChange` is negative in this case, the histogram's element is plotted below the zero line.
- Finally, we plot a zero line. As an alternative, using `hline(0)` achieves the same effect.
- We use `format = format.volume` in our indicator() call so that large values displayed for this script are abbreviated like those of the built-in "Volume" indicator.

A plot() call must always be in the script's *global* scope, i.e., at the *beginning* of a non-indented line. Scripts cannot call the plot() function from the indented (local) blocks of user-defined functions or conditional structures like if blocks, for loops, etc. Calls to plot() **can**, however, be designed to plot conditionally in two ways, which we cover in the Plotting conditionally section of this page.

If the plot() call includes `force_overlay = true`, the result always displays on the main chart pane, even if the script is running in a separate pane. Otherwise, the script displays the plot in the same pane in which it runs. By contrast, scripts can only color bars in the main chart pane, regardless of where they run.

`plot()` parameters

The plot() function has the following signature:

```
plot(series, title, color, linewidth, style, trackprice, histbase, offset, join
```

The parameters of plot() are:

`series`

It is the only mandatory parameter. Its argument must be of "series int/float" type. Note that because the auto-casting rules in Pine Script® convert in the "int" → "float" → "bool" direction, a "bool" type variable cannot be used as is for the `series`; it must be converted to an equivalent "int" or "float" type argument. For example, to plot a series based on a "bool" variable `newDay`, we can pass `newDay ? 1 : 0` as the `series` argument to plot 1 when the variable is `true` and 0 when it is `false`.

`title`

Requires a “const string” argument, so it must be known at compile time. The string appears:

- In the script’s scale when the “Chart settings/Scales/Indicator Name Label” field is checked.
- In the Data Window.
- In the “Settings/Style” tab.
- In the dropdown of input.source() fields.
- In the “Condition” field of the “Create Alert” dialog box, when the script is selected.
- As the column header when exporting chart data to a CSV file.

`color`

Accepts “series color”, so can be calculated on the fly, bar by bar. Plotting with na as the color, or any color with a transparency of 100, is one way to hide plots when they are not needed.

`linewidth`

Is the plotted element’s size, but it does not apply to all styles. When a line is plotted, the unit is pixels. It has no impact when plot.style_columns is used.

`style`

The available arguments are:

- plot.style_line (the default): It plots a continous line using the `linewidth` argument in pixels for its width. It does not plot any na values, but does draw a line through them by joining the most recent non-na value to the next non-na value. comes in.
- plot.style_linebr: Allows the plotting of discontinuous lines by not plotting on na values, and not joining such gaps.
- plot.style_stepline: Plots using a staircase effect. Transitions between changes in values are done using a vertical line drawn in middle of bars, as opposed to a point-to-point diagonal joining the midpoints of bars. Can also be used to achieve an effect similar to that of plot.style_linebr, but only if care is taken to plot no color on na values.
- plot.style_area: Plots a line of `linewidth` width, filling the area between the line and the `histbase`. The `color` argument is used for both the line and the fill. You can make the line a different color by using another plot() call. Positive values are plotted above the `histbase`, negative values below it.
- plot.style_areabr: This is similar to plot.style_area but it doesn’t bridge over na values. Another difference is how the indicator’s scale is calculated. Only the plotted values serve in the calculation of the *y* range of the script’s visual space. If only high values situated far away from the `histbase` are plotted, for example, those values will be used to calculate the *y* scale of the script’s visual space. Positive values are plotted above the `histbase`, negative values below it.
- plot.style_columns: Plots columns similar to those of the “Volume” built-in indicator. The `linewidth` value does **not** affect the width of the columns. Positive values are plotted above the `histbase`, negative values below it. Always includes the value of `histbase` in the *y* scale of the script’s visual space.
- plot.style_histogram: Plots columns similar to those of the “Volume” built-in indicator, except that the `linewidth` value is used to determine the width of the histogram’s bars in pixels. Note that since `linewidth` requires an “input int” value, the width of the histogram’s bars cannot vary bar to bar. Positive values are plotted above the `histbase`, negative values below it. Always includes the value of `histbase` in the *y* scale of the script’s visual space.

- plot.style_circles and plot.style_cross: These plot a shape that is not joined across bars unless `join = true` is also used. For these styles, the `linewidth` argument becomes a relative sizing measure — its units are not pixels.

`trackprice`

The default value of this is `false`. When it is `true`, a dotted line made up of small squares will be plotted the full width of the script's visual space. It is often used in conjuction with `show_last = 1, offset = -99999` to hide the actual plot and only leave the residual dotted line.

`histbase`

It is the reference point used with plot.style_area, plot.style_columns and plot.style_histogram. It determines the level separating positive and negative values of the `series` argument. It cannot be calculated dynamically, as an "input int/float" is required.

`offset`

This allows shifting the plot in the past/future using a negative/positive offset in bars. The value cannot change during the script's execution.

`join`

This only affect styles plot.style_circles or plot.style_cross. When `true`, the shapes are joined by a one-pixel line.

`editable`

This boolean parameter controls whether or not the plot's properties can be edited in the "Settings/Style" tab. Its default value is `true`.

`show_last`

Controls the number of bars on which the plot values are visible, counting *backward* from the last bar. Bars beyond the specified amount show na values for this plot. It accepts an "input int" type argument, so it cannot be calculated dynamically.

`display`

Controls the locations where plot values appear, which include the script pane, status line, price scale, and Data Window. The default is display.all. This parameter supports addition and subtraction of `display.*` options, e.g., `display.all - display.pane` hides the pane visuals but displays the plot's numeric results everywhere else, while `display.status_line + display.data_window` displays results only in those two locations.

When this parameter is set to display.none, the script calculates the plot values, but does not display them in the script pane, status line, or Data Window, and the hidden plot does not affect the scale of the script's visual space. This display option can be useful for plots intended for use as external inputs for other scripts, or for plots used with the `{{plot("[plot_title]")}}` placeholder in alertcondition() calls, e.g.:

//@version=6 indicator("") r = ta.rsi(close, 14) xUp = ta.crossover(r, 50) plot(r, "RSI", display = display.none) alertcondition(xUp, "xUp alert", message = 'RSI is bullish at: {{plot("RSI")}}')

`format`

Specifies the numeric format used to represent plot values in the status line, Data Window, and price scale. It accepts one of the following values: format.price, format.percent, or format.volume.

By default, the plot uses the same format as the script's indicator() or strategy() declaration statement. If the plot and declaration use different formats, then the plot's `format` argument takes precedence.

`precision`

Specifies the number of digits included after the decimal point for the plot's numeric values. It accepts a positive integer argument from 0 to 16. This precision affects the results displayed in the status line, Data Window, and price scale.

By default, the plot uses the same precision as the script's indicator() or strategy() declaration statement. If the plot and declaration use different precisions, then the plot's `precision` argument takes precedence.

The `precision` parameter does not affect plot results formatted using the format.volume argument, because the decimal precision rules of the volume formatting take precedence.

`force_overlay`

If `true`, the plotted results display on the main chart pane, even when the script occupies a separate pane. Optional. The default is `false`.

`linestyle`

Controls the style of plotted lines, using one of the following arguments: plot.linestyle_solid, plot.linestyle_dashed, or plot.linestyle_dotted.

This parameter only modifies lines, so the `style` parameter must use one of the following arguments: plot.style_line (the default), plot.style_linebr, plot.style_stepline, plot.style_stepline_diamond, or plot.style_area for it to take effect.

Plotting conditionally

Scripts cannot call the plot() function from conditional structures such as if blocks, but can control plots by varying their plotted values or their color. When no plot is required, you can either plot na values, or plot values using na color or any color with 100 transparency (which also makes it invisible).

Value control

One way to control the display of plots is to plot na values when no plot is needed. Sometimes, values returned by functions such as request.security() will return na values, when `gaps = barmerge.gaps_on` is used, for example. In both these cases it is sometimes useful to plot discontinuous lines. This script shows a few ways to do it:

//@version=6 indicator("Discontinuous plots", "", true) bool plotValues = bar_index % 3 == 0 plot(plotValues ? high : na, color = color.fuchsia, linewidth = 6, style = plot.style_linebr) plot(plotValues ? high : na) plot(plotValues ? math.max(open, close) : na, color = color.navy, linewidth = 6, style = plot.style_cross) plot(plotValues ? math.min(open, close) : na, color = color.navy, linewidth = 6, style = plot.style_circles) plot(plotValues ? low : na, color = plotValues ? color.green : na, linewidth = 6, style = plot.style_stepline)

Note that:

- We define the condition determining when we plot using `bar_index % 3 == 0`, which becomes `true` when the remainder of the division of the bar index by 3 is zero. This will happen every three bars.
- In the first plot, we use plot.style_linebr, which plots the fuchsia line on highs. It is centered on the bar's horizontal midpoint.
- The second plot shows the result of plotting the same values, but without using special care to break the line. What's happening here is that the thin blue line of the plain plot() call is automatically bridged over na values (or *gaps*), so the plot does not interrupt.
- We then plot navy blue crosses and circles on the body tops and bottoms. The plot.style_circles and plot.style_cross style are a simple way to plot discontinuous values, e.g., for stop or take profit levels, or support & resistance levels.
- The last plot in green on the bar lows is done using plot.style_stepline. Note how its segments are wider than the fuchsia line segments plotted with plot.style_linebr. Also note how on the last bar, it only plots halfway until the next bar comes in.
- The plotting order of each plot is controlled by their order of appearance in the script.

This script shows how you can restrict plotting to bars after a user-defined date. We use the input.time() function to create an input widget allowing script users to select a date and time, using Jan 1st 2021 as its default value:

//@version=6 indicator("", "", true) startInput = input.time(timestamp("2021-01-01")) plot(time > startInput ? close : na)

Color control

The Conditional coloring section of the Colors page discusses color control for plots. We'll look here at a few examples.

The value of the `color` parameter in plot() can be a constant, such as one of the built-in constant colors or a color literal. In Pine Script, the qualified type of such colors is called **"const color"** (see the Type system page). They are known at compile time:

//@version=6 indicator("", "", true) plot(close, color = color.gray)

The color of a plot can also be determined using information that is only known when the script begins execution on the first historical bar of a chart (bar zero, i.e., `bar_index == 0` or `barstate.isfirst == true`), as will be the case when the information needed to determine a color depends on the chart the script is running on. Here, we calculate a plot color using the syminfo.type built-in variable, which returns the type of the chart's symbol. The qualified type of `plotColor` in this case will be **"simple color"**:

//@version=6 indicator("", "", true) plotColor = switch syminfo.type "stock" => color.purple "futures" => color.red "index" => color.gray "forex" => color.fuchsia "crypto" => color.lime "fund" => color.orange "dr" => color.aqua "cfd" => color.blue plot(close, color = plotColor) printTable(txt) => var table t = table.new(position.middle_right, 1, 1), table.cell(t, 0, 0, txt, bgcolor = color.yellow) printTable(syminfo.type)

Plot colors can also be chosen through a script's inputs. In this case, the `lineColorInput` variable is of the **"input color"** type:

//@version=6 indicator("", "", true) color lineColorInput = input(#1848CC, "Line color") plot(close, color = lineColorInput)

Finally, plot colors can also be *dynamic* values, i.e., calculated values that can change on each bar. These values are of the **"series color"** type:

//@version=6 indicator("", "", true) plotColor = close >= open ? color.lime : color.red plot(close, color = plotColor)

When plotting pivot levels, one common requirement is to avoid plotting level transitions. Using lines is one alternative, but you can also use plot() like this:

//@version=6 indicator("Pivot plots", "", true) pivotHigh = fixnan(ta.pivothigh(3,3)) plot(pivotHigh, "High pivot", ta.change(pivotHigh) != 0 ? na : color.olive, 3) plotchar(ta.change(pivotHigh), "ta.change(pivotHigh)", "•", location.top, size = size.small)

Note that:

- We use `pivotHigh = fixnan(ta.pivothigh(3,3))` to hold our pivot values. Because ta.pivothigh() only returns a value when a new pivot is found, we use fixnan() to fill the gaps with the last pivot value returned. The gaps here refer to the na values ta.pivothigh() returns when no new pivot is found.
- Our pivots are detected three bars after they occur because we use the argument `3` for both the `leftbars` and `rightbars` parameters in our ta.pivothigh() call.
- The last plot is plotting a continuous value, but it is setting the plot's color to na when the pivot's value changes, so the plot isn't visible then. Because of this, a visible plot will only appear on the bar following the one where we plotted using na color.
- The blue dot indicates when a new high pivot is detected and no plot is drawn between the preceding bar and that one. Note how the pivot on the bar indicated by the arrow has just been detected in the realtime bar, three bars later, and how no plot is drawn. The plot will only appear on the next bar, making the plot visible **four bars** after the actual pivot.

Levels

The hline() function plots horizontal lines at fixed levels (see the page on Levels). The hline() function is useful because it has some unique line styles that are not available with plot(), and is often more performant than similar plot() lines. However, it also has some limitations, namely that it does not accept "series color" arguments, and that its `price` parameter requires an "input int/float" type, so it cannot vary during the script's execution. In cases where scripts need to use dynamically calculated prices or colors, the plot() function can create similar horizontal levels.

You can plot levels with plot() in a few different ways. This script shows a CCI indicator with levels plotted using plot():

//@version=6 indicator("CCI levels with `plot()`") plot(ta.cci(close, 20)) plot(0, "Zero", color.gray, 1, plot.style_circles) plot(bar_index % 2 == 0 ? 100 : na, "100", color.lime, 1, plot.style_linebr) plot(bar_index % 2 == 0 ? -100 : na, "-100", color.fuchsia, 1, plot.style_linebr) plot(200, "200", color.green, 2, trackprice = true, show_last = 1, offset = -99999) plot(-200, "-200", color.red, 2, trackprice = true, show_last = 1, offset = -99999) plot(300, "300", color.new(color.green, 50), 1) plot(-300, "-300", color.new(color.red, 50), 1)

Note that:

- The zero level is plotted using plot.style_circles.

- The 100 levels are plotted using a conditional value that only plots every second bar. In order to prevent the na values from being bridged, we use the plot.style_linebr line style.
- The 200 levels are plotted using `trackprice = true` to plot a distinct pattern of small squares that extends the full width of the script's visual space. The `show_last = 1` in there displays only the last plotted value, which would appear as a one-bar straight line if the next trick wasn't also used: the `offset = -99999` pushes that one-bar segment far away in the past so that it is never visible.
- The 300 levels are plotted using a continuous line, but a lighter transparency is used to make them less prominent.

Offsets

The `offset` parameter specifies the number of bars into the past or future that a script plots a given series. By default, a plot's offset is zero, so each plot point aligns with its bar. Negative offsets display plots on bars *before* the current bar, while positive offsets display them on future bars.

For example, this script plots two close series. It displays the values in the *red* series five bars to the left because its `offset` argument is negative, while displaying the values in the *green* series five bars to the right because its `offset` is positive:

//@version=6 indicator("", "", true) plot(close, "Offset of -5 (in the past)", color = color.red, offset = -5) plot(close, "Offset of +5 (in the future)", color = color.lime, offset = 5)

Note that:

- The argument for the `offset` parameter cannot be of type "series"; it must be a "simple" value, which does *not* change during script execution.

Plot count limit

Each script is limited to a maximum plot count of 64. All `plot*()` calls and alertcondition() calls count towards the plot count of a script. Depending on the complexity of the plot and its arguments, certain calls count as *more than one* plot in the total plot count.

For example, a plot() call counts as *one* plot in the total plot count if it uses a "const color" argument for its `color` parameter, because the color is known at compile time:

plot(close, color = color.green)

A plot() call counts as *two* plots in the total plot count if it uses a stronger qualified type for its `color` argument, such as any one of the following, because the resulting color is dynamic:

plot(close, color = syminfo.mintick > 0.0001 ? color.green : color.red) //□ "simple color" plot(close, color = input.color(color.purple)) //□ "input color" plot(close, color = close > open ? color.green : color.red) //□ "series color" plot(close, color = color.new(color.silver, close > open ? 40 : 0)) //□ "series color"

See the Plot limits section of the Limitations page for more information.

Scale

Not all values can be plotted everywhere. Your script's visual space is always bound by upper and lower limits that are dynamically adjusted with the values plotted. An RSI indicator will plot values between 0 and

100, which is why it is usually displayed in a distinct *pane* — or area — above or below the chart. If RSI values were plotted as an overlay on the chart, the effect would be to distort the symbol's normal price scale, unless it just hapenned to be close to RSI's 0 to 100 range. This shows an RSI signal line and a centerline at the 50 level, with the script running in a separate pane:

```
//@version=6 indicator("RSI") myRSI = ta.rsi(close, 20) bullColor = color.from_gradient(myRSI, 50, 80, color.new(color.lime, 70), color.new(color.lime, 0)) bearColor = color.from_gradient(myRSI, 20, 50, color.new(color.red, 0), color.new(color.red, 70)) myRSIColor = myRSI > 50 ? bullColor : bearColor plot(myRSI, "RSI", myRSIColor, 3) hline(50)
```

Note that the *y* axis of our script's visual space is automatically sized using the range of values plotted, i.e., the values of RSI. See the page on Colors for more information on the color.from_gradient() function used in the script.

If we try to plot the symbol's close values in the same space by adding the following line to our script:

```
plot(close)
```

This is what happens:

The chart is on the BTCUSD symbol, whose close prices are around 40000 during this period. Plotting values in the 40000 range makes our RSI plots in the 0 to 100 range indiscernible. The same distorted plots would occur if we placed the RSI indicator on the chart as an overlay.

Merging two indicators

If you are planning to merge two signals in one script, first consider the scale of each. It is impossible, for example, to correctly plot an RSI and a MACD in the same script's visual space because RSI has a fixed range (0 to 100) while MACD doesn't, as it plots moving averages calculated on price.

If both your indicators used fixed ranges, you can shift the values of one of them so they do not overlap. We could, for example, plot both RSI (0 to 100) and the True Strength Indicator (TSI) (-100 to +100) by displacing one of them. Our strategy here will be to compress and shift the TSI values so they plot over RSI:

```
//@version=6 indicator("RSI and TSI") myRSI = ta.rsi(close, 20) bullColor = color.from_gradient(myRSI, 50, 80, color.new(color.lime, 70), color.new(color.lime, 0)) bearColor = color.from_gradient(myRSI, 20, 50, color.new(color.red, 0), color.new(color.red, 70)) myRSIColor = myRSI > 50 ? bullColor : bearColor plot(myRSI, "RSI", myRSIColor, 3) hline(100) hline(50) hline(0) // 1. Compress TSI's range from -100/100 to -50/50. // 2. Shift it higher by 150, so its -50 min value becomes 100. myTSI = 150 + (100 * ta.tsi(close, 13, 25) / 2) plot(myTSI, "TSI", color.blue, 2) plot(ta.ema(myTSI, 13), "TSI EMA", #FF006E) hline(200) hline(150)
```

Note that:

- We have added levels using hline() to situate both signals.
- In order for both signal lines to oscillate on the same range of 100, we divide the TSI value by 2 because it has a 200 range (-100 to +100). We then shift this value up by 150 so it oscillates between 100 and 200, making 150 its centerline.
- The manipulations we make here are typical of the compromises required to bring two indicators with different scales in the same visual space, even when their values, contrary to MACD, are bounded in a fixed range.

Tables

Introduction

Tables are drawing objects that display text and colors within organized cells. They visually arrange cells in columns and rows, similar to a spreadsheet. Unlike other visuals in Pine Script®, tables do not anchor to specific bars or chart coordinates. Instead, scripts display each table drawing at one of nine *fixed positions* within a chart pane – one of the four corners, one of the midpoints between the corners, or the space's center – or stack the tables vertically in the bottom panel. Tables do not move or vanish as users scroll or zoom on the chart; they consistently remain visible at their position until the script directly modifies or deletes them.

A script creates and displays tables in two distinct steps:

- First, it must call the table.new() function to create a table object. The function call defines specific *base properties* of the table, including the table's position, the number of columns and rows, and the default border and background colors, then returns the table's reference (ID).
- After creating the table object, the script can pass its ID in calls to the table.cell() function to populate and display each cell. Each call to the function defines the properties for *one* specific cell, including the cell's text, colors, formatting, and size. The overall size of the table depends on the width and height of each displayed cell. See the Sizing table cells section to learn more.

Scripts can modify all base properties of a table, excluding the `force_overlay` setting, and the number of columns and rows, by passing the object's ID to the built-in `table.set_*()` functions. Likewise, they can modify all properties of *individual cells* by passing the ID to either the `table.cell_set_*()` functions or to additional calls to the table.cell() function. Scripts can also *merge* a specific subset of a table's cells via the table.merge_cells() function. Refer to the Modifying table and cell properties and Merging cells sections to learn more about these functions and how to use them.

A single script can display a maximum of *nine* tables at once – one for each available position. If a script creates a table using the same position as an existing table, it *replaces* the existing table with the new one. The contents of each table always reflect the *final* state of the table as of the *last* chart bar. Past states of the table, or any replaced tables, are *never visible*.

> NoteTables often require more computational resources than other types of drawings, especially if a script creates or updates them frequently across bars or populates many cells. See the Updating tables only when necessary section below to learn the best practices for minimizing the resource demands of table drawings. The examples across this page also demonstrate these practices in action.

Creating and displaying tables

To create a table, start by calling the table.new() function. The function creates a table object with specified base properties and returns its ID. The function's signature is as follows:

```
table.new(position, columns, rows, bgcolor, frame_color, frame_width, border_co
```

Where:

- The `position`, `columns`, and `rows` parameters require arguments. Supplying arguments to all other parameters is optional.
- The `position` parameter specifies the table's position on the chart. It accepts one of the following `position.*` constants: position.top_left, position.top_center, position.top_right, position.middle_left, position.middle_center, position.middle_right, position.bottom_left, position.bottom_center, or position.bottom_right.
- The `columns` parameter requires an "int" value specifying the maximum number of columns that the table can display.
- The `rows` parameter requires an "int" value specifying the maximum number of rows that the table can display.
- The `bgcolor` parameter sets the table's main background color. If not specified, the default background color is na (invisible). If the table's cells also have specified background colors, those colors *overlay* onto the table's main background.
- The `frame_color` parameter sets the color of the table's *outer frame*. The default color is na.
- The `frame_width` parameter sets the width of the frame, in pixels. It accepts an "int" value greater than or equal to 0. A value of 0 (the default) makes the frame invisible.
- The `border_color` parameter sets the color of the inner borders *between* each table cell. The default color is na.
- The `border_width` parameter sets the width of the inner borders, in pixels. It does not affect the table's outer frame. It accepts an "int" value greater than or equal to 0. A value of 0 (the default) makes the borders invisible.
- The `force_overlay` parameter accepts a "bool" value specifying which pane the table occupies when displayed on the chart. If `true`, the table appears on the main chart pane, even if the script occupies a separate pane. If `false`, it appears in the script's pane. Users can *move* the table into a tab in the chart's *bottom panel*, regardless of the `force_overlay` argument, by selecting "Move tables to bottom" from the script's "More" menu. See the Displaying tables in the bottom panel section to learn more.

The table object created by a table.new() call *does not* initially display any visuals, because it does not contain data for any *cells*. For the table to display visuals, the script must call the table.cell() function to define and populate at least one cell. The function has the following signature:

```
table.cell(table_id, column, row, text, width, height, text_color, text_halign,
```

Where:

- The `table_id`, `column`, and `row` parameters require arguments. Supplying arguments to all other parameters is optional.
- The `table_id` parameter requires the ID of the table object.
- The `column` parameter specifies the column index of the cell to populate. It requires an "int" value from 0 to one less than the `columns` argument in the table.new() call. A value of 0 refers to the first column,

1 refers to the second, and so on.

- The `row` parameter specifies the row index of the cell to populate. It requires an “int” value from 0 to one less than the `rows` argument in the table.new() call. A value of 0 refers to the table’s first row, 1 refers to the second, and so on.
- The `text` parameter accepts a “string” argument specifying the text to display in the table’s cell. The default is an empty string.
- The `width` parameter accepts a “float” value specifying the base width of the cell. A value greater than 0 sets the width as a percentage of the pane’s width if the table is on the chart, or as a proportion of the table’s total width if the table is in the bottom panel. A value of 0 (the default) specifies that the width automatically adjusts based on the cell’s text. The cell’s final width is the maximum of the base width and the widths of all other cells in the same column.
- The `height` parameter accepts a “float” value specifying the base height of the cell. A value greater than 0 sets the height as a percentage of the pane’s height if the table is on the chart, or as a proportion of the total allocated height if the table is in the bottom panel. A value of 0 (the default) specifies that the height automatically adjusts based on the cell’s text. The cell’s final height is the maximum of the base height and the heights of all other cells in the same row.
- The `text_color` parameter specifies the color of the displayed text. The default is `#363a45`.
- The `text_halign` parameter sets the horizontal alignment of the cell’s text. The possible arguments are text.align_left, text.align_center, and text.align_right. The default is text.align_center.
- The `text_valign` parameter sets the vertical alignment of the cell’s text. The possible arguments are text.align_top, text.align_center, and text.align_bottom. The default is text.align_center.
- The `text_size` parameter sets the size of the cell’s text. It accepts an “int” value greater than or equal to 0 for specifying the size in typographic points, or one of the `size.*` built-in “string” constants for specifying a preset size. The `size.*` constants and the equivalent “int” sizes are as follows: size.auto (0), size.tiny (8), size.small (10), size.normal (14), size.large (20), and size.huge (36). The default is size.normal.
- The `bgcolor` parameter controls the background color of the cell. The specified color *overlays* onto the table’s main background defined by the table.new() call. If the `bgcolor` argument of the table.cell() call is na (the default), the cell displays the table’s main background color. If the argument is a “color” value with zero transparency, the cell displays that color instead of the main background color. If the argument represents a partially transparent color, the cell displays a *mixture* between that color and the table’s main background color.
- The `tooltip` parameter accepts a “string” value that defines the *tooltip* text for the cell. If not na or an empty string, the cell displays the specified text within a tooltip when the user hovers over the cell. The default is an empty string.
- The `text_font_family` parameter specifies the font family of the cell’s main text. The possible arguments are font.family_default and font.family_monospace. The default is font.family_default.
- The `text_formatting` parameter sets the formatting behavior of the displayed text. It accepts one of the `text.format_*` constants (text.format_none, text.format_italic, or text.format_bold) or an addition expression that combines formatting settings (e.g., `text.format_bold + text.format_italic` for bold, italicized text). The default is text.format_none.

> NoteThe `table.cell_set_*()` setter functions can also populate table cells. However, these functions define only one specific cell property at a time. For consistency, we recommend using the table.cell() function to set a cell’s initial properties, then using additional calls to the function or to the

> `table.cell_set_*()` functions to update the cell as needed. See the Modifying table and cell properties section below to learn more about updating cells.

The following example script demonstrates how to create and display a basic table. It displays a fixed, single-cell table at an input-specified position on the first chart bar. The script first calls the table.new() function to create the table object, then assigns the returned ID to the persistent `displayTable` variable. Then, within a once structure, the script passes the variable to a table.cell() call to populate the table's cell with a formatted string and display the result on the chart:

```
//@version=6 indicator("Creating and displaying tables demo", overlay = true, behind_chart = false) //@variable Sets the position of the table in the chart pane. // The "string" values of the `position.*` constants (e.g., `"middle_center"`) appear as the input's options. string positionInput = input.string( position.middle_center, "Position", [ position.top_left, position.top_center, position.top_right, position.middle_left, position.middle_center, position.middle_right, position.bottom_left, position.bottom_center, position.bottom_right ], tooltip = "Sets the table's position in the pane. The table does not move as you scroll or zoom on the chart." ) //@variable References a `table` object that can display up to one cell. // This `table.new()` call **does not** display a table on the chart. // It only creates an empty `table` object and sets the object's base properties. var table displayTable = table.new(positionInput, columns = 1, rows = 1, bgcolor = chart.fg_color) // This `once` structure executes its code block once, on the first bar. once //@variable A formatted string containing the text to display in the table. string displayText = str.format( "Symbol: {0}\nTimeframe: {1}\nBars: {2}", ticker.standard(syminfo.tickerid), timeframe.period, last_bar_index - bar_index + 1 ) // Call `table.cell()` to create the table's cell and display it on the chart. // Without a call to this function, the script compiles but does not display any data. table.cell(displayTable, column = 0, row = 0, text = displayText, text_color = chart.bg_color, text_size = 30)
```

Note that:

- The table.new() call does not display any visuals; it only creates the table object and defines the object's base properties. The table displays data only after the script populates its cell with the table.cell() call.
- The script assigns the table.new() call to a persistent variable declared with the var keyword to prevent repeated table creation after the first bar. Without using the keyword, an alternative way to achieve the same effect in our example is to move the variable declaration into the once structure.
- The string input in the script displays the "string" values of the `position.*` constants as options. Programmers can use these strings directly in table.new() calls to set a table's position.
- The table does not move as the user scrolls or zooms on the chart. It remains fixed at the input-specified pane position, even while viewing the first chart bar.
- The table.cell() call in this script does not specify a `width` or `height` argument. Therefore, the script sets the table to use the minimum width and height that accommodates the cell's text. See the Sizing table cells section below for more information.
- This script creates and populates the table on only a single bar because the data it displays *does not change* across bars. To display dynamic data in tables, the typical best approach is to use the table.cell() function or other `table.*()` setter functions inside conditional structures that execute code on the *last available bar* (i.e., where the barstate.islast value is `true`). See the Modifying table and cell properties section to learn more about these functions and how to use them efficiently.

A script *must* call the table.cell() function or other setter functions for *each cell* to display. If a script does not define data for a cell at least once, the table automatically *excludes* that cell from its display. Likewise, if the

script does not define cell data for an entire row or column, the table automatically *omits* that row/column when possible.

For example, the simple script below creates a table that can display up to three rows and three columns. It then uses one call to the table.cell() function to populate the table's middle cell, but does not define data for the other cells. Consequently, the table displays only *one* cell rather than nine:

//@version=6 indicator("Unpopulated cells demo") //@variable References a `table` object that can display up to three columns and three rows. // The table includes a frame and cell borders to visually separate displayed cells. var table t = table.new(position.middle_center, columns = 3, rows = 3, frame_color = chart.fg_color, frame_width = 1, border_color = chart.fg_color, border_width = 1) // On the first bar, call `table.cell()` as a method to populate the table's middle cell at column 1, row 1. // Although the table can display nine cells, it displays only one, because the other possible cells are not defined. once t.cell(1, 1, "Col 1\nRow 1", text_color = color.white, text_size = 30, bgcolor = color.blue)

Note that:

- Most built-in functions for reference types, including tables, are callable as functions or methods. This script calls the table.cell() function using *method syntax* for convenience. We also use method syntax across other examples on this page.

To configure the table to display all possible cells, the script must call the table.cell() function separately for each one. The modified script below calls the function within nested for loops. The outer loop iterates over the table's *column* indices, and the inner loop iterates over the *row* indices. Each iteration of the inner loop calls the function to populate the cell at the current column and row. The table in this version displays all nine cells, because the script directly defines data for each one:

//@version=6 indicator("Populating all cells demo") //@variable References a `table` object that can display up to three columns and three rows. // The table includes a frame and cell borders to visually separate displayed cells. var table t = table.new(position.middle_center, columns = 3, rows = 3, frame_color = chart.fg_color, frame_width = 1, border_color = chart.fg_color, border_width = 1) once // Loop through all column indices from 0 to 2. for col = 0 to 2 // Loop through all row indices from 0 to 2 once for each column index. for row = 0 to 2 //@variable A formatted string indicating the current column and row indices. string displayText = str.format("Col {0}\nRow {1}", col, row) // Define the cell at the `col` and `row` indices and populate it with the formatted text. t.cell(col, row, displayText, text_color = color.white, text_size = 30, bgcolor = color.blue)

Sizing table cells

The total width and height of a table depend on the width and height of each displayed *cell*. The `width`, `height`, `text`, and `text_size` arguments in each call to the table.cell() function and the corresponding `table.cell_set_*()` functions affect the visual sizes of table cells. When a table displays on the *chart*, the script sizes each cell using the following logic:

- If the `width` or `height` value specified for a cell is 0, the script sets the initial width or height of the cell to fit the horizontal or vertical span of the text, which depends on both the `text` and `text_size` values.
- If the `width` or `height` value for a cell is a positive number, the script sets the initial width or height as a *percentage* of the chart pane's total width or height. For example, a `width` value of 50 specifies that the cell is at least half (50%) of the width of the chart pane. The specified width/height takes

precedence over the cell's text; if the values are too small to fit the entire text, the cell shows only a *truncated portion* of the text.
- After determining the initial width and height of a cell, the script checks the width of all other cells in the same column and the height of all other cells on the same row. If necessary, it then *increases* the cell's width or height to align with the neighboring cells.
- The directions in which the table expands the width and height of its cells depend on the table's *position* in the pane. For instance, a table in the middle-center position of the pane expands in all directions, whereas a table in the bottom-right corner expands leftward and upward.

> NoteTable cells follow *different* sizing rules when displayed in the chart's *bottom panel*. Refer to the Displaying tables in the bottom panel section below for details.

The following script provides an interactive way to understand the sizing behavior of table cells on the chart. The script creates a single-cell table that displays the text from a text area input. It also includes other inputs to specify the text size, the cell width, the cell height, and the table's chart position. The table expands in all possible directions from the input position based on the span of the rendered text, or the specified "Cell width" and "Cell height" inputs if their values are greater than 0:

//@version=6 indicator("Sizing table cells demo") //@variable Sets the position of the table in the chart pane. string positionInput = input.string(position.middle_center, "Position", [position.top_left, position.top_center, position.top_right, position.middle_left, position.middle_center, position.middle_right, position.bottom_left, position.bottom_center, position.bottom_right]) //@variable A string containing the text to display in the table. Uses "Lorem ipsum" placeholder text by default. string textInput = input.text_area("""Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.""") //@variable The size of the table's text, in typographic points. Use 0 for the "auto" size. int textSizeInput = input.int(24, "Text size", minval = 0) //@variable Specifies the width of the table cell. Use 0 to set the width based on the width of the text. int widthInput = input.int(0, "Cell width", minval = 0) //@variable Specifies the height of the table cell. Use 0 to set the height based on the height of the text. int heightInput = input.int(0, "Cell height", minval = 0) // Create and populate the table on the first bar. once //@variable References a single-cell table with a blue default background. table t = table.new(positionInput, 1, 1, bgcolor = color.blue) // Populate the table's cell using the input values. t.cell(0, 0, textInput, widthInput, heightInput, color.white, text_size = textSizeInput, text_halign = text.align_left)

Note that:

- The script uses a multiline literal string to set the text area input's default value. The string automatically includes the newline control character for each separate line of text, without requiring the `\n` escape sequence. See the Multiline strings section of the Strings page for more information.

When explicitly sizing table cells with nonzero `width` and `height` arguments, we recommend configuring each cell's dimensions carefully to ensure the total width and height of the table do not exceed 100% of the available space, and configuring the text in each cell to avoid truncation. If a table exceeds the available display space, or if a cell's width or height percentage is too small to display the specified text, there is no

way to view all of the table's contents when displaying it on the chart. However, users *can* view all the contents by moving the table to the *bottom panel*. See the next section below to learn more.

> NoteThe *exact* horizontal and vertical span of a table varies with the user's viewport, and tables are limited in the minimum cell width and height they can display. Consequently, in some cases, a table can span *past* the chart's space even if both the total width and height are specified as 100%, especially if it contains many cells. To ensure the best results when programming tables, we recommend testing the visuals on multiple pane and viewport sizes.

A common technique to fit multi-cell tables into a specified space is to divide the required total width and height of a table by the number of columns and rows it contains. For instance, if a table contains five columns, and the required total width is 50% of the pane, set each cell's `width` value to 10 (50 / 5). If a table requires *non-uniform* cell sizes, one possible technique to set dimensions is to compute normalized width and height weights that sum to one, then multiply the weights by the table's total required width/height percentage.

For example, the advanced script below includes a user-defined function named `calcCellDims()`, which creates an array containing the cell width or height percentages for fitting a specified total pane percentage. The function divides elements from an array of arbitrary weights by the array's sum to normalize them, then multiplies the elements by a total percentage to calculate the final width or height values. The script uses the function to calculate cell widths and heights for a multi-cell table on the chart. Each cell in the table displays formatted text indicating the cell's final width and height percentages:

//@version=6 indicator("Non-uniform cell sizing demo") //@function Calculates an array of width or height percentages for sizing table cells. //@param numCells The number of cells per row (for width) or column (for height). //@param totalPercent The total percentage of the chart pane's width or height. //@param weights Optional. The ID of an array containing a proportional weight for each cell. // If `na`, the function creates an array of uniform percentages. //@returns The ID of an array of width or height percentages. The script can use the elements in // `table.cell()` calls to fit a multi-cell table within a specific percentage of the pane. calcCellDims(int numCells, float totalPercent, array<float> weights = na) => if na(weights) array<float> result = array.new<float>(numCells, totalPercent / numCells) else array<float> result = array.new<float>() float totalWt = weights.sum() for weight in weights float cellPercent = totalPercent * weight / totalWt result.push(cellPercent) result //@variable References an array containing relative weights for both the width and height of the table cells. // These weights represent relative proportions of the table's display space. var array<float> cellWeights = array.from(1.0, 2.0, 3.0, 4.0) once //@variable The number of columns and rows in the table. int numColsRows = cellWeights.size() //@variable References a table supporting `numColsRows` columns and rows, with visible cell borders. table t = table.new(position.middle_center, columns = numColsRows, rows = numColsRows, bgcolor = chart.bg_color, border_color = chart.fg_color, border_width = 1) //@variable References an array containing the width and height percentages for each row and column. array<float> cellPercents = calcCellDims(numColsRows, 90, cellWeights) // Loop through the `cellPercents` array to get column indices and width values. for [col, width] in cellPercents // Loop through the array again, once for each column, to get the row index and height. for [row, height] in cellPercents //@variable A formatted string containing the calculated width and height of each cell. string cellText = str.format("W: {0}%, H: {1}%", width, height) // Define the cell at the `col` and `row` indices and display the `cellText` value. table.cell(t, col, row, cellText, width = width, height = height, text_color = chart.fg_color)

Displaying tables in the bottom panel

When a user applies a table-drawing script to the chart, the tables appear in the script's pane by default, or in the main chart pane if the `force_overlay` argument of the table.new() call is `true`. Users can then optionally display the tables within a *tab* in the chart's *bottom panel* by selecting the "Move tables to bottom" option from the "More" menu in the script's status line:

Selecting the option moves all tables drawn by the script into the same tab in the bottom panel, regardless of which chart pane each table occupies. The user can move the tables back onto the chart at any time by selecting "Move tables to chart" from the script's "More" menu, or from the *context menu* opened by a dropdown arrow in the tab's header:

Users can move the tables from more than one script into the bottom panel's tab. However, the panel displays the tables from only **one** script at a time. To specify which script's tables to analyze, select the script name from the list in the tab's context menu:

Tables use different positioning and sizing rules in the bottom panel, and often appear differently from how they appear on the chart. Key differences in the display behaviors include the following:

- The panel stacks all tables *vertically* based on the *order* in which the script creates them, regardless of the `position` or `force_overlay` argument in each table.new() call. The *first* table created by the script automatically appears at the *top*, and the *last* table appears at the *bottom*.
- Each table automatically *stretches horizontally* to at least the width of the panel, regardless of the `width` argument in the table.cell() calls.
- The cells of tables in the bottom panel do not truncate displayed text to fit a specified width or height. They fit the entire text when possible. If a table's text is too tall or wide to fit within the panel's visible space, the panel adds vertical or horizontal scroll bars as necessary to make the full text viewable.
- If a table contains more than one column, a positive `width` argument in each table.cell() call specifies the width of the cell as a relative *proportion* of the table's total width rather than a percentage of the panel's width. For example, suppose a two-column table defines a width of 10 for the cells in the first column and a width of 30 for those in the second. In this case, the first column spans approximately *one-fourth* (10 / (10 + 30)) of the table's total width, and the second spans approximately *three-fourths* (30 / (10 + 30)) of the width.
- If at least *one* table.cell() call includes a positive `height` argument, the corresponding table automatically stretches other necessary rows *vertically* to fit the allocated display space in the pane, and it treats the specified value as a relative proportion of the space's height rather than a direct percentage. Otherwise, the table automatically uses the minimum amount of vertical space required to display the specified text.
- The available vertical space for each table depends on the total number of tables and the rows that they contain. If a specified height proportion is *too small* for the text on a row, the table automatically adjusts its row height to fit the text.

> NoteThere are no script parameters for selectively activating this feature or moving tables to the bottom by default. When creating a script that displays tables, we recommend manually checking their outputs on the chart and in the bottom panel to confirm they appear as intended.

The following example demonstrates how tables behave differently when moved to the bottom panel. The script below creates up to nine tables on the first bar, one for each `position.*` constant. It stores each `position.*` string in an array, then loops through a slice of the array. On each loop iteration, the script calls table.new() to create a table that supports two columns and three rows, then populates it with table.cell() calls using defined widths and heights. The first row in each table includes a number indicating the table's creation order, where 0 refers to the first one, as well as the string indicating the table's chart position:

```
//@version=6 indicator("Displaying tables in the bottom panel demo") //@variable Specifies the number of tables to draw. int tableNumInput = input.int(9, "# of tables to draw", 1, 9) once //@variable References a slice of an array containing each position string. // The script loops through the slice and draws a table for each element. array<string> positions = array.from( position.bottom_left, position.bottom_center, position.bottom_right, position.middle_left, position.middle_center, position.middle_right, position.top_left, position.top_center, position.top_right ).slice(0, tableNumInput) //@variable The width of the first table column. The width increments by two on each loop iteration. float col1Width = 5 // Loop over the indices and position strings in the `positions` array slice. for [i, pos] in positions //@variable References a two-column, three-row table with a thin black frame and inner borders. table t = table.new( pos, 2, 3, frame_color = #000000, frame_width = 1, border_color = #000000, border_width = 1 ) // Merge both cells on the first row. t.merge_cells(0, 0, 1, 0) // Increment the first column width by two, and subtract it from 30 to calculate the second column's width. col1Width += 2 float col2Width = 30 - col1Width // Populate the merged cell in the first row with text indicating the table's creation order. // The cell uses a height of 10. t.cell( 0, 0, str.format("Table {0} ({1})", i, pos), height = 10, text_color = #000000, bgcolor = color.silver ) // Populate the other cells with row and column information. // The second row uses a height of 5, and the third uses a height of 15. t.cell(0, 1, "Col 0, Row 1", width = col1Width, height = 5, text_color = #000000, bgcolor = color.aqua) t.cell(1, 1, "Col 1, Row 1", width = col2Width, height = 5, text_color = #000000, bgcolor = color.orange) t.cell(0, 2, "Col 0, Row 2", width = col1Width, height = 15, text_color = #000000, bgcolor = color.lime) t.cell(1, 2, "Col 1, Row 2", width = col2Width, height = 15, text_color = #000000, bgcolor = color.red)
```

Note that:

- The script uses the table.merge_cells() function to merge the cells on each table's first row. The merged cell inherits all properties, excluding width and height, from the *first cell* in the merge range (at column 0, row 0). See the Merging cells section below to learn more about this function.

When we apply the script to a chart, it displays all the tables in a separate chart pane at their specified positions by default. The total defined width and height of each table is 30. Therefore, when displaying in the pane, each table occupies approximately 30% of the pane's total width and height:

If we select "Move tables to bottom" from the script's "More" menu, all tables move into a tab in the chart's bottom panel, and their display behavior changes. Rather than appearing at the same relative positions in the panel, the tables stack vertically in order from first to last. Each table stretches horizontally to fit the tab, and the tab automatically allocates the necessary amount of vertical space required to display each one:

Note that:

- The columns in each table span a proportional amount of the table's width when they display in the panel. For instance, the first table defines a width of 7 (5 + 2) for the first column and 23 (30 - 7) for the second. Therefore, the first column spans approximately 7 / 30 of the total width, and the second spans approximately 23 / 30 of the total width.

- The rows in each table also span a proportional amount of the total allocated height when possible. However, the available height for each table depends on the total number of tables and their contents. If the bottom panel displays more than one table, it automatically adjusts the row heights to fit the space, because the table.cell() calls specify `height` arguments. If the panel displays enough tables to require scrolling, it sets each row to use the *minimum* height that fits the displayed text. To see this behavior in action, adjust the "# of tables to draw" input in the script's "Settings/Inputs" tab.

Tables in the bottom panel also process text differently from those on the chart. When a table displays on the chart, there is no direct way to interact with the table's text. Users can only view the text or hover over cells to inspect defined tooltips. By contrast, when a table displays in the bottom panel, users can select any of the main text in the table and copy it to their clipboard. For instance, in the following image, we selected all the text in the first table from our example script. We can then copy the entire text at once and paste it elsewhere:

Note that:

- This behavior applies only to the *main text* in a table's cells. Users can view any tooltips in a bottom-panel table by hovering over its cells, just like with tables on the chart, but they cannot select the text from those tooltips.

Modifying table and cell properties

The `table` namespace features multiple *setter functions*, which enable scripts to update the data and visual properties of a table and its cells over time. Programmers often use these functions when creating tables that display dynamic "series" data, or when modifying the appearance of a table in response to specific conditions.

The `table.set_*()` functions modify specific *base properties* of a table, including its position, frame settings, and border settings. The available `table.set_*()` functions are table.set_bgcolor(), table.set_border_color(), table.set_border_width(), table.set_frame_color(), table.set_frame_width(), and table.set_position(). Each of these functions has the following general signature:

```
function_name(table_id, table_property) → void
```

Where:

- `function_name` is the function's identifier.
- The `table_id` parameter requires the ID of the table object.
- `table_property` refers to the table's base property to update. The corresponding parameter name in each function matches one of the parameters in the table.new() function (e.g., `position` for table.set_position()).

For updating specific table *cells*, scripts can use either the table.cell() function or the `table.cell_set_*()` functions. Each successive call to these functions overwrites the current properties for a specific cell. The table.cell() function overwrites *all* cell properties at once. In essence, it redefines the cell entirely on each call. By contrast, each `table.cell_set_*()` function updates only *one* specific cell property, such as the cell's width, height, text, or background color, while leaving all other properties unchanged.

The available `table.cell_set_*()` functions include table.cell_set_bgcolor(), table.cell_set_width(), table.cell_set_height(), table.cell_set_text(), table.cell_set_text_color(), table.cell_set_text_size(), table.cell_set_text_halign(), table.cell_set_text_valign(), table.cell_set_text_font_family(), table.cell_set_text_formatting(), and table.cell_set_tooltip(). The general signature for each of these functions is as follows:

```
function_name(table_id, column, row, cell_property) → void
```

Where:

- `function_name` is the function's identifier.
- The `table_id` parameter requires the ID of the table object.
- The `column` parameter requires the column index of the cell to modify. The index of the first column is 0, and the index of the last is one less than the `columns` argument of the table.new() call.
- The `row` parameter requires the row index of the cell to modify. The index of the first row is 0, and the index of the last is one less than the `rows` argument of the table.new() call.
- `cell_property` refers to the cell property to update. The corresponding parameter name in each function matches one of the parameters of the table.cell() function (e.g., `text` for table.cell_set_text()).

To learn more about the table.cell() function and its signature, refer to the Creating and displaying tables section above.

When using any of these functions to update a table, remember that only the *latest state* of the table, as of the *last available bar*, is visible to the user. All previous states of a table are **never visible**. Therefore, updating tables *before* the last available bar is typically **unnecessary**. Furthermore, updating tables across a chart's history can significantly *increase* the script's runtime requirements, especially if the tables display many dynamic strings or colors.

The most efficient way to populate a table with dynamic data is to call the `table.set_*()`, table.cell(), or `table.cell_set_*()` functions only on the bars where the output is *visible*, i.e., the last historical bar or the current realtime bar. Restricting these calls to the latest available bar avoids consuming resources on repeated display updates that users cannot view. The simplest approach to achieve this effect is to place the setter calls that must work across bars into an if structure that uses barstate.islast in its condition. If the table's outputs depend on data from previous historical bars, use variables, collections, or history-referencing operations to make necessary data available for the setter calls on the last bar.

The following example script calculates the change in the close series over multiple lengths and displays formatted values in a two-column, nine-row table. It performs all calculations on the *last bar*, within an if structure that uses barstate.islast as the condition. On the first bar where the condition is `true`, the script executes multiple table.cell() calls within a nested once structure to set the initial properties of all the table cells. Then, within a loop that executes on all bars where the if structure executes, the script calculates the change values and updates all cells after the first in the second column. The script calls the table.cell_set_text() function to update each cell's text, and it calls table.cell_set_bgcolor() to update the background color:

//@version=6 indicator("Modifying table and cell properties demo", overlay = true, behind_chart = false) if barstate.islast //@variable References a `table` object that supports two columns and nine rows. var table t = table.new(position.top_right, 2, 9, bgcolor = #ffffff, frame_color = #000000, frame_width = 1, border_color

= #000000, border_width = 1) // This `once` structure's block executes only on the first bar where `barstate.islast` is `true`. once // Set the column header cells with fixed, black text and defaults for all other properties. t.cell(0, 0, "Length", text_color = #000000, bgcolor = #cccccc) t.cell(1, 0, "Change", text_color = #000000, bgcolor = #cccccc) //@variable The offset value for the row header text. The value starts at 1 and then doubles within a loop. int offset = 1 // Loop to initialize the cells for the remaining eight rows. for row = 1 to 8 // Set the cells in the first column to display the `offset` value with black text. t.cell(0, row, str.tostring(offset), text_color = #000000) // Set the cells in the second column to use a white text color. No text is initially specified. t.cell(1, row, text_color = #ffffff) offset *= 2 // The code in this scope executes on all bars where `barstate.islast` is `true`. //@variable The bar offset for `[]` operations. The value starts at 1 and then doubles within a loop. int offset = 1 // Loop through all row indices after the first. for row = 1 to 8 // Calculate the absolute and percentage difference between the current `close` value and the // value from `offset` bars back. float change = close - close[offset] float changePct = 100 * change / close[offset] // Multiply the `offset` value by two for the next iteration. offset *= 2 // Both of these function calls modify a specific property of a cell without overwriting other properties. // Set the cell in the second column on the row to display formatted text representing the current change values. t.cell_set_text(1, row, str.format("{0,number,0.0000} ({1,number,0.000}%)", change, changePct)) // Set the cell's background to teal if the `change` value is positive, and maroon otherwise. t.cell_set_bgcolor(1, row, change > 0 ? color.teal : color.maroon)

Note that:

- This script can produce the same result by calling the table.cell() function instead of the table.cell_set_text() and table.cell_set_bgcolor() functions. However, each call to the function overwrites *all* properties of a cell. If a property is not specified, the table.cell() function uses the default property when updating a cell. Therefore, all necessary properties must be *explicitly defined* in each call. For instance, if we replace the two `table.cell_set_*()` calls with a single table.cell() call, the script displays the same result only if the call includes `text`, `bgcolor`, *and* `text_color` arguments.

Clearing cells

Scripts can *clear* specific cells within a table by using the table.clear() function. The function deletes the properties for one or more cells within a rectangular region from a starting column and row to an ending column and row. The function's signature is as follows:

```
table.clear(table_id, start_column, start_row, end_column, end_row) → void
```

Where:

- The `table_id` parameter requires the ID of the table object.
- The `start_column` parameter requires the column index of the first cell to clear. The argument must be less than or equal to the `end_column` value.
- The `start_row` parameter requires the row index of the first cell to clear. The argument must be less than or equal to the `end_row` value.
- The `end_column` parameter requires the column index of the last cell to clear. The argument must be greater than or equal to the `start_column` value.
- The `end_row` parameter requires the row index of the last cell to clear. The argument must be greater than or equal to the `start_row` value.

This function is often useful if a script must dynamically reduce the number of cells in a table's display. After a script clears a table cell with this function, the cell's data is no longer defined. Therefore, the table automatically *excludes* the cell from its display. If the call clears an entire row or column, the table automatically omits that row/column when possible. To display a cell after clearing it, the script must *repopulate* the cell by calling the table.cell() or `table.cell_set_*()` functions.

The following example script demonstrates the behavior of cleared cells. The script populates and then clears a table in 50-bar cycles, starting on the last bar. The script first creates a table that supports 10 columns and 5 rows. It then calls the table.cell() function to populate one specific cell on each successive bar. After the script populates all 50 cells, it calls the table.clear() function on the next bar to clear the cells, then begins populating the table again:

//@version=6 indicator("Clearing cells demo") if barstate.islast //@variable References a table that supports 10 columns and 5 rows. var table t = table.new(position.middle_center, 10, 5, chart.bg_color, chart.fg_color, 1, chart.fg_color, 1) //@variable The number of cells currently populated by `table.cell()` calls. var int cellCount = 0 // If the `cellCount` value is 50, clear all cells from column 0, row 0 to column 9, row 4, then reset the value to 0. if cellCount == 50 t.clear(start_column = 0, start_row = 0, end_column = 9, end_row = 4) cellCount := 0 // Calculate the column and row indices of the cell to populate. The script populates each row from left to right. int col = cellCount % 10 int row = int(cellCount / 10) //@variable The color of the current cell, based on the direction of the latest change in the `close` value. color cellColor = close > close[1] ? color.aqua : close < close[1] ? color.fuchsia : na //@variable A string representing the current bar's opening date and time in the exchange time zone. // Each cell displays this text in a tooltip. string cellTooltip = str.format_time(time) // Populate the cell at the `col` and `row` indices. t.cell(col, row, str.tostring(close), text_color = #000000, bgcolor = cellColor, tooltip = cellTooltip) // Increment the `cellCount` value by one. cellCount += 1

Note that:

- The table does not initially display any data after each table.clear() call, because the call in our example erases the properties of every cell. The table displays data again only after the table.cell() call. The table gradually expands its display across bars to show all columns and rows as the script redefines them.

Merging cells

Scripts can *merge* multiple cells within a table into a single cell by calling the table.merge_cells() function. The function merges all cells within a rectangular region from a starting column and row to an ending column and row. After a table merges cells, it cannot separate those cells or merge them with other cells. The final merged cell inherits all properties from the *first cell* in the merge range, excluding its width and height. The dimensions of the merged cell depend on the width and height of all neighboring cells in the columns and rows covered by the range.

The signature of the table.merge_cells() function is as follows:

```
table.merge_cells(table_id, start_column, start_row, end_column, end_row) → voi
```

Where:

- The `table_id` parameter requires the ID of the table object.
- The `start_column` parameter requires the column index of the first cell to merge. The argument must be less than or equal to the `end_column` value.
- The `start_row` parameter requires the row index of the first cell to merge. The argument must be less than or equal to the `end_row` value.
- The `end_column` parameter requires the column index of the final cell to merge. The argument must be greater than or equal to the `start_column` value.
- The `end_row` parameter requires the row index of the final cell to merge. The argument must be greater than or equal to the `start_row` value.

> NoteA merged cell *ignores* the text and styling of all cells in the merge range except for the first. Therefore, to modify the properties of a merged cell, scripts must call the table.cell() or `table.cell_set_*()` functions using the **first** column and row indices in the merge range. For instance, to update a merged cell from column 0, row 1 to column 2, row 3, the calls must use 0 as the `column` argument and 1 as the `row` argument. Using other indices from the merge range *does not* affect the cell.

The following example script demonstrates the behavior of merged cells. On the first bar, the script creates a five-column, five-row table with a white background and a black border and frame, then populates its cells with single-character alphabetical strings from `"A"` to `"Y"`. The script calls the table.merge_cells() function as a method to merge all cells from an input start column and row to an input end column and row. It also uses the table.cell_set_bgcolor() function to color the background of the first cell in the merge range blue:

//@version=6 indicator("Merging cells demo") // Define inputs specifying the indices of the cells to merge. int startColInput = input.int(1, "Start column", 0, 4) int startRowInput = input.int(1, "Start row", 0, 4) int endColInput = input.int(3, "End column", 0, 4) int endRowInput = input.int(3, "End row", 0, 4) if barstate.isfirst //@variable References a table that can display five rows and five columns. var table t = table.new(position.middle_center, 5, 5, #ffffff, #000000, 1, #000000, 1) // Merge all cells from the specified starting column and row to the ending column and row. // This function can define merged cell positions before or after any `table.cell()` calls. t.merge_cells(startColInput, startRowInput, endColInput, endRowInput) //@variable References an array of single-character strings. var array<string> cellStrings = str.split("ABCDEFGHIJKLMNOPQRSTUVWXY", "") // Loop through the `cellStrings` array's indices and elements and populate the table. for [i, letter] in cellStrings // Calculate the column and row index for the cell based on the `i` value. // These indices access cells in order from left to right, moving to the next row for each multiple of five. int col = i % 5 int row = math.floor(i / 5) // Populate the cell at the `col` and `row` indices. t.cell(col, row, letter, width = 16, height = 16, text_color = #000000) // Set the background color of the first cell in the merge range to blue. // This setter call causes the entire merged cell to inherit a blue background. if col == startColInput and row == startRowInput t.cell_set_bgcolor(col, row, color.blue)

Note that:

- The merged cell automatically displays the text and background color of the first cell in the merge range. It ignores the properties of all other cells in the range, even though the script also defines them with the looped table.cell() call.
- If the start and end column indices are equal, as well as the start and end row indices, the table.merge_cells() call does not affect the table's appearance, because the merge range spans only a single cell. If the start row or column index is greater than the end index, a runtime error occurs, because tables merge cells exclusively from top to bottom and left to right.

- The table.merge_cells() function can merge cells *before or after* the script defines their properties with the table.cell() function. In this example, we placed the call before the loop that executes the table.cell() calls. Moving the call after the loop achieves the same effect.

Deleting and replacing tables

Tables *persist* indefinitely on the chart or in the bottom panel until a script deletes them. Scripts can delete a table, if necessary, by doing either of the following:

- Calling the table.delete() function. Each call explicitly deletes a table object and removes its ID. Any variables or collections that reference the table store na after the call. This function is most useful if a script must display a table on only *some* of the latest bars.
- Calling the table.new() function with the same `position` argument as a previous call. Each successive call to the function automatically *deletes* the existing table using the specified position and *replaces* it with a new table object that displays cells at that position. Replacing a table can be useful if a script must increase the number of cells in a table or dynamically change merged cell ranges.

The following example script uses tables to create a right-to-left scrolling display of colors indicating the direction of changes in the close series over the latest 30 bars. The script uses an array as a queue to store the change direction values. Then, on the last historical bar or the current realtime bar, it creates a new table object and loops through the array to populate the cells. It populates the second row's cells by calling the table.cell() function for each element that does not match the previous element. For consecutive elements that have the same value, the script calls table.merge_cells() instead to merge all the corresponding cells in the second row, preventing the table's black border from appearing between each one.

Each execution of the table.new() call automatically replaces the table from the previous bar with a new one, because the call uses an unchanging `position` argument. Replacing the table on each realtime bar is useful for our intended effect in this case, as the script cannot dynamically overwrite the merged cell ranges in a single table across bars:

//@version=6 indicator("Replacing tables demo", overlay = true, behind_chart = false) //@variable References an array whose elements represent the direction of the latest 30 `close` values. var array<float> directions = array.new<float>(30, 0) // Push the sign of the change in the current `close` value into the end of the array, and remove the first element. directions.push(math.sign(close - close[1])) directions.shift() if barstate.islast // Call `table.new()` to create a new 30-column, 2-row `table` object on each bar. // Because each successive execution of this call uses the same position, it automatically deletes the previous // table and creates a new table to populate. table t = table.new(position.bottom_right, 30, 2, #ececec, #000000, 1, #000000, 1) // Declare variables to track the starting index and previous direction state for setting merged cells in the loop. int mergeStart = 0 float prevDir = 0 // Loop through the `directions` array. for [i, dir] in directions // Set each cell in the first row to display the analyzed bar offset. t.cell(i, 0, str.tostring(29 - i), width = 2, height = 5, text_color = #000000) // Set the properties of the first cell on the second row and set `prevDir` during the first iteration. if i == 0 t.cell(i, 1, height = 10, bgcolor = dir == 1 ? color.green : dir == -1 ? color.red : color.gray) prevDir := dir continue // On each successive iteration where the `dir` and `prevDir` values do not match, merge the second row's // cells from the `mergeStart` column to the `i - 1` column, update the `mergeStart` and `prevDir` variables, // then set the properties of the cell at column `i` cell in the second row. if dir != prevDir t.merge_cells(mergeStart, 1, i - 1, 1) mergeStart := i prevDir := dir t.cell(i, 1, height = 10, bgcolor = dir == 1 ? color.green : dir == -1 ? color.red : color.gray) // Merge all remaining cells if the `mergeStart` index is less than the final column index. if mergeStart < 29 t.merge_cells(mergeStart, 1, 29, 1)

Tips

Follow these tips to use tables efficiently, and to display updates to tables dynamically in strategies.

Updating tables only when necessary

As mentioned in the introduction, tables often require more computational resources than other types of drawings, especially if a script creates or updates them frequently or populates many cells. In some cases, excessive table creation or updates can significantly increase a script's runtime requirements.

To minimize resource use when working with tables, ensure the script creates and updates them **only when necessary**. Below are the best practices for optimizing table performance:

- If the table or cell properties do not require changes across bars, place all the required `table.*()` function calls inside a conditional structure that executes its local block on a **single bar**. All table visuals persist across executions until explicitly deleted or replaced. Therefore, calling `table.*()` functions repeatedly is **not** necessary to display fixed visuals continuously.
- If any table or cell properties do require changes across bars, place the required `table.set_*()`, table.cell(), or `table.cell_set_*()` calls inside a conditional structure that executes only on the **last** available bar, where the result is **visible**. Users cannot view the past states of tables. Therefore, updating tables repeatedly across history is an unnecessary waste of runtime resources.
- When creating and modifying tables, it is typically best to assign the table.new() call to a persistent variable declared with the var keyword to prevent repeated table creation. Each successive call with the same `position` argument replaces any existing table at that position with a new one to populate, as explained above. Replacing tables when not required can cause excessive resource use.
- If a script *must* replace a table over time to change its cell count, merged cells, etc., keep all successive table.new() calls inside a conditional structure that executes only on the *last bar*. Repeatedly recreating and repopulating tables across history can significantly elevate resource use.

> TipSimilar best practices can apply to other drawing types. For example, if a script draws a label that always displays only the latest values from a series, limiting label.new() and `label.set_*()` calls to the latest bar, where the output is visible, can help improve script performance. See the Profiling and optimization page to learn more about runtime optimization techniques.

Displaying dynamic tables in strategies

Strategies differ from indicators in the way they execute across a dataset. By default, they execute only on the *closing tick* of each bar, regardless of whether that bar has a historical or realtime state. Consequently, with this behavior, if a strategy populates tables with data that can fluctuate during realtime bars, the tables *lag* behind the latest data while a bar is open and show new results only when the bar *closes*.

For a strategy to update its tables or other drawings on each update to an open realtime bar, the script must execute on *each new tick* – not just each bar's close. Programmers can configure a strategy to execute on each new tick by default by including `calc_on_every_tick = true` in the strategy() declaration statement. Script users can also override a strategy's default calculation behavior and enable executions on each new tick by selecting "On realtime bar tick" in the strategy's Script execution settings.

> NoteEnabling strategy executions on each realtime tick can cause the strategy's orders to behave *differently* on realtime bars. Each new update from the data feed is a valid tick for filling orders on realtime bars, while executions on historical bars are limited to filling orders on chart prices or the prices from a lower timeframe. See the `calc_on_every_tick` section of the Strategies page for more information. If a script must place orders only after a realtime bar closes for historical consistency, regardless of this setting, include barstate.isconfirmed in the conditions that control the script's order placement commands.

The example strategy below places market orders to enter and close trades based on the value of a 14-length RSI. The script also draws a two-column, two-row table to display the current strategy.equity value and its one-bar change. The script includes `calc_on_every_tick = true` in the strategy() statement, enabling the table to update after each tick in a realtime bar by default. To promote consistency in the order logic across historical and realtime bars, the script uses the order placement commands inside an if structure that executes only when the barstate.isconfirmed value is `true`:

//@version=6 strategy("Dynamic tables in strategies demo", calc_on_every_tick = true, default_qty_type = strategy.percent_of_equity, default_qty_value = 2) //@variable The 14-length RSI of the `close` series. float rsi = ta.rsi(close, 14) // On a closed bar, place an entry market order if the RSI is above 60, or a closing market order if it's below 50. if barstate.isconfirmed switch rsi > 60 => strategy.entry("Long", strategy.long) rsi < 50 => strategy.close_all() // Plot the RSI and the entry and exit levels on every bar. plot(rsi, "RSI") hline(60, "Entry level", color.green) hline(50, "Exit level", color.red) // Limit the table logic to the last bar, where the output is visible. // Because we use `calc_on_every_tick = true` in the declaration statement, this logic executes on each realtime // tick by default, enabling the display to update in real time. // Without this argument, the display updates only after a bar closes by default. if barstate.islast //@variable References a table that supports two columns and two rows. Displays on the main chart pane by default. var table infoDisplay = table.new(position.top_right, 2, 2, na, chart.fg_color, 1, chart.fg_color, 1, force_overlay = true) // Populate the table's header cells on one bar, because their visuals do not require changes. once infoDisplay.cell(0, 0, "Equity", text_color = #ffffff, bgcolor = #2196f3) infoDisplay.cell(1, 0, "Change", text_color = #ffffff, bgcolor = #2196f3) // Update the other cells that display dynamic values on every subsequent bar. infoDisplay.cell(0, 1, str.tostring(strategy.equity, "0.00"), text_color = chart.fg_color) infoDisplay.cell(1, 1, str.tostring(strategy.equity - strategy.equity[1], "0.00"), text_color = chart.fg_color)

Text and shapes

Introduction

Pine Script® features five different ways to display text or shapes on the chart:

- plotchar()
- plotshape()
- plotarrow()
- Labels created with label.new()
- Tables created with table.new() (see Tables)

Which one to use depends on your needs:

- Tables can display text in various relative positions on a chart, which do not move as users zoom in or scroll the chart horizontally. Their content is not tethered to bars. In contrast, text displayed with plotchar(), plotshape() or label.new() is always tethered to a specific bar, so it will move with the bar's position on the chart. See the page on Tables for more information on them.
- Three elements can display pre-defined shapes: plotshape(), plotarrow() and labels created with label.new().
- plotarrow() cannot display text, only up or down arrows.
- plotchar() and plotshape() can display non-dynamic text on any bar or all bars of the chart.
- plotchar() can only display one character while plotshape() can display strings, including line breaks.
- label.new() can display a maximum of 500 labels on the chart. Its text **can** contain dynamic text, or "series strings". Line breaks are also supported in label text.
- While plotchar() and plotshape() can display text at a fixed offset in the past or the future, which cannot change during the script's execution, each label.new() call can use a "series" offset that can be calculated on the fly.

These are a few things to keep in mind concerning Pine Script strings:

- Since the `text` parameter in both plotchar() and plotshape() require a "const string" argument, it cannot contain values such as prices that can only be known on the bar ("series string").
- To include "series" values in text displayed using label.new(), they will first need to be converted to strings using str.tostring().
- The concatenation operator for strings in Pine is `+`. It is used to join string components into one string, e.g., `msg = "Chart symbol: " + syminfo.tickerid` (where syminfo.tickerid is a built-in variable that returns the chart's exchange and symbol information in string format).
- Characters displayed by all these functions can be Unicode characters, which may include Unicode symbols. See this Exploring Unicode script to get an idea of what can be done with Unicode characters.
- Some functions have parameters that can specify the color, size, font family, and formatting of displayed text. For example, drawing objects like labels, tables, and boxes support text formatting such as bold, italics, and monospace.
- Pine scripts display strings using the system default font. The exact font may vary based on the user's operating system.

This script displays text using the four methods available in Pine Script:

//@version=6 indicator("Four displays of text", overlay = true) plotchar(ta.rising(close, 5), "`plotchar()`", "□", location.belowbar, color.lime, size = size.small) plotshape(ta.falling(close, 5), "`plotchar()`", location = location.abovebar, color = na, text = "•`plotshape()•`\n□", textcolor = color.fuchsia, size = size.huge) if bar_index % 25 == 0 label.new(bar_index, na, "•LABEL•\nHigh = " + str.tostring(high, format.mintick) + "\n□", yloc = yloc.abovebar, style = label.style_none, textcolor = color.black, size = size.normal) printTable(txt) => var table t = table.new(position.middle_right, 1, 1), table.cell(t, 0, 0, txt, bgcolor = color.yellow) printTable("•TABLE•\n" + str.tostring(bar_index + 1) + " bars\nin the dataset")

Note that:

- The method used to display each text string is shown with the text, except for the lime up arrows displayed using plotchar(), as it can only display one character.
- Label and table calls can be inserted in conditional structures to control when their are executed, whereas plotchar() and plotshape() cannot. Their conditional plotting must be controlled using their first argument, which is a "series bool" whose `true` or `false` value determines when the text is displayed.
- Numeric values displayed in the table and labels is first converted to a string using str.tostring().
- We use the + operator to concatenate string components.
- plotshape() is designed to display a shape with accompanying text. Its `size` parameter controls the size of the shape, not of the text. We use na for its `color` argument so that the shape is not visible.
- Contrary to other texts, the table text will not move as you scroll or scale the chart.
- Some text strings contain the □ Unicode arrow (U+1F807).
- Some text strings contain the `\n` sequence that represents a new line.

`plotchar()`

This function is useful to display a single character on bars. It has the following syntax:

```
plotchar(series, title, char, location, color, offset, text, textcolor, editabl
```

See the Reference Manual entry for plotchar() for details on its parameters.

As explained in the Plotting without affecting the scale section of our page on Debugging, the function can be used to display and inspect values in the Data Window or in the indicator values displayed to the right of the script's name on the chart:

//@version=6 indicator("", "", true) plotchar(bar_index, "Bar index", "", location.top)

Note that:

- The cursor is on the chart's last bar.
- The value of bar_index on **that** bar is displayed in indicator values (1) and in the Data Window (2).
- We use location.top because the default location.abovebar will put the price into play in the script's scale, which will often interfere with other plots.

plotchar() also works well to identify specific points on the chart or to validate that conditions are `true` when we expect them to be. This example displays an up arrow under bars where close, high and volume have all been rising for two bars:

//@version=6 indicator("", "", true) bool longSignal = ta.rising(close, 2) and ta.rising(high, 2) and (na(volume) or ta.rising(volume, 2)) plotchar(longSignal, "Long", "▲", location.belowbar, color = na(volume) ? color.gray : color.blue, size = size.tiny)

Note that:

- We use `(na(volume) or ta.rising(volume, 2))` so our script will work on symbols without volume data. If we did not make provisions for when there is no volume data, which is what `na(volume)` does by being `true` when there is no volume, the `longSignal` variable's value would never be `true` because `ta.rising(volume, 2)` yields `false` in those cases.
- We display the arrow in gray when there is no volume, to remind us that all three base conditions are not being met.
- Because plotchar() is now displaying a character on the chart, we use `size = size.tiny` to control its size.
- We have adapted the `location` argument to display the character under bars.

If you don't mind plotting only circles, you could also use plot() to achieve a similar effect:

//@version=6 indicator("", "", true) longSignal = ta.rising(close, 2) and ta.rising(high, 2) and (na(volume) or ta.rising(volume, 2)) plot(longSignal ? low - ta.tr : na, "Long", color.blue, 2, plot.style_circles)

This method has the inconvenience that, since there is no relative positioning mechanism with plot() one must shift the circles down using something like ta.tr (the bar's “True Range”):

`plotshape()`

This function is useful to display pre-defined shapes and/or text on bars. It has the following syntax:

```
plotshape(series, title, style, location, color, offset, text, textcolor, editab
```

See the Reference Manual entry for plotshape() for details on its parameters.

Let's use the function to achieve more or less the same result as with our second example of the previous section:

//@version=6 indicator("", "", true) longSignal = ta.rising(close, 2) and ta.rising(high, 2) and (na(volume) or ta.rising(volume, 2)) plotshape(longSignal, "Long", shape.arrowup, location.belowbar)

Note that here, rather than using an arrow character, we are using the `shape.arrowup` argument for the `style` parameter.

It is possible to use different plotshape() calls to superimpose text on bars. You need to use the newline character sequence, `\n`. The newline needs to be the **last** one in the string for text going up, and the **first** one when you are plotting under the bar and text is going down:

//@version=6 indicator("Lift text", "", true) plotshape(true, "", shape.arrowup, location.abovebar, color.green, text = "A") plotshape(true, "", shape.arrowup, location.abovebar, color.lime, text = "B\n") plotshape(true, "", shape.arrowdown, location.belowbar, color.red, text = "C") plotshape(true, "", shape.arrowdown, location.belowbar, color.maroon, text = "\nD")

The available shapes you can use with the `style` parameter are:

Argument	Shape	With Text	Argument	Shape	With Text
`shape.xcross`			`shape.arrowup`		
`shape.cross`			`shape.arrowdown`		
`shape.circle`			`shape.square`		
`shape.triangleup`			`shape.diamond`		
`shape.triangledown`			`shape.labelup`		
`shape.flag`			`shape.labeldown`		

`plotarrow()`

The plotarrow() function displays up or down arrows of variable length, based on the relative value of the series used in the function's first argument. It has the following syntax:

```
plotarrow(series, title, colorup, colordown, offset, minheight, maxheight, edita
```

See the Reference Manual entry for plotarrow() for details on its parameters.

The `series` parameter in plotarrow() is not a “series bool” as in plotchar() and plotshape(); it is a “series int/float” and there's more to it than a simple `true` or `false` value determining when the arrows are plotted. This is the logic governing how the argument supplied to `series` affects the behavior of plotarrow():

- `series > 0`: An up arrow is displayed, the length of which will be proportional to the relative value of the series on that bar in relation to other series values.
- `series < 0`: A down arrow is displayed, proportionally-sized using the same rules.
- `series == 0 or na(series)`: No arrow is displayed.

The maximum and minimum possible sizes for the arrows (in pixels) can be controlled using the `minheight` and `maxheight` parameters.

Here is a simple script illustrating how plotarrow() works:

//@version=6 indicator("", "", true) body = close - open plotarrow(body, colorup = color.teal, colordown = color.orange)

Note how the height of arrows is proportional to the relative size of the bar bodies.

You can use any series to plot the arrows. Here we use the value of the “Chaikin Oscillator” to control the location and size of the arrows:

//@version=6 indicator("Chaikin Oscillator Arrows", overlay = true) fastLengthInput = input.int(3, minval = 1) slowLengthInput = input.int(10, minval = 1) osc = ta.ema(ta.accdist, fastLengthInput) - ta.ema(ta.accdist, slowLengthInput) plotarrow(osc)

Note that we display the actual “Chaikin Oscillator” in a pane below the chart, so you can see what values are used to determine the position and size of the arrows.

Labels

Labels are only available in v4 and higher versions of Pine Script. They work very differently than plotchar() and plotshape().

Labels are objects, like lines and boxes, or tables. Like them, they are referred to using an ID, which acts like a pointer. Label IDs are of “label” type. As with other objects, labels IDs are “time series” and all the functions used to manage them accept “series” arguments, which makes them very flexible.

> NoteThe Supercharts interface features a set of *drawing tools* that enable users to draw on the chart using mouse actions. Although some of those drawings might resemble the outputs of a script’s drawing objects, it’s crucial to understand that they are **unrelated** entities. Scripts cannot interact with the chart’s drawing tools. Additionally, mouse actions do not directly affect a script’s drawing objects.

Labels are advantageous because:

- They allow “series” values to be converted to text and placed on charts. This means they are ideal to display values that cannot be known before time, such as price values, support and resistance levels, of any other values that your script calculates.
- Their positioning options are more flexible that those of the `plot*()` functions.
- They offer more display modes.
- Contrary to `plot*()` functions, label-handling functions can be inserted in conditional or loop structures, making it easier to control their behavior.
- You can add tooltips to labels.

One drawback to using labels versus plotchar() and plotshape() is that you can only draw a limited quantity of them on the chart. The default is ~50, but you can use the `max_labels_count` parameter in your indicator() or strategy() declaration statement to specify up to 500. Labels, like lines and boxes, are managed using a garbage collection mechanism which deletes the oldest ones on the chart, such that only the most recently drawn labels are visible.

Your toolbox of built-ins to manage labels are all in the `label` namespace. They include:

- label.new() to create labels.
- `label.set_*()` functions to modify the properties of an existing label.
- `label.get_*()` functions to read the properties of an existing label.
- label.delete() to delete labels
- The label.all array which always contains the IDs of all the visible labels on the chart. The array's size will depend on the maximum label count for your script and how many of those you have drawn. `aray.size(label.all)` will return the array's size.

Creating and modifying labels

The label.new() function creates a new label object on the chart. It has the following signatures:

```
label.new(point, text, xloc, yloc, color, style, textcolor, size, textalign, to
```

```
label.new(x, y, text, xloc, yloc, color, style, textcolor, size, textalign, too
```

The difference between the two signatures is how they specify the label's coordinates on the chart. The first signature uses a `point` parameter, which accepts a chart point object. The second signature uses `x` and `y`

either a bar index or time value, depending on the `xloc` property.

The *setter* functions allowing you to change a label's properties are:

- label.set_x()
- label.set_y()
- label.set_xy()
- label.set_point()
- label.set_text()
- label.set_xloc()
- label.set_yloc()
- label.set_color()
- label.set_style()
- label.set_textcolor()
- label.set_size()
- label.set_textalign()
- label.set_tooltip()
- label.set_text_font_family()
- label.set_text_formatting()

They all have a similar signature. The one for label.set_color() is:

```
label.set_color(id, color) → void
```

where:

- `id` is the ID of the label whose property is to be modified.
- The next parameter is the property of the label to modify. It depends on the setter function used. label.set_xy() changes two properties, so it has two such parameters.

This is how you can create labels in their simplest form:

//@version=6 indicator("", "", true) label.new(bar_index, high)

Note that:

- The label is created with the parameters `x = bar_index` (the index of the current bar, bar_index) and `y = high` (the bar's high value).
- We do not supply an argument for the function's `text` parameter. Its default value being an empty string, no text is displayed.
- No logic controls our label.new() call, so labels are created on every bar.
- Only the last 54 labels are displayed because our indicator() call does not use the `max_labels_count` parameter to specify a value other than the ~50 default.

- Labels persist on bars until your script deletes them using label.delete(), or garbage collection removes them.

In the next example we display a label on the bar with the highest high value in the last 50 bars:

//@version=6 indicator("", "", true) // Find the highest `high` in last 50 bars and its offset. Change it's sign so it is positive. LOOKBACK = 50 hi = ta.highest(LOOKBACK) highestBarOffset = -ta.highestbars(LOOKBACK) // Create label on bar zero only. var lbl = label.new(na, na, "", color = color.orange, style = label.style_label_lower_left) // When a new high is found, move the label there and update its text and tooltip. if ta.change(hi) != 0 // Build label and tooltip strings. labelText = "High: " + str.tostring(hi, format.mintick) tooltipText = "Offest in bars: " + str.tostring(highestBarOffset) + "\nLow: " + str.tostring(low[highestBarOffset], format.mintick) // Update the label's position, text and tooltip. label.set_xy(lbl, bar_index[highestBarOffset], hi) label.set_text(lbl, labelText) label.set_tooltip(lbl, tooltipText)

Note that:

- We create the label on the first bar only by using the var keyword to declare the `lbl` variable that contains the label's ID. The `x`, `y` and `text` arguments in that label.new() call are irrelevant, as the label will be updated on further bars. We do, however, take care to use the `color` and `style` we want for the labels, so they don't need updating later.
- On every bar, we detect if a new high was found by testing for changes in the value of `hi`
- When a change in the high value occurs, we update our label with new information. To do this, we use three `label.set*()` calls to change the label's relevant information. We refer to our label using the `lbl` variable, which contains our label's ID. The script is thus maintaining the same label throughout all bars, but moving it and updating its information when a new high is detected.

Here we create a label on each bar, but we set its properties conditionally, depending on the bar's polarity:

//@version=6 indicator("", "", true) lbl = label.new(bar_index, na) if close >= open label.set_text(lbl, "green") label.set_color(lbl, color.green) label.set_yloc(lbl, yloc.belowbar) label.set_style(lbl, label.style_label_up) else label.set_text(lbl, "red") label.set_color(lbl, color.red) label.set_yloc(lbl, yloc.abovebar) label.set_style(lbl, label.style_label_down)

Positioning labels

Labels are positioned on the chart according to *x* (bars) and *y* (price) coordinates. Five parameters affect this behavior: `x`, `y`, `xloc`, `yloc` and `style`:

`x`

Is either a bar index or a time value. When a bar index is used, the value can be offset in the past or in the future (up to a maximum of 500 bars in the future and 10,000 bars in the past). Past or future offsets can also be calculated when using time values. The `x` value of an existing label can be modified using label.set_x() or label.set_xy().

`xloc`

Is either xloc.bar_index (the default) or xloc.bar_time. It determines which type of argument must be used with `x`. With xloc.bar_index, `x` must be an absolute bar index. With xloc.bar_time, `x` must be a UNIX time in milliseconds corresponding to the time value of a bar's open. The `xloc` value of an existing label can be modified using label.set_xloc().

`y`

Is the price level where the label is positioned. It is only taken into account with the default `yloc` value of yloc.price. If `yloc` is yloc.abovebar or yloc.belowbar then the `y` argument is ignored. The `y` value of an existing label can be modified using label.set_y() or label.set_xy().

`yloc`

Can be yloc.price (the default), yloc.abovebar or yloc.belowbar. The argument used for `y` is only taken into account with yloc.price. The `yloc` value of an existing label can be modified using label.set_yloc().

`style`

The argument used has an impact on the visual appearance of the label and on its position relative to the reference point determined by either the `y` value or the top/bottom of the bar when yloc.abovebar or yloc.belowbar are used. The `style` of an existing label can be modified using label.set_style().

These are the available `style` arguments:

Argument	Label	Label with text	Argument	Label	Label with text
`label.style_xcross`			`label.style_label_up`		
`label.style_cross`			`label.style_label_down`		
`label.style_flag`			`label.style_label_left`		
`label.style_circle`			`label.style_label_right`		
`label.style_square`			`label.style_label_lower_left`		
`label.style_diamond`			`label.style_label_lower_right`		
`label.style_triangleup`			`label.style_label_upper_left`		
`label.style_triangledown`			`label.style_label_upper_right`		
`label.style_arrowup`			`label.style_label_center`		
`label.style_arrowdown`			`label.style_none`		

When using xloc.bar_time, the `x` value must be a UNIX timestamp in milliseconds. See the page on Time for more information. The start time of the current bar can be obtained from the time built-in variable. The bar time of previous bars is `time[1]`, `time[2]` and so on. Time can also be set to an absolute value with the timestamp() function. You may add or subtract periods of time to achieve relative time offset.

Let's position a label one day ago from the date on the last bar:

//@version=6 indicator("") daysAgoInput = input.int(1, tooltip = "Use negative values to offset in the future") if barstate.islast MS_IN_ONE_DAY = 24 * 60 * 60 * 1000 oneDayAgo = time - (daysAgoInput * MS_IN_ONE_DAY) label.new(oneDayAgo, high, xloc = xloc.bar_time, style = label.style_label_right)

Note that because of varying time gaps and missing bars when markets are closed, the positioning of the label may not always be exact. Time offsets of the sort tend to be more reliable on 24x7 markets.

You can also offset using a bar index for the `x` value, e.g.:

label.new(bar_index + 10, high) label.new(bar_index - 10, high[10]) label.new(bar_index[10], high[10])

Reading label properties

The following *getter* functions are available for labels:

- label.get_x()
- label.get_y()
- label.get_text()

They all have a similar signature. The one for label.get_text() is:

```
label.get_text(id) → series string
```

where `id` is the label whose text is to be retrieved.

Cloning labels

The label.copy() function is used to clone labels. Its syntax is:

```
label.copy(id) → void
```

Deleting labels

The label.delete() function is used to delete labels. Its syntax is:

```
label.delete(id) → void
```

To keep only a user-defined quantity of labels on the chart, one could use code like this:

//@version=6 MAX_LABELS = 500 indicator("", max_labels_count = MAX_LABELS) qtyLabelsInput = input.int(5, "Labels to keep", minval = 0, maxval = MAX_LABELS) myRSI = ta.rsi(close, 20) if myRSI > ta.highest(myRSI, 20)[1] label.new(bar_index, myRSI, str.tostring(myRSI, "#.00"), style = label.style_none) if array.size(label.all) > qtyLabelsInput label.delete(array.get(label.all, 0)) plot(myRSI)

Note that:

- We define a `MAX_LABELS` constant to hold the maximum quantity of labels a script can accommodate. We use that value to set the `max_labels_count` parameter's value in our indicator() call, and also as the `maxval` value in our input.int() call to cap the user value.
- We create a new label when our RSI breaches its highest value of the last 20 bars. Note the offset of `[1]` we use in `if myRSI > ta.highest(myRSI, 20)[1]`. This is necessary. Without it, the value returned by ta.highest() would always include the current value of `myRSI`, so `myRSI` would never be higher than the function's return value.
- After that, we delete the oldest label in the label.all array that is automatically maintained by the Pine Script runtime and contains the ID of all the visible labels drawn by our script. We use the array.get() function to retrieve the array element at index zero (the oldest visible label ID). We then use label.delete() to delete the label linked with that ID.

Note that if one wants to position a label on the last bar only, it is unnecessary and inefficent to create and delete the label as the script executes on all bars, so that only the last label remains:

// INEFFICENT! //@version=6 indicator("", "", true) lbl = label.new(bar_index, high, str.tostring(high, format.mintick)) label.delete(lbl[1])

This is the efficient way to realize the same task:

//@version=6 indicator("", "", true) if barstate.islast // Create the label once, the first time the block executes on the last bar. var lbl = label.new(na, na) // On all iterations of the script on the last bar, update the label's information. label.set_xy(lbl, bar_index, high) label.set_text(lbl, str.tostring(high, format.mintick))

Realtime behavior

Labels are subject to both *commit* and *rollback* actions, which affect the behavior of a script when it executes on the realtime bar. See the Execution model page to learn more.

This script demonstrates the effect of rollback when running on the realtime bar:

//@version=6 indicator("", "", true) label.new(bar_index, high)

On realtime bars, label.new() creates a new label on every script update, but because of the rollback process, the label created on the previous update on the same bar is deleted. Only the last label created before the realtime bar's close will be committed, and thus persist.

Text formatting

Drawing objects like labels, tables, and boxes have text-related properties that allow users to customize how an object's text appears on the chart. Some common properties include the text color, size, font family, and typographic emphasis.

Programmers can set an object's text properties when initializing it using the label.new(), box.new(), or table.cell() parameters. Alternatively, they can use the corresponding setter functions, e.g., label.set_text_font_family(), table.cell_set_text_color(), box.set_text_halign(), etc.

All three drawing objects have a `text_formatting` parameter, which sets the typographic emphasis to display **bold**, *italicized*, or unformatted text. It accepts the constants text.format_bold, text.format_italic, or text.format_none (no special formatting; default value). It also accepts `text.format_bold + text.format_italic` to display text that is both ***bold and italicized***.

The `size` parameter in label.new() and the `text_size` parameter in box.new() and table.cell() specify the size of the text displayed in the drawn objects. The parameters accept both "string" `size.*` constants and "int" typographic sizes. A "string" `size.*` constant represents one of six fixed sizing options. An "int" size value can be any positive integer, allowing scripts to replicate the `size.*` values or use other customized sizing.

This table lists the `size.*` constants and their equivalent "int" sizes for tables, boxes, and labels:

"string" constant	**"int"** `text_size` **in tables and boxes**	**"int"** `size` **in labels**
`size.auto`	0	0
`size.tiny`	8	~7

“string” constant	“int” `text_size` in tables and boxes	“int” `size` in labels
`size.small`	10	~10
`size.normal`	14	12
`size.large`	20	18
`size.huge`	36	24

The example below creates a label and table on the last available bar. The label displays a string representation of the current close value. The single-cell table displays a string representing the price and percentage difference between the current close and open values. The label’s text size is defined by a string input that returns the value of a built-in `size.*` constant, and the table’s text size is defined by an integer input. Additionally, the script creates a box that visualizes the range from the highest to lowest price over the last 20 bars. The box displays custom text, with a constant `text_size` of 19, to show the distance from the close value to the current highest or lowest price. The two Boolean inputs specify whether all three drawings apply bold and italic text formats to their displayed text:

//@version=6 indicator("Text formatting demo", overlay = true) //@variable The size of the `closeLabel` text, set using "string" `size.*` constants. string closeLabelSize = input.string(size.large, "Label text size", [size.auto, size.tiny, size.small, size.normal, size.large, size.huge], group = "Text size") //@variable The size of the `barMoveTable` text, set using "int" sizes. int tableTextSize = input.int(25, "Table text size", minval = 0, group = "Text size") // Toggles for the text formatting of all the drawing objects (`label`, `table` cell, and `box` texts). bool formatBold = input.bool(false, "Bold emphasis", group = "Text formatting (all objects)") bool formatItalic = input.bool(true, "Italic emphasis", group = "Text formatting (all objects)") // Track the highest and lowest prices in 20 bars. Used to draw a `box` of the high-low range. float recentHighest = ta.highest(20) float recentLowest = ta.lowest(20) if barstate.islast //@variable Label displaying `close` price on last bar. Text size is set using "string" constants. label closeLabel = label.new(bar_index, close, "Close price: " + str.tostring(close, "$0.00"), color = #EB9514D8, style = label.style_label_left, size = closeLabelSize) // Create a `table` cell to display the bar move (difference between `open` and `close` price). float barMove = close - open //@variable Single-cell table displaying the `barMove`. Cell text size is set using "int" values. var table barMoveTable = table.new(position.bottom_right, 1, 1, bgcolor = barMove > 0 ? #31E23FCC : #EE4040CC) barMoveTable.cell(0, 0, "Bar move = " + str.tostring(barMove, "$0.00") + "\n Percent = " + str.tostring(barMove / open, "0.00%"), text_halign = text.align_right, text_size = tableTextSize) // Draw a box to show where current price falls in the range of `recentHighest` to `recentLowest`. //@variable Box drawing the range from `recentHighest` to `recentLowest` in last 20 bars. Text size is set at 19. box rangeBox = box.new(bar_index - 20, recentHighest, bar_index + 1, recentLowest, text_size = 19, bgcolor = #A4B0F826, text_valign = text.align_top, text_color = #4A07E7D8) // Set box text to display how far current price is from the high or low of the range, depending on which is closer. rangeBox.set_text("Current price is " + (close >= (recentHighest + recentLowest) / 2 ? str.tostring(recentHighest - close, "$0.00") + " from box high" : str.tostring(close - recentLowest, "$0.00") + " from box low")) // Set the text formatting of the `closeLabel`, `barMoveTable` cell, and `rangeBox` objects. // `formatBold` and `formatItalic` can both be `true` to combine formats, or both `false` for no special formatting. switch formatBold and formatItalic => closeLabel.set_text_formatting(text.format_bold + text.format_italic) barMoveTable.cell_set_text_formatting(0, 0, text.format_bold + text.format_italic) rangeBox.set_text_formatting(text.format_bold + text.format_italic) formatBold => closeLabel.set_text_formatting(text.format_bold) barMoveTable.cell_set_text_formatting(0, 0, text.format_bold) rangeBox.set_text_formatting(text.format_bold) formatItalic => closeLabel.set_text_formatting(text.format_italic) barMoveTable.cell_set_text_formatting(0, 0,

text.format_italic) rangeBox.set_text_formatting(text.format_italic) => closeLabel.set_text_formatting(text.format_none) barMoveTable.cell_set_text_formatting(0, 0, text.format_none) rangeBox.set_text_formatting(text.format_none)

Welcome to Pine Script® v6

Pine Script® is TradingView's programming language. It allows traders to create their own trading tools and run them on our servers. We designed Pine Script as a lightweight, yet powerful, language for developing indicators and strategies that you can then backtest. Most of TradingView's built-in indicators are written in Pine Script, and our thriving community of Pine Script programmers has published more than 150,000 Community Scripts, half of which are open-source.

Requirements

It’s our explicit goal to keep Pine Script accessible and easy to understand for the broadest possible audience. Pine Script is cloud-based and therefore different from client-side programming languages. While we likely won’t develop Pine Script into a full-fledged language, we do constantly improve it and are always happy to consider requests for new features.

Because each script uses computational resources in the cloud, we must impose limits in order to share these resources fairly among our users. We strive to set as few limits as possible, but will of course have to implement as many as needed for the platform to run smoothly. Limitations apply to the amount of data requested from additional symbols, execution time, memory usage and script size.

Where can I get more information?

- A description of all the Pine Script® operators, variables, and functions can be found in the Reference Manual.
- Use the code from one of TradingView's built-in scripts to start from. Open a new chart and select the "Pine Editor" button in the toolbar. Once the editor window opens, click the dropdown menu and select "Create new/Built in…" to open a dialog box containing a list of TradingView's built-in scripts.
- Information about major releases and modifications to Pine Script (as well as other features) is regularly published on TradingView's blog.
- TradingView's Community scripts contain all user-published scripts. Access them from TradingView's homepage by selecting "Community/Indicators and strategies", or from the charts by using the "Indicators, metrics, and strategies" button and selecting the "Community" tab of the script searching dialog box.
- The Knowledge Base provides many articles about all aspects of the TradingView platform, including indicators, alerts, and Pine Script.

External resources

- PineCoders maintain a public Telegram chat, PineCoders Pine Script Q&A, where programmers can ask questions about how to achieve their goals with Pine Script.
- To raise, vote on, and discuss TradingView-related feature requests, including suggestions about Pine Script , see the `/r/TradingView` subreddit.
- Programmers can also ask questions about Pine Script in the `[pine-script]` tag on StackOverflow.

Debugging

Introduction

TradingView's close integration between the Pine Editor and the Supercharts interface enables efficient, interactive debugging of Pine Script® code. Pine scripts can create dynamic outputs in multiple locations, on and off the chart. Programmers can use these outputs to validate their scripts' behaviors and ensure everything works as expected.

Understanding the most effective tools and methods for inspecting a script helps programmers quickly find and fix potential problems in their code, which improves the overall coding experience. This page explains the script outputs that are the most useful for debugging, along with helpful tips and techniques.

> TipEffective debugging in the Pine Script environment requires an understanding of the Execution model, Time series structure, and Type system. We recommend reviewing these topics, along with string formatting, which the following techniques often use.

Common debug outputs

Pine scripts can create outputs in several ways, each of which has different advantages. While programmers can use any of them to debug their code, some outputs are more optimal for debugging than others.

The functions in the `log.*` namespace log *interactive messages* in the Pine Logs pane. These logging functions are the most convenient and flexible tools for debugging Pine code. Scripts can call `log.*()` functions on any execution from global or local scopes, enabling programmers to analyze historical and realtime script behaviors in depth with minimal code, for example:

//@version=6 indicator("Common debug outputs - Pine Logs") //@variable The natural logarithm of the current `high - low` range. float logRange = math.log(high - low) // Plot the `logRange`. plot(logRange, "logRange") if barstate.isconfirmed // Generate an "error" or "info" message on the confirmed bar, depending on whether `logRange` is defined. switch na(logRange) => log.error("Undefined `logRange` value.") => log.info("`logRange` value: " + str.tostring(logRange)) else // Generate a "warning" message for unconfirmed values. log.warning("Unconfirmed `logRange` value: " + str.tostring(logRange))

Pine drawings display visuals in the main chart pane or the script's separate pane. Although they do not output results in other locations, such as the Data Window or Pine Logs pane, drawings provide convenient ways to visualize a script's data and logic within global or local scopes. Labels are the most flexible drawings for debugging, because they can display colored shapes with formatted text and tooltips at any available chart location, for example:

//@version=6 indicator("Common debug outputs - Pine drawings", overlay = true) //@variable Is `true` when a new bar opens on the "1D" timeframe. bool newDailyBar = timeframe.change("1D") //@variable The previous bar's `bar_index` from when `newDailyBar` last occurred. int closedIndex = ta.valuewhen(newDailyBar, bar_index - 1, 0) //@variable The previous bar's `close` from when `newDailyBar` last occurred. float closedPrice = ta.valuewhen(newDailyBar, close[1], 0) if newDailyBar // Draw a line from the previous `closedIndex` and `closedPrice` to the current values. line.new(closedIndex[1],

closedPrice[1], closedIndex, closedPrice, width = 2) //@variable A string containing debug information to display in a label. string debugText = "'1D' bar closed at: \n(" + str.tostring(closedIndex) + ", " + str.tostring(closedPrice) + ")" //@variable Draws a label at the current `closedIndex` and `closedPrice`. label debugLabel = label.new(closedIndex, closedPrice, debugText, color = color.purple, textcolor = color.white)

The `plot*()` functions can help to debug numeric values, conditions, and colors from a script's global scope. They can output results in up to *four* locations: the main chart pane or the script's pane, the status line, the price scale, and the Data Window. The display on the chart provides a quick view of the series' history, and the numbers in the other output locations show calculated information for specific bars:

//@version=6 indicator("Common debug outputs - Plots") // Plot the `bar_index` in all available locations. plot(bar_index, "bar_index", color.teal, 3)

The bgcolor() function displays colors in the background of the main chart pane or the script's pane. The barcolor() function colors the main chart's bars or candles. Although these outputs are less flexible than Pine Logs, drawings, and plots, they provide a quick way to inspect calculated colors and visualize conditions from the global scope:

//@version=6 indicator("Common debug outputs - Background and bar colors") //@variable Is `true` if the `close` is rising over 2 bars. bool risingPrice = ta.rising(close, 2) // Highlight the chart background and color the main chart bars based on `risingPrice`. bgcolor(risingPrice ? color.new(color.green, 70) : na, title= "`risingPrice` highlight") barcolor(risingPrice ? color.aqua : chart.bg_color, title = "`risingPrice` bar color")

Programmers can use any of these outputs individually or in combination to debug their scripts, depending on the data types and structures that require inspection. See the sections below for detailed information about these outputs and various debugging techniques.

Pine Logs

Pine Logs are interactive, user-defined messages that scripts can create from within global or local scopes at *any point* during code executions on the chart's dataset or requested datasets. They provide a simple, powerful way for programmers to inspect a script's calculations, logic, and execution flow with human-readable text. Using Pine Logs is the primary, most universal technique for debugging Pine Script code.

Pine Logs do not appear on the chart or in the Data Window. Instead, scripts print logged messages with prefixed date and time information in the dedicated *Pine Logs pane*. The inspection and filtering options in the Pine Logs pane help users analyze and navigate logs efficiently.

To access the pane, select "Pine Logs" from the Pine Editor's "More" menu or from the "More" menu in the status line of a script on the chart that uses the `log.*()` functions:

> NoticeOnly *personal* scripts can generate Pine Logs. A published script **cannot** create logs, even if its source code contains `log.*()` function calls. Published libraries can export functions containing `log.*()` calls for use in personal scripts, but they cannot generate logs directly.

Creating logs

Scripts create Pine Logs by calling the functions in the `log.*` namespace: log.info(), log.warning(), or log.error(). All these logging functions have the following two signatures:

```
log.*(message) → void
log.*(formatString, arg0, arg1, ...) → void
```

Where:

- The first overload prints the specified "string" `message` in the Pine Logs pane.
- The second overload creates a formatted string based on its `formatString` and additional arguments, similar to str.format(), then displays the resulting text inside the pane.

Each `log.*()` function has a different *logging level*, allowing programmers to *categorize* the messages shown in the Pine Logs pane:

- The log.info() function creates a message with the "info" level (gray text).
- The log.warning() function creates a message with the "warning" level (orange text).
- The log.error() function creates a message with the "error" level (red text).

This simple script demonstrates the difference between all three `log.*()` functions. It calls log.info(), log.warning(), and log.error() on the first chart bar to print the values of three literal strings in the Pine Logs pane:

//@version=6 indicator("Logging levels demo", overlay = true) // Display logs with all three logging levels in the Pine Logs pane on the first bar. if barstate.isfirst log.info("This is an 'info' message.") log.warning("This is a 'warning' message.") log.error("This is an 'error' message.")

Note that:

- The Pine Logs pane can *filter* messages by their logging level using the menu accessible from the rightmost icon above the logs. See the Filtering logs section to learn more.

Scripts can generate logs at *any point* during their executions, allowing programmers to track information from historical bars, and monitor script behaviors on open realtime bars.

During historical executions, scripts log a new message *once* for each `log.*()` call on any bar. During realtime executions, scripts can call the `log.*()` functions to log messages for *any available tick*, regardless of whether the bar is confirmed. The logs created on realtime ticks are *not* subject to rollback. All logs remain available in the Pine Logs pane until the script restarts.

The example script below calculates the average ratio of each bar's `close - open` value to its `high - low` range. When the range is nonzero, the script prints the values of the calculation's variables in the Pine Logs pane using log.info() if the bar is confirmed or log.warning() if the bar is still open (*unconfirmed*). If the bar's range is zero, making the calculated ratio *undefined*, the script logs an "error" message using log.error():

//@version=6 indicator("Historical and realtime logs demo", "Average bar ratio") //@variable The current bar's change from the `open` to `close`. float numerator = close - open //@variable The current bar's `low` to `high` range. float denominator = high - low //@variable The ratio of the bar's open-to-close change to its full range. float ratio = numerator / denominator //@variable The average `ratio` over 10 *non-na* values. float average = ta.sma(ratio, 10) // Plot the `average`. plot(average, "average", color.purple, 3) if

barstate.isconfirmed switch denominator // Log an "error" message when the `denominator` is 0. 0.0 => log.error("Division by 0 on confirmed bar!\nBar excluded from the average.") // Otherwise, log an "info" message containing a formatted representation of the variables' confirmed values. => log.info("Values (Confirmed):\nnumerator: {0,number,#.########}\ndenominator: {1,number,#.########}" + "\nratio: {2,number,#.########}\naverage: {3,number,#.########}", numerator, denominator, ratio, average) else switch denominator // Log an "error" message for the unconfirmed bar when the `denominator` is 0. 0.0 => log.error("Division by 0 on unconfirmed bar!") // Otherwise, log a "warning" message containing a formatted representation of the unconfirmed values. => log.warning("Values (unconfirmed):\nnumerator: {0,number,#.########}\ndenominator: {1,number,#.########}" + "\nratio: {2,number,#.########}\naverage: {3,number,#.########}", numerator, denominator, ratio, average)

Note that:

- Programmers can use barstate.isconfirmed in the conditions that trigger `log.*()` calls to allow logs for any realtime bar only once, on its *closing tick*, as shown in the example code.
- Users can pause realtime logs by selecting the “Disable logging” button at the top of the Pine Logs pane.
- Allowing logging on any tick of an open bar can result in a large number of logged messages over time. Therefore, we recommend including unique information in the messages or using different logging levels for easy *filtering* from the Pine Logs pane.
- The Pine Logs pane can display the most recent 10,000 logs for historical bars. If a programmer needs to view earlier logs, they can add logic in the code to filter specific `log.*()` calls. See the Custom code filters section for an example.

The following sections use the example script above to demonstrate the Pine Logs pane’s log inspection and filtering features.

Inspecting logs

When a script generates a log by calling any `log.*()` function call, the Pine Logs pane automatically prefixes the logged message with an ISO 8601 *timestamp* representing the log’s assigned time, expressed in the chart’s time zone. The timestamp prefixed to a log on a historical bar represents the bar’s *opening time*, whereas the timestamp for a realtime log represents the *system time* of the log event:

Additionally, each log includes “Source code” and “Scroll to bar” options, which appear when hovering over the message in the Pine Logs pane. These features provide convenient ways for users to inspect and verify a log’s conditions:

The “Source code” option opens the script in the Pine Editor and highlights the code line containing the specific `log.*()` call that triggered the log event:

The “Scroll to bar” option navigates the chart to the bar where the `log.*()` call occurred, then displays a temporary label above the bar, containing its date and time information:

Note that:

- The label’s time information depends on the chart’s timeframe. For example, the label on a “1D” chart contains only the weekday and date, whereas the label on an intraday chart also includes the time of day.

It's important to note that every script on the chart that generates logs maintains an *independent* log history. The Pine Logs pane shows logs for only *one* script at a time. To inspect the logs from a specific script when multiple are on the chart, select its title from the dropdown menu at the top of the pane:

Filtering logs

The Pine Logs pane displays up to 10,000 logged messages from script executions on historical bars. It then appends a new log for each `log.*()` call executed on *any* realtime tick.

To help users navigate high volumes of logs efficiently, the pane includes *filters* that isolate logs based on logging level, start date and time, or search queries. Users can apply these log filters individually or in combination to show only the messages that meet specific criteria. The filters are accessible from the icons below the "x" in the top-right portion of the pane:

For custom filtering options, programmers can use conditional logic to activate specific `log.*()` calls selectively across a script's executions. See the Custom code filters section below to learn more.

Logging level

Selecting the rightmost icon above the messages in the Pine Logs pane opens a "Filter levels" dropdown menu containing checkboxes for each logging level ("Info", "Warning", and "Error"). To remove logs with a specific logging level from the displayed results, uncheck the level from this menu.

In the example below, we deactivated the "info" and "warning" levels for our script's logs, allowing only "error" messages in the Pine Logs pane:

Note that:

- Deactivating logging levels in this menu hides the relevant messages but *does not* stop the execution of those `log.*()` calls in the code. For instance, a log.info() call still executes and adds to the historical log count even when the "Info" option is unchecked.

Start date

The "Start date" option above the logs in the Pine Logs pane opens a dialog box where users can specify a starting date and time to filter the displayed messages:

After the user sets the filter in the dialog box, a tag showing the selected date and time appears above the logs, indicating it is active. With this filter, only logs with prefixed timestamps from the specified start point onward appear in the Pine Logs pane:

Character and pattern search

The "Search" option above the logs in the Pine Logs pane opens a search bar where users can *match* logs containing specific character sequences or patterns, similar to the Pine Editor's "Find/Replace" tool for matching code.

When the search bar is not empty, the pane shows only the messages that fully or partially match the text or pattern, with the matched portion of each message highlighted in blue for visual reference.

Below, we searched “Confirmed” to identify all logs from our example script that contain the term anywhere in their text:

Note that:

- The filtered results include logs containing “confirmed” with a lowercase “c” because the search filter performs *case-insensitive* matching on ASCII characters by default.
- The results also include logs containing “unconfirmed” because the default filter behavior does not exclusively match *whole-word* terms.

The rightmost icon in the search bar opens a dropdown menu containing three options to adjust the search filter’s behavior: Match case, Whole word, and Regex:

Match case

The “Match case” search option activates *case-sensitive* matching. With this setting, the filter’s results include only the logs containing the search query with identical cases for ASCII letter characters.

Here, we enabled the “Match case” setting for our “Confirmed” search, preventing all the script’s logs containing “confirmed” with a *lowercase* “c” from appearing in the results:

Note that:

- The “Match case” setting *does not* affect the search behavior for Unicode letter characters outside the ASCII range (U+0000 - U+007F).

Whole word

The “Whole word” search option activates *whole-word* matching. With this setting enabled, the filter includes logs containing the searched term, but only if it is separated from other text by *whitespace* characters or any of the following non-word characters: `.` (period), `,` (comma), `:` (colon), `;` (semicolon), `'` (apostrophe), or `"` (quotation mark).

For example, searching for “Confirmed” in our script’s logs with the “Whole word” setting prevents the messages containing “unconfirmed” from appearing in the results:

Note that:

- With the “Whole word” setting active, the search filter *cannot* match terms containing whitespaces or the other non-word characters listed above.
- Whole-word search queries can include other Unicode characters outside the ASCII range.

Regex

The “Regex” search option enables advanced, flexible log filtering with regular expressions (regex). In contrast to plain text searches, which only match *literal* character sequences, regex searches can match variable *text patterns* based on the rules defined by the query’s syntax.

With regular expressions, the Pine Logs search filter can isolate logs containing various text structures, simple or complex, such as dates and times with a defined format, alphanumeric sequences with varying digits or letters, sequences of characters within specified Unicode subsets, and more.

For instance, this regex search query specifies that the displayed logs must contain “average:”, with optional trailing whitespace characters, followed by a sequence of characters representing a number greater than 0.5 and less than or equal to 1.0:

```
average:\s*(?:0\.5\d*[1-9]\d*|0\.[6-9]\d*|(?:1\.0*|1))
```

The more advanced search query below specifies that the logs must contain prefixed timestamps representing any time of day equal to or after 09:30 and before 16:00 in the chart’s time zone:

```
(?<=^\[\d{4}-\d{2}-\d{2}\x54)(?:09:3\d:[0-5]\d\.\d{3}|1[1-5]:(?:[0-5]\d[:\.]){2
```

For more information about regular expressions, consult the Regex syntax reference in this manual’s Strings page. Most of the described syntax works the same within the Pine Logs search filter, with a few notable differences:

- The strings used as `regex` arguments in str.match() calls require *two* consecutive backslashes (`\\`) for specifying escape sequences in the pattern (e.g., "`\\w`" means the regex matches a character from the `\w` class). In contrast, the Pine Logs search filter requires only a *single* backslash for escape sequences. Double backslashes in the search bar match the literal `\` character.
- The regex search query can use the syntax `\xhh` or `\uhhhh` to reference Unicode code points in the Basic Multilingual Plane, where each `h` is a *hexadecimal* digit (e.g., `\x67` and `\u0067` refer to U+0067, the `a` character). However, the full-range syntax (`\x{...}`) is not supported.
- The search query *cannot* use Unicode property references, such as `\p{Lu}`, `\p{IsGreek}`, etc.
- The search query can use only the `^` and `$` *boundary assertions* to match a logged message’s start and end boundaries. The `\A`, `\Z`, and `\z` assertions are *not* supported.
- The search query cannot use *pattern modifiers* globally (e.g., `(?m)^abc`). However, it can use some modifiers locally inside *non-capturing groups* (e.g., `(?m:^abc)`).

Custom code filters

If the filtering options in the Pine Logs pane are not sufficient, programmers can control specific `log.*()` calls using inputs and conditional logic.

The script below calculates an RMA of close prices and creates a compound condition from four distinct individual conditions. It plots the RMA on the chart and highlights the background when the `compoundCondition` value is `true`. For debugging, the script uses log.info() to display a formatted string representing the `close` and `rma` values, the values of all the “bool” variables that form the compound condition, and the final `compoundCondition` value.

The `filterLogsInput`, `logStartInput`, and `logEndInput` variables define a custom *time filter* for generating logs. When `filterLogsInput` is `true`, the script uses the time inputs assigned to `logStartInput` and `logEndInput` to filter the log.info() calls, allowing a new log only when the bar’s time is within the specified range:

//@version=6 indicator("Custom code filters demo", overlay = true) //@variable The length for moving average calculations. int lengthInput = input.int(20, "Length", 2) //@variable If `true`, only allows logs within

```
the input time range. bool filterLogsInput = input.bool(true, "Only log in time range", group = "Log filter")
//@variable The starting time for logs if `filterLogsInput` is `true`. int logStartInput = input.time(0, "Start time",
group = "Log filter", confirm = true) //@variable The ending time for logs if `filterLogsInput` is `true`. int
logEndInput = input.time(0, "End time", group = "Log filter", confirm = true) //@variable The RMA of `close`
prices. float rma = ta.rma(close, lengthInput) //@variable Is `true` when `close` exceeds the `rma`. bool
priceBelow = close <= rma //@variable Is `true` when the current `close` is greater than the max of the
previous `hl2` and `close`. bool priceRising = close > math.max(hl2[1], close[1]) //@variable Is `true` when
the `rma` is positively accelerating. bool rmaAccelerating = rma - 2.0 * rma[1] + rma[2] > 0.0 //@variable Is
`true` when the difference between `rma` and `close` exceeds 2 times the current ATR. bool
closeAtThreshold = rma - close > ta.atr(lengthInput) * 2.0 //@variable Is `true` when all the above conditions
occur. bool compoundCondition = priceBelow and priceRising and rmaAccelerating and closeAtThreshold //
Plot the `rma`. plot(rma, "RMA", color.teal, 3) // Highlight the chart background when the
`compoundCondition` occurs. bgcolor(compoundCondition ? color.new(color.aqua, 80) : na, title =
"Compound condition highlight") //@variable If `filterLogsInput` is `true`, is only `true` in the input time range.
Otherwise, always `true`. bool showLog = filterLogsInput ? time >= logStartInput and time <= logEndInput :
true // Log results for a confirmed bar when `showLog` is `true`. if barstate.isconfirmed and showLog
log.info( "\nclose:          {0,number,#.#####}" + "\nrma:           {1,number,#.#####}" + "\npriceBelow:
{2}" + "\npriceRising:     {3}" + "\nrmaAccelerating:  {4}" + "\ncloseAtThreshold:  {5}" + "\n" +
"\ncompoundCondition: {6}", close, rma, priceBelow, priceRising, rmaAccelerating, closeAtThreshold,
compoundCondition )
```

Note that:

- The `input.*()` calls assigned to the `filterLogsInput`, `logStartInput`, and `logEndInput` variables include a `group` argument to group the inputs in the “Settings/Inputs” tab.
- Users can adjust time input values directly on the chart by selecting the script’s status line and moving the displayed time markers with the mouse pointer. Additionally, users can select “Reset points” in the script’s “More” menu to clear the inputs and choose new values.
- The `formatString` argument of the log.info() call uses the Em Space character (U+2003) to align the represented values vertically in the logged text. In contrast to the standard space and tab characters, leading or repeated Em and En spaces are *not* removed from the Pine Logs pane’s displayed messages.

Pine drawings

Pine’s drawing types create chart drawings with specified properties. Scripts can place drawings at any valid chart location during code executions on any bar. Programmers can use these types in a script’s global or local scopes to visualize numeric data, conditions, colors, and strings on the chart. The flexibility of Pine drawings makes them helpful for debugging scripts when other methods do not suffice, namely when a programmer wants to inspect information graphically outside the Pine Logs pane.

However, before debugging a script using drawings, it is crucial to note the following limitations:

- The `expression` argument of a `request.*()` call *cannot* depend on code that creates or modifies drawings. Likewise, an indicator that specifies another context in its declaration statement cannot create drawings from *anywhere* in the code. To debug code that executes on requested data, use Pine Logs instead.
- In contrast to Pine Logs, drawings do not have built-in navigation features. Therefore, users must manually scroll across the chart to inspect drawings created on specific bars.

- Scripts can maintain only a *limited* number of objects of each drawing type. When the number of drawings exceeds the limit, Pine's garbage collector automatically *removes* the oldest ones.

The sections below explain some simple debugging methods using labels and tables. These drawings, especially *labels*, are the most effective for on-chart debugging because they can use dynamic strings to express information from other data types as custom text.

Labels

Labels display colored shapes and text at specified chart coordinates. In contrast to the outputs of the plotshape() and plotchar() functions, labels can display text from "series string" values that change across script executions. Programmers often use labels to visualize the logic of conditional structures and show text representing information from a script's global or local scopes.

The most common techniques for debugging with labels include:

- Drawing a label containing key information anchored to *every* bar that requires inspection.
- Drawing a single label containing information from specific executions at the *end* of the dataset or visible chart.

Drawing on successive bars

When inspecting values of varying magnitudes or different types across bars, a simple approach is to create formatted strings containing the necessary debug information and display them in labels on each bar requiring analysis.

In this example, we've modified the "Average bar ratio" script from the Pine Logs section above. Instead of creating formatted text and displaying information using `log.*()` function calls, this script formats the values separately, then calls label.new() to show the results on the chart within labels anchored to each bar's high:

//@version=6 indicator("Drawing on successive bars demo", "Average bar ratio") //@variable The current bar's change from the `open` to `close`. float numerator = close - open //@variable The current bar's `low` to `high` range. float denominator = high - low //@variable The ratio of the bar's open-to-close change to its full range. float ratio = numerator / denominator //@variable The average `ratio` over 10 *non-na* values. float average = ta.sma(ratio, 10) // Plot the `average`. plot(average, "average", color.purple, 3) if barstate.isconfirmed if denominator == 0 string debugText = "Division by 0 on confirmed bar!\nBar excluded from the average." label.new(bar_index, high, debugText, color = color.red, textcolor = #000000, force_overlay = true) else string debugText = str.format("Values (Confirmed):\nnumerator: {0,number,#.########}\ndenominator: {1,number,#.########}" + "\nratio: {2,number,#.########}\naverage: {3,number,#.########}", numerator, denominator, ratio, average) label.new(bar_index, high, debugText, textcolor = #ffffff, force_overlay = true) else if denominator == 0 string debugText = "Division by 0 on unconfirmed bar!" label.new(bar_index, high, debugText, color = color.red, textcolor = #000000, force_overlay = true) else string debugText = str.format("Values (unconfirmed):\nnumerator: {0,number,#.########}\ndenominator: {1,number,#.########}" + "\nratio: {2,number,#.########}\naverage: {3,number,#.########}", numerator, denominator, ratio, average) label.new(bar_index, high, debugText, color = color.orange, textcolor = #000000, force_overlay = true)

Note that:

- The label.new() calls include `force_overlay = true`, meaning the labels always appear on the main chart pane.
- Unlike the example in the Pine Logs section, this script's outputs **are** subject to *rollback*, meaning the information shown on a bar reflects only the bar's *latest* data. The script does **not** show information for *all* realtime bar updates.

The above example allows users to inspect the script's confirmed values or latest updates on any bar that has a label drawing. However, each bar's results are legible only when the labels do not *overlap*.

An alternative, more *compact* way to display text with labels on successive bars is to utilize the label.new() function's `tooltip` parameter instead of the `text` parameter, as labels show their tooltips only when the mouse pointer *hovers* over them.

In the script version below, we changed all the label.new() calls to use `debugText` as the `tooltip` argument instead of the `text` argument. Now, we can view a specific bar's information without visual clutter from other nearby labels:

//@version=6 indicator("Drawing tooltips on successive bars demo", "Average bar ratio") //@variable The current bar's change from the `open` to `close`. float numerator = close - open //@variable The current bar's `low` to `high` range. float denominator = high - low //@variable The ratio of the bar's open-to-close change to its full range. float ratio = numerator / denominator //@variable The average `ratio` over 10 *non-na* values. float average = ta.sma(ratio, 10) // Plot the `average`. plot(average, "average", color.purple, 3) if barstate.isconfirmed if denominator == 0 string debugText = "Division by 0 on confirmed bar!\nBar excluded from the average." label.new(bar_index, high, color = color.red, tooltip = debugText, force_overlay = true) else string debugText = str.format("Values (Confirmed):\nnumerator: {0,number,#.########}\ndenominator: {1,number,#.########}" + "\nratio: {2,number,#.########}\naverage: {3,number,#.########}", numerator, denominator, ratio, average) label.new(bar_index, high, tooltip = debugText, force_overlay = true) else if denominator == 0 string debugText = "Division by 0 on unconfirmed bar!" label.new(bar_index, high, color = color.red, tooltip = debugText, force_overlay = true) else string debugText = str.format("Values (unconfirmed):\nnumerator: {0,number,#.########}\ndenominator: {1,number,#.########}" + "\nratio: {2,number,#.########}\naverage: {3,number,#.########}", numerator, denominator, ratio, average) label.new(bar_index, high, color = color.orange, tooltip = debugText, force_overlay = true)

When drawing labels across successive bars, it's important to note that the maximum number of labels a script can display is 500. As such, the examples above allow users to inspect information for only the most recent 500 chart bars.

For successive labels on earlier bars, programmers can create conditional logic that limits the drawings to specific *time ranges*, e.g.:

```
if time >= startTime and time <= endTime
    <create_drawing_id>
```

Below, we added a condition to the script that draws a label only when the bar's time is between the chart.left_visible_bar_time and chart.right_visible_bar_time values. This logic restricts the drawings to *visible* chart bars, allowing us to scroll through the chart and inspect labels on any bar:

//@version=6 indicator("Drawing in visible ranges demo", "Average bar ratio") //@variable The current bar's change from the `open` to `close`. float numerator = close - open //@variable The current bar's `low` to `high` range. float denominator = high - low //@variable The ratio of the bar's open-to-close change to its full range. float ratio = numerator / denominator //@variable The average `ratio` over 10 *non-na* values. float average = ta.sma(ratio, 10) // Plot the `average`. plot(average, "average", color.purple, 3) if time >= chart.left_visible_bar_time and time <= chart.right_visible_bar_time if barstate.isconfirmed if denominator == 0 string debugText = "Division by 0 on confirmed bar!\nBar excluded from the average." label.new(bar_index, high, color = color.red, tooltip = debugText, force_overlay = true) else string debugText = str.format("Values (Confirmed):\nnumerator: {0,number,#.########}\ndenominator: {1,number,#.########}" + "\nratio: {2,number,#.########}\naverage: {3,number,#.########}", numerator, denominator, ratio, average) label.new(bar_index, high, tooltip = debugText, force_overlay = true) else if denominator == 0 string debugText = "Division by 0 on unconfirmed bar!" label.new(bar_index, high, color = color.red, tooltip = debugText, force_overlay = true) else string debugText = str.format("Values (unconfirmed):\nnumerator: {0,number,#.########}\ndenominator: {1,number,#.########}" + "\nratio: {2,number,#.########}\naverage: {3,number,#.########}", numerator, denominator, ratio, average) label.new(bar_index, high, color = color.orange, tooltip = debugText, force_overlay = true)

Note that:

- The script *restarts* each time the UNIX timestamps of the chart.left_visible_bar_time or chart.right_visible_bar_time variables change after the user scrolls or zooms on the chart.

Drawing at the end of the chart

When debugging information does not change frequently across executions, or only the information from a specific execution requires inspection, programmers often display it using labels anchored to the *end* of the chart.

The following example displays price and chart information in four separate labels at the end of the chart. The script's `printLabel()` function renders a specified string in a label that always anchors to the last available time in the dataset, regardless of when the function call occurs:

//@version=6 indicator("Drawing labels at the end of the chart demo", "Chart info", true, behind_chart = false) //@function Draws a label to display the `info` text at the latest available time. // Each instance of a call to this function updates its label text across executions. //@param info The string to display. //@param price Optional. The y-coordinate of the label. If `na`, the function draws the label above the last bar. // The default is `na`. //@param textColor Optional. The color of the displayed text. If `na`, the label uses `chart.fg_color`. // The default is `na`. //@param size Optional. The size of the label in typographic points. The default is 18. //@returns A `label` object with dynamic text. printLabel(string info, simple float price = na, simple color textColor = na, simple int size = 18) => var int anchorTime = math.max(last_bar_time, chart.right_visible_bar_time) var color col = nz(textColor, chart.fg_color) var yloc = na(price) ? yloc.abovebar : yloc.price var label result = label.new(anchorTime, price, na, xloc.bar_time, yloc, na, label.style_none, col, size, force_overlay = true) result.set_text(info) // Call `printLabel()` on the first bar to display "Chart info:" and formatted chart information. if barstate.isfirst printLabel("Chart info:" + str.repeat("\n", 6), textColor = color.teal) printLabel(str.format("Symbol: {0}, Type: {1}, Timeframe: {2}\nStandard chart: {3}, Replay active: {4}", ticker.standard(), syminfo.type, timeframe.period, chart.is_standard, str.contains(syminfo.tickerid, "replay")) + str.repeat("\n", 3)) // On the last available bar, call `printLabel()` to display the latest OHLCV values and total bar count. if barstate.islast printLabel(str.format("O: {0,number,#.#####}, H:

{1,number,#.#####}, L: {2,number,#.#####}, C: {3,number,#.#####}, V: {4}", open, high, low, close, str.tostring(volume, format.volume)) + "\n") printLabel("Total bars: " + str.tostring(bar_index + 1))

Note that:

- The `printLabel()` function draws *one* label per function call instance. The label's `x` property is the maximum of the last_bar_time and chart.right_visible_bar_time values, ensuring it appears above the last available bar.
- On each execution of a `printLabel()` instance, the label's `text` property updates to reflect the latest `info` value.
- The label.new() call in the `printLabel()` function includes `force_overlay = true`, meaning the drawing always appears in the main chart pane.
- This script uses *four* distinct `printLabel()` calls. The first three append repeated newline characters (`\n`) in the `info` argument to prevent the label text from overlapping.

Tables

Tables display text within cells arranged in columns and rows at fixed locations in the chart pane's visual space. In contrast to other drawing types, which create visuals on the chart at specified *coordinates*, tables appear at one of nine unique, *bar-agnostic* locations defined by the `position.*` constants.

Because tables appear at consistent relative locations in the pane, unaffected by scroll or zoom actions, programmers occasionally use them for on-chart debugging. The most common technique is to draw a *single-cell* table containing information from specific script executions.

This example contains a `printTable()` function that calls table.new() and table.cell() to create a single-cell table that displays dynamic text in a relative location on the main chart pane. The script uses a single call to this function to display the same chart information shown by the example script from the previous section:

//@version=6 indicator("Debugging with single-cell tables demo", "Chart info", true, behind_chart = false) //@function Draws a single-cell table to display the `info` text in the top-right corner of the chart. //@param info The string to display. //@param textColor Optional. The color of the displayed text. If `na`, the table uses `chart.fg_color`. // The default is `na`. //@param size Optional. The size of the table's text in typographic points. The default is 18. //@returns A single-cell table with dynamic text. printTable(string info, simple color textColor = na, simple int size = 18) => var color col = nz(textColor, chart.fg_color) var table result = table.new(position.top_right, 1, 1, na, force_overlay = true) table.cell(result, 0, 0, info, text_color = col, text_size = size) // Call `printTable()` on the latest available bar to display chart information in the top-right corner. if barstate.islast printTable(str.format("Chart info:\n\nSymbol: {0}, Type: {1}, Timeframe: {2}\nStandard chart: {3}, Replay active: {4}" + "\n\nO: {5,number,#.#####}, H: {6,number,#.#####}, L: {7,number,#.#####}, C: {8,number,#.#####}, V: {9}" + "\nTotalBars: {10}", ticker.standard(), syminfo.type, timeframe.period, chart.is_standard, str.contains(syminfo.tickerid, "replay"), open, high, low, close, str.tostring(volume, format.volume), bar_index + 1))

Note that:

- Every new table drawing *replaces* any existing one that has the same specified position. Therefore, scripts cannot call the `printTable()` function multiple times to place multiple drawings in a single location, unlike the `printLabel()` function from the previous section.
- This script calls `printTable()` only on the last historical bar and all realtime bars because updating tables on each historical bar is an unnecessary use of runtime resources. See the Reducing drawing

updates section of the Profiling and optimization page for more information.

Plots and chart colors

The built-in `plot*()` functions display results from a value's series in up to four locations: the chart pane, the script's status line, the Data Window, and the price scale. Programmers often use these output functions as a quick way to display the history of a script's numeric values, conditions, and colors. Two other functions, bgcolor() and barcolor(), color a chart pane's background and the main chart's bars or candles. Although not as versatile as other output functions, they offer a quick way to display conditions and colors on the chart.

All these functions, especially plot(), plotchar(), and plotshape(), can serve as helpful tools for debugging a script's calculations and logic. For instance, the outputs of a single plot() call can show the complete available history of a script's series on the chart and provide information for any bar in other locations.

Before using plots or chart colors for debugging, it is important to note the following limitations:

- Unlike Pine Logs or drawings, these outputs *cannot* display results for values that are accessible from local scopes only. Scripts must extract values from local scopes into the *global scope* to debug them with plots or chart colors.
- The only `plot*()` functions that can display *text* on the chart – plotchar() and plotshape() – require "const string" values. Therefore, they cannot display dynamic strings or calculated string conversions of other types.
- Similar to drawings, plots *do not* have built-in navigation features. Users must scroll across the chart to find plotted information for specific bars.
- The maximum *plot count* for any script is 64. Each call to these functions contributes a different number to the total, depending on its arguments. See the Plot limits section of the Limitations page to learn more.

Plotting numbers

One of the simplest methods to inspect global numeric series ("int" or "float" values) is to plot them using the plot(), plotchar(), or plotshape() function. The outputs on the chart pane provide a graphical view of the series' history. The other possible output locations (status line, price scale, and Data Window) show formatted numbers representing the values calculated on a specific bar.

Let's look at a simple debugging example. The following script calculates a custom oscillator whose value is the average of three separate oscillators. It displays the `oscillator` value in four output locations using a plot() call:

//@version=6 indicator("Plotting numbers demo") //@variable The length of each oscillator. int lengthInput = input.int(20, "Length", 2) //@variable The correlation between `close` and `bar_index` over `lengthInput` bars. float osc1 = ta.correlation(close, bar_index, lengthInput) //@variable The RSI of `close` over `lengthInput` bars, scaled to the range [-1, 1]. float osc2 = (ta.rsi(close, lengthInput) - 50) / 50 //@variable The percent rank of `close` compared to `lengthInput` past values, scaled to the range [-1, 1]. float osc3 = (ta.percentrank(close, lengthInput) - 50) / 50 //@variable The average of `osc1`, `osc2`, and `osc3`. float oscillator = math.avg(osc1, osc2, osc3) // Plot the `oscillator`. plot(oscillator, "Combined oscillator", color.purple, 3)

The above script's outputs allow inspection of the final `oscillator`, but not the three constituent oscillators that determine its value. Because the script calculates all three series in the global scope, we can inspect

them using additional plots. Here, we add three plot() calls to the script to display each oscillator, allowing us to verify the script's calculated values and understand how they affect the final result:

//@version=6 indicator("Plotting numbers demo") //@variable The length of each oscillator. int lengthInput = input.int(20, "Length", 2) //@variable The correlation between `close` and `bar_index` over `lengthInput` bars. float osc1 = ta.correlation(close, bar_index, lengthInput) //@variable The RSI of `close` over `lengthInput` bars, scaled to the range [-1, 1]. float osc2 = (ta.rsi(close, lengthInput) - 50) / 50 //@variable The percent rank of `close` compared to `lengthInput` past values, scaled to the range [-1, 1]. float osc3 = (ta.percentrank(close, lengthInput) - 50) / 50 //@variable The average of `osc1`, `osc2`, and `osc3`. float oscillator = math.avg(osc1, osc2, osc3) // Plot the `oscillator`. plot(oscillator, "Combined oscillator", color.purple, 3) // Plot the `osc1`, `osc2`, and `osc3` series for inspection. plot(osc1, "osc1", color.red, 2, plot.style_circles, join = true) plot(osc2, "osc2", color.maroon, 2, plot.style_circles, join = true) plot(osc3, "osc3", color.blue, 2, plot.style_circles, join = true)

Note that:

- The numbers in the script's status line and the Data Window represent the values plotted on the bar at the mouse pointer's location. When the pointer is *not* on the chart, these numbers represent the *latest* bar's data.
- The labels in the price scale show the latest *non-na* values available in the plotted series up to the last visible bar. If a plotted series does not have a non-na value at any point before that bar, the price scale does *not* show a label for it.

> TipWhen debugging numbers, it is crucial to consider the *decimal precision* (i.e., number of *fractional digits*) required to inspect them effectively. Programmers can set the precision for a script's plots using the `precision` parameter of the indicator(), strategy(), or `plot*()` functions. Alternatively, users can change the precision from the "Precision" field in the script's "Settings" menu or the chart's settings. Note that when a `plot*()` function includes a `precision` argument, it uses that value to determine the output's decimal precision, ignoring the script's global precision setting.

Plotting without affecting the scale

Debugging multiple numeric series by plotting them on the chart can make the results hard to read if the plots affect the price scale, especially if each plotted series has a significantly different value range. Programmers can specify a plot's *display locations* to avoid distorting the scale by passing a `display.*` constant or expression to the `display` parameter of the `plot*()` call.

Let's look at a simple example that calculates a few numeric series with different ranges. This script calculates a weighted moving average with custom weights and plots the result on the chart:

//@version=6 indicator("Plotting without affecting the scale demo", "Weighted average", true, precision = 5) //@variable The number of bars in the average. int lengthInput = input.int(20, "Length", 1) //@variable The weight applied to the price on each bar. float weight = math.pow(close - open, 2) //@variable The numerator of the average. float numerator = math.sum(weight * close, lengthInput) //@variable The denominator of the average. float denominator = math.sum(weight, lengthInput) //@variable The weighted average over `lengthInput` bars. float average = numerator / denominator // Plot the `average`. plot(average, "Weighted average", linewidth = 3)

Note that:

- This script includes `precision = 5` in the indicator() declaration statement, which specifies that it plots numbers with five fractional digits instead of using the chart's default precision setting.

Suppose we want to inspect *all* the values in the `average` calculation using plots. If we use `plot*()` functions with the default `display` argument (display.all), the plotted results appear in all possible locations, including the chart pane. Unlike the example script from the Plotting numbers section, this script's visuals become hard to read in the pane because each plot has a significantly different range:

//@version=6 indicator("Plotting without affecting the scale demo", "Weighted average", true, precision = 5) //@variable The number of bars in the average. int lengthInput = input.int(20, "Length", 1) //@variable The weight applied to the price on each bar. float weight = math.pow(close - open, 2) //@variable The numerator of the average. float numerator = math.sum(weight * close, lengthInput) //@variable The denominator of the average. float denominator = math.sum(weight, lengthInput) //@variable The weighted average over `lengthInput` bars. float average = numerator / denominator // Plot the `average`. plot(average, "Weighted average", linewidth = 3) // Create debug plots for the `weight`, `numerator`, and `denominator`. plot(weight, "weight", color.purple) plot(numerator, "numerator", color.teal) plot(denominator, "denominator", color.maroon)

We can change the `display` argument in each debug plot() call to view all the calculated values while preserving the chart's scale. Below, we set the argument to `display.all - display.pane`, meaning all the debug plots show information in all locations *except* the chart pane. Now, we can visualize how the calculated values affect each bar's `average` result without distorting the scale:

//@version=6 indicator("Plotting without affecting the scale demo", "Weighted average", true, precision = 5) //@variable The number of bars in the average. int lengthInput = input.int(20, "Length", 1) //@variable The weight applied to the price on each bar. float weight = math.pow(close - open, 2) //@variable The numerator of the average. float numerator = math.sum(weight * close, lengthInput) //@variable The denominator of the average. float denominator = math.sum(weight, lengthInput) //@variable The weighted average over `lengthInput` bars. float average = numerator / denominator // Plot the `average`. plot(average, "Weighted average", linewidth = 3) //@variable The display locations of all debug plots. debugLocations = display.all - display.pane // Create debug plots for the `weight`, `numerator`, and `denominator`. plot(weight, "weight", color.purple, display = debugLocations) plot(numerator, "numerator", color.teal, display = debugLocations) plot(denominator, "denominator", color.maroon, display = debugLocations)

Note that:

- The `display.*` constants support *addition* and *subtraction* operations for customized display settings. This script uses subtraction to remove display.pane from the output locations allowed by display.all. Operations that remove valid display constants more than once do not cause errors. For instance, this script produces the *same* outputs if it subtracts display.pane once, twice, or more times in the `debugLocations` expression.

Plotting and coloring conditions

Programmers can inspect a script's conditions ("bool" values) with the `plot*()`, bgcolor(), and barcolor() functions in several ways, including:

- Using the “bool” condition as the `series` argument in a plotshape() or plotchar() call. The call creates a shape/character with specified text on the chart when the condition is `true`, and it shows a *numeric text* representation of the condition in the status line and Data Window (`1` for `true` and `0` for `false`).
- Creating a logical expression that returns different “int” or “float” values for the condition’s `true` and `false` states, then using the result as the `series` argument in a `plot*()` call. When using plotchar() or plotshape(), note that these functions show visuals on the chart only when the `series` value is not na or 0.
- Creating a logical expression that returns different “color” values based on the condition’s `true` or `false` state, then using the result to color the chart with bgcolor() or barcolor(), or to color a plot or fill.

The following example uses the above methods to debug a simple condition. The script calculates an RSI with an input length and defines a `crossBelow` condition that is `true` when the RSI crosses 30. It uses plotshape(), plotchar(), and bgcolor() calls to visualize the `crossBelow` condition in different ways:

//@version=6 indicator("Plotting and coloring conditions demo") //@variable The length of the RSI. int lengthInput = input.int(14, "Length", 1) //@variable The calculated RSI value. float rsi = ta.rsi(close, lengthInput) //@variable Is `true` when the `rsi` crosses below 30, `false` otherwise. bool crossBelow = ta.crossunder(rsi, 30.0) // Plot the `rsi`. plot(rsi, "RSI", color.rgb(136, 76, 146), linewidth = 3) // Plot a circle near the top of the pane when `crossBelow` is `true`. // The status line and Data Window show 1 when the condition is `true` and 0 when it is `false`. plotshape(crossBelow, "plotshape debug", shape.circle, location.top, color.red, size = size.small) // Plot the `⤱` character at the `rsi` value when `crossBelow` is `true`. // The status line and Data Window show the `rsi` value when the condition is `true` and `na` when it is `false`. plotchar(crossBelow ? rsi : na, "plotchar debug", "⤱", location.absolute, color.maroon, size = size.normal) // Highlight the background when `crossBelow` is `true`. Does not add information to the status line or Data Window. bgcolor(crossBelow ? color.new(color.red, 60) : na, title = "bgcolor debug")

Note that:

- The `plot*()` functions that display text or shapes on the chart – plotshape(), plotchar(), and plotarrow() – *do not* display data in the price scale.
- The plotshape() call uses `crossUnder` as its `series` argument. The chart pane shows a shape at the top when the condition occurs. The status line and Data Window show 1 when the `series` is `true` and 0 when it is `false`.
- The plotchar() call plots the result of a ternary expression that returns the `rsi` when `crossUnder` is `true` and na otherwise. It shows the character U+2930 at the `rsi` location when the expression does not evaluate to na. Because the `series` argument is a “float” value, the number in the status line and Data Window represents that value directly.
- The bgcolor() call highlights the chart’s background when `crossUnder` is `true`, but it does not display information in the status line or Data Window.

The plotshape() and plotchar() functions have a `text` parameter that adds “const string” text to the plotted shapes/characters. When debugging multiple global conditions, it is often helpful to call these functions with `text` arguments to label each condition for simple on-chart inspection. The arguments can contain the newline character (`\n` escape sequence), allowing scripts to plot multiple shapes in identical locations with non-overlapping text.

Let’s explore a debugging example using this approach. The script below calculates an RSI and its median over `lengthInput` bars. Then, it creates five singular conditions and uses them to form a *compound*

condition. The script plots the `rsi` and `median` values with the `plot()` function, and it colors the background with bgcolor() when the `compoundCondition` is `true`:

```
//@version=6 indicator("Plotting and coloring compound conditions demo") //@variable The length of the RSI and median RSI calculations. int lengthInput = input.int(14, "Length", 2) //@variable The RSI of `close` with a smoothing factor defined by `lengthInput`. float rsi = ta.rsi(close, lengthInput) //@variable The median of the `rsi` over `lengthInput` bars. float median = ta.median(rsi, lengthInput) //@variable Condition #1: Is `true` when the 1-bar `rsi` change switches from 1 to -1. bool changeNegative = ta.change(math.sign(ta.change(rsi))) == -2 //@variable Condition #2: Is `true` when the previous bar's `rsi` is greater than 70. bool prevAbove70 = rsi[1] > 70.0 //@variable Condition #3: Is `true` when the current `close` is lower than the previous bar's `open`. bool closeBelow = close < open[1] //@variable Condition #4: Is `true` when the `rsi` is between 60 and 70. bool betweenLevels = bool(math.max(70.0 - rsi, 0.0) * math.max(rsi - 60.0, 0.0)) //@variable Condition #5: Is `true` when the `rsi` is above the `median`. bool aboveMedian = rsi > median //@variable Is `true` when the first condition occurs alongside conditions 2 and 3 or 4 and 5. bool compundCondition = changeNegative and ((prevAbove70 and closeBelow) or (betweenLevels and aboveMedian)) //Plot the `rsi` and the `median`. plot(rsi, "RSI", color.teal, 3) plot(median, "RSI Median", color.gray, 2) // Highlight the background red when the `compundCondition` occurs. bgcolor(compundCondition ? color.new(color.red, 60) : na, title = "compundCondition")
```

To verify that the script's logic works as intended, we can inspect each of the conditions that affect the final `compoundCondition` value. Below, we added five plotchar() calls to display information for these conditions, each with the same `location` argument. To label the conditions on the chart, each plotchar() call uses a string containing newline characters (`\n`) and a digit from 1 to 5 as the `text` argument. With these outputs, we can see which sets of conditions trigger each `compoundCondition` occurrence:

```
//@version=6 indicator("Plotting and coloring compound conditions demo") //@variable The length of the RSI and median RSI calculations. int lengthInput = input.int(14, "Length", 2) //@variable The RSI over `lengthInput` bars. float rsi = ta.rsi(close, lengthInput) //@variable The median of the `rsi` over `lengthInput` bars. float median = ta.median(rsi, lengthInput) //@variable Condition #1: Is `true` when the 1-bar `rsi` change switches from 1 to -1. bool changeNegative = ta.change(math.sign(ta.change(rsi))) == -2 //@variable Condition #2: Is `true` when the previous bar's `rsi` is greater than 70. bool prevAbove70 = rsi[1] > 70.0 //@variable Condition #3: Is `true` when the current `close` is lower than the previous bar's `open`. bool closeBelow = close < open[1] //@variable Condition #4: Is `true` when the `rsi` is between 60 and 70. bool betweenLevels = bool(math.max(70.0 - rsi, 0.0) * math.max(rsi - 60.0, 0.0)) //@variable Condition #5: Is `true` when the `rsi` is above the `median`. bool aboveMedian = rsi > median //@variable Is `true` when the first condition occurs alongside conditions 2 and 3 or 4 and 5. bool compundCondition = changeNegative and ((prevAbove70 and closeBelow) or (betweenLevels and aboveMedian)) //Plot the `rsi` and the `median`. plot(rsi, "RSI", color.teal, 3) plot(median, "RSI Median", color.gray, 2) // Highlight the background red when the `compundCondition` occurs. bgcolor(compundCondition ? color.new(color.red, 60) : na, title = "compundCondition") // Use `plotshape()` to show `compundCondition` values in the status line and Data Window. plotshape( compundCondition, "compundCondition (1 and (2 and 3) or (4 and 5))", color = chart.fg_color, display = display.all - display.pane ) // Plot characters on the chart and numbers in the status line and Data Window when conditions 1-5 occur. plotchar(changeNegative, "changeNegative (1)", "", location.top, text = "1", textcolor = chart.fg_color) plotchar(prevAbove70, "prevAbove70 (2)", "", location.top, text = "\n2", textcolor = chart.fg_color) plotchar(closeBelow, "closeBelow (3)", "", location.top, text = "\n\n3", textcolor = chart.fg_color) plotchar(betweenLevels, "betweenLevels (4)", "", location.top, text = "\n\n\n4", textcolor = chart.fg_color) plotchar(aboveMedian, "aboveMedian (5)", "", location.top, text = "\n\n\n\n5", textcolor = chart.fg_color)
```

Note that:

- The `char` argument of each plotchar() call is an empty string, meaning the function displays its `text` value without a character above it.
- Because each plotchar() call outputs results at the same relative location (`location.top`), we included different numbers of leading `\n` sequences in the `text` arguments to move the displayed numerals down and ensure they do not overlap.
- The `title` argument of each plotchar() call contains the condition number to distinguish it in the Data Window.
- The plotshape() call's title describes the compound condition's structure in the Data Window.

To learn more about the plotshape() and plotchar() functions and how their outputs differ from labels, refer to the Text and shapes page.

Tips and techniques

The following sections explain several additional tips and helpful techniques for effective Pine Script debugging.

Decomposing expressions

One of the best practices for efficient debugging is to *split expressions*, especially those with multiple calculations or logical operations, into *smaller parts* assigned to separate variables. Decomposing expressions enables programmers to inspect each critical part individually, making it easier to verify calculations or logic and isolate potential issues in the code. Additionally, complex code broken down into smaller parts is typically simpler to read, maintain, and profile.

The following script calculates a custom oscillator representing the smoothed median change in the differences between the close price and two EMAs over different lengths. The script performs all the calculations in a single expression assigned to the `osc` variable. Then, it creates a compound condition in another expression assigned to the `upSignal` variable and uses that variable to trigger order placement commands. The script plots the `osc` series as columns with different colors based on the `upSignal` value:

//@version=6 strategy("Decomposing expressions demo") //@variable The length used for the first part of the oscillator. int length1Input = input.int(20) //@variable The length used for the second part of the oscillator. int length2Input = input.int(40) //@variable Oscillator smoothing length. int smoothingInput = input.int(10) //@variable The maximum of `length1Input` and `length2Input`. int maxLength = math.max(length1Input, length2Input) //@variable The smoothed median change in the differences between `close` and two EMAs over different lengths. float osc = ta.ema(math.avg(ta.change(close - ta.ema(close, length1Input), length1Input), ta.change(close - ta.ema(close, length2Input), length2Input)), smoothingInput) //@variable `true` if `osc` is positive, above the last two-bar average, and below twice the stdev for `maxLength` bars. bool upSignal = osc < 2 * ta.stdev(osc, maxLength) and osc > 0 and math.avg(osc[1], osc[2]) < osc // Plot the `osc` as columns colored based on the `upSignal`. plot(osc, "Custom oscillator", upSignal ? color.aqua : color.gray, style = plot.style_columns) // Place a "Buy" market order when `upSignal` is `true`, and a closing market order when it is `false`. if upSignal strategy.entry("Buy", strategy.long) else strategy.close("Buy")

Because the `osc` and `upSignal` values depend on multiple calculations and conditions, inspecting only the final values does not provide complete information about the script's behaviors. To verify the script's

workings, we can decompose the expressions assigned to `osc` and `upCondition` into smaller parts and inspect them individually.

The script version below declares several extra variables to hold *different parts* of the original `osc` and `upCondition` expressions. With this expanded structure, we can inspect each part of the calculations and logic step-by-step using various outputs. In this script, we included a single log.info() call at the end that displays formatted text containing each variable's information in the Pine Logs pane:

```
//@version=6 strategy("Decomposing expressions demo") //@variable The length used for the first part of the oscillator. int length1Input = input.int(20) //@variable The length used for the second part of the oscillator. int length2Input = input.int(40) //@variable Oscillator smoothing length. int smoothingInput = input.int(10) //@variable The maximum of `length1Input` and `length2Input`. int maxLength = math.max(length1Input, length2Input) //#region Split the `osc` calculations into smaller parts: // 1. Calculate the EMAS over `length1Input` and `length2Input` bars. float ema1 = ta.ema(close, length1Input), float ema2 = ta.ema(close, length2Input) // 2. Calculate the differences between `close` and `ema1` and `ema2`. float diff1 = close - ema1, float diff2 = close - ema2 // 3. Calculate the changes in `diff1` and `diff2` over `length1Input` and `length2Input` bars. float change1 = ta.change(diff1, length1Input), float change2 = ta.change(diff2, length2Input) // 4. Calculate the median of `change1` and `change2`. float medChange = math.avg(change1, change2) //#endregion //@variable The smoothed median change in the differences between `close` and two EMAs over different lengths. float osc = ta.ema(medChange, smoothingInput) //#region Split the `upSignal` calculations and logic into smaller parts: // 1. Assign the calculations in the expression to separate variables. float oscDev = 2 * ta.stdev(osc, maxLength), float pastAvg = math.avg(osc[1], osc[2]) // 2. Assign each singular condition to a separate variable. bool cond1 = osc < oscDev, bool cond2 = osc > 0, bool cond3 = pastAvg < osc //#endregion //@variable Is `true` if `osc` is positive, above the past two-bar average, and below twice its stdev over `maxLength` bars. bool upSignal = cond1 and cond2 and cond3 // Plot the `osc` as columns colored based on the `upSignal`. plot(osc, "Custom oscillator", upSignal ? color.aqua : color.gray, style = plot.style_columns) // Place a "Buy" market order when `upSignal` is `true`, and a closing market order when it is `false`. if upSignal strategy.entry("Buy", strategy.long) else strategy.close("Buy") // Call `log.info()` to display a formatted message containing debug information in the Pine Logs pane. if barstate.isconfirmed log.info( "\nema1: {0,number,0.00000}, diff1: {1,number,0.00000}, change1: {2,number,0.00000}" + "\nema2: {3,number,0.00000}, diff2: {4,number,0.00000}, change2: {5,number,0.00000}" + "\nmedChange: {6,number,0.00000}\n\nosc: {7,number,0.00000}\n----\noscDev: {8,number,0.00000}" + "\npastAvg: {9,number,0.00000}\ncond1: {10}, cond2: {11}, cond3: {12}\n\nupSignal: {13}", ema1, diff1, change1, ema2, diff2, change2, medChange, osc, oscDev, pastAvg, cond1, cond2, cond3, upSignal )
```

Note that:

- This script declares some extra variables on the *same line*, separated by commas, to reduce the number of lines added to the code.
- The script calls log.info() only when barstate.isconfirmed is `true`, preventing unnecessary logs on the ticks of unconfirmed bars.
- All the placeholders with the `number` modifier in the log.info() call's formatting string include the `0.00000` pattern, which forces the formatted numbers to always show five fractional digits. Refer to the Formatting strings section of the Strings page for more information.
- The Pine Logs pane displays up to 10,000 historical logs. To view earlier logs, add another condition to the if structure that limits the log.info() call to specific bars. See the Custom code filters section above for an example that restricts `log.*()` calls using time inputs.

Extracting data from local scopes

The *scope* of an identifier (e.g., a variable) refers to the part of a script where it is defined and *accessible* during the script's execution.

All identifiers declared outside user-defined functions, methods, loops, conditional structures, or user-defined type and enum type declarations belong to the *global scope*. Identifiers in the global scope are accessible to most *inner* (local) scopes after declaration. Every Pine script has exactly **one** global scope.

All user-defined functions, methods, loops, and conditional structures in a script create unique, separate *local scopes*. All identifiers within a local scope belong exclusively to that scope, meaning their values or references are *inaccessible* to any *outer* or *containing* scope.

A common practice when debugging variables declared in a local scope is to extract their data to an outer scope or the global scope, making it usable in debugging outputs with different scope requirements.

The following sections explain techniques for extracting data from local scopes using return expressions and reference types. We demonstrate these techniques on the following script, which contains a `customMA()` function that calculates a custom adaptive moving average of a `source` series based on the distance from its current value to its 25th and 75th percentiles over `length` bars. The script contains a local function scope, and a nested block scope from the if structure that sets the `outerRange` value:

//@version=6 indicator("Extracting from local scopes initial demo", overlay = true) //@variable The number of bars in the `customMA()` calculation. int lengthInput = input.int(50, "Length", 2) //@function Calculates a moving average that changes only when `source` is outside the first and third quartiles. //@param source The series of values to process. //@param length The number of bars in the quartile calculation. //@returns The adaptive moving average value. customMA(float source, int length) => //@variable The custom moving average. var float result = na // Calculate the 25th and 75th `source` percentiles (first and third quartiles) over `length` bars. float q1 = ta.percentile_linear_interpolation(source, length, 25) float q3 = ta.percentile_linear_interpolation(source, length, 75) //@variable The distance from `source` to its interquartile range. float outerRange = 0.0 // Calculate the `outerRange` value when `source` is not `na`. if not na(source) float upperRange = source - q3 float lowerRange = q1 - source outerRange := math.max(upperRange, lowerRange, 0.0) //@variable The total range of `source` values over `length` bars. float totalRange = ta.range(source, length) //@variable Half the ratio of the `outerRange` to the `totalRange`. float alpha = 0.5 * outerRange / totalRange // Mix the `source` with the `result` based on the `alpha` value. result := (1.0 - alpha) * nz(result, source) + alpha * source // Return the `result`. result //@variable The `customMA()` of `close` over `lengthInput` bars. float maValue = customMA(close, lengthInput) // Plot the `maValue`. plot(maValue, "Custom MA", color.blue, 3)

Extraction using return expressions

In Pine Script, any user-defined function or method call, loop, or conditional structure *returns* the result of the *final* expression or nested structure within its local scope. Scripts can use these structures' returned results, excluding void, by assigning them to variables declared in the *outer* scope.

When debugging functions and conditional structures that contain multiple local variables, a common technique to extract data from their scopes is to return tuples containing the data that requires inspection.

Here, we've modified the previous example script's `customMA()` function to return a tuple containing values calculated from the local scopes. With this change, the script can call the function with a *tuple declaration* to

make all the data available to the global scope. The script plots the `q1Dbg` and `q3Dbg` values, highlights the background when `alphaDbg` is 0, and uses log.info() to display a formatted string containing all the extracted data in the Pine Logs pane:

```
//@version=6 indicator("Extraction using return expressions demo", overlay = true) //@variable The number of bars in the `customMA()` calculation. int lengthInput = input.int(50, "Length", 2) //@function Calculates a moving average that changes only when `source` is outside the first and third quartiles. //@param source The series of values to process. //@param length The number of bars in the quartile calculation. //@returns The adaptive moving average value. customMA(float source, int length) => //@variable The custom moving average. var float result = na // Calculate the 25th and 75th `source` percentiles (first and third quartiles) over `length` bars. float q1 = ta.percentile_linear_interpolation(source, length, 25) float q3 = ta.percentile_linear_interpolation(source, length, 75) //@variable The distance from `source` to its interquartile range. float outerRange = 0.0 // To extract `upperRange` and `lowerRange` values, we need to make them accessible to the function's main scope. // Here, we added a tuple at the end of the `if` statement's local block, then declared a tuple in the function's // scope to hold the returned values. [upper, lower] = if not na(source) float upperRange = source - q3 float lowerRange = q1 - source outerRange := math.max(upperRange, lowerRange, 0.0) [upperRange, lowerRange] //@variable The total range of `source` values over `length` bars. float totalRange = ta.range(source, length) //@variable Half the ratio of the `outerRange` to the `totalRange`. float alpha = 0.5 * outerRange / totalRange // Mix the `source` with the `result` based on the `alpha` value. result := (1.0 - alpha) * nz(result, source) + alpha * source // Return a tuple containing the `result` and other local variables. [result, q1, q3, upper, lower, outerRange, totalRange, alpha] //@variable The `customMA()` of `close` over `lengthInput` bars. [maValue, q1Dbg, q3Dbg, upperDbg, lowerDbg, outerRangeDbg, totalRangeDbg, alphaDbg] = customMA(close, lengthInput) // Plot the `maValue`. plot(maValue, "Custom MA", color.blue, 3) // When the bar is confirmed, log an "info" message containing formatted debug information for each variable. if barstate.isconfirmed log.info( "maValue: {0,number,#.#####}\nq1Dbg: {1,number,#.#####}, q3Dbg: {2,number,#.#####}" + "\nupperDbg: {3,number,#.#####}, lowerDbg: {4,number,#.#####}" + "\nouterRangeDbg: {5,number,#.#####}, totalRangeDbg: {6,number,#.#####}\nalphaDbg: {7,number,#.#####}", maValue, q1Dbg, q3Dbg, upperDbg, lowerDbg, outerRangeDbg, totalRangeDbg, alphaDbg ) // Display the extracted `q1` and `q3` data in all plot locations. plot(q1Dbg, "q1Dbg", color.new(color.maroon, 50)) plot(q3Dbg, "q3Dbg", color.new(color.teal, 50)) // Highlight the chart's background when the extracted `alpha` value is 0. bgcolor(alphaDbg == 0.0 ? color.new(color.orange, 90) : na, title = "`alpha == 0.0` highlight")
```

Note that:

- We added a tuple at the *end* of the if structure's block to *return* the `upperRange` and `lowerRange` values from its local scope. The function assigns the result to a two-variable tuple in its main scope, enabling it to include the if structure's local values in the return expression.

Extraction using reference types

Reference types, including all special types and user-defined types (UDTs), serve as structures for creating *objects*. Each object has an associated *reference* that distinguishes it and provides access to its data. Unlike fundamental types, variables of reference types **do not** store values directly. Instead, they hold the references for specific objects in memory.

An advanced, flexible way to extract data from local scopes is to initialize reference-type objects – such as instances of collections or UDTs – in the global scope and store local variable data in their elements or fields.

This technique is especially useful for extracting data from user-defined functions and methods. Although functions can access global variables, they cannot *reassign* them like global conditional structures and loops can. Consequently, they *cannot update* the data held by global variables of fundamental types. However, scripts do not modify reference types by reassigning their variables; they access objects via their references and use *methods* or *field reassignments* to update their data. As such, scripts *can* update global collections or UDT instances from inside function scopes.

For example, this modified version of our initial script declares a global `debugData` variable that holds the reference of a map with “string” keys and “float” values. Each map.put() call inside the `customMA()` scope modifies the map by adding a *key-value pair* containing a local variable’s *name* and *value*. After calling `customMA()`, the script uses map.get() calls on `debugData` to retrieve the stored information for its debugging outputs:

```
//@version=6 indicator("Extraction using reference types demo", overlay = true) //@variable The number of bars in the `customMA()` calculation. int lengthInput = input.int(50, "Length", 2) //@variable A global map of "string" keys and "float" values to store debug information from local scopes. var map<string, float> debugData = map.new<string, float>() //@function Calculates a moving average that changes only when `source` is outside the first and third quartiles. //@param source The series of values to process. //@param length The number of bars in the quartile calculation. //@returns The adaptive moving average value. customMA(float source, int length) => //@variable The custom moving average. var float result = na // Calculate the 25th and 75th percentiles (first and third quartiles). float q1 = ta.percentile_linear_interpolation(source, length, 25), debugData.put("q1", q1) float q3 = ta.percentile_linear_interpolation(source, length, 75), debugData.put("q3", q3) //@variable The distance from `source` to its interquartile range. float outerRange = 0.0 // Calculate the `outerRange` value when `source` is not `na`. if not na(source) float upperRange = source - q3, debugData.put("upperRange", upperRange) float lowerRange = q1 - source, debugData.put("lowerRange", lowerRange) outerRange := math.max(upperRange, lowerRange, 0.0), debugData.put("outerRange", outerRange) //@variable The total range of `source` values over `length` bars. float totalRange = ta.range(source, length), debugData.put("totalRange", totalRange) //@variable Half the ratio of the `outerRange` to the `totalRange`. float alpha = 0.5 * outerRange / totalRange, debugData.put("alpha", alpha) // Mix the `source` with the `result` based on the `alpha` value. result := (1.0 - alpha) * nz(result, source) + alpha * source // Return the `result`. result //@variable The `customMA()` of `close` over `lengthInput` bars. float maValue = customMA(close, lengthInput) // Plot the `maValue`. plot(maValue, "Custom MA", color.blue, 3) // When the bar is confirmed, log an "info" message containing formatted debug information for each value. if barstate.isconfirmed log.info( "maValue: {0,number,#.#####}\nq1: {1,number,#.#####}, q3: {2,number,#.#####}" + "\nupperRange: {3,number,#.#####}, lowerRange: {4,number,#.#####}" + "\nouterRange: {5,number,#.#####}, totalRange: {6,number,#.#####}\nalpha: {7,number,#.#####}", maValue, debugData.get("q1"), debugData.get("q3"), debugData.get("upperRange"), debugData.get("lowerRange"), debugData.get("outerRange"), debugData.get("totalRange"), debugData.get("alpha") ) // Display the extracted `q1` and `q3` data in all plot locations. plot(debugData.get("q1"), "q1", color.new(color.maroon, 50)) plot(debugData.get("q3"), "q3", color.new(color.teal, 50)) // Highlight the chart's background when the extracted `alpha` value is 0. bgcolor(debugData.get("alpha") == 0.0 ? color.new(color.orange, 90) : na, title = "`alpha == 0.0` highlight")
```

Note that:

- The script declares `debugData` with the var keyword, meaning the assigned map reference *persists* across script executions.

- A function executes its local code only when the script calls it. Therefore, the `debugData` map contains new information only after the `customMA()` call.
- Because the map.put() calls in `customMA()` assign keys to the map that do not change across executions, each `customMA()` call *replaces* the `debugData` map's existing data. Programmers can *preserve* data from specific executions with this technique by making a *copy* of the global collection after the function call.

Inspecting loops

Loops are structures that execute a local code block repeatedly based on a *counter* (for), the *contents* of a collection (for…in), or a *condition* (while). These structures allow scripts to perform repetitive tasks without redundant lines of code.

Because loops can execute their local code multiple times, programmers must use techniques to track local variables *across iterations* to debug them effectively. As with other structures, there are many ways to inspect loops. These sections cover two helpful techniques: collecting loop information and tracing loop executions.

Collecting loop information

One of the most effective loop inspection techniques is to use collections or strings to gather information from the local scope on each iteration requiring inspection, then use the information in output functions *after* the loop terminates.

Let's look at a simple loop debugging example using this technique. The following script calculates the average rate of change in the close price over lengths from 1 to `lookbackInput` bars inside a for loop. It declares an `aroc` variable in the global scope, sums the rates of change inside the loop, and then divides the sum by the `lookbackInput` to calculate the average:

//@version=6 indicator("Collecting loop information demo", "Average ROC") //@variable The number of past bars in the calculation. int lookbackInput = input.int(20, "Lookback", 1) //@variable The average ROC of `close` prices over each length from 1 to `lookbackInput` bars. float aroc = 0.0 // Calculation loop. for length = 1 to lookbackInput //@variable The `close` value `length` bars ago. float pastClose = close[length] //@variable The `close` rate of change over `length` bars. float roc = (close - pastClose) / pastClose // Add the `roc` to the `aroc` value. aroc += roc // Divide `aroc` by the `lookbackInput` to get the average. aroc /= lookbackInput // Plot the `aroc` series. plot(aroc, "aroc", color.blue, 3)

To debug the script's loop and ensure it works as intended, we can collect data from the local scope on each iteration and pass the result to the available output functions after the loop ends. In the script version below, we demonstrate two extraction methods. The first declares a global `logText` variable and concatenates formatted strings containing each loop iteration's `length` and `roc` values. The second declares a global `rocArray` variable and pushes each iteration's `roc` value into the referenced array.

After terminating the loop, the script calls log.info() to display the `logText` in the Pine Logs pane if the bar is confirmed. It then displays a "string" representation of the `rocArray` inside label tooltips. Lastly, it shows the array's *first* and *last* element values in all possible plot locations with the plot() function:

//@version=6 indicator("Collecting loop information demo", "Average ROC", max_labels_count = 500) //@variable The number of bars in the calculation. int lookbackInput = input.int(20, "Lookback", 1) //@variable An array containing the `roc` value from each loop iteration. array<float> debugValues =

array.new<float>() //@variable A string containing information about the `roc` value on each iteration. string logText = "" //@variable The average ROC of `close` over lags from 1 to `lookbackInput` bars. float aroc = 0.0 // Calculation loop. for length = 1 to lookbackInput //@variable The `close` value `length` bars ago. float pastClose = close[length] //@variable The `close` rate of change over `length` bars. float roc = (close - pastClose) / pastClose // Add the `roc` to `aroc`. aroc += roc // Concatenate a new "string" representation with the `debugText`. logText += "\nlength: " + str.tostring(length) + ", roc: " + str.tostring(roc) // Push the `roc` value into the `debugValues` array. array.push(debugValues, roc) // Divide `aroc` by the `lookbackInput`. aroc /= lookbackInput // Plot the `aroc`. plot(aroc, "aroc", color.blue, 3) // Log the `logText` in the Pine Logs pane when the bar is confirmed. if barstate.isconfirmed log.info(logText) // Draw a label with a tooltip containing a "string" representation of the `debugValues` array. label.new(bar_index, aroc, color = color.new(color.blue, 70), tooltip = str.tostring(debugValues)) // Plot the `roc` values from the first and last iteration. plot(array.first(debugValues), "First iteration roc", color.new(color.teal, 50), 2) plot(array.last(debugValues), "Last iteration roc", color.new(color.maroon, 50), 2)

Note that:

- Scripts can generate Pine Logs and drawings directly from within a loop's local scope. However, because loops usually execute their local code *more than once*, calling `log.*()` or label.new() functions inside the scope can result in numerous logs or labels per bar. Logging on each iteration helps trace execution patterns, but it also limits the number of *historical* bars with available debug data. See the next section, Tracing loop executions, for an example.
- Strings can contain up to *40,960* characters, and large strings or repeated concatenation can impact a script's performance. Therefore, extracting loop information with string concatenation is suitable for relatively *small* loops or inspecting specific variables. To extract large amounts of data from loops, use collections instead.

Tracing loop executions

An alternative way to inspect a loop, without collecting information for use in the *outer scope*, is to add `log.*()` calls directly to the loop's *local block*. Each iteration that activates the call results in a new message in the Pine Logs pane, allowing programmers to trace the loop's execution pattern in detail.

This simple script calculates a pseudorandom sample from a binomial distribution using a for loop. The plotted `sample` series represents the number of math.random() calls across `trialsInput` iterations that return a value not exceeding the `probabilityInput` value. On each iteration where `success` is `false`, the loop *skips* the rest of its block and moves to the next iteration. On other iterations, it increments the `sample` value by one:

//@version=6 indicator("Tracing loop executions demo", "Binomial sample") //@variable The probability that each random trial succeeds. float probabilityInput = input.float(0.5, "Success probability", 0.0, 1.0) //@variable The number of random trials to test. float trialsInput = input.int(10, "Trials", 1) //@variable Random sample from a binomial distribution, i.e., the number of successes from `trialsInput` random trials. int sample = 0 // Execute `trialsInput` loop iterations to calculate the `sample`. for trial = 1 to trialsInput //@variable A pseudorandom value between 0 and 1. float randValue = math.random() //@variable `true` if the `randValue` is less than or equal to the `probabilityInput`, `false` otherwise. bool success = randValue <= probabilityInput // Skip the rest of the iteration if `success` is `false`. if not success continue // Otherwise, add 1 to the `sample`. sample += 1 // Plot the `sample` as teal columns. plot(sample, "Binomial sample", color.teal, 1, plot.style_columns)

Below, we added `log.*()` function calls to generate Pine Logs at specific points in the loop's local block across iterations. Each loop iteration creates two new logs. The first log shows formatted text containing the local `trial`, `randValue`, and `success` variables' values. The second log depends on the if statement. When the statement's local code executes, the log is a `"CONTINUE"` message with the "warning" level. Otherwise, the second log is an "info" message containing the current iteration's `sample` value:

```
//@version=6 indicator("Tracing loop executions demo") //@variable The probability that each random trial succeeds. float probabilityInput = input.float(0.5, "Success probability", 0.0, 1.0) //@variable The number of random trials to test. float trialsInput = input.int(10, "Trials", 1) //@variable Random sample from a binomial distribution, i.e., the number of successes from `trialsInput` random trials. int sample = 0 // Log a message to mark the point before the start of the loop. log.warning("---------- LOOP START (bar {0,number,#})", bar_index) // Execute `trialsInput` loop iterations to calculate the `sample`. for trial = 1 to trialsInput //@variable A pseudorandom value between 0 and 1. float randValue = math.random() //@variable `true` if the `randValue` is less than or equal to the `probabilityInput`, `false` otherwise. bool success = randValue <= probabilityInput // Log a message containing the `trial`, `randValue`, and `success` information. log.info("trial: {0}, randValue: {1,number,#.########}, success: {2}", trial, randValue, success) // Skip the rest of the iteration if `success` is `false`. if not success // Log a message before the `continue` statement. log.warning("CONTINUE") continue // Otherwise, add 1 to the `sample`. sample += 1 // Log a message showing the iteration's `sample` value. log.info("sample: {0}", sample) // Log a message to mark the point after the loop ends. log.warning("---------- LOOP END\n\n") // Plot the `sample` as teal columns. plot(sample, "Binomial sample", color.teal, 1, plot.style_columns)
```

Note that:

- The script includes log.warning() calls before and after the loop to mark its start and end in the Pine Logs pane. The message marking the start of the loop also displays the current bar_index value.
- The Pine Logs pane shows only the most recent 10,000 logs created on historical bars. Because this script creates multiple logs per bar, the earliest message in the pane is from less than 10,000 bars back. Programmers can use conditional logic that limits `log.*()` calls in order to inspect a loop's execution flow on earlier bars with this technique. See the Custom code filters section to learn more.

Debugging collections

Collections are data structures that store values or references as *elements*, which scripts access using indices or keys, depending on the type. These structures can contain a lot of information, as the maximum number of elements across all instances of each collection type is 100,000.

Programmers can inspect a collection's data using various techniques, depending on the types they contain and their sizes. The most common approaches include:

- Creating a "string" representation of the collection with str.tostring() and displaying the result using Pine Logs or other text outputs.
- Retrieving specific elements from the collection, then creating formatted strings for logging, or using the element values or references in other output processes.

Displaying collection strings

The simplest way to inspect the data of arrays and matrices of "int", "float", "bool", and "string" types is to generate "string" representations with the str.tostring() function, then display the results using Pine Logs or other "string" outputs.

The following script calls request.security_lower_tf() to retrieve a “float” array containing close prices for each lower-timeframe bar within the current chart bar, which it uses to calculate an average intrabar price. Then, it calculates the ratio of the difference between the bar’s price and the intrabar average to the bar’s total range. The script plots the resulting ratio and its EMA in a separate pane:

//@version=6 indicator("Displaying collection strings demo") //@variable The length of the EMA. int lengthInput = input.int(20, "EMA length", 1) //@variable An array of `close` prices requested for the chart's symbol at the 1-minute timeframe. array<float> intrabarPrices = request.security_lower_tf("", "1", close) //@variable The average `close` price of the intrabars within the current chart bar. float avgPrice = intrabarPrices.avg() //@variable The bar's total range. float barRange = high - low //@variable The difference between `close` and `avgPrice`, normalized by the `barRange`. float ratio = (close - avgPrice) / barRange //@variable The EMA of the `ratio`. float smoothed = ta.ema(ratio, lengthInput) // Plot the `ratio` series as conditionally-colored columns. plot(ratio, "", ratio > 0 ? color.teal : color.maroon, 1, plot.style_columns) // Display the `smoothed` series as a translucent orange area plot. plot(smoothed, "", color.new(color.orange, 40), 1, plot.style_area)

To verify the ratio’s calculations, we can inspect the data stored in the `intrabarPrices` array by converting it to a “string” value and displaying the result for each bar.

The script version below declares a `debugText` variable that holds a formatted string representing the `intrabarPrices` array, the array’s size, and the `avgPrice` value. The script calls the `log.*()` functions to display the `debugText` value for each bar in the Pine Logs pane:

//@version=6 indicator("Displaying collection strings demo") //@variable The length of the EMA. int lengthInput = input.int(20, "EMA length", 1) //@variable An array of `close` prices requested for the chart's symbol at the 1-minute timeframe. array<float> intrabarPrices = request.security_lower_tf("", "1", close) //@variable The average `close` price of the intrabars within the current chart bar. float avgPrice = intrabarPrices.avg() //@variable The bar's total range. float barRange = high - low //@variable The difference between `close` and `avgPrice`, normalized by the `barRange`. float ratio = (close - avgPrice) / barRange //@variable The EMA of the `ratio`. float smoothed = ta.ema(ratio, lengthInput) // Plot the `ratio` series as conditionally-colored columns. plot(ratio, "", ratio > 0 ? color.teal : color.maroon, 1, plot.style_columns) // Display the `smoothed` series as a translucent orange area plot. plot(smoothed, "", color.new(color.orange, 40), 1, plot.style_area) //@variable A "string" representation of `intrabarPrices`, `intrabarPrices.size()`, and the `avgPrice`. string debugText = str.format("\nintrabarPrices: {0}\nsize: {1}\navgPrice: {2,number,#.#####}", str.tostring(intrabarPrices), intrabarPrices.size(), avgPrice) // Log the `debugText` with the "info" or "warning" level, depending on whether the bar is confirmed. switch barstate.isconfirmed => log.info(debugText) => log.warning(debugText)

Note that:

- The script calls log.info() on confirmed bars and log.warning() on the open bar. Users can filter the logs by logging level to inspect confirmed and unconfirmed bars’ logs separately.
- For larger collections whose “string” representations *exceed* 40,960 characters or cause excessive memory use, programmers can split them into *smaller parts* and convert them to strings separately. Alternatively, they can inspect individual elements via the `*.get()` method or for…in loops.

> TipIn contrast to arrays and matrices of numeric values, Boolean values, or strings, maps of such types *do not* have built-in “string” representations. However, programmers can inspect a map’s contents with

> this technique by using map.keys() and map.values() to retrieve *key* and *value* arrays, then calling str.tostring() to convert those arrays to “string” values.

Inspecting individual elements

Collections of “color” or non-fundamental types (e.g., labels) *do not* have built-in “string” representations. Consequently, the technique described in the Displaying collection strings section does not work for them.

To inspect a collection that does not have a built-in “string” format, programmers can retrieve elements individually within for…in loops or using methods such as `*.get()`, then use those elements in custom “string” constructions or other output routines.

Consider the following example, which calculates the ratio of close changes to the overall close range over `lengthInput` bars. It plots the resulting `osc` in a separate pane, and it draws a label on the main chart pane each time the variable’s absolute value is 1:

//@version=6 indicator("Inspecting individual elements demo") //@variable The number of bars in the calculation. int lengthInput = input.int(20, "Length", 2) //@variable The change in price across `lengthInput` - 1 bars. float priceChange = ta.change(close, lengthInput - 1) //@variable The total `close` range over `lengthInput` bars. float priceRange = ta.range(close, lengthInput) //@variable The ratio of the `priceChange` to the `priceRange`. float osc = priceChange / priceRange //@variable Teal if `osc` is positive, maroon otherwise. color oscColor = osc > 0 ? color.teal : color.maroon // Draw a label at the current bar's `bar_index` and `close` displaying `priceChange` when `osc` is 1 or -1. if math.abs(osc) == 1 string labelText = str.format("priceChange: {0,number,#.####}", priceChange) label.new(bar_index, close, labelText, color = oscColor, textcolor = color.white, force_overlay = true) // Plot the `osc` using the `oscColor`. plot(osc, "Oscillator", oscColor, 1, plot.style_area)

When a script creates labels, it *automatically* maintains an array containing each active label’s reference. Programmers can access this array using the label.all variable, and thus inspect each individual label’s properties on any bar.

In the version below, the script executes a log.info() call to display the current bar_index and the size of the label.all array for the latest bar. Then, it iterates through the array with a for…in loop. On each iteration, the script calls log.info() to log formatted text containing the array index and the corresponding label’s `x`, `y`, and `text` properties. Additionally, the script plots the oldest and newest active labels’ y-coordinates on each bar:

//@version=6 indicator("Inspecting individual elements demo") //@variable The number of bars in the calculation. int lengthInput = input.int(20, "Length", 2) //@variable The change in price across `lengthInput` - 1 bars. float priceChange = ta.change(close, lengthInput - 1) //@variable The total `close` range over `lengthInput` bars. float priceRange = ta.range(close, lengthInput) //@variable The ratio of the `priceChange` to the `priceRange`. float osc = priceChange / priceRange //@variable Teal if `osc` is positive, maroon otherwise. color oscColor = osc > 0 ? color.teal : color.maroon // Draw a label at the current bar's `bar_index` and `close` displaying `priceChange` when `osc` is 1 or -1. if math.abs(osc) == 1 string labelText = str.format("priceChange: {0,number,#.####}", priceChange) label.new(bar_index, close, labelText, color = oscColor, textcolor = color.white, force_overlay = true) // Plot the `osc` using the `oscColor`. plot(osc, "Oscillator", oscColor, 1, plot.style_area) // On the first or last tick of the latest bar, inspect all labels on the chart. if barstate.islast and (barstate.isnew or barstate.isconfirmed) // Log a message containing the current `bar_index` and `label.all.size()`. log.info("Current bar: {0,number,#}, Active labels: {1}", bar_index,

label.all.size()) // Loop through the `label.all` array. for [i, lbl] in label.all // Log a message containing the array index (`i`) and the label's `x`, `y`, and `text` properties. log.info("{0}, x: {1,number,#}, y: {2,number,#.#####}, text: {3}", i, lbl.get_x(), lbl.get_y(), lbl.get_text()) // Initialize variables for the oldest and newest active labels. label oldestLabel = na label newestLabel = na // Reassign the variables to the first and last labels in `label.all` when the array is not empty. if label.all.size() > 0 oldestLabel := label.all.first() newestLabel := label.all.last() // Plot the y-coordinate history of the `oldestLabel` and `newestLabel`. plot(label.get_y(oldestLabel), "oldestLabel y-coordinate", color.fuchsia, force_overlay = true) plot(label.get_y(newestLabel), "newestLabel y-coordinate", color.aqua, force_overlay = true)

Note that:

- It is not possible to obtain *all* properties from drawing objects. For example, there is no built-in method to retrieve a label's color. Some other types, such as table, do not have `*.get_*()` methods. If an object's properties are not directly accessible, programmers can create separate variables for the arguments of the drawing's `*.new()` or `*.set_*()` function, and then use those variables for debugging.
- In the above image, the logs show that the label.all array contains 55 elements. By default, Pine limits the number of labels to approximately 50, but the *precise* number of active labels varies. Programmers can increase the label drawing limit using the `max_labels_count` parameter of the indicator() or strategy() declaration statement.

Debugging objects of UDTs

User-defined types (UDTs) define the structures of objects. Objects contain a fixed set of *fields*, where each field can hold a separate value or reference to another specified type, even to another instance of the *same* user-defined type.

Because UDT objects can organize values and references to an arbitrary number of various *different* types, Pine does not have a built-in method to convert UDT objects to strings. Instead, to debug these structures, programmers must retrieve data from each *field* that requires inspection.

The following example defines a custom `Data` type with three fields. The first two fields reference arrays that hold successive price and time values. The third field specifies the number of bars between each new data sample. The script creates a new object of this type with a randomized `length` field on the first bar, then updates its arrays on bars whose bar_index values are divisible by that field.

The script uses array.covariance() and array.variance() on the object's `prices` and `times` arrays to calculate a time-based slope of the collected data, and then plots the result on the chart:

//@version=6 indicator("Debugging objects of UDTs demo") //@type A structure for storing time and price information once every `sampleMult` bars. //@field prices References an array of "float" price values. //@field times References an array of "int" UNIX timestamps. //@field sampleMult Number of bars per sample. type Data array<float> prices array<int> times int sampleMult //@variable The initial seed for the `math.random()` function. int seedInput = input.int(1234, "Seed", 1) //@variable References a `Data` object with arrays of 10 elements and a pseudorandom `sampleMult` value. var Data data = Data.new(array.new<float>(10), array.new<int>(10), int(math.random(1, 11, seedInput))) // Queue new data through the `prices` and `times` arrays of the `Data` object once every `data.sampleMult` bars. if bar_index % data.sampleMult == 0 data.prices.push(close) data.times.push(time) data.prices.shift() data.times.shift() //@variable The time-based slope calculated from the `data` array fields. float slope =

array.covariance(data.prices, data.times) / data.times.variance() // Plot the `slope` value. plot(slope, "Slope", slope > 0 ? color.teal : color.maroon, 3)

To verify and understand the script's calculations, we can *extract* information from the `Data` object's *fields* and inspect the data with Pine Logs or other outputs.

The script version below includes a log.info() call inside the if structure. The call displays formatted text representing information from the `Data` object's `prices`, `times`, and `length` fields in the Pine Logs pane. Now, we can view each change to the object's data to confirm the script's behavior:

//@version=6 indicator("Debugging objects of UDTs demo") //@type A structure for storing time and price information once every `sampleMult` bars. //@field prices References an array of "float" price values. //@field times References an array of "int" UNIX timestamps. //@field sampleMult Number of bars per sample. type Data array<float> prices array<int> times int sampleMult //@variable The initial seed for the `math.random()` function. int seedInput = input.int(1234, "Seed", 1) //@variable References a `Data` object with arrays of 10 elements and a pseudorandom `sampleMult` value. var Data data = Data.new(array.new<float>(10), array.new<int>(10), int(math.random(1, 11, seedInput))) // Queue new data through the `prices` and `times` arrays of the `Data` object once every `data.sampleMult` bars. if bar_index % data.sampleMult == 0 data.prices.push(close) data.times.push(time) data.prices.shift() data.times.shift() // Log formatted text containing information from the `Data` object's `prices`, `times`, and `sampleMult` // fields with the "info" or "warning" level. string fString = "Data object fields:\n\nprices: {0}\n\ntimes: {1}\n\nsampleMult: {2}\n-------" switch barstate.isconfirmed => log.info(fString, str.tostring(data.prices), str.tostring(data.times), data.sampleMult) => log.warning(fString, str.tostring(data.prices), str.tostring(data.times), data.sampleMult) //@variable The time-based slope calculated from the `data` array fields. float slope = array.covariance(data.prices, data.times) / data.times.variance() // Plot the `slope` value. plot(slope, "Slope", slope > 0 ? color.teal : color.maroon, 3)

Note that:

- The script calls log.info() on confirmed bars and log.warning() on open bars, allowing users to filter the results by logging level in the Pine Logs pane.

Organization and readability

Source code that is organized and easy to read is typically simpler to debug. Furthermore, well-written code is more straightforward for programmers to maintain and improve over time. Therefore, we recommend prioritizing organization and readability throughout the script-writing process, especially while debugging.

Below are a few helpful coding recommendations based on our Style guide and best practices:

- Follow the script organization guidelines. Organizing scripts based on this structure makes different parts of the code simple to locate and inspect.
- Use *identifiers* that you can read, distinguish, and understand. When a code contains unclear identifiers, it is often harder to debug efficiently. See our Naming conventions to learn our recommended identifier format.
- Use *type keywords* to signify the qualified types that variables and parameters can accept. Although Pine can usually *infer* variable and parameter types, declaring them explicitly improves readability and helps programmers distinguish between assignment and reassignment operations. Plus, it enables Pine's autosuggest feature to display more relevant type-based suggestions.

- Document the code using comments and compiler annotations (`//@function`, `//@variable`, etc.). The Pine Editor's autosuggest displays the text from annotations when the mouse pointer hovers over identifiers, making it simple to recall what different parts of the code represent.

Limitations

Introduction

As is mentioned in our Welcome page:

> *Because each script uses computational resources in the cloud, we must impose limits in order to share these resources fairly among our users. We strive to set as few limits as possible, but will of course have to implement as many as needed for the platform to run smoothly. Limitations apply to the amount of data requested from additional symbols, execution time, memory usage and script size.*

If you develop complex scripts using Pine Script®, sooner or later you will run into some of the limitations we impose. This section provides you with an overview of the limitations that you may encounter. There are currently no means for Pine Script programmers to get data on the resources consumed by their scripts. We hope this will change in the future.

In the meantime, when you are considering large projects, it is safest to make a proof of concept in order to assess the probability of your script running into limitations later in your project.

Below, we describe the limits imposed in the Pine Script environment.

Time

Script compilation

Scripts must compile before they are executed on charts. Compilation occurs when you save a script from the Pine Editor or when you add a script to the chart. A two-minute limit is imposed on compilation time, which will depend on the size and complexity of your script, and whether or not a cached version of a previous compilation is available. When a compile exceeds the two-minute limit, a warning is issued. Heed that warning by shortening your script because after three consecutive warnings a one-hour ban on compilation attempts is enforced. The first thing to consider when optimizing code is to avoid repetitions by using functions to encapsulate oft-used segments, and call functions instead of repeating code.

Script execution

Once a script is compiled it can be executed. See the Events that trigger script executions section of the Execution model page for a list of the events triggering the execution of a script. The time allotted for the script to execute on all bars of a dataset varies with account types. The limit is 20 seconds for basic accounts, 40 for others.

Loop execution

The execution time for any loop on any single bar is limited to 500 milliseconds. The outer loop of embedded loops counts as one loop, so it will time out first. Keep in mind that even though a loop may execute under the 500 ms time limit on a given bar, the time it takes to execute on all the dataset's bars may nonetheless cause your script to exceed the total execution time limit. For example, the limit on total execution time will make it impossible for you script to execute a 400 ms loop on each bar of a 20,000-bar dataset because your script would then need 8000 seconds to execute.

Chart visuals

Plot limits

A maximum of 64 plot counts are allowed per script. The functions that generate plot counts are:

- plot()
- plotarrow()
- plotbar()
- plotcandle()
- plotchar()
- plotshape()
- alertcondition()
- bgcolor()
- barcolor()
- fill(), but only if its `color` is of the series form.

The following functions do not generate plot counts:

- hline()
- line.new()
- label.new()
- table.new()
- box.new()

One function call can generate up to seven plot counts, depending on the function and how it is called. When your script exceeds the maximum of 64 plot counts, the runtime error message will display the plot count generated by your script. Once you reach that point, you can determine how many plot counts a function call generates by commenting it out in a script. As long as your script still throws an error, you will be able to see how the actual plot count decreases after you have commented out a line.

The following example shows different function calls and the number of plot counts each one will generate:

//@version=6 indicator("Plot count example") bool isUp = close > open color isUpColor = isUp ? color.green : color.red bool isDn = not isUp color isDnColor = isDn ? color.red : color.green // Uses one plot count each. p1 = plot(close, color = color.white) p2 = plot(open, color = na) // Uses two plot counts for the `close` and `color` series. plot(close, color = isUpColor) // Uses one plot count for the `close` series. plotarrow(close, colorup = color.green, colordown = color.red) // Uses two plot counts for the `close` and `colorup` series. plotarrow(close, colorup = isUpColor) // Uses three plot counts for the `close`, `colorup`, and the `colordown` series. plotarrow(close - open, colorup = isUpColor, colordown = isDnColor) // Uses four plot counts for the `open`, `high`, `low`, and `close` series. plotbar(open, high, low, close, color = color.white) // Uses five plot counts for the `open`, `high`, `low`, `close`, and `color` series. plotbar(open, high, low, close, color = isUpColor) // Uses four plot counts for the `open`, `high`, `low`, and `close` series. plotcandle(open, high, low, close, color = color.white, wickcolor = color.white, bordercolor = color.purple) // Uses five plot counts for the `open`, `high`, `low`, `close`, and `color` series. plotcandle(open, high, low, close, color = isUpColor, wickcolor = color.white, bordercolor = color.purple) // Uses six plot counts for the `open`, `high`, `low`, `close`, `color`, and `wickcolor` series. plotcandle(open, high, low, close, color = isUpColor, wickcolor = isUpColor , bordercolor = color.purple) // Uses seven plot counts for the `open`, `high`, `low`, `close`, `color`, `wickcolor`, and `bordercolor` series. plotcandle(open, high, low, close, color = isUpColor, wickcolor =

isUpColor , bordercolor = isUp ? color.lime : color.maroon) // Uses one plot count for the `close` series. plotchar(close, color = color.white, text = "|", textcolor = color.white) // Uses two plot counts for the `close`` and `color` series. plotchar(close, color = isUpColor, text = "—", textcolor = color.white) // Uses three plot counts for the `close`, `color`, and `textcolor` series. plotchar(close, color = isUpColor, text = "O", textcolor = isUp ? color.yellow : color.white) // Uses one plot count for the `close` series. plotshape(close, color = color.white, textcolor = color.white) // Uses two plot counts for the `close` and `color` series. plotshape(close, color = isUpColor, textcolor = color.white) // Uses three plot counts for the `close`, `color`, and `textcolor` series. plotshape(close, color = isUpColor, textcolor = isUp ? color.yellow : color.white) // Uses one plot count. alertcondition(close > open, "close > open", "Up bar alert") // Uses one plot count. bgcolor(isUp ? color.yellow : color.white) // Uses one plot count for the `color` series. fill(p1, p2, color = isUpColor)

This example generates a plot count of 56. If we were to add two more instances of the last call to plotcandle(), the script would throw an error stating that the script now uses 70 plot counts, as each additional call to plotcandle() generates seven plot counts, and 56 + (7 * 2) is 70.

Line, box, polyline, and label limits

Contrary to plots, which can cover the chart's entire dataset, scripts will only show the last 50 lines, boxes, polylines, and labels on the chart by default. One can increase the maximum number for each of these drawing types via the `max_lines_count`, `max_boxes_count`, `max_polylines_count`, and `max_labels_count` parameters of the script's indicator() or strategy() declaration statement. The maximum number of line, box, and label IDs is 500, and the maximum number of polyline IDs is 100.

In this example, we set the maximum number of recent labels shown on the chart to 100:

//@version=6 indicator("Label limits example", max_labels_count = 100, overlay = true) label.new(bar_index, high, str.tostring(high, format.mintick))

It's important to note when setting any of a drawing object's properties to na that its ID still exists and thus contributes to a script's drawing totals. To demonstrate this behavior, the following script draws a “Buy” and “Sell” label on each bar, with `x` values determined by the `longCondition` and `shortCondition` variables.

The “Buy” label's `x` value is na when the bar index is even, and the “Sell” label's `x` value is na when the bar index is odd. Although the `max_labels_count` is 10 in this example, we can see that the script displays fewer than 10 labels on the chart since the ones with na values also count toward the total:

//@version=6 // Approximate maximum number of label drawings MAX_LABELS = 10 indicator("labels with na", overlay = false, max_labels_count = MAX_LABELS) // Add background color for the last MAX_LABELS bars. bgcolor(bar_index > last_bar_index - MAX_LABELS ? color.new(color.green, 80) : na) longCondition = bar_index % 2 != 0 shortCondition = bar_index % 2 == 0 // Add "Buy" and "Sell" labels on each new bar. label.new(longCondition ? bar_index : na, 0, text = "Buy", color = color.new(color.green, 0), style = label.style_label_up) label.new(shortCondition ? bar_index : na, 0, text = "Sell", color = color.new(color.red, 0), style = label.style_label_down) plot(longCondition ? 1 : 0) plot(shortCondition ? 1 : 0)

To display the desired number of labels, we must eliminate label drawings we don't want to show rather than setting their properties to na. The example below uses an if structure to conditionally draw the “Buy” and “Sell” labels, preventing the script from creating new label IDs when it isn't necessary:

//@version=6 // Approximate maximum number of label drawings MAX_LABELS = 10 indicator("conditional labels", overlay = false, max_labels_count = MAX_LABELS) // Add background color for the last

MAX_LABELS bars. bgcolor(bar_index > last_bar_index - MAX_LABELS ? color.new(color.green, 80) : na) longCondition = bar_index % 2 != 0 shortCondition = bar_index % 2 == 0 // Add a "Buy" label when `longCondition` is true. if longCondition label.new(bar_index, 0, text = "Buy", color = color.new(color.green, 0), style = label.style_label_up) // Add a "Sell" label when `shortCondition` is true. if shortCondition label.new(bar_index, 0, text = "Sell", color = color.new(color.red, 0), style = label.style_label_down) plot(longCondition ? 1 : 0) plot(shortCondition ? 1 : 0)

Table limits

Scripts can display a maximum of nine tables on the chart, one for each of the possible locations: position.bottom_center, position.bottom_left, position.bottom_right, position.middle_center, position.middle_left, position.middle_right, position.top_center, position.top_left, and position.top_right. When attempting to place two tables in the same location, only the newest instance will show on the chart.

`request.*()` calls

Number of calls

A script can use up to 40 *unique* calls to the functions in the `request.*()` namespace, or up to 64 unique calls if the user has the Ultimate plan. A subsequent call to the same `request.*()` function with the same arguments is not typically unique. This limitation applies when using any of the following `request.*()` functions:

- request.security()
- request.security_lower_tf()
- request.currency_rate()
- request.dividends()
- request.splits()
- request.earnings()
- request.quandl()
- request.financial()
- request.economic()
- request.seed()

> NoteThe request.footprint() function has unique limitations. A script can execute only *one* unique call to this function. Additionally, only users who have a Premium or Ultimate plan can use scripts that call it. To learn more about this function and how to use it, refer to the `request.footprint()` section of the Other timeframes and data page.

When a script executes two or more identical `request.*()` function calls, only the *first* call usually counts toward this limit. The repeated calls do not count because they *reuse* the data from the first call rather than executing a redundant request. Note that when a script imports library functions containing `request.*()` calls within their scopes, those calls **do** count toward this limit, even if the script already calls the same `request.*()` function with the same arguments in its main scope.

The script below calls request.security() with the same arguments 50 times within a for loop. Although the script contains more than 40 `request.*()` calls, it *does not* raise an error because each call is **identical**. In

this case, it reuses the data from the first iteration's request.security() call for the repeated calls on all subsequent iterations:

//@version=6 indicator("`request.*()` call limit demo") //@variable The sum of values requested from all `request.security()` calls. float reqSum = 0.0 // Call `request.security()` 50 times within a loop. // More than 40 `request.*()` calls occur, but each call is identical. Redundant calls do not count toward the limit. for i = 1 to 50 reqSum += request.security(syminfo.tickerid, "1D", close) plot(reqSum)

Here, we modified the above script to call request.security() with a different `timeframe` argument on each iteration, meaning all 50 calls are now **unique**. This time, the script will reach the `request.*()` call limit while executing the loop and raise a runtime error because it requests a *distinct* dataset on each iteration:

//@version=6 indicator("`request.*()` call limit demo") //@variable The sum of values requested from all `request.security()` calls. float reqSum = 0.0 // Call `request.security()` 50 times within a loop with different `timeframe` arguments. // This loop causes a runtime error when `i == 41` because each iteration executes a unique request. for i = 1 to 50 reqSum += request.security(syminfo.tickerid, str.tostring(i), close) plot(reqSum)

Note that:

- These example scripts can call request.security() within a loop and allow "series string" `timeframe` arguments because Pine v6 scripts enable dynamic requests by default. See this section of the Other timeframes and data page for more information.

Intrabars

Scripts can retrieve up to the most recent 200,000 *intrabars* (lower-timeframe bars) via the request.security() or request.security_lower_tf() functions, depending on the user's plan:

- All non-professional plans – Basic, Essential, Plus, and Premium – can request up to 100K bars of data.
- Expert plans have access to 125K bars of data.
- Ultimate plans can request 200K lower-timeframe bars.

The `calc_bars_count` parameter of the `request.*()` functions limits the intrabar data retrieved by a request. If a `request.*()` call does not include a `calc_bars_count` argument, the number of requested bars is the same as the number of chart bars available for the symbol and timeframe. Otherwise, the function retrieves up to the specified number of bars, depending on the span of the dataset. The largest possible number of bars in the request depends on the limits listed above.

The number of bars on the chart's timeframe covered by a lower-timeframe request varies with the number of intrabars available for each chart bar. For example, if a script running on a 60-minute chart uses a `request.*()` call that requests data from the 1-minute timeframe, that call can retrieve data for up to 60 intrabars per chart bar. If the call uses the argument `calc_bars_count = 100000`, the minimum number of chart bars covered by the request is 1666, because 100000 / 60 = 1666.67. However, it's important to note that a that a provider might not report data for *every* minute within an hour. Therefore, such a request might cover more chart bars, depending on the available data.

Tuple element limit

All the `request.*()` function calls in a script taken together cannot return more than 127 tuple elements. When the combined tuple size of all `request.*()` calls will exceed 127 elements, one can instead utilize user-defined types (UDTs) to request a greater number of values.

The example below outlines this limitation and the way to work around it. The first request.security() call represents using a tuple with 128 elements as the `expression` argument. Since the number of elements is greater than 127, it would result in an error.

To avoid the error, we can use those same values as *fields* within an object of a UDT and pass its ID to the `expression` instead:

//@version=6 indicator("Tuple element limit") s1 = close s2 = close * 2 ... s128 = close * 128 // Causes an error. [v1, v2, v3, ..., v128] = request.security(syminfo.tickerid, "1D", [s1, s2, s3, ..., s128]) // Works fine: type myType float v1 float v2 float v3 ... float v128 myObj = request.security(syminfo.tickerid, "1D", myType.new(s1, s2, s3, ..., s128))

Note that:

- This example outlines a scenario where the script tries to evaluate 128 tuple elements in a single request.security() call. The same limitation applies if we were to split the tuple request across *multiple* calls. For example, two request.security() calls that each retrieve a tuple with 64 elements will also cause an error.

Script size and memory

Compiled tokens

Pine Script is a compiled language optimized for efficient, repeated script execution. When a user saves a script or applies it to a dataset, the compiler translates the Pine Script code into a tokenized *intermediate language (IL)*, then generates the final compiled code from the script's IL form. Using an IL representation enables the compiler to perform multiple transformations to optimize the script's structure. The compiler measures the size of a script based on the *number of tokens* in the script's *IL form* after transformations. It **does not** measure size as the number of characters or lines in the written source code.

The IL form of each indicator, strategy, and library script can contain a maximum of *100,256 tokens*. Libraries are compiled separately from the scripts that import them. In other words, code imported from a library does *not* directly affect a script's compiled size. However, the total size of all libraries that a script imports cannot exceed *1 million tokens*. There is no direct way to inspect a script's IL form or its size if the script compiles successfully. The only time that a script's size is visible is when it exeeds the size limit.

In most cases, the compiled size of a script will likely not reach the token limit. However, if it does, there are several techniques that can help reduce the size, including reducing *embedded data*, shortening or removing large constant strings, replacing repetitive statements with loops or concise function calls, splitting logic into multiple scripts, and moving code into separate libraries.

It's important to note that, due to the compiler's automatic optimizations, several types of manual code changes have *little to no effect* on a compiled script's size. For instance, the compiler automatically *discards* any *unused* code that does not affect the script's plots, drawings, or other *outputs*. Manually removing unused code thus has zero impact on the tokens in a script's IL form. Refer to the Compiled script contains

too many tokens error page to learn more about how code changes affect a script's compiled size, as well as how to reduce the size if it reaches the token limits.

The following example demonstrates how the compiler omits *unused* code, and why removing such code manually does not affect a script's size. The script below declares a user-defined type and a user-defined method, then creates a simple sequence of calculations:

//@version=6 indicator("My Script") plot(close) type myType float field = 10.0 method m(array<myType> a, myType v) => a.push(v) var arr = array.new<myType>() arr.push(myType.new(25)) arr.m(myType.new())

Although the above script contains multiple pieces of code, most of it *does not* contribute to the script's outputs. The *only* part of the script that produces an output is the `plot(close)` call. The compiler recognizes that the rest of the code is unused and *excludes* it from the script's IL translation. Therefore, the compiled form of the script is *identical* to that of the script below:

//@version=6 indicator("My Script") plot(close)

Variables per scope

Scripts can contain up to 1,000 variables in each of its scopes. Pine scripts always contain one global scope, represented by non-indented code, and they may contain zero or more local scopes. Local scopes are sections of indented code representing procedures executed within functions and methods, as well as if, switch, for, for…in, and while structures, which allow for one or more local blocks. Each local block counts as one local scope.

The branches of a conditional expression using the ?: ternary operator do not count as local blocks.

Compilation request size

The size of the compilation request for a script cannot exceed 5MB. The compilation request is all of the information that is sent to the compiler. This information comprises the script itself and any libraries the script imports.

Unlike the limit for compiled tokens, the request size limit includes unused parts of code. This is because the script is not compiled yet, so any unused code has not yet been optimized out.

To reduce the compilation request size, you can:

- Reduce the size of the script by optimizing the code.
- Reduce the number of script inputs (script inputs are counted separately).
- Remove any imported libraries that are not needed.
- Use smaller libraries. The entire library is sent for compilation, regardless of which functions are called.

Collections

Pine Script collections (arrays, matrices, and maps) can have a maximum of 100,000 elements. Each key-value pair in a map contains two elements, meaning maps can contain a maximum of 50,000 key-value pairs.

Other limitations

Maximum bars back

References to past values using the [] history-referencing operator are dependent on the size of the historical buffer maintained by the Pine Script runtime, which is limited to a maximum of 5000 bars for most series. Some built-in series like open, high, low, close, and time have larger historical buffers that can reference up to 10,000 bars.

If a script references values beyond the historical buffer's limit, it causes a runtime error. For more information about this error, refer to this section of the Error messages page, which discusses the historical buffer and how to change its size using either the max_bars_back() function or the `max_bars_back` parameter of the indicator() or strategy() declaration statement.

Drawings using xloc.bar_index can be positioned a maximum of 10,000 bars in the past.

Maximum bars forward

When positioning drawings using xloc.bar_index, it is possible to use bar index values greater than that of the current bar as *x* coordinates. A maximum of 500 bars in the future can be referenced.

This example shows how we use the `maxval` parameter in our input.int() function call to cap the user-defined number of bars forward we draw a projection line so that it never exceeds the limit:

//@version=6 indicator("Max bars forward example", overlay = true) // This function draws a `line` using bar index x-coordinates. drawLine(bar1, y1, bar2, y2) => // Only execute this code on the last bar. if barstate.islast // Create the line only the first time this function is executed on the last bar. var line lin = line.new(bar1, y1, bar2, y2, xloc.bar_index) // Change the line's properties on all script executions on the last bar. line.set_xy1(lin, bar1, y1) line.set_xy2(lin, bar2, y2) // Input determining how many bars forward we draw the `line`. int forwardBarsInput = input.int(10, "Forward Bars to Display", minval = 1, maxval = 500) // Calculate the line's left and right points. int leftBar = bar_index[2] float leftY = high[2] int rightBar = leftBar + forwardBarsInput float rightY = leftY + (ta.change(high)[1] * forwardBarsInput) // This function call is executed on all bars, but it only draws the `line` on the last bar. drawLine(leftBar, leftY, rightBar, rightY)

Chart bars

The number of bars appearing on charts is dependent on the amount of historical data available for the chart's symbol and timeframe, and on the type of account you hold. When the required historical date is available, the minimum number of chart bars is:

- 40000 historical bars for the Ultimate plan.
- 25000 historical bars for the Expert plan.
- 20000 historical bars for the Premium plan.
- 10000 historical bars for Essential and Plus plans.
- 5000 historical bars for other plans.

Trade orders in backtesting

A script can place a maximum of 9000 orders when backtesting strategies. Once it reaches that limit, the earlier orders are *trimmed* to store the information of new orders. Programmers can use the strategy.closedtrades.first_index variable to reference the index of the earliest untrimmed trade.

When using Deep Backtesting, the order limit is 1,000,000.

Profiling and optimization

Introduction

Pine Script® is a cloud-based compiled language geared toward efficient repeated script execution. When a user adds a Pine script to a chart, it executes *numerous* times, once for each available bar or tick in the data feeds it accesses, as explained in this manual's Execution model page.

The Pine Script compiler automatically performs several internal optimizations to accommodate scripts of various sizes and help them run smoothly. However, such optimizations *do not* prevent performance bottlenecks in script executions. As such, it's up to programmers to profile a script's runtime performance and identify ways to modify critical code blocks and lines when they need to improve execution times.

This page covers how to profile and monitor a script's runtime and executions with the Pine Profiler and explains some ways programmers can modify their code to optimize runtime performance.

For a quick introduction, see the following video, where we profile an example script and optimize it step-by-step, examining several common script inefficiencies and explaining how to avoid them along the way:

Play

Pine Profiler

Before diving into optimization, it's prudent to evaluate a script's runtime and pinpoint *bottlenecks*, i.e., areas in the code that substantially impact overall performance. With these insights, programmers can ensure they focus on optimizing where it truly matters instead of spending time and effort on low-impact code.

Enter the *Pine Profiler*, a powerful utility that analyzes the executions of all significant code lines and blocks in a script and displays helpful performance information next to the lines inside the Pine Editor. By inspecting the Profiler's results, programmers can gain a clearer perspective on a script's overall runtime, the distribution of runtime across its significant code regions, and the critical portions that may need extra attention and optimization.

Profiling a script

The Pine Profiler can analyze the runtime performance of any *editable* script coded in Pine Script v6. To profile a script, add it to the chart, open the source code in the Pine Editor, and turn on the “Profiler mode” switch in the dropdown accessible via the “More” option in the top-right corner:

We will use the script below for our initial profiling example, which calculates a custom `oscillator` based on average distances from the close price to upper and lower percentiles over `lengthInput` bars. It includes a few different types of *significant* code regions, which come with some differences in interpretation while profiling:

//@version=6 indicator("Pine Profiler demo") //@variable The number of bars in the calculations. int lengthInput = input.int(100, "Length", 2) //@variable The percentage for upper percentile calculation. float upperPercentInput = input.float(75.0, "Upper percentile", 50.0, 100.0) //@variable The percentage for lower percentile calculation. float lowerPercentInput = input.float(25.0, "Lower percentile", 0.0, 50.0) // Calculate percentiles using the linear interpolation method. float upperPercentile =

```
ta.percentile_linear_interpolation(close, lengthInput, upperPercentInput) float lowerPercentile =
ta.percentile_linear_interpolation(close, lengthInput, lowerPercentInput) // Declare arrays for upper and
lower deviations from the percentiles on the same line. var upperDistances = array.new<float>(lengthInput),
var lowerDistances = array.new<float>(lengthInput) // Queue distance values through the `upperDistances`
and `lowerDistances` arrays based on excessive price deviations. if math.abs(close - 0.5 * (upperPercentile
+ lowerPercentile)) > 0.5 * (upperPercentile - lowerPercentile) array.push(upperDistances, math.max(close -
upperPercentile, 0.0)) array.shift(upperDistances) array.push(lowerDistances, math.max(lowerPercentile -
close, 0.0)) array.shift(lowerDistances) //@variable The average distance from the `upperDistances` array.
float upperAvg = upperDistances.avg() //@variable The average distance from the `lowerDistances` array.
float lowerAvg = lowerDistances.avg() //@variable The ratio of the difference between the `upperAvg` and
`lowerAvg` to their sum. float oscillator = (upperAvg - lowerAvg) / (upperAvg + lowerAvg) //@variable The
color of the plot. A green-based gradient if `oscillator` is positive, a red-based gradient otherwise. color
oscColor = oscillator > 0 ? color.from_gradient(oscillator, 0.0, 1.0, color.gray, color.green) :
color.from_gradient(oscillator, -1.0, 0.0, color.red, color.gray) // Plot the `oscillator` with the `oscColor`.
plot(oscillator, "Oscillator", oscColor, style = plot.style_area)
```

Once enabled, the Profiler collects information from all executions of the script's significant code lines and blocks, then displays bars and approximate runtime percentages to the left of the code lines inside the Pine Editor:

Note that:

- The Profiler tracks every execution of a significant code region, including the executions on *realtime ticks*. Its information updates over time as new executions occur.
- Profiler results **do not** appear for script declaration statements, type declarations, other *insignificant* code lines such as variable declarations with no tangible impact, *unused code* that the script's outputs do not depend on, or *repetitive code* that the compiler optimizes during translation. See this section for more information.

When a script contains at least *four* significant lines of code, the Profiler will include “flame” icons next to the *top three* code regions with the highest performance impact. If one or more of the highest-impact code regions are *outside* the lines visible inside the Pine Editor, a “flame” icon and a number indicating how many critical lines are outside the view will appear at the top or bottom of the left margin. Clicking the icon will vertically scroll the Editor's window to show the nearest critical line:

Hovering the mouse pointer over the space next to a line highlights the analyzed code and exposes a tooltip with additional information, including the time spent and the number of executions. The information shown next to each line and in the corresponding tooltip depends on the profiled code region. The section below explains different types of code the Profiler analyzes and how to interpret their performance results.

> Note
>
> Similar to profiling tools for other languages, the Pine Profiler *wraps* a script and its significant code with extra calculations to collect performance data. Therefore, a script’s resource usage **increases** while profiling, and the results are thus **estimates** rather than precise performance measurements.

> Furthermore, the Profiler cannot collect and display individual performance data for the *internal calculations* that also affect runtime, including the calculations required to track performance, meaning the time values shown for all a script's code regions **do not** add up to exactly 100% of its overall runtime.

Interpreting profiled results

Single-line results

For a code line containing single-line expressions, the Profiler bar and displayed percentage represent the relative portion of the script's total runtime spent on that line. The corresponding tooltip displays three fields:

- The "Line number" field indicates the analyzed code line.
- The "Time" field shows the runtime percentage for the line of code, the runtime spent on that line, and the script's total runtime.
- The "Executions" field shows the number of times that specific line executed while running the script.

Here, we hovered the pointer over the space next to line 12 of our profiled code to view its tooltip:

float upperPercentile = ta.percentile_linear_interpolation(close, lengthInput, upperPercentInput)

Note that:

- The time information for the line represents the time spent completing *all* executions, **not** the time spent on a single execution.
- To estimate the *average* time spent per execution, divide the line's time by the number of executions. In this case, the tooltip shows that line 12 took about 14.1 milliseconds to execute 20,685 times, meaning the average time per execution was approximately 14.1 ms / 20685 = 0.0006816534 milliseconds (0.6816534 microseconds).

When a line of code consists of more than one expression separated by commas, the number of executions shown in the tooltip represents the *sum* of each expression's total executions, and the time value displayed represents the total time spent evaluating all the line's expressions.

For instance, this global line from our initial example includes two variable declarations separated by commas. Each uses the var keyword, meaning the script only executes them once on the first available bar. As we see in the Profiler tooltip for the line, it counted *two* executions (one for each expression), and the time value shown is the *combined* result from both expressions on the line:

var upperDistances = array.new<float>(lengthInput), var lowerDistances = array.new<float>(lengthInput)

Note that:

- When analyzing scripts with more than one expression on the same line, we recommend moving each expression to a *separate line* for more detailed insights while profiling, namely if they may contain *higher-impact* calculations.

When using line wrapping for readability or stylistic purposes, the Profiler considers all portions of a wrapped line as part of the *first line* where it starts in the Pine Editor.

For example, although this code from our initial script occupies more than one line in the Pine Editor, it's still treated as a *single* line of code, and the Profiler tooltip displays single-line results, with the "Line number"

field showing the *first* line in the Editor that the wrapped line occupies:

color oscColor = oscillator > 0 ? color.from_gradient(oscillator, 0.0, 1.0, color.gray, color.green) : color.from_gradient(oscillator, -1.0, 0.0, color.red, color.gray)

Code block results

For a line at the start of a loop or conditional structure, the Profiler bar and percentage represent the relative portion of the script's runtime spent on the **entire code block**, not just the single line. The corresponding tooltip displays four fields:

- The “Code block range” field indicates the range of lines included in the structure.
- The “Time” field shows the code block's runtime percentage, the time spent on all block executions, and the script's total runtime.
- The “Line time” field shows the runtime percentage for the block's initial line, the time spent on that line, and the script's total runtime. The interpretation differs for switch blocks or if blocks *with* `else if` statements, as the values represent the total time spent on **all** the structure's conditional statements. See below for more information.
- The “Executions” field shows the number of times the code block executed while running the script.

Here, we hovered over the space next to line 19 in our initial script, the beginning of a simple if structure *without* `else if` statements. As we see below, the tooltip shows performance information for the entire code block and the current line:

if math.abs(close - 0.5 * (upperPercentile + lowerPercentile)) > 0.5 * (upperPercentile - lowerPercentile) array.push(upperDistances, math.max(close - upperPercentile, 0.0)) array.shift(upperDistances) array.push(lowerDistances, math.max(lowerPercentile - close, 0.0)) array.shift(lowerDistances)

Note that:

- The “Time” field shows that the total time spent evaluating the structure 20,685 times was 7.2 milliseconds.
- The “Line time” field indicates that the runtime spent on the *first line* of this if structure was about three milliseconds.

Users can also inspect the results from lines and nested blocks within a code block's range to gain more granular performance insights. Here, we hovered over the space next to line 20 within the code block to view its single-line result:

Note that:

- The number of executions shown is *less than* the result for the entire code block, as the condition that controls the execution of this line does not return `true` all the time. The opposite applies to the code inside loops since each execution of a loop statement can trigger **several** executions of the loop's local block.

When profiling a switch structure or an if structure that includes `else if` statements, the “Line time” field will show the time spent executing **all** the structure's conditional expressions, **not** just the block's first line. The results for the lines inside the code block range will show runtime and executions for each **local block**. This format is necessary for these structures due to the Profiler's calculation and display constraints. See this section for more information.

For example, the “Line time” for the switch structure in this script represents the time spent evaluating *all four* conditional statements within its body, as the Profiler *cannot* track them separately. The results for each line in the code block's range represent the performance information for each *local block*:

//@version=6 indicator("`switch` and `if...else if` results demo") //@variable The upper band for oscillator calculation. var float upperBand = close //@variable The lower band for oscillator calculation. var float lowerBand = close // Update the `upperBand` and `lowerBand` based on the proximity of the `close` to the current band values. // The "Line time" field on line 11 represents the time spent on all 4 conditional expressions in the structure. switch close > upperBand => upperBand := close close < lowerBand => lowerBand := close upperBand - close > close - lowerBand => upperBand := 0.9 * upperBand + 0.1 * close close - lowerBand > upperBand - close => lowerBand := 0.9 * lowerBand + 0.1 * close //@variable The ratio of the difference between `close` and `lowerBand` to the band range. float oscillator = 100.0 * (close - lowerBand) / (upperBand - lowerBand) // Plot the `oscillator` as columns with a dynamic color. plot(oscillator, "Oscillator", oscillator > 50.0 ? color.teal : color.maroon, style = plot.style_columns, histbase = 50.0)

When the conditional logic in such structures involves significant calculations, programmers may require more granular performance information for each calculated condition. An effective way to achieve this analysis is to use *nested* if blocks instead of the more compact switch or `if...else if` structures. For example, instead of:

switch <expression1> => <localBlock1> <expression2> => <localBlock2> => <localBlock3>

or:

if <expression1> <localBlock1> else if <expression2> <localBlock2> else <localBlock3>

one can use nested if blocks for more in-depth profiling while maintaining the same logical flow:

if <expression1> <localBlock1> else if <expression2> <localBlock2> else <localBlock3>

Below, we changed the previous switch example to an equivalent nested if structure. Now, we can view the runtime and executions for each significant part of the conditional pattern individually:

//@version=6 indicator("`switch` and `if...else if` results demo") //@variable The upper band for oscillator calculation. var float upperBand = close //@variable The lower band for oscillator calculation. var float lowerBand = close // Update the `upperBand` and `lowerBand` based on the proximity of the `close` to the current band values. if close > upperBand upperBand := close else if close < lowerBand lowerBand := close else if upperBand - close > close - lowerBand upperBand := 0.9 * upperBand + 0.1 * close else if close - lowerBand > upperBand - close lowerBand := 0.9 * lowerBand + 0.1 * close //@variable The ratio of the difference between `close` and `lowerBand` to the band range. float oscillator = 100.0 * (close - lowerBand) / (upperBand - lowerBand) // Plot the `oscillator` as columns with a dynamic color. plot(oscillator, "Oscillator", oscillator > 50.0 ? color.teal : color.maroon, style = plot.style_columns, histbase = 50.0)

Note that:

- This same process can also apply to ternary operations. When a complex ternary expression's operands contain significant calculations, reorganizing the logic into a nested if structure allows more detailed Profiler results, making it easier to spot critical parts.

User-defined function calls

User-defined functions and methods are functions written by users. They encapsulate code sequences that a script may execute several times. Users often write functions and methods for improved code modularity, reusability, and maintainability.

The indented lines of code within a function represent its *local scope*, i.e., the sequence that executes *each time* the script calls it. Unlike code in a script's global scope, which a script evaluates once on each execution, the code inside a function may activate zero, one, or *multiple times* on each script execution, depending on the conditions that trigger the calls, the number of calls that occur, and the function's logic.

This distinction is crucial to consider while interpreting Profiler results. When a profiled code contains user-defined function or method calls:

- The results for each *function call* reflect the runtime allocated toward it and the total number of times the script activated that specific call.
- The time and execution information for all local code *inside* a function's scope reflects the combined results from **all** calls to the function.

This example contains a user-defined `similarity()` function that estimates the similarity of two series, which the script calls only *once* from the global scope on each execution. In this case, the Profiler's results for the code inside the function's body correspond to that specific call:

//@version=6 indicator("User-defined function calls demo") //@function Estimates the similarity between two standardized series over `length` bars. // Each individual call to this function activates its local scope. similarity(float sourceA, float sourceB, int length) => // Standardize `sourceA` and `sourceB` for comparison. float normA = (sourceA - ta.sma(sourceA, length)) / ta.stdev(sourceA, length) float normB = (sourceB - ta.sma(sourceB, length)) / ta.stdev(sourceB, length) // Calculate and return the estimated similarity of `normA` and `normB`. float abSum = math.sum(normA * normB, length) float a2Sum = math.sum(normA * normA, length) float b2Sum = math.sum(normB * normB, length) abSum / math.sqrt(a2Sum * b2Sum) // Plot the similarity between the `close` and an offset `close` series. plot(similarity(close, close[1], 100), "Similarity 1", color.red)

Let's increase the number of times the script calls the function each time it executes. Here, we changed the script to call our user-defined function *five times*:

//@version=6 indicator("User-defined function calls demo") //@function Estimates the similarity between two standardized series over `length` bars. // Each individual call to this function activates its local scope. similarity(float sourceA, float sourceB, int length) => // Standardize `sourceA` and `sourceB` for comparison. float normA = (sourceA - ta.sma(sourceA, length)) / ta.stdev(sourceA, length) float normB = (sourceB - ta.sma(sourceB, length)) / ta.stdev(sourceB, length) // Calculate and return the estimated similarity of `normA` and `normB`. float abSum = math.sum(normA * normB, length) float a2Sum = math.sum(normA * normA, length) float b2Sum = math.sum(normB * normB, length) abSum / math.sqrt(a2Sum * b2Sum) // Plot the similarity between the `close` and several offset `close` series. plot(similarity(close, close[1], 100), "Similarity 1", color.red) plot(similarity(close, close[2], 100), "Similarity 2", color.orange) plot(similarity(close, close[4], 100), "Similarity 3", color.green) plot(similarity(close, close[8], 100), "Similarity 4", color.blue) plot(similarity(close, close[16], 100), "Similarity 5", color.purple)

In this case, the local code results no longer correspond to a *single* evaluation per script execution. Instead, they represent the *combined* runtime and executions of the local code from **all five** calls. As we see below, the results after running this version of the script across the same data show 137,905 executions of the local code, *five times* the number from when the script only contained one `similarity()` function call:

> NoteIf the local scopes of a script's user-defined functions or methods contain calls to `request.*()` functions, the *translated form* of the script extracts such calls **outside** the functions' scopes to evaluate them **separately**. Consequently, the Profiler's results for lines with calls to those user-defined functions **do not** include the time spent on the `request.*()` calls. See the section below to learn more.

When requesting other contexts

Pine scripts can request data from other *contexts*, i.e., different symbols, timeframes, or data modifications than what the chart's data uses by calling the `request.*()` family of functions or specifying an alternate `timeframe` in the indicator() declaration statement.

When a script requests data from another context, it evaluates all required scopes and calculations within that context, as explained in the Other timeframes and data page. This behavior can affect the runtime of a script's code regions and the number of times they execute.

The Profiler information for any code line or block represents the results from executing the code in *all necessary contexts*, which may or may not include the chart's data. Pine Script determines which contexts to execute code within based on the calculations required by a script's data requests and outputs.

Let's look at a simple example. This initial script only uses the chart's data for its calculations. It declares a `pricesArray` variable with the varip keyword, meaning the array assigned to it persists across the data's history and all available realtime ticks. On each execution, the script calls array.push() to push a new close value into the array, and it plots the array's size.

After profiling the script across all the bars on an intraday chart, we see that the number of elements in the `pricesArray` corresponds to the number of executions the Profiler shows for the array.push() call on line 8:

//@version=6 indicator("When requesting other contexts demo") //@variable An array containing the `close` value from every available price update. varip array<float> pricesArray = array.new<float>() // Push a new `close` value into the `pricesArray` on each update. array.push(pricesArray, close) // Plot the size of the `pricesArray`. plot(array.size(pricesArray), "Total number of chart price updates")

Now, let's try evaluating the size of the `pricesArray` from *another context* instead of using the chart's data. Below, we've added a request.security() call with array.size(pricesArray) as its `expression` argument to retrieve the value calculated on the "1D" timeframe and plotted that result instead.

In this case, the number of executions the Profiler shows on line 8 still corresponds to the number of elements in the `pricesArray`. However, it did not execute the same number of times since the script did not require the *chart's data* in the calculations. It only needed to initialize the array and evaluate array.push() across all the requested *daily data*, which has a different number of price updates than our current intraday chart:

//@version=6 indicator("When requesting other contexts demo") //@variable An array containing the `close` value from every available price update. varip array<float> pricesArray = array.new<float>() // Push a new `close` value into the `pricesArray` on each update. array.push(pricesArray, close) // Plot the size of the `pricesArray` requested from the daily timeframe. plot(request.security(syminfo.tickerid, "1D", array.size(pricesArray)), "Total number of daily price updates")

Note that:

- The requested EOD data in this example had fewer data points than our intraday chart, so the array.push() call required fewer executions in this case. However, EOD feeds *do not* have history limitations, meaning it's also possible for requested HTF data to span **more** bars than a user's chart, depending on the timeframe, the data provider, and the user's plan.

If this script were to plot the array.size() value directly in addition to the requested daily value, it would then require the creation of *two* arrays (one for each context) and the execution of array.push() across both the chart's data *and* the data from the daily timeframe. As such, the declaration on line 5 will execute *twice*, and the results on line 8 will reflect the time and executions accumulated from evaluating the array.push() call across **both separate datasets**:

//@version=6 indicator("When requesting other contexts demo") //@variable An array containing the `close` value from every available price update. varip array<float> pricesArray = array.new<float>() // Push a new `close` value into the `pricesArray` on each update. array.push(pricesArray, close) // Plot the size of the `pricesArray` from the daily timeframe and the chart's context. // Including both in the outputs requires executing line 5 and line 8 across BOTH datasets. plot(request.security(syminfo.tickerid, "1D", array.size(pricesArray)), "Total number of daily price updates") plot(array.size(pricesArray), "Total number of chart price updates")

It's important to note that when a script calls a user-defined function or method that contains `request.*()` calls in its local scope, the script's *translated form* extracts the `request.*()` calls **outside** the scope and encapsulates the expressions they depend on within **separate functions**. When the script executes, it evaluates the required `request.*()` calls first, then *passes* the requested data to a *modified form* of the user-defined function.

Since the translated script executes a user-defined function's data requests separately **before** evaluating non-requested calculations in its local scope, the Profiler's results for lines containing calls to the function **will not** include the time spent on its `request.*()` calls or their required expressions.

As an example, the following script contains a user-defined `getCompositeAvg()` function with a request.security() call that requests the math.avg() of 10 ta.wma() calls with different `length` arguments from a specified `symbol`. The script uses the function to request the average result using a Heikin Ashi ticker ID:

//@version=6 indicator("User-defined functions with `request.*()` calls demo", overlay = true) int multInput = input.int(10, "Length multiplier", 1) string tickerID = ticker.heikinashi(syminfo.tickerid) getCompositeAvg(string symbol, int lengthMult) => request.security(symbol, timeframe.period, math.avg(ta.wma(close, lengthMult), ta.wma(close, 2 * lengthMult), ta.wma(close, 3 * lengthMult), ta.wma(close, 4 * lengthMult), ta.wma(close, 5 * lengthMult), ta.wma(close, 6 * lengthMult), ta.wma(close, 7 * lengthMult), ta.wma(close, 8 * lengthMult), ta.wma(close, 9 * lengthMult), ta.wma(close, 10 * lengthMult))) plot(getCompositeAvg(tickerID, multInput), "Composite average", linewidth = 3)

After profiling the script, users might be surprised to see that the runtime results shown inside the function's body heavily **exceed** the results shown for the *single* `getCompositeAvg()` call:

The results appear this way since the translated script includes internal modifications that *moved* the request.security() call and its expression **outside** the function's scope, and the Profiler has no way to represent the results from those calculations other than displaying them next to the request.security() line in this scenario. The code below roughly illustrates how the translated script looks:

//@version=6 indicator("User-defined functions with `request.*()` calls demo", overlay = true) int multInput = input.int(10, "Length multiplier") string tickerID = ticker.heikinashi(syminfo.tickerid) secExpr(int lengthMult)=> math.avg(ta.wma(close, lengthMult), ta.wma(close, 2 * lengthMult), ta.wma(close, 3 * lengthMult), ta.wma(close, 4 * lengthMult), ta.wma(close, 5 * lengthMult), ta.wma(close, 6 * lengthMult), ta.wma(close, 7 * lengthMult), ta.wma(close, 8 * lengthMult), ta.wma(close, 9 * lengthMult), ta.wma(close, 10 * lengthMult)) float sec = request.security(tickerID, timeframe.period, secExpr(multInput)) getCompositeAvg(float s) => s plot(getCompositeAvg(sec), "Composite average", linewidth = 3)

Note that:

- The `secExpr()` code represents the *separate function* used by request.security() to calculate the required expression in the requested context.
- The request.security() call takes place in the **outer scope**, outside the `getCompositeAvg()` function.
- The translation substantially reduced the local code of `getCompositeAvg()`. It now solely returns a value passed into it, as all the function's required calculations take place **outside** its scope. Due to this reduction, the function call's performance results **will not** reflect any of the time spent on the data request's required calculations.

Insignificant, unused, and redundant code

When inspecting a profiled script's results, it's crucial to understand that *not all* code in a script necessarily impacts runtime performance. Some code has no direct performance impact, such as a script's declaration statement and type declarations. Other code regions with insignificant expressions, such as most `input.*()` calls, variable references, or variable declarations without significant calculations, have little to *no effect* on a script's runtime. Therefore, the Profiler will **not** display performance results for these types of code.

Additionally, Pine scripts do not execute code regions that their *outputs* (plots, drawings, logs, etc.) do not depend on, as the compiler automatically **removes** them during translation. Since unused code regions have *zero* impact on a script's performance, the Profiler will **not** display any results for them.

The following example contains a `barsInRange` variable and a for loop that adds 1 to the variable's value for each historical close price between the current high and low over `lengthInput` bars. However, the script **does not use** these calculations in its outputs, as it only plots the close price. Consequently, the script's compiled form **discards** that unused code and only considers the plot(close) call.

The Profiler does not display **any** results for this script since it does not execute any **significant** calculations:

//@version=6 indicator("Unused code demo") //@variable The number of historical bars in the calculation. int lengthInput = input.int(100, "Length", 1) //@variable The number of closes over `lengthInput` bars between the current bar's `high` and `low`. int barsInRange = 0 for i = 1 to lengthInput //@variable The `close` price from `i` bars ago. float pastClose = close[i] // Add 1 to `barsInRange` if the `pastClose` is between the current bar's `high` and `low`. if pastClose > low and pastClose < high barsInRange += 1 // Plot the `close` price. This is the only output. // Since the outputs do not require any of the above calculations, the compiled script will not execute them. plot(close)

Note that:

- Although this script does not use the input.int() from line 5 and discards all its associated calculations, the “Length” input *will* still appear in the script's settings, as the compiler **does not** completely remove

unused inputs.

If we change the script to plot the `barsInRange` value instead, the declared variables and the for loop are no longer unused since the output depends on them, and the Profiler will now display performance information for that code:

//@version=6 indicator("Unused code demo") //@variable The number of historical bars in the calculation. int lengthInput = input.int(100, "Length", 1) //@variable The number of closes over `lengthInput` bars between the current bar's `high` and `low`. int barsInRange = 0 for i = 1 to lengthInput //@variable The `close` price from `i` bars ago. float pastClose = close[i] // Add 1 to `barsInRange` if the `pastClose` is between the current bar's `high` and `low`. if pastClose > low and pastClose < high barsInRange += 1 // Plot the `barsInRange` value. The above calculations will execute since the output requires them. plot(barsInRange, "Bars in range")

Note that:

- The Profiler does not show performance information for the `lengthInput` declaration on line 5 or the `barsInRange` declaration on line 8 since the expressions on these lines do not impact the script's performance.

When possible, the compiler also simplifies certain instances of *redundant code* in a script, such as some forms of identical expressions with the same fundamental type values. This optimization allows the compiled script to only execute such calculations *once*, on the first occurrence, and *reuse* the calculated result for each repeated instance that the outputs depend on.

If a script contains repetitive code and the compiler simplifies it, the Profiler will only show results for the **first occurrence** of the code since that's the only time the script requires the calculation.

For example, this script contains a code line that plots the value of ta.sma(close, 100) and 12 code lines that plot the value of ta.sma(close, 500):

//@version=6 indicator("Redundant calculations demo", overlay = true) // Plot the 100-bar SMA of `close` values one time. plot(ta.sma(close, 100), "100-bar SMA", color.teal, 3) // Plot the 500-bar SMA of `close` values 12 times. After compiler optimizations, only the first `ta.sma(close, 500)` // call on line 9 requires calculation in this case. plot(ta.sma(close, 500), "500-bar SMA", #001aff, 12) plot(ta.sma(close, 500), "500-bar SMA", #4d0bff, 11) plot(ta.sma(close, 500), "500-bar SMA", #7306f7, 10) plot(ta.sma(close, 500), "500-bar SMA", #920be9, 9) plot(ta.sma(close, 500), "500-bar SMA", #ae11d5, 8) plot(ta.sma(close, 500), "500-bar SMA", #c618be, 7) plot(ta.sma(close, 500), "500-bar SMA", #db20a4, 6) plot(ta.sma(close, 500), "500-bar SMA", #eb2c8a, 5) plot(ta.sma(close, 500), "500-bar SMA", #f73d6f, 4) plot(ta.sma(close, 500), "500-bar SMA", #fe5053, 3) plot(ta.sma(close, 500), "500-bar SMA", #ff6534, 2) plot(ta.sma(close, 500), "500-bar SMA", #ff7a00, 1)

Since the last 12 lines all contain identical ta.sma() calls, the compiler can automatically simplify the script so that it only needs to evaluate ta.sma(close, 500) *once* per execution rather than repeating the calculation 11 more times.

As we see below, the Profiler only shows results for lines 5 and 9. These are the only parts of the code requiring significant calculations since the ta.sma() calls on lines 10-20 are redundant in this case:

Another type of repetitive code optimization occurs when a script contains two or more user-defined functions or methods with identical compiled forms. In such a case, the compiler simplifies the script by

removing the redundant functions, and the script will treat all calls to the redundant functions as calls to the **first** defined version. Therefore, the Profiler will only show local code performance results for the *first* function since the discarded “clones” will never execute.

For instance, the script below contains two user-defined functions, `metallicRatio()` and `calcMetallic()`, that calculate a metallic ratio of a given order raised to a specified exponent:

//@version=6 indicator("Redundant functions demo") //@variable Controls the base ratio for the `calcMetallic()` call. int order1Input = input.int(1, "Order 1", 1) //@variable Controls the base ratio for the `metallicRatio()` call. int order2Input = input.int(2, "Order 2", 1) //@function Calculates the value of a metallic ratio with a given `order`, raised to a specified `exponent`. //@param order Determines the base ratio used. 1 = Golden Ratio, 2 = Silver Ratio, 3 = Bronze Ratio, and so on. //@param exponent The exponent applied to the ratio. metallicRatio(int order, float exponent) => math.pow((order + math.sqrt(4.0 + order * order)) * 0.5, exponent) //@function A function with the same signature and body as `metallicRatio()`. // The script discards this function and treats `calcMetallic()` as an alias for `metallicRatio()`. calcMetallic(int ord, float exp) => math.pow((ord + math.sqrt(4.0 + ord * ord)) * 0.5, exp) // Plot the results from a `calcMetallic()` and `metallicRatio()` call. plot(calcMetallic(order1Input, bar_index % 5), "Ratio 1", color.orange, 3) plot(metallicRatio(order2Input, bar_index % 5), "Ratio 2", color.maroon)

Despite the differences in the function and parameter names, the two functions are otherwise identical, which the compiler detects while translating the script. In this case, it **discards** the redundant `calcMetallic()` function, and the compiled script treats the `calcMetallic()` call as a `metallicRatio()` call.

As we see here, the Profiler shows performance information for the `calcMetallic()` and `metallicRatio()` calls on lines 21 and 22, but it does **not** show any results for the local code of the `calcMetallic()` function on line 18. Instead, the Profiler's information on line 13 within the `metallicRatio()` function reflects the local code results from **both** function calls:

A look into the Profiler's inner workings

The Pine Profiler wraps all necessary code regions with specialized *internal functions* to track and collect required information across script executions. It then passes the information to additional calculations that organize and display the performance results inside the Pine Editor. This section gives users a peek into how the Profiler applies internal functions to wrap Pine code and collect performance data.

There are two main internal **(non-Pine)** functions the Profiler wraps significant code with to facilitate runtime analysis. The first function retrieves the current system time at specific points in the script's execution, and the second maps cumulative elapsed time and execution data to specific code regions. We represent these functions in this explanation as `System.timeNow()` and `registerPerf()` respectively.

When the Profiler detects code that requires analysis, it adds `System.timeNow()` above the code to get the initial time before execution. Then, it adds `registerPerf()` below the code to map and accumulate the elapsed time and number of executions. The elapsed time added on each `registerPerf()` call is the `System.timeNow()` value *after* the execution minus the value *before* the execution.

The following *pseudocode* outlines this process for a single line of code, where `_startX` represents the starting time for the `lineX` line:

long _startX = System.timeNow() <code_line_to_analyze> registerPerf(System.timeNow() - _startX, lineX)

The process is similar for code blocks. The difference is that the `registerPerf()` call maps the data to a *range of lines* rather than a single line. Here, `lineX` represents the *first* line in the code block, and `lineY` represents the block's *last* line:

long _startX = System.timeNow() <code_block_to_analyze> registerPerf(System.timeNow() - _startX, lineX, lineY)

Note that:

- In the above snippets, `long`, `System.timeNow()`, and `registerPerf()` represent *internal code*, **not** Pine Script code.

Let's now look at how the Profiler wraps a full script and all its significant code. We will start with this script, which calculates three pseudorandom series and displays their average result. The script utilizes an object of a user-defined type to store a pseudorandom state, a method to calculate new values and update the state, and an if…else if structure to update each series based on generated values:

//@version=6 indicator("Profiler's inner workings demo") int seedInput = input.int(12345, "Seed") type LCG float state method generate(LCG this, int generations = 1) => float result = 0.0 for i = 1 to generations this.state := 16807 * this.state % 2147483647 result += this.state / 2147483647 result / generations var lcg = LCG.new(seedInput) var float val0 = 1.0 var float val1 = 1.0 var float val2 = 1.0 if lcg.generate(10) < 0.5 val0 *= 1.0 + (2.0 * lcg.generate(50) - 1.0) * 0.1 else if lcg.generate(10) < 0.5 val1 *= 1.0 + (2.0 * lcg.generate(50) - 1.0) * 0.1 else if lcg.generate(10) < 0.5 val2 *= 1.0 + (2.0 * lcg.generate(50) - 1.0) * 0.1 plot(math.avg(val0, val1, val2), "Average pseudorandom result", color.purple)

The Profiler will wrap the entire script and all necessary code regions, excluding any insignificant, unused, or redundant code, with the aforementioned **internal** functions to collect performance data. The *pseudocode* below demonstrates how this process applies to the above script:

long _startMain = System.timeNow() // Start time for the script's overall execution. // <Additional internal code executes here> //@version=6 indicator("Profiler's inner workings demo") // Declaration statements do not require profiling. int seedInput = input.int(12345, "Seed") // Variable declaration without significant calculation. type LCG // Type declarations do not require profiling. float state method generate(LCG this, int generations = 1) => // Function signature does not affect runtime. float result = 0.0 // Variable declaration without significant calculation. long _start11 = System.timeNow() // Start time for the loop block that begins on line 11. for i = 1 to generations // Loop header calculations are not independently wrapped. long _start12 = System.timeNow() // Start time for line 12. this.state := 16807 * this.state % 2147483647 registerPerf(System.timeNow() - _start12, line12) // Register performance info for line 12. long _start13 = System.timeNow() // Start time for line 13. result += this.state / 2147483647 registerPerf(System.timeNow() - _start13, line13) // Register performance info for line 13. registerPerf(System.timeNow() - _start11, line11, line13) // Register performance info for the block (line 11 - 13). long _start14 = System.timeNow() // Start time for line 14. result / generations registerPerf(System.timeNow() - _start14, line14) // Register performance info for line 14. long _start16 = System.timeNow() // Start time for line 16. var lcg = LCG.new(seedInput) registerPerf(System.timeNow() - _start16, line16) // Register performance info for line 16. var float val0 = 1.0 // Variable declarations without significant calculations. var float val1 = 1.0 var float val2 = 1.0 long _start22 = System.timeNow() // Start time for the `if` block that begins on line 22. if lcg.generate(10) < 0.5 // `if` statement is not independently wrapped. long _start23 = System.timeNow() // Start time for line 23. val0 *= 1.0 + (2.0 * lcg.generate(50) - 1.0) * 0.1 registerPerf(System.timeNow() - _start23, line23) // Register performance info for line 23. else if lcg.generate(10) < 0.5 // `else if` statement is not independently wrapped. long _start25 = System.timeNow() // Start time for line 25. val1 *= 1.0 + (2.0 *

lcg.generate(50) - 1.0) * 0.1 registerPerf(System.timeNow() - _start25, line25) // Register performance info for line 25. else if lcg.generate(10) < 0.5 // `else if` statement is not independently wrapped. long _start27 = System.timeNow() // Start time for line 27. val2 *= 1.0 + (2.0 * lcg.generate(50) - 1.0) * 0.1 registerPerf(System.timeNow() - _start27, line27) // Register performance info for line 27. registerPerf(System.timeNow() - _start22, line22, line28) // Register performance info for the block (line 22 - 28). long _start29 = System.timeNow() // Start time for line 29. plot(math.avg(val0, val1, val2), "Average pseudorandom result", color.purple) registerPerf(System.timeNow() - _start29, line29) // Register performance info for line 29. // <Additional internal code executes here> registerPerf(System.timeNow() - _startMain, total) // Register the script's overall performance info.

Note that:

- This example is **pseudocode** that provides a basic outline of the **internal calculations** the Profiler applies to collect performance data. Saving this example in the Pine Editor will result in a compilation error since `long`, `System.timeNow()`, and `registerPerf()` **do not** represent Pine Script code.
- These internal calculations that the Profiler wraps a script with require **additional** computational resources, which is why a script's runtime **increases** while profiling. Programmers should always interpret the results as **estimates** since they reflect a script's performance with the extra calculations included.

After running the wrapped script to collect performance data, *additional* internal calculations organize the results and display relevant information inside the Pine Editor:

The *"Line time"* calculation for code blocks also occurs at this stage, as the Profiler cannot individually wrap loop headers or the conditional statements in if or switch structures. This field's value represents the *difference* between a block's total time and the sum of its local code times, which is why the "Line time" value for a switch block or an if block with `else if` expressions represents the time spent on **all** the structure's conditional statements, not just the block's *initial line* of code. If a programmer requires more granular information for each conditional expression in such a block, they can reorganize the logic into a *nested* if structure, as explained here.

> NoteThe Profiler **cannot** collect individual performance data for any required *internal* calculations and display their results inside the Pine Editor. Consequently, the time values the Profiler displays for all code regions in a script **do not** add up to 100% of its total runtime.

Profiling across configurations

When a code's time complexity is not constant or its execution pattern varies with its inputs, function arguments, or available data, it's often wise to profile the code across *different configurations* and data feeds for a more well-rounded perspective on its general performance.

For example, this simple script uses a for loop to calculate the sum of squared distances between the current close price and `lengthInput` previous prices, then plots the square root of that sum on each bar. In this case, the `lengthInput` directly impacts the calculation's runtime since it determines the number of times the loop executes its local code:

//@version=6 indicator("Profiling across configurations demo") //@variable The number of previous bars in the calculation. Directly affects the number of loop iterations. int lengthInput = input.int(25, "Length", 1)

//@variable The sum of squared distances from the current `close` to `lengthInput` past `close` values. float total = 0.0 // Look back across `lengthInput` bars and accumulate squared distances. for i = 1 to lengthInput float distance = close - close[i] total += distance * distance // Plot the square root of the `total`. plot(math.sqrt(total))

Let's try profiling this script with different `lengthInput` values. First, we'll use the default value of 25. The Profiler's results for this specific run show that the script completed 20,685 executions in about 96.7 milliseconds:

Here, we've increased the input's value to 50 in the script's settings. The results for this run show that the script's total runtime was 194.3 milliseconds, close to *twice* the time from the previous run:

In the next run, we changed the input's value to 200. This time, the Profiler's results show that the script finished all executions in approximately 0.8 seconds, around *four times* the previous run's time:

We can see from these observations that the script's runtime appears to scale *linearly* with the `lengthInput` value, excluding other factors that may affect performance, as one might expect since the bulk of the script's calculations occur within the loop and the input's value controls how many times the loop must execute.

> TipProfiling each configuration *more than once* helps reduce the impact of outliers while assessing how a script's performance varies with its inputs or data. See the Repetitive profiling section below for more information.

Repetitive profiling

The runtime resources available to a script *vary* over time. Consequently, the time it takes to evaluate a code region, even one with constant complexity, *fluctuates* across executions, and the cumulative performance results shown by the Profiler **will vary** with each independent script run.

Users can enhance their analysis by *restarting* a script several times and profiling each independent run. Averaging the results from each profiled run and evaluating the dispersion of runtime results can help users establish more robust performance benchmarks and reduce the impact of *outliers* (abnormally long or short runtimes) in their conclusions.

Incorporating a *dummy input* (i.e., an input that does nothing) into a script's code is a simple technique that enables users to *restart* it while profiling. The input will not directly affect any calculations or outputs. However, as the user changes its value in the script's settings, the script restarts and the Profiler re-analyzes the executed code.

For example, this script queues pseudorandom values with a constant seed through an array with a fixed size, and it calculates and plots the array.avg() value on each bar. For profiling purposes, the script includes a `dummyInput` variable with an input.int() value assigned to it. The input does nothing in the code aside from allowing us to *restart* the script each time we change its value:

//@version=6 indicator("Repetitive profiling demo") //@variable An input not connected to script calculations. Changing its value in the "Inputs" tab restarts the script. int dummyInput = input.int(0, "Dummy input") //@variable An array of pseudorandom values. var array<float> randValues = array.new<float>(2500, 0.0) // Push a new `math.random()` value with a fixed `seed` into the `randValues` array and remove the oldest

value. array.push(randValues, math.random(seed = 12345)) array.shift(randValues) // Plot the average of all elements in the `randValues` array. plot(array.avg(randValues), "Pseudorandom average")

After the first script run, the Profiler shows that it took 308.6 milliseconds to execute across all of the chart's data:

Now, let's change the dummy input's value in the script's settings to restart it without changing the calculations. This time, it completed the same code executions in 424.6 milliseconds, 116 milliseconds longer than the previous run:

Restarting the script again yields another new result. On the third run, the script finished all code executions in 227.4 milliseconds, the shortest time so far:

After repeating this process several times and documenting the results from each run, one can manually calculate their *average* to estimate the script's expected total runtime:

```
AverageTime = (time1 + time2 + ... + timeN) / N
```

> NoticeWhether profiling a single script run or multiple, it’s crucial to understand that **results will vary**. Averaging results across several profiled script runs can help programmers derive more stable performance estimates. However, those estimates do not necessarily indicate how the script will perform in the future.

Optimization

Code optimization, not to be confused with indicator or strategy optimization, involves modifying a script's source code for improved execution time, resource efficiency, and scalability. Programmers may use various approaches to optimize a script when they need enhanced runtime performance, depending on what a script's calculations entail.

Fundamentally, most techniques one will use to optimize Pine code involve *reducing* the number of times critical calculations occur or *replacing* significant calculations with simplified formulas or built-ins. Both of these paradigms often overlap.

The following sections explain several straightforward concepts programmers can apply to optimize their Pine Script code.

> TipBefore looking for ways to optimize a script, profile it to gauge its performance and identify the **critical code regions** that can benefit the most from optimization.

Using built-ins

Pine Script features a variety of *built-in* functions and variables that help streamline script creation. Many of Pine's built-ins feature internal optimizations to help maximize efficiency and minimize execution time. As such, one of the simplest ways to optimize Pine code is to utilize these efficient built-ins in a script's calculations when possible.

Let's look at an example where one can replace user-defined calculations with a concise built-in call to substantially improve performance. Suppose a programmer wants to calculate the highest value of a series over a specified number of bars. Someone not familiar with all of Pine's built-ins might approach the task using a code like the following, which uses a loop on each bar to compare `length` historical values of a `source` series:

//@variable A user-defined function to calculate the highest `source` value over `length` bars. pineHighest(float source, int length) => float result = na if bar_index + 1 >= length result := source if length > 1 for i = 1 to length - 1 result := math.max(result, source[i]) result

Alternatively, one might devise a more optimized Pine function by reducing the number of times the loop executes, as iterating over the history of the `source` to achieve the result is only necessary when specific conditions occur:

//@variable A faster user-defined function to calculate the highest `source` value over `length` bars. // This version only requires a loop when the highest value is removed from the window, the `length` // changes, or when the number of bars first becomes sufficient to calculate the result. fasterPineHighest(float source, int length) => var float result = na if source[length] == result or length != length[1] or bar_index + 1 == length result := source if length > 1 for i = 1 to length - 1 result := math.max(result, source[i]) else result := math.max(result, source) result

The built-in ta.highest() function will outperform **both** of these implementations, as its internal calculations are highly optimized for efficient execution. Below, we created a script that plots the results of calling `pineHighest()`, `fasterPineHighest()`, and ta.highest() to compare their performance using the Profiler:

//@version=6 indicator("Using built-ins demo") //@variable A user-defined function to calculate the highest `source` value over `length` bars. pineHighest(float source, int length) => float result = na if bar_index + 1 >= length result := source if length > 1 for i = 1 to length - 1 result := math.max(result, source[i]) result //@variable A faster user-defined function to calculate the highest `source` value over `length` bars. // This version only requires a loop when the highest value is removed from the window, the `length` // changes, or when the number of bars first becomes sufficient to calculate the result. fasterPineHighest(float source, int length) => var float result = na if source[length] == result or length != length[1] or bar_index + 1 == length result := source if length > 1 for i = 1 to length - 1 result := math.max(result, source[i]) else result := math.max(result, source) result plot(pineHighest(close, 20)) plot(fasterPineHighest(close, 20)) plot(ta.highest(close, 20))

The profiled results over 20,735 script executions show the call to `pineHighest()` took the most time to execute, with a runtime of 57.9 milliseconds, about 69.3% of the script's total runtime. The `fasterPineHighest()` call performed much more efficiently, as it only took about 16.9 milliseconds, approximately 20.2% of the total runtime, to calculate the same values.

The most efficient *by far*, however, was the ta.highest() call, which only required 3.2 milliseconds (~3.8% of the total runtime) to execute across all the chart's data and compute the same values in this run:

While these results effectively demonstrate that the built-in function outperforms our user-defined functions with a small `length` argument of 20, it's crucial to consider that the calculations required by the functions *will vary* with the argument's value. Therefore, we can profile the code while using different arguments to gauge how its runtime scales.

Here, we changed the `length` argument in each function call from 20 to 200 and profiled the script again to observe the changes in performance. The time spent on the `pineHighest()` function in this run increased to about 0.6 seconds (~86% of the total runtime), and the time spent on the `fasterPineHighest()` function increased to about 75 milliseconds. The ta.highest() function, on the other hand, *did not* experience a substantial runtime change. It took about 5.8 milliseconds this time, only a couple of milliseconds more than the previous run.

In other words, while our user-defined functions experienced significant runtime growth with a higher `length` argument in this run, the change in the built-in ta.highest() function's runtime was relatively marginal in this case, thus further emphasizing its performance benefits:

Note that:

- In many scenarios, a script's runtime can benefit from using built-ins where applicable. However, the relative performance edge achieved from using built-ins depends on a script's *high-impact code* and the specific built-ins used. In any case, one should always profile their scripts, preferably several times, when exploring optimized solutions.
- The calculations performed by the functions in this example also depend on the sequence of the chart's data. Therefore, programmers can gain further insight into their general performance by profiling the script across different datasets as well.

Reducing repetition

The Pine Script compiler can automatically simplify some types of repetitive code without a programmer's intervention. However, this automatic process has its limitations. If a script contains repetitive calculations that the compiler *cannot* reduce, programmers can reduce the repetition *manually* to improve their script's performance.

For example, this script contains a `valuesAbove()` method that counts the number of elements in an array above the element at a specified index. The script plots the number of values above the element at the last index of a `data` array with a calculated `plotColor`. It calculates the `plotColor` within a switch structure that calls `valuesAbove()` in all 10 of its conditional expressions:

//@version=6 indicator("Reducing repetition demo") //@function Counts the number of elements in `this` array above the element at a specified `index`. method valuesAbove(array<float> this, int index) => int result = 0 float reference = this.get(index) for [i, value] in this if i == index continue if value > reference result += 1 result //@variable An array containing the most recent 100 `close` prices. var array<float> data = array.new<float>(100) data.push(close) data.shift() //@variable Returns `color.purple` with a varying transparency based on the `valuesAbove()`. color plotColor = switch data.valuesAbove(99) <= 10 => color.new(color.purple, 90) data.valuesAbove(99) <= 20 => color.new(color.purple, 80) data.valuesAbove(99) <= 30 => color.new(color.purple, 70) data.valuesAbove(99) <= 40 => color.new(color.purple, 60) data.valuesAbove(99) <= 50 => color.new(color.purple, 50) data.valuesAbove(99) <= 60 => color.new(color.purple, 40) data.valuesAbove(99) <= 70 => color.new(color.purple, 30) data.valuesAbove(99) <= 80 => color.new(color.purple, 20) data.valuesAbove(99) <= 90 => color.new(color.purple, 10) data.valuesAbove(99) <= 100 => color.new(color.purple, 0) // Plot the number values in the `data` array above the value at its last index. plot(data.valuesAbove(99), color = plotColor, style = plot.style_area)

The profiled results for this script show that it spent about 2.5 seconds executing 21,201 times. The code regions with the highest impact on the script's runtime are the for loop within the `valuesAbove()` local scope

starting on line 8 and the switch block that starts on line 21:

Notice that the number of executions shown for the local code within `valuesAbove()` is substantially *greater* than the number shown for the code in the script's global scope, as the script calls the method up to 11 times per execution, and the results for a function's local code reflect the *combined* time and executions from each separate call:

Although each `valuesAbove()` call uses the *same* arguments and returns the *same* result, the compiler cannot automatically reduce this code for us during translation. We will need to do the job ourselves. We can optimize this script by assigning the value of `data.valuesAbove(99)` to a *variable* and *reusing* the value in all other areas requiring the result.

In the version below, we modified the script by adding a `count` variable to reference the `data.valuesAbove(99)` value. The script uses this variable in the `plotColor` calculation and the plot() call:

```
//@version=6 indicator("Reducing repetition demo") //@function Counts the number of elements in `this` array above the element at a specified `index`. method valuesAbove(array<float> this, int index) => int result = 0 float reference = this.get(index) for [i, value] in this if i == index continue if value > reference result += 1 result //@variable An array containing the most recent 100 `close` prices. var array<float> data = array.new<float>(100) data.push(close) data.shift() //@variable The number values in the `data` array above the value at its last index. int count = data.valuesAbove(99) //@variable Returns `color.purple` with a varying transparency based on the `valuesAbove()`. color plotColor = switch count <= 10 => color.new(color.purple, 90) count <= 20 => color.new(color.purple, 80) count <= 30 => color.new(color.purple, 70) count <= 40 => color.new(color.purple, 60) count <= 50 => color.new(color.purple, 50) count <= 60 => color.new(color.purple, 40) count <= 70 => color.new(color.purple, 30) count <= 80 => color.new(color.purple, 20) count <= 90 => color.new(color.purple, 10) count <= 100 => color.new(color.purple, 0) // Plot the `count`. plot(count, color = plotColor, style = plot.style_area)
```

With this modification, the profiled results show a significant improvement in performance, as the script now only needs to evaluate the `valuesAbove()` call **once** per execution rather than up to 11 separate times:

Note that:

- Since this script only calls `valuesAbove()` once, the method's local code will now reflect the results from that specific call. See this section to learn more about interpreting profiled function and method call results.

Minimizing `request.*()` calls

The built-in functions in the `request.*()` namespace allow scripts to retrieve data from other contexts. While these functions provide utility in many applications, it's important to consider that each call to these functions can have a significant impact on a script's resource usage.

A single script can contain up to 40 unique calls to the `request.*()` family of functions, or up to 64 if the user has the Ultimate plan. However, we recommend programmers aim to keep their scripts' `request.*()` calls far *below* this limit to keep the performance impact of their data requests as low as possible.

When a script requests the values of several expressions from the *same* context with multiple request.security() or request.security_lower_tf() calls, one effective way to optimize such requests is to *condense* them into a single `request.*()` call that uses a tuple as its `expression` argument. This optimization

not only helps improve the runtime of the requests; it also helps reduce the script's *memory usage* and compiled size.

As a simple example, the following script requests nine ta.percentrank() values with different lengths from a specified symbol using nine separate calls to request.security(). It then plots all nine requested values on the chart to utilize them in the outputs:

//@version=6 indicator("Minimizing `request.*()` calls demo") //@variable The symbol to request data from. string symbolInput = input.symbol("BINANCE:BTCUSDT", "Symbol") // Request 9 `ta.percentrank()` values from the `symbolInput` context using 9 `request.security()` calls. float reqRank1 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 10)) float reqRank2 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 20)) float reqRank3 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 30)) float reqRank4 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 40)) float reqRank5 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 50)) float reqRank6 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 60)) float reqRank7 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 70)) float reqRank8 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 80)) float reqRank9 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 90)) // Plot the `reqRank*` values. plot(reqRank1) plot(reqRank2) plot(reqRank3) plot(reqRank4) plot(reqRank5) plot(reqRank6) plot(reqRank7) plot(reqRank8) plot(reqRank9)

The results from profiling the script show that it took the script 340.8 milliseconds to complete its requests and plot the values in this run:

Since all the request.security() calls request data from the **same context**, we can optimize the code's resource usage by merging all of them into a single request.security() call that uses a tuple as its `expression` argument:

//@version=6 indicator("Minimizing `request.*()` calls demo") //@variable The symbol to request data from. string symbolInput = input.symbol("BINANCE:BTCUSDT", "Symbol") // Request 9 `ta.percentrank()` values from the `symbolInput` context using a single `request.security()` call. [reqRank1, reqRank2, reqRank3, reqRank4, reqRank5, reqRank6, reqRank7, reqRank8, reqRank9] = request.security(symbolInput, timeframe.period, [ta.percentrank(close, 10), ta.percentrank(close, 20), ta.percentrank(close, 30), ta.percentrank(close, 40), ta.percentrank(close, 50), ta.percentrank(close, 60), ta.percentrank(close, 70), ta.percentrank(close, 80), ta.percentrank(close, 90)]) // Plot the `reqRank*` values. plot(reqRank1) plot(reqRank2) plot(reqRank3) plot(reqRank4) plot(reqRank5) plot(reqRank6) plot(reqRank7) plot(reqRank8) plot(reqRank9)

As we see below, the profiled results from running this version of the script show that it took 228.3 milliseconds this time, a decent improvement over the previous run:

Note that:

- The computational resources available to a script **fluctuate** over time. As such, it's typically a good idea to profile a script multiple times to help solidify performance conclusions.
- Another way to request multiple values from the same context with a single `request.*()` call is to pass an object of a user-defined type (UDT) as the `expression` argument. See this section of the Other timeframes and data page to learn more about requesting UDTs.

- Programmers can also reduce the total runtime of a request.security(), request.security_lower_tf(), or request.seed() call by passing an argument to the function's `calc_bars_count` parameter, which *restricts* the number of *historical* data points it can access from a context and execute required calculations on. In general, if calls to these `request.*()` functions retrieve *more* historical data than what a script *needs*, limiting the requests with `calc_bars_count` can help improve the script's performance.

Avoiding redrawing

Pine Script's drawing types allow scripts to draw custom visuals on a chart that one cannot achieve through other outputs such as plots. While these types provide greater visual flexibility, they also have a *higher* runtime and memory cost, especially when a script unnecessarily *recreates* drawings instead of directly updating their properties to change their appearance.

Most drawing types, excluding polylines, feature built-in *setter functions* in their namespaces that allow scripts to modify a drawing *without* deleting and recreating it. Utilizing these setters is typically less computationally expensive than creating a new drawing object when only *specific properties* require modification.

For example, the script below compares deleting and redrawing boxes to using `box.set*()` functions. On the first bar, it declares the `redrawnBoxes` and `updatedBoxes` arrays and executes a loop to push 25 box elements into them.

The script uses a separate for loop to iterate across the arrays and update the drawings on each execution. It *recreates* the boxes in the `redrawnBoxes` array using box.delete() and box.new(), whereas it *directly modifies* the properties of the boxes in the `updatedBoxes` array using box.set_lefttop() and box.set_rightbottom(). Both approaches achieve the same visual result. However, the latter is more efficient:

//@version=6 indicator("Avoiding redrawing demo") //@variable An array of `box` IDs deleted with `box.delete()` and redrawn with `box.new()` on each execution. var array<box> redrawnBoxes = array.new<box>() //@variable An array of `box` IDs with properties that update across executions update via `box.set*()` functions. var array<box> updatedBoxes = array.new<box>() // Populate both arrays with 25 elements on the first bar. if barstate.isfirst for i = 1 to 25 array.push(redrawnBoxes, box(na)) array.push(updatedBoxes, box.new(na, na, na, na)) for i = 0 to 24 // Calculate coordinates. int x = bar_index - i float y = close[i + 1] - close // Get the `box` ID from each array at the `i` index. box redrawnBox = redrawnBoxes.get(i) box updatedBox = updatedBoxes.get(i) // Delete the `redrawnBox`, create a new `box` ID, and replace that element in the `redrawnboxes` array. box.delete(redrawnBox) redrawnBox := box.new(x - 1, y, x, 0.0) array.set(redrawnBoxes, i, redrawnBox) // Update the properties of the `updatedBox` rather than redrawing it. box.set_lefttop(updatedBox, x - 1, y) box.set_rightbottom(updatedBox, x, 0.0)

The results from profiling this script show that line 24, which contains the box.new() call, is the *heaviest* line in the code block that executes on each bar, with a runtime close to **double** the combined time spent on the box.set_lefttop() and box.set_rightbottom() calls on lines 27 and 28:

Note that:

- The number of executions shown for the loop's *local code* is 25 times the number shown for the code in the script's *global scope*, as each execution of the loop statement triggers 25 executions of the local block.

- This script updates its drawings over *all bars* in the chart's history for **testing** purposes. However, it does **not** actually need to execute all these historical updates since users will only see the **final** result from the *last historical bar* and the changes across *realtime bars*. See the next section to learn more.

Reducing drawing updates

When a script produces drawing objects that change across *historical bars*, users will only ever see their **final results** on those bars since the script completes its historical executions when it first loads on the chart. The only time one will see such drawings *evolve* across executions is during *realtime bars*, as new data flows in.

Since the evolving outputs from dynamic drawings on historical bars are **never visible** to a user, one can often improve a script's performance by *eliminating* the historical updates that don't impact the final results.

For example, this script creates a table with two columns and 21 rows to visualize the history of an RSI in a paginated, tabular format. The script initializes the cells of the `infoTable` on the first bar, and it references the history of the calculated `rsi` to update the `text` and `bgcolor` of the cells in the second column within a for loop on each bar:

//@version=6 indicator("Reducing drawing updates demo") //@variable The first offset shown in the paginated table. int offsetInput = input.int(0, "Page", 0, 249) * 20 //@variable A table that shows the history of RSI values. var table infoTable = table.new(position.top_right, 2, 21, border_color = chart.fg_color, border_width = 1) // Initialize the table's cells on the first bar. if barstate.isfirst table.cell(infoTable, 0, 0, "Offset", text_color = chart.fg_color) table.cell(infoTable, 1, 0, "RSI", text_color = chart.fg_color) for i = 0 to 19 table.cell(infoTable, 0, i + 1, str.tostring(offsetInput + i)) table.cell(infoTable, 1, i + 1) float rsi = ta.rsi(close, 14) // Update the history shown in the `infoTable` on each bar. for i = 0 to 19 float historicalRSI = rsi[offsetInput + i] table.cell_set_text(infoTable, 1, i + 1, str.tostring(historicalRSI)) table.cell_set_bgcolor(infoTable, 1, i + 1, color.from_gradient(historicalRSI, 30, 70, color.red, color.green)) plot(rsi, "RSI")

After profiling the script, we see that the code with the highest impact on performance is the for loop that starts on line 20, i.e., the code block that updates the table's cells:

This critical code region executes **excessively** across the chart's history, as users will only see the table's **final** historical result. The only time that users will see the table update is on the **last historical bar** and across all subsequent **realtime bars**. Therefore, we can optimize this script's resource usage by restricting the executions of this code to only the last available bar.

In this script version, we placed the loop that updates the table cells within an if structure that uses barstate.islast as its condition, effectively restricting the code block's executions to only the last historical bar and all realtime bars. Now, the script *loads* more efficiently since all the table's calculations only require **one** historical execution:

//@version=6 indicator("Reducing drawing updates demo") //@variable The first offset shown in the paginated table. int offsetInput = input.int(0, "Page", 0, 249) * 20 //@variable A table that shows the history of RSI values. var table infoTable = table.new(position.top_right, 2, 21, border_color = chart.fg_color, border_width = 1) // Initialize the table's cells on the first bar. if barstate.isfirst table.cell(infoTable, 0, 0, "Offset", text_color = chart.fg_color) table.cell(infoTable, 1, 0, "RSI", text_color = chart.fg_color) for i = 0 to 19 table.cell(infoTable, 0, i + 1, str.tostring(offsetInput + i)) table.cell(infoTable, 1, i + 1) float rsi = ta.rsi(close, 14) // Update the history shown in the `infoTable` on the last available bar. if barstate.islast for i = 0 to 19 float historicalRSI = rsi[offsetInput + i] table.cell_set_text(infoTable, 1, i + 1, str.tostring(historicalRSI))

table.cell_set_bgcolor(infoTable, 1, i + 1, color.from_gradient(historicalRSI, 30, 70, color.red, color.green)) plot(rsi, "RSI")

Note that:

- The script will still update the cells when new **realtime** updates come in, as users can observe those changes on the chart, unlike the changes that the script used to execute across historical bars.

Storing calculated values

When a script performs a critical calculation that changes *infrequently* throughout all executions, one can reduce its runtime by **saving the result** to a variable declared with the var or varip keywords and **only** updating the value if the calculation changes. If the script calculates *multiple* values excessively, one can store them within collections, matrices, and maps or objects of user-defined types.

Let's look at an example. This script calculates a weighted moving average with custom weights based on a generalized window function. The `numerator` is the sum of weighted close values, and the `denominator` is the sum of the calculated weights. The script uses a for loop that iterates `lengthInput` times to calculate these sums, then it plots their ratio, i.e., the resulting average:

//@version=6 indicator("Storing calculated values demo", overlay = true) //@variable The number of bars in the weighted average calculation. int lengthInput = input.int(50, "Length", 1, 5000) //@variable Window coefficient. float coefInput = input.float(0.5, "Window coefficient", 0.0, 1.0, 0.01) //@variable The sum of weighted `close` prices. float numerator = 0.0 //@variable The sum of weights. float denominator = 0.0 //@variable The angular step in the cosine calculation. float step = 2.0 * math.pi / lengthInput // Accumulate weighted sums. for i = 0 to lengthInput - 1 float weight = coefInput - (1 - coefInput) * math.cos(step * i) numerator += close[i] * weight denominator += weight // Plot the weighted average result. plot(numerator / denominator, "Weighted average", color.purple, 3)

After profiling the script's performance over our chart's data, we see that it took about 241.3 milliseconds to calculate the default 50-bar average across 20,155 chart updates, and the critical code with the *highest impact* on the script's performance is the loop block that starts on line 17:

Since the number of loop iterations *depends* on the `lengthInput` value, let's test how its runtime scales with another configuration requiring heavier looping. Here, we set the value to 2500. This time, the script took about 12 seconds to complete all of its executions:

Now that we've pinpointed the script's *high-impact* code and established a benchmark to improve, we can inspect the critical code block to identify optimization opportunities. After examining the calculations, we can observe the following:

- The only value that causes the `weight` calculation on line 18 to vary across loop iterations is the *loop index*. All other values in its calculation remain consistent. Consequently, the `weight` calculated on each loop iteration **does not vary** across chart bars. Therefore, rather than calculating the weights on **every update**, we can calculate them **once**, on the first bar, and **store them** in a collection for future access across subsequent script executions.
- Since the weights never change, the resulting `denominator` never changes. Therefore, we can add the var keyword to the variable declaration and only calculate its value **once** to reduce the number of executed addition assignment (+=) operations.

- Unlike the `denominator`, we **cannot** store the `numerator` value to simplify its calculation since it consistently *changes* over time.

In the modified script below, we've added a `weights` variable to reference an array that stores each calculated `weight`. This variable and the `denominator` both include the var keyword in their declarations, meaning the values assigned to them will *persist* throughout all script executions until explicitly reassigned. The script calculates their values using a for loop that executes only on the first chart bar. Across all other bars, it calculates the `numerator` using a for…in loop that references the *saved values* from the `weights` array:

//@version=6 indicator("Storing calculated values demo", overlay = true) //@variable The number of bars in the weighted average calculation. int lengthInput = input.int(50, "Length", 1, 5000) //@variable Window coefficient. float coefInput = input.float(0.5, "Window coefficient", 0.0, 1.0, 0.01) //@variable An array that stores the `weight` values calculated on the first chart bar. var array<float> weights = array.new<float>() //@variable The sum of weighted `close` prices. float numerator = 0.0 //@variable The sum of weights. The script now only calculates this value on the first bar. var float denominator = 0.0 //@variable The angular step in the cosine calculation. float step = 2.0 * math.pi / lengthInput // Populate the `weights` array and calculate the `denominator` only on the first bar. if barstate.isfirst for i = 0 to lengthInput - 1 float weight = coefInput - (1 - coefInput) * math.cos(step * i) array.push(weights, weight) denominator += weight // Calculate the `numerator` on each bar using the stored `weights`. for [i, w] in weights numerator += close[i] * w // Plot the weighted average result. plot(numerator / denominator, "Weighted average", color.purple, 3)

With this optimized structure, the profiled results show that our modified script with a high `lengthInput` value of 2500 took about 5.9 seconds to calculate across the same data, about *half* the time of our previous version:

Note that:

- Although we've significantly improved this script's performance by saving its *execution-invariant* values to variables, it does still involve a higher computational cost with **large** `lengthInput` values due to the remaining loop calculations that execute on each bar.
- Another, more *advanced* way one can further enhance this script's performance is by storing the weights in a *single-row* matrix on the first bar, using an array as a queue to hold recent close values, then replacing the for…in loop with a call to matrix.mult(). See the Matrices page to learn more about working with `matrix.*()` functions.

Eliminating loops

Loops allow Pine scripts to perform *iterative* calculations on each execution. Each time a loop activates, its local code may execute *several times*, often leading to a *substantial increase* in resource usage.

Pine loops are necessary for *some* calculations, such as manipulating elements within collections or looking backward through a dataset's history to calculate values *only* obtainable on the current bar. However, in many other cases, programmers use loops when they **don't need to**, leading to suboptimal runtime performance. In such cases, one may eliminate unnecessary loops in any of the following ways, depending on what their calculations entail:

- Identifying simplified, **loop-free expressions** that achieve the same result without iteration
- Replacing a loop with optimized built-ins where possible
- Distributing a loop's iterations *across bars* when feasible rather than evaluating them all at once

This simple example contains an `avgDifference()` function that calculates the average difference between the current bar's `source` value and all the values from `length` previous bars. The script calls this function to calculate the average difference between the current close price and `lengthInput` previous prices, then it plots the result on the chart:

//@version=6 indicator("Eliminating loops demo") //@variable The number of bars in the calculation. int lengthInput = input.int(20, "Length", 1) //@function Calculates the average difference between the current `source` and `length` previous `source` values. avgDifference(float source, int length) => float diffSum = 0.0 for i = 1 to length diffSum += source - source[i] diffSum / length plot(avgDifference(close, lengthInput))

After inspecting the script's profiled results with the default settings, we see that it took about 64 milliseconds to execute 20,157 times:

Since we use the `lengthInput` as the `length` argument in the `avgDifference()` call and that argument controls how many times the loop inside the function must iterate, our script's runtime will **grow** with the `lengthInput` value. Here, we set the input's value to 2000 in the script's settings. This time, the script completed its executions in about 3.8 seconds:

As we see from these results, the `avgDifference()` function can be costly to call, depending on the specified `lengthInput` value, due to its for loop that executes on each bar. However, loops are **not** necessary to achieve the output. To understand why, let's take a closer look at the loop's calculations. We can represent them with the following expression:

(source - source[1]) + (source - source[2]) + ... + (source - source[length])

Notice that it adds the *current* `source` value `length` times. These iterative additions are not necessary. We can simplify that part of the expression to `source * length`, which reduces it to the following:

source * length - source[1] - source[2] - ... - source[length]

or equivalently:

source * length - (source[1] + source[2] + ... + source[length])

After simplifying and rearranging this representation of the loop's calculations, we see that we can compute the result in a simpler way and **eliminate** the loop by subtracting the previous bar's rolling sum (math.sum()) of `source` values from the `source * length` value, i.e.:

source * length - math.sum(source, length)[1]

The `fastAvgDifference()` function below is a **loop-free** alternative to the original `avgDifference()` function that uses the above expression to calculate the sum of `source` differences, then divides the expression by the `length` to return the average difference:

//@function A faster way to calculate the `avgDifference()` result. // Eliminates the `for` loop using the relationship: // `(x - x[1]) + (x - x[2]) + ... + (x - x[n]) = x * n - math.sum(x, n)[1]`. fastAvgDifference(float source, int length) => (source * length - math.sum(source, length)[1]) / length

Now that we've identified a potential optimized solution, we can compare the performance of `fastAvgDifference()` to the original `avgDifference()` function. The script below is a modified form of the previous version that plots the results from calling both functions with the `lengthInput` as the `length` argument:

//@version=6 indicator("Eliminating loops demo") //@variable The number of bars in the calculation. int lengthInput = input.int(20, "Length", 1) //@function Calculates the average difference between the current `source` and `length` previous `source` values. avgDifference(float source, int length) => float diffSum = 0.0 for i = 1 to length diffSum += source - source[i] diffSum / length //@function A faster way to calculate the `avgDifference()` result. // Eliminates the `for` loop using the relationship: // `(x - x[1]) + (x - x[2]) + ... + (x - x[n]) = x * n - math.sum(x, n)[1]`. fastAvgDifference(float source, int length) => (source * length - math.sum(source, length)[1]) / length plot(avgDifference(close, lengthInput)) plot(fastAvgDifference(close, lengthInput))

The profiled results for the script with the default `lengthInput` of 20 show a substantial difference in runtime spent on the two function calls. The call to the original function took about 47.3 milliseconds to execute 20,157 times on this run, whereas our optimized function only took 4.5 milliseconds:

Now, let's compare the performance with the *heavier* `lengthInput` value of 2000. As before, the runtime spent on the `avgDifference()` function increased significantly. However, the time spent executing the `fastAvgDifference()` call remained very close to the result from the previous configuration. In other words, while our original function's runtime scales directly with its `length` argument, our optimized function demonstrates relatively *consistent* performance since it does not require a loop:

> NoteNot all iterative calculations have loop-free alternatives. If the **only** way to achieve a calculation is through iteration, programmers can still aim to identify ways to optimize their loops for improved performance. See the Optimizing loops section below for more information.

Optimizing loops

Although Pine's execution model and the available built-ins often *eliminate* the need for loops in many cases, there are still instances where a script **will** require loops for some types of tasks, including:

- Manipulating collections or executing calculations over a collection's elements when the available built-ins **will not** suffice
- Performing calculations across historical bars that one **cannot** achieve with simplified *loop-free* expressions or optimized *built-ins*
- Calculating values that are **only** obtainable through iteration

When a script uses loops that a programmer cannot eliminate, there are several techniques one can use to reduce their performance impact. This section explains two of the most common, useful techniques that can help improve a required loop's efficiency.

> TipBefore identifying ways to *optimize* a loop, we recommend searching for ways to eliminate it first. If **no solution** exists that makes the loop unnecessary, then proceed with attempting to reduce its overhead.

Reducing loop calculations

The code executed within a loop's local scope can have a **multiplicative** impact on its overall runtime, as each time a loop statement executes, it will typically trigger *several* iterations of the local code. Therefore, programmers should strive to keep a loop's calculations as simple as possible by eliminating unnecessary

structures, function calls, and operations to minimize the performance impact, especially when the script must evaluate its loops *numerous times* throughout all its executions.

For example, this script contains a `filteredMA()` function that calculates a moving average of up to `length` unique `source` values, depending on the `true` elements in a specified `mask` array. The function queues the unique `source` values into a `data` array, uses a for…in loop to iterate over the `data` and calculate the `numerator` and `denominator` sums, then returns the ratio of those sums. Within the loop, it only adds values to the sums when the `data` element is not na and the `mask` element at the `index` is `true`. The script utilizes this user-defined function to calculate the average of up to 100 unique close prices filtered by a `randMask` and plots the result on the chart:

//@version=6 indicator("Reducing loop calculations demo", overlay = true) //@function Calculates a moving average of up to `length` unique `source` values filtered by a `mask` array. filteredMA(float source, int length, array<bool> mask) => // Raise a runtime error if the size of the `mask` doesn't equal the `length`. if mask.size() != length runtime.error("The size of the `mask` array used in the `filteredMA()` call must match the `length`.") //@variable An array containing `length` unique `source` values. var array<float> data = array.new<float>(length) // Queue unique `source` values into the `data` array. if not data.includes(source) data.push(source) data.shift() // The numerator and denominator of the average. float numerator = 0.0 float denominator = 0.0 // Loop to calculate sums. for item in data if na(item) continue int index = array.indexof(data, item) if mask.get(index) numerator += item denominator += 1.0 // Return the average, or the last non-`na` average value if the current value is `na`. fixnan(numerator / denominator) //@variable An array of 100 pseudorandom "bool" values. var array<bool> randMask = array.new<bool>(100, true) // Push the first element from `randMask` to the end and queue a new pseudorandom value. randMask.push(randMask.shift()) randMask.push(math.random(seed = 12345) < 0.5) randMask.shift() // Plot the `filteredMA()` of up to 100 unique `close` values filtered by the `randMask`. plot(filteredMA(close, 100, randMask))

After profiling the script, we see it took about two seconds to execute 21,778 times. The code with the highest performance impact is the expression on line 37, which calls the `filteredMA()` function. Within the `filteredMA()` function's scope, the for…in loop has the highest impact, with the `index` calculation in the loop's scope (line 22) contributing the most to the loop's runtime:

The above code demonstrates suboptimal usage of a for…in loop, as we **do not** need to call array.indexof() to retrieve the `index` in this case. The array.indexof() function can be *costly* to call within a loop since it must search through the array's contents and locate the corresponding element's index *each time* the script calls it.

To eliminate this costly call from our for…in loop, we can use the *second form* of the structure, which produces a *tuple* containing the **index** and the element's value on each iteration:

for [index, item] in data

In this version of the script, we removed the array.indexof() call on line 22 since it is **not** necessary to achieve the intended result, and we changed the for…in loop to use the alternative form:

//@version=6 indicator("Reducing loop calculations demo", overlay = true) //@function Calculates a moving average of up to `length` unique `source` values filtered by a `mask` array. filteredMA(float source, int length, array<bool> mask) => // Raise a runtime error if the size of the `mask` doesn't equal the `length`. if mask.size() != length runtime.error("The size of the `mask` array used in the `filteredMA()` call must match the `length`.") //@variable An array containing `length` unique `source` values. var array<float> data =

array.new<float>(length) // Queue unique `source` values into the `data` array. if not data.includes(source) data.push(source) data.shift() // The numerator and denominator of the average. float numerator = 0.0 float denominator = 0.0 // Loop to calculate sums. for [index, item] in data if na(item) continue if mask.get(index) numerator += item denominator += 1.0 // Return the average, or the last non-`na` average value if the current value is `na`. fixnan(numerator / denominator) //@variable An array of 100 pseudorandom "bool" values. var array<bool> randMask = array.new<bool>(100, true) // Push the first element from `randMask` to the end and queue a new pseudorandom value. randMask.push(randMask.shift()) randMask.push(math.random(seed = 12345) < 0.5) randMask.shift() // Plot the `filteredMA()` of up to 100 unique `close` values filtered by the `randMask`. plot(filteredMA(close, 100, randMask))

With this simple change, our loop is much more efficient, as it no longer needs to redundantly search through the array on each iteration to keep track of the index. The profiled results from this script run show that it took only 0.6 seconds to complete its executions, a significant improvement over the previous version's result:

Loop-invariant code motion

Loop-invariant code is any code region within a loop's scope that produces an **unchanging** result on each iteration. When a script's loops contain loop-invariant code, it can substantially impact performance in some cases due to excessive, **unnecessary** calculations.

Programmers can optimize a loop with invariant code by *moving* the unchanging calculations **outside** the loop's scope so the script only needs to evaluate them once per execution rather than repetitively.

The following example contains a `featureScale()` function that creates a rescaled version of an array. Within the function's for…in loop, it scales each element by calculating its distance from the array.min() and dividing the value by the array.range(). The script uses this function to create a `rescaled` version of a `prices` array, then plots the difference between the array's array.first() and array.avg() method call results on the chart:

//@version=6 indicator("Loop-invariant code motion demo") //@function Returns a feature scaled version of `this` array. featureScale(array<float> this) => array<float> result = array.new<float>() for item in this result.push((item - array.min(this)) / array.range(this)) result //@variable An array containing the most recent 100 `close` prices. var array<float> prices = array.new<float>(100, close) // Queue the `close` through the `prices` array. prices.unshift(close) prices.pop() //@variable A feature scaled version of the `prices` array. array<float> rescaled = featureScale(prices) // Plot the difference between the first element and the average value in the `rescaled` array. plot(rescaled.first() - rescaled.avg())

As we see below, the profiled results for this script after 20,187 executions show it completed its run in about 3.3 seconds. The code with the highest impact on performance is the line containing the `featureScale()` function call, and the function's critical code is the for…in loop block starting on line 7:

Upon examining the loop's calculations, we can see that the array.min() and array.range() calls on line 8 are **loop-invariant**, as they will always produce the **same result** across each iteration. We can make our loop much more efficient by assigning the results from these calls to variables **outside** its scope and referencing them as needed.

The `featureScale()` function in the script below assigns the array.min() and array.range() values to `minValue` and `rangeValue` variables *before* executing the for…in loop. Inside the loop's local scope, it *references* the variables across its iterations rather than repetitively calling these `array.*()` functions:

//@version=6 indicator("Loop-invariant code motion demo") //@function Returns a feature scaled version of `this` array. featureScale(array<float> this) => array<float> result = array.new<float>() float minValue = array.min(this) float rangeValue = array.range(this) for item in this result.push((item - minValue) / rangeValue) result //@variable An array containing the most recent 100 `close` prices. var array<float> prices = array.new<float>(100, close) // Queue the `close` through the `prices` array. prices.unshift(close) prices.pop() //@variable A feature scaled version of the `prices` array. array<float> rescaled = featureScale(prices) // Plot the difference between the first element and the average value in the `rescaled` array. plot(rescaled.first() - rescaled.avg())

As we see from the script's profiled results, moving the *loop-invariant* calculations outside the loop leads to a substantial performance improvement. This time, the script completed its executions in only 289.3 milliseconds:

Minimizing historical buffer calculations

Pine scripts create *historical buffers* for all variables and function calls their outputs depend on. Each buffer contains information about the range of historical values the script can access with the history-referencing operator [].

A script *automatically* determines the required buffer size for all its variables and function calls by analyzing the historical references executed during the **first 244 bars** in a dataset. When a script only references the history of a calculated value *after* those initial bars, it will **restart** its executions repetitively across previous bars with successively larger historical buffers until it either determines the appropriate size or raises a runtime error. Those repetitive executions can significantly increase a script's runtime in some cases.

When a script *excessively* executes across a dataset to calculate historical buffers, one effective way to improve its performance is *explicitly* defining suitable buffer sizes using the max_bars_back() function. With appropriate buffer sizes declared explicitly, the script does not need to re-execute across past data to determine the sizes.

For example, the script below uses a polyline to draw a basic histogram representing the distribution of calculated `source` values over 500 bars. On the last available bar, the script uses a for loop to look back through historical values of the calculated `source` series and determine the chart points used by the polyline drawing. It also plots the value of `bar_index + 1` to verify the number of bars it executed across:

//@version=6 indicator("Minimizing historical buffer calculations demo", overlay = true) //@variable A polyline with points that form a histogram of `source` values. var polyline display = na //@variable The difference Q3 of `high` prices and Q1 of `low` prices over 500 bars. float innerRange = ta.percentile_nearest_rank(high, 500, 75) - ta.percentile_nearest_rank(low, 500, 25) // Calculate the highest and lowest prices, and the total price range, over 500 bars. float highest = ta.highest(500) float lowest = ta.lowest(500) float totalRange = highest - lowest //@variable The source series for histogram calculation. Its value is the midpoint between the `open` and `close`. float source = math.avg(open, close) if barstate.islast polyline.delete(display) // Calculate the number of histogram bins and their size. int bins = int(math.round(5 * totalRange / innerRange)) float binSize = totalRange / bins //@variable An array of chart points for the polyline. array<chart.point> points = array.new<chart.point>(bins, chart.point.new(na, na, na)) // Loop to build the histogram. for i = 0 to 499 //@variable The histogram bin number. Uses past values of the `source` for its calculation. // The script must execute across all previous bars AGAIN to determine the historical buffer for // `source`, as initial references to the calculated series occur AFTER the first 244 bars. int index = int((source[i] - lowest) / binSize) if na(index) continue chart.point currentPoint = points.get(index) if

na(currentPoint.index) points.set(index, chart.point.from_index(bar_index + 1, (index + 0.5) * binSize + lowest)) continue currentPoint.index += 1 // Add final points to the `points` array and draw the new `display` polyline. points.unshift(chart.point.now(lowest)) points.push(chart.point.now(highest)) display := polyline.new(points, closed = true) plot(bar_index + 1, "Number of bars", display = display.data_window)

Since the script *only* references past `source` values on the *last bar*, it will **not** construct a suitable historical buffer for the series within the first 244 bars on a larger dataset. Consequently, it will **re-execute** across all historical bars to identify the appropriate buffer size.

As we see from the profiled results after running the script across 20,320 bars, the number of *global* code executions was 162,560, which is **eight times** the number of chart bars. In other words, the script had to *repeat* the historical executions **seven more times** to determine the appropriate buffer for the `source` series in this case:

This script will only reference the most recent 500 `source` values on the last historical bar and all realtime bars. Therefore, we can help it establish the correct buffer *without* re-execution by defining a 500-bar referencing length with max_bars_back().

In the following script version, we added max_bars_back(source, 500) after the variable declaration to explicitly specify that the script will access up to 500 historical `source` values throughout its executions:

//@version=6 indicator("Minimizing historical buffer calculations demo", overlay = true) //@variable A polyline with points that form a histogram of `source` values. var polyline display = na //@variable The difference Q3 of `high` prices and Q1 of `low` prices over 500 bars. float innerRange = ta.percentile_nearest_rank(high, 500, 75) - ta.percentile_nearest_rank(low, 500, 25) // Calculate the highest and lowest prices, and the total price range, over 500 bars. float highest = ta.highest(500) float lowest = ta.lowest(500) float totalRange = highest - lowest //@variable The source series for histogram calculation. Its value is the midpoint between the `open` and `close`. float source = math.avg(open, close) // Explicitly define a 500-bar historical buffer for the `source` to prevent recalculation. max_bars_back(source, 500) if barstate.islast polyline.delete(display) // Calculate the number of histogram bins and their size. int bins = int(math.round(5 * totalRange / innerRange)) float binSize = totalRange / bins //@variable An array of chart points for the polyline. array<chart.point> points = array.new<chart.point>(bins, chart.point.new(na, na, na)) // Loop to build the histogram. for i = 0 to 499 //@variable The histogram bin number. Uses past values of the `source` for its calculation. // Since the `source` now has an appropriate predefined buffer, the script no longer needs // to recalculate across previous bars to determine the referencing length. int index = int((source[i] - lowest) / binSize) if na(index) continue chart.point currentPoint = points.get(index) if na(currentPoint.index) points.set(index, chart.point.from_index(bar_index + 1, (index + 0.5) * binSize + lowest)) continue currentPoint.index += 1 // Add final points to the `points` array and draw the new `display` polyline. points.unshift(chart.point.now(lowest)) points.push(chart.point.now(highest)) display := polyline.new(points, closed = true) plot(bar_index + 1, "Number of bars", display = display.data_window)

With this change, our script no longer needs to re-execute across all the historical data to determine the buffer size. As we see in the profiled results below, the number of global code executions now aligns with the number of chart bars, and the script took substantially less time to complete all of its historical executions:

Note that:

- This script only requires up to the most recent 501 historical bars to calculate its drawing output. In this case, another way to optimize resource usage is to include `calc_bars_count = 501` in the

indicator() function, which reduces unnecessary script executions by restricting the historical data the script can calculate across to 501 bars.

> Notice
>
> When using max_bars_back() to explicitly define the buffer size for a series, ensure that the script **does not** reference more past bars than specified during its executions. If the specified buffer size is insufficient, the runtime system still re-executes the script across historical bars to calculate an appropriate size, leading to increased resource use.
>
> Additionally, it's crucial to understand that large buffers elevate a script's *memory use*. Choosing buffer sizes that are larger than what a script needs is a suboptimal practice that yields no benefit. In some cases, excessively large buffers can cause a script to exceed its memory limits. Therefore, when defining a buffer's size, choose the **smallest** possible size that accommodates the script's historical references. For example, if a script requires only 500 past values from a series, set the buffer's size to 500 bars. Setting the buffer to include 5000 bars in such a case causes the script to use significantly more memory than necessary.

Tips

Working around Profiler overhead

Since the Pine Profiler must perform *extra calculations* to collect performance data, as explained in this section, the time it takes to execute a script **increases** while profiling.

Most scripts will run as expected with the Profiler's overhead included. However, when a complex script's runtime approaches a plan's limit, using the Profiler on it may cause its runtime to *exceed* the limit. Such a case indicates that the script likely needs optimization, but it can be challenging to know where to start without being able to profile the code. The most effective workaround in this scenario is reducing the number of bars the script must execute on. Users can achieve this reduction in any of the following ways:

- Selecting a dataset that has fewer data points in its history, e.g., a higher timeframe or a symbol with limited data
- Using conditional logic to limit code executions to a specific time or bar range
- Including a `calc_bars_count` argument in the script's declaration statement to specify how many recent historical bars it can use

Reducing the number of data points works in most cases because it directly decreases the number of times the script must execute, typically resulting in less accumulated runtime.

As a demonstration, this script contains a `gcd()` function that uses a *naive* algorithm to calculate the greatest common divisor of two integers. The function initializes its `result` using the smallest absolute value of the two numbers. Then, it reduces the value of the `result` by one within a while loop until it can divide both numbers without remainders. This structure entails that the loop will iterate up to *N* times, where *N* is the smallest of the two arguments.

In this example, the script plots the value of `gcd(10000, 10000 + bar_index)`. The smallest of the two arguments is always 10,000 in this case, meaning the while loop within the function will require up to 10,000 iterations per script execution, depending on the bar_index value:

//@version=6 indicator("Script takes too long while profiling demo") //@function Calculates the greatest common divisor of `a` and `b` using a naive algorithm. gcd(int a, int b) => //@variable The greatest common divisor. int result = math.max(math.min(math.abs(a), math.abs(b)), 1) // Reduce the `result` by 1 until it divides `a` and `b` without remainders. while result > 0 if a % result == 0 and b % result == 0 break result -= 1 // Return the `result`. result plot(gcd(10000, 10000 + bar_index), "GCD")

When we add the script to our chart, it takes a while to execute across our chart's data, but it does not raise an error. However, *after* enabling the Profiler, the script raises a runtime error stating that it exceeded the Premium plan's runtime limit (40 seconds):

Our current chart has over 20,000 historical bars, which may be too many for the script to handle within the alloted time while the Profiler is active. We can try limiting the number of historical executions to work around the issue in this case.

Below, we included `calc_bars_count = 10000` in the indicator() function, which limits the script's available history to the most recent 10,000 historical bars. After restricting the script's historical executions, it no longer exceeds the Premium plan's limit while profiling, so we can now inspect its performance results:

//@version=6 indicator("Script takes too long while profiling demo", calc_bars_count = 10000) //@function Calculates the greatest common divisor of `a` and `b` using a naive algorithm. gcd(int a, int b) => //@variable The greatest common divisor. int result = math.max(math.min(math.abs(a), math.abs(b)), 1) // Reduce the `result` by 1 until it divides `a` and `b` without remainders. while result > 0 if a % result == 0 and b % result == 0 break result -= 1 // Return the `result`. result plot(gcd(10000, 10000 + bar_index), "GCD")

> TipThis process might require trial and error, because identifying the number of executions that a computationally heavy script can handle before timing out is not necessarily straightforward. If a script takes too long to execute after enabling the Profiler, experiment with different ways to limit its executions until you can profile it successfully.

Publishing scripts

Introduction

TradingView hosts a large global community of Pine Script® programmers, and millions of traders. Script authors can publish their custom indicator scripts, strategies, and libraries publicly in the Community scripts repository, allowing others in our community to use and learn from them. They can also publish *private* scripts to create *drafts* for public releases, test features, or collaborate with friends.

This page explains the script publishing process and provides recommendations to help authors publish their Pine scripts effectively.

> NoticeBefore you publish a script, ensure you read and understand our House Rules, Script Publishing Rules, and Vendor Requirements.

Script publications

When an *editable* script is on the chart and opened in the Pine Editor, users can select the “Publish indicator/strategy/library” button in the top-right corner to open the “Publish script” window and create a *script publication*:

After the author follows all the necessary steps to prepare the publication and selects the “Publish private/public script” button on the last page of the “Publish script” window, TradingView generates a dedicated *script widget* and *script page*, which feature options for users to boost, share, report, and comment on the publication.

The script widget is a *preview* of the publication that appears in all relevant locations on TradingView, depending on the specified privacy and visibility settings. It shows the script’s title, a compressed view of the published chart, and a brief preview of the script’s description. An icon in the top-right corner of the widget indicates whether the published script is an indicator, strategy, or library:

Clicking on the widget opens the script page. The top of the page shows information about the script’s visibility, its title, and an enlarged view of the published chart:

For published strategies, the script page also includes the option for users to view the strategy report results below the publication’s title.

Below the chart or strategy report are the publication’s complete description, release notes from script updates, additional information, and user comments.

Privacy types

Script publications have one of two *privacy types*, which determine how users can discover them: public or private. Public scripts are discoverable to all members of the TradingView community, whereas private scripts are accessible only via their URLs. Authors set a script publication’s privacy type using the “Privacy settings” field on the *second page* of the “Publish script” window:

> NoticeEnsure you select the correct option in this field when you prepare a publication, as you **cannot** change a script's privacy type after you publish it.

Public

A script published with the "Public" setting is available in the Community scripts feed and discoverable to all TradingView users worldwide. Unlike public ideas, everyone accesses the same *global repository* for public scripts, regardless of which localized TradingView version they use.

Users can discover public scripts by navigating the Community scripts feed directly, viewing the Scripts tab of an author's profile, searching the "Community" tab of the "Indicators, Metrics & Strategies" menu, or specifying script keywords in the search bar at the top of many TradingView pages. We also feature exceptional public scripts in our Editors' picks.

Because public scripts are available to our global community and are **not** for private use, they must meet the criteria defined in our House Rules, Script Publishing Rules, and Vendor Requirements. Our *script moderators* analyze public scripts using these criteria. Script publications that do not follow these rules become *hidden* from the community.

> NoticeWhen you publish a public script, you have only **15 minutes** to edit or delete it. After that period expires, the publication is finalized and **cannot** be changed or removed. Therefore, before you publish a public script, validate that everything appears as intended and complies with our rules. The recommended approach is to start with a private script, which you can *always* edit or delete.

Private

A script published with the "Private" setting is *not* available in the Community scripts feed, and users cannot find the publication using TradingView's search features. The script widget is visible only to the author, from their profile's Scripts tab. Other users cannot see the script widget, and they cannot view the script page without having access to its URL.

Authors can **always** edit or delete private script publications, unlike public scripts, using the available options in the top-right corner of the script page. This capability makes private scripts ideal for testing features, collaborating with friends, and creating draft publications before committing to public releases. To learn more about how private publications differ from public ones, see this article in our Help Center.

> NoticePrivate scripts are strictly for **private use**. Our script moderators do not analyze privately published scripts as long as they *remain* private. As per our Script Publishing Rules and Vendor Requirements, you cannot reference or link to private publications in any public TradingView content. Additionally, if you share links to private scripts in social networks or other public content, those scripts are *not* considered private.

Visibility types

A script publication's *visibility type* determines whether other users can see the source code, and whether anyone or only authorized individuals can use the script. The possible types are open-source, protected, and invite-only. The "Visibility" options on the *second page* of the "Publish script" window specify a script's visibility type:

> NoticeAs with the privacy type, you **cannot** change a script's visibility type after you publish it. Make sure you select the appropriate option while preparing your publication.

Open

A script published with the "Open" setting is *open-source*, meaning anyone who views the publication or uses the script can access its Pine Script code. Most script publications on TradingView use this setting because it allows programmers to demonstrate their Pine knowledge and provide code for others to verify, learn from, modify, and build upon.

An open-source script's page displays the source code in an expandable window above the comments. The window also includes the option to view the source code directly inside the Pine Editor in a separate tab:

When a user adds the script to their chart, they can also view the source code in the Pine Editor at any time by selecting the "Source code" option in the script's status line:

Note that:

- When a published script's code is open inside the Pine Editor, it is *read-only*. Users cannot edit the code without creating a *working copy*, and any changes to that copied code do **not** affect the original published script.
- All open-source scripts on TradingView use the Mozilla Public License 2.0 by default. Authors wanting to use alternative licenses can specify them in the source code.
- All script publications that *reuse* code from another open-source script must meet the "Open-source reuse" criteria outlined in our Script Publishing Rules. These rules take precedence over any provisions from an open-source license.

> TipOpen-source scripts are eligible for inclusion in our Editors' picks section, which showcases exceptional publications from our growing community of script authors. The Editors' picks are selected from public, open-source scripts that are original, provide potential value to users, include a helpful description, and comply with our House Rules.

Protected

A script published with the "Protected" setting has *closed-source* code, meaning the code is protected and not viewable to any user except the author. Although users cannot access the source code, they can add the script to their charts and use it freely. This visibility option is available only to script authors with paid plans.

Closed-source script publications are ideal for authors wanting to share their unique Pine Script creations with the community without exposing their distinct calculations and logic. They are *not* for sharing closed-source scripts that reproduce the behaviors of open-source ones. As such, when an author publishes a

closed-source script, the publication's description should include information that helps users understand the script's unique characteristics that require protecting the code. See our Script Publishing Rules to learn more.

Invite-only

A script published with the "Invite-only" setting has closed-source code. No user except the author can view the code. Additionally, unlike a protected script, only users *invited* by the author can add the script to their charts and use it. This visibility option is available only to script authors with Premium and higher-tier plans.

Below the description on the invite-only script page, the author can see a *"Manage access"* button. This button opens a dialog box where the author specifies which users have access to the script:

Script authors typically use invite-only publications to provide interested users with unique scripts, often in exchange for payment. As such, invite-only script authors are considered *vendors*. In addition to the House Rules and Script Publishing Rules, which apply to *all* script authors, vendors must understand and follow our Vendor Requirements.

> Notice
>
> Public invite-only scripts are the **only** published scripts for which authors can require payment to access. Selling access to private scripts is prohibited, and authors cannot charge users for access to open-source or protected scripts because they are, by definition, *free* to use.
>
> TradingView does not benefit from script sales. Transactions concerning invite-only scripts are strictly between *users* and *vendors*; they do not involve TradingView.

Preparing a publication

At the start of the script publishing process, authors verify and refine their source code to ensure correct functionality. Then, they prepare their chart visuals and, for strategies, the strategy report, to showcase their script's behaviors. After finalizing these details, authors select the "Publish…" button to open the "Publish script" window, where they set the title, write a helpful description, and then define the publication's settings.

The sections below provide a step-by-step overview of this preparation process and list practical recommendations for creating helpful, user-friendly publications based on our Script Publishing Rules and best practices.

Source code

When an author publishes a script, the publication creates an independent copy of the source code, which becomes part of the publication's *version history*. If the published code contains incorrect or misleading calculations, produces unexpected behaviors, or uses excessive runtime resources, those issues are only fixable through script updates.

Therefore, regardless of a publication's intended visibility type, we recommend validating the source code *before* publishing it to confirm that the script is readable, usable, programmed correctly, and compliant.

When preparing source code to publish:

- Ensure the code is original to you and provides a potentially helpful script for the community.
- Use debugging techniques such as Pine Logs to verify that the script works as intended, and to find and fix any issues in its calculations or logic.
- Fix any higher-timeframe request.security() calls that use a *non-offset* `expression` argument and barmerge.lookahead_on as the `lookahead` argument on historical bars. These calls are not suitable for script publications because they cause *lookahead bias*. See the `lookahead` section of the Other timeframes and data page for more information.
- Use the Pine Profiler to analyze the script's runtime performance. If the script contains unnecessary loops or other inefficient calculations, consider optimizing them to help ensure efficiency and usability.
- Include `minval`, `maxval`, and `options` arguments in applicable `input.*()` calls to prevent users from supplying *unintended* input values. It is also helpful to include runtime.error() calls for other unintended use cases.
- Organize the source code, add helpful titles to inputs and plots, use readable names for identifiers, and include informative comments to make the code simpler to maintain and easier to understand. See the Style guide page for more information.
- Document exported functions and types of libraries with compiler annotations. Annotation text is visible when hovering over an imported library's identifiers or by using parameter hints. Additionally, the description field of the "Publish script" window automatically adds the text to exported code signatures.
- Use a meaningful, searchable title relating to the script's purpose as the `title` argument of the indicator(), strategy(), or library() declaration statement. The title field of the "Publish script" window uses this text by default.

Chart

When an author publishes a script, the publication *copies* their current chart to showcase the visual outputs. If the author has drawings, images, or other scripts on their chart, the published chart also includes them. Therefore, before opening the "Publish script" window, confirm that the chart is clean and ready for publishing.

When preparing a chart for a script publication:

- The script must be active on the chart. If the script is not running on the current chart, open its source code in the Pine Editor and select "Add to chart" in the top-right corner.
- Ensure the chart contains only *necessary* visuals and is easy for users to understand. Remove any other scripts, drawings, or images unless using or demonstrating the script *requires* them. If the publication requires extra scripts or other visuals on the chart, explain their use in the description.
- The chart's *status line* should show the current symbol and timeframe, and the script's status line should show its name. These details help users understand what the displayed data represents. Enable the "Title/Titles" checkboxes in the "Status line" tab of the chart's settings. If the text in the status lines is the same color as the chart's background, change its color in the "Canvas" tab.
- The symbol's price series and the script's visual outputs should be visible on the chart. If the script is on the chart but hidden, select the "Show" icon in its status line to make it visible. If the symbol's price series is invisible, select the "Show" option in the "More" menu of the chart's status line.
- Show the script's *default* behavior so that users know what to expect when they add it to their charts. If an instance of the script on the chart does not use the default settings, select "Reset settings" from the "Defaults" dropdown tab at the bottom of the script's "Settings" menu.

- Do not use a *non-standard chart* (Heikin Ashi, Renko, Line Break, Kagi, Point & Figure, or Range) if the script is a strategy, issues alerts, or displays trade signals of *any kind* in its outputs. The OHLC series on non-standard charts represent *synthetic* (calculated) prices, **not** real-world prices. Scripts that create alert conditions or simulate trades on these charts can **mislead** users and produce **unrealistic** results.

Strategy report

Strategies simulate trades based on programmed rules, displaying their hypothetical performance results in a dedicated strategy report in the chart's bottom panel. When an author publishes a strategy script, the script page uses that report's results along with the strategy's properties and chart's configuration details to populate the publication's *"Strategy report"* display.

Because traders often use a strategy script's performance information to determine the potential viability of a trading system, programmers must verify that their scripts have *realistic* properties and results. Before publishing a strategy script, check the strategy report in the chart's bottom panel to validate that everything appears as intended.

To maintain realism when publishing strategies, follow these guidelines based on our Script Publishing Rules:

- In the strategy() declaration statement, choose an `initial_capital` argument representing realistic starting capital for the average trader in the market. Do not use an excessive value to exaggerate hypothetical returns.
- Specify `commission_*` and `slippage` arguments that approximate real-world commission and slippage amounts. We also recommend using `margin_*` arguments that reflect realistic margin/leverage levels for the chart symbol's exchange.
- Set the strategy's order placement logic to risk *sustainable* capital in the simulated trades. In most real-world settings, risking more than 10% of equity on a single trade is *not* typically considered sustainable.
- Choose a dataset and default strategy configuration that produces a reasonable number of simulated trades, ideally *100 or more*. A strategy report with significantly fewer trades, especially over a short duration, does not typically provide enough information to help traders gauge a strategy's hypothetical performance.
- Ensure the strategy uses the default properties set in the strategy() declaration statement, and explain these defaults in the description.
- Resolve any warnings shown in the strategy report before publishing the script.

Title and description

After preparing the source code, chart visuals, and strategy report for a script publication, open the "Publish Script" window and draft a meaningful title and description to help users understand the script. First, confirm that the correct code is open in the Pine Editor, then select the "Publish…" button in the top-right corner.

The first page of the "Publish Script" window contains two text fields that *cannot* be empty:

The first field determines the publication's *title*, which appears at the top of the script widget and page. TradingView also uses the specified title to determine the publication's *URL*. By default, this field proposes the text from the `title` argument of the script's declaration statement. It is typically best to use that title. However, some authors prefer to use different or modified titles.

When defining the title of a script publication:

- Use text that hints at the script's purpose or functionality. A meaningful title helps users understand and search for the publication.
- Use English text only. If the script is public, it is available to the *global* TradingView community. To help ensure the script is understandable, English is required by our Script Publishing Rules because it is the most common language used for international communication.
- Include only standard 7-bit ASCII characters to ensure readability and searchability. Do not include emoji or other special characters in this text.
- Avoid using all capital letters in the text, except for abbreviations such as "RSI", "EMA", etc. Text with whole words written in ALL CAPS is distracting for users.
- Do not include misleading or unsubstantiated statements about the script (e.g., "90% win rate").
- Do not include website references, social media handles, or other forms of advertisement.

The second text field determines the publication's *description*. The toolbar at the top contains several options that insert *markup tags* into the field for adding text formats, Pine code blocks, lists, and more. The script page displays the complete, parsed text from this field below the published chart or strategy report:

Most of the markup for publication descriptions requires surrounding raw text with an *opening tag* (e.g., `[b]`) and a matching *closing tag* with a forward slash (e.g., `[/b]`). Some tags also require additional syntax. Here, we list the available tags and explain how they work:

- The `[b][/b]`, `[i][/i]`, and `[s][/s]` tag pairs respectively apply **bold**, *italic*, and ~~strikethrough~~ formatting to the enclosed text.
- The `[pine][/pine]` tags format the enclosed multi-line text as a Pine code block with syntax highlighting on a new line.
- The `[list][/list]` tags create a bulleted list. Each line between these tags that starts with the special `[*]` tag defines a separate bullet. To create a *numbered* list, use `[list=1]` as the *opening tag*.
- The `[quote][/quote]` tags format the enclosed multi-line text as a *block quotation*.
- The `[url=][/url]` tags create a hyperlink to a specified URL. For example, `[url=https://www.tradingview.com/]myLink[/url]` formats the text "myLink" as a link to TradingView's home page. Use these tags to create links to relevant TradingView pages and standard reference materials. Avoid linking to social media or other websites, as our House Rules forbid advertising in publications.
- The `[image][/image]` tags render a *chart image* from an enclosed *URL* for either a snapshot or an idea publication. These tags are *optional*, as publications can render images from snapshot and idea URLs automatically. Before taking a snapshot, prepare the chart for readability, as you would for a publication's chart.
- The `$` character adds a hyperlink to a specific symbol's *overview page* when it precedes a valid *symbol* or *ticker identifier*. For example, `$AMEX:SPY` creates a link to the SPY symbol overview.

Writing a helpful description is a **critical step** in the script publishing process, as users rely on the provided information to understand a published script. Below, we list a few helpful recommendations for preparing descriptions based on some of the key criteria outlined in our Script Publishing Rules:

- Include relevant, self-contained details that help users understand the script's purpose, how it works, how to use it, and why it is original, regardless of the intended visibility type. Even if the publication is open-source, the description should cover this information because not all users understand a script

by reading its Pine Script code. Furthermore, an informative description helps users verify that the script works as intended.

- If the publication is closed-source (protected or invite-only), include accurate details about the script's *unique qualities* that require hiding the source code. Closed-source scripts that match the behaviors of open-source scripts *do not* benefit our community.
- Do not make unsubstantiated statements about the script's capabilities or performance. If the text contains claims about the script, it should include details substantiating them to avoid misleading traders.
- If the text contains emoji or other non-ASCII characters, ensure it uses them *sparingly* to maintain readability. Likewise, avoid using all capital letters throughout the text because it reduces readability.
- The description *can* include languages other than English. However, the text should *begin* with an English explanation to help users in *different regions* understand the publication. Additionally, if the source code does not use English for input titles or other user interface text, the description should contain English translations of those elements.

Publication settings

The *second* page of the "Publish script" window is where authors specify a script publication's settings and search tags. This page is accessible only after adding a title and description for the script on the previous page and selecting the "Continue" button:

The two fields at the top of the page specify the script's privacy and visibility types. Ensure both fields use the correct options, as these settings **cannot** change after the script is published.

> TipEven if you intend to share your script publicly, we recommend publishing a private version first. You can use the private publication as a *draft* of the release to ensure the content is correct, then create a new public version with the verified description. See the section on private drafts below to learn more.

Note that setting the publication's visibility type to invite-only reveals an additional *"Author's instructions"* field, which cannot remain empty. This field is where vendors provide necessary information for users to *request access* to their script, such as direct contact details and links to instructional pages. The contents of this field will appear below the description on the invite-only script page:

The remaining input fields on this page provide options to assign *tags* (keywords) to the publication for discoverability. The "Category" field contains a menu where the author can select up to *three* preset category tags for the publication. If the script is public, users can search the specified categories to discover it:

The publication can also include *custom*, non-preset search tags for additional discoverability. To add custom tags to the publication, select the "Show more" option, then enter a list of searchable keywords in the "Tags" field:

Publishing and editing

After following all necessary steps to prepare a script publication, including fine-tuning the source code, cleaning the chart, and adding a helpful title and description, select the "Publish…" button at the bottom of the last page of the "Publish script" window to publish the script.

If the publication's privacy type is set to public, there is a checkbox above the "Publish…" button, which the author must select before they can create the publication. This checkbox confirms awareness of the House Rules and the consequence of the script becoming *hidden* from the community if it does not follow them:

When the script is published, the "Publish script" window closes automatically, and TradingView opens the new publication's script page. The page includes "Edit" and "Delete" buttons in the top-right corner. If the script is public, these buttons are available for only *15 minutes*. If private, they are *always* available.

Selecting the "Edit" button opens the "Edit script" window, where the author can change the title, description, and search tags:

Note that:

- The "Privacy settings" and "Visibility" fields on the second page of this window are **not** editable.
- The "Edit script" window does **not** provide options to edit the published source code, chart, or strategy report. To change these details, publish a script update.

Script updates

Authors can *update* their public or private scripts over time to add new features, fix bugs, optimize performance, etc. To publish an update to an existing script, confirm that the new source code differs from the code in the last published version. Then, add the updated script to the chart and select the "Publish…" option in the top-right of the Pine Editor to open the "Publish script" window.

After opening the window, select the "Update existing script" option at the top of the first page:

In this publishing mode, the first text field specifies the *existing* script to update, **not** the title of a new publication. Enter the existing publication's title in the field or select the title from the available options in the dropdown menu:

Below the title field is a checkbox specifying whether the update will affect the publication's chart. If unchecked (default), the script page will copy the author's *current chart* to showcase the changes. If checked, the publication will continue using its *existing* chart display:

> NoticeIf you plan to update the publication's chart, prepare the chart before opening the "Publish script" window, just as you would with a new publication.

The text field below the checkbox is where the author explains the *changes* made to the script. The publication will display the parsed text from this field beneath the description as dated *release notes* on the script page. The contents of this field **do not** modify the publication's original description and are displayed *in addition* to it:

When publishing release notes, prepare them similarly to the description. Provide self-contained information allowing users to understand the changes included in the update, how they impact the script's functionality, and what benefits the changes provide over the previous version.

> NoticeAfter you publish a script update, the release notes are finalized *immediately* and **cannot** be changed. Therefore, we recommend using private drafts to validate script updates before committing to

> public releases.

The bottom of the page contains an expandable *difference checker*, which displays a side-by-side or inline comparison between the new source code and the last published version. We recommend inspecting and confirming the code differences *before* publishing an update, because all updates are preserved in the script's *version history*:

After confirming the details on the first page of the "Publish script" window, select "Continue" to move to the final page, then select the "Publish new version" button at the bottom to finalize the script update.

Note that:

- The "Privacy settings" and "Visibility" fields appear grayed out on the last page of the window for script updates because authors **cannot** change these settings for existing script publications.

Tips

Use the following tips and our recommendations in the Preparing a publication section above to create helpful, compliant script publications.

Private drafts

New script authors occasionally overlook the importance of reviewing their content before sharing it publicly, leading to unintentional errors in their published script descriptions, such as typos, incorrect statements, or House Rule violations.

The title and description of a public script are editable for only 15 minutes. After that time, the content becomes **final**. If the published text contains mistakes, the author **cannot** edit or update the publication to fix them.

In contrast, private scripts are always editable, making them valuable tools for *drafting* public script releases. Private drafts help authors avoid uncaught mistakes in their public versions and ensure quality for script users. Therefore, we strongly recommend starting *every* script publication with a private draft.

When using private publications as drafts for public releases, follow this simple process:

1. Prepare the draft publication's content as you would for a public script, but set the "Privacy settings" field to "Private" on the last page of the "Publish script" window.
2. Check the private draft's script widget and script page to verify whether the publication's content appears as intended. If there are mistakes in the draft's source code, chart, or strategy report, fix them by publishing an update. To fix errors in the draft's title or description, select the "Edit" option on the script page and add the corrected text to the appropriate field.
3. After validating the draft, open the "Edit script" window and copy the raw text from the description field.
4. Prepare a new, public script publication using the updated source code and verified description text.
5. After publishing the public version, you can delete the private draft using the "Delete" option at the top-right of its script page.

House Rules

Many traders use public scripts in their analysis to reinforce trade decisions. Likewise, many programmers learn from public scripts and use published libraries in their Pine projects. New and experienced users alike should be able to rely on the script publications from our community for helpful content and original, potentially beneficial tools.

Our Script Publishing Rules establish the core criteria for publishing scripts on TradingView, and our Vendor Requirements define additional criteria for vendors. The script moderators curate the Community scripts based on these rules and our House Rules. If a publication does not meet these criteria, it becomes *hidden*, and our moderators send the author a message explaining the issues that need correction. The author can then prepare a *new publication* with the necessary corrections if they want to share their script publicly.

We recommend all authors review and understand our rules and verify a script publication's compliance *before* publishing it. Below, we list a few simple tips:

Publish original content

Publish a script publicly if you believe it is original and might benefit the community. Avoid rehashing, mimicking, or copying existing scripts or other public domain code. Likewise, avoid publishing scripts that combine available indicators or other code without a clear purpose. In other words, aim to provide a helpful tool for the community based on *your* unique interests and expertise.

Reuse code responsibly

Authors can publish scripts that reuse open-source code from other publications. However, they must meet the "Open-source reuse" criteria in our Script Publishing Rules, which take precedence over all open-source licenses. These criteria include crediting the original author, making meaningful improvements to the code, and sharing the code open-source unless the original author grants *explicit permission* to publish it closed-source.

Use a clear chart

A script publication's chart showcases the script's visual outputs to help users understand how it works. This display is not for demonstrating complex charting setups with multiple scripts or drawing tools. If the chart of a published script contains unnecessary scripts or drawings, it will not add clarity for users, and it can potentially mislead them.

Therefore, when publishing a public script, ensure the chart only includes what is *necessary* to demonstrate its outputs and behaviors. See the "Chart" section of our Script Publishing Rules to understand our chart criteria, and this portion of the Preparing a publication section above for detailed recommendations.

Provide helpful documentation

Similar to how users rely on our documentation to understand Pine, users rely on the documentation in an author's publications to understand their scripts. When a script publication does not include a helpful description that explains the script's workings and how to use it, users often struggle to understand and use it effectively. Therefore, when sharing a script publicly, include a clear description explaining everything users need to know about it and its use.

See the "Description" and "Language" sections of our Script Publishing Rules to understand the criteria for helpful script descriptions. The Title and description section above provides detailed recommendations based on these criteria.

For examples of compliant script descriptions, refer to the publications featured in our Editors' picks. To see examples of our recommended description format, refer to the publications from the TradingView and PineCoders accounts.

Style guide

Introduction

This style guide provides recommendations on how to name variables and organize your Pine scripts in a standard way that works well. Scripts that follow our best practices will be easier to read, understand and maintain.

You can see scripts using these guidelines published from the TradingView and PineCoders accounts on the platform.

Naming Conventions

We recommend the use of:

- `camelCase` for all identifiers, i.e., variable or function names: `ma`, `maFast`, `maLengthInput`, `maColor`, `roundedOHLC()`, `pivotHi()`.
- All caps `SNAKE_CASE` for constants: `BULL_COLOR`, `BEAR_COLOR`, `MAX_LOOKBACK`.
- The use of qualifying suffixes when it provides valuable clues about the type or provenance of a variable: `maShowInput`, `bearColor`, `bearColorInput`, `volumesArray`, `maPlotID`, `resultsTable`, `levelsColorArray`.

Script organization

The Pine Script® compiler is quite forgiving of the positioning of specific statements or the version compiler annotation in the script. While other arrangements are syntactically correct, this is how we recommend organizing scripts:

```
<license>
<version>
<declaration_statement>
<import_statements>
<constant_declarations>
<inputs>
<function_declarations>
<calculations>
<strategy_calls>
<visuals>
<alerts>
```

<license>

If you publish your open-source scripts publicly on TradingView (scripts can also be published privately), your open-source code is by default protected by the Mozilla license. You may choose any other license you prefer.

The reuse of code from those scripts is governed by our House Rules on Script Publishing which preempt the author's license.

The standard license comments appearing at the beginning of scripts are:

// This source code is subject to the terms of the Mozilla Public License 2.0 at https://mozilla.org/MPL/2.0/ // © username

<version>

This is the compiler annotation defining the version of Pine Script the script will use. If none is present, v1 is used. For v6, use:

//@version=6

<declaration_statement>

This is the mandatory declaration statement which defines the type of your script. It must be a call to either indicator(), strategy(), or library().

<import_statements>

If your script uses one or more Pine Script libraries, your import statements belong here.

<constant_declarations>

Scripts can declare variables qualified as “const”, i.e., ones referencing a constant value.

We refer to variables as “constants” when they meet these criteria:

- Their declaration uses the optional `const` keyword (see our User Manual's section on type qualifiers for more information).
- They are initialized using a literal (e.g., `100` or `"AAPL"`) or a built-in qualified as “const” (e.g., `color.green`).
- Their value does not change during the script's execution.

We use `SNAKE_CASE` to name these variables and group their declaration near the top of the script. For example:

// ——— Constants int MS_IN_MIN = 60 * 1000 int MS_IN_HOUR = MS_IN_MIN * 60 int MS_IN_DAY = MS_IN_HOUR * 24 color GRAY = #808080ff color LIME = #00FF00ff color MAROON = #800000ff color ORANGE = #FF8000ff color PINK = #FF0080ff color TEAL = #008080ff color BG_DIV = color.new(ORANGE, 90) color BG_RESETS = color.new(GRAY, 90) string RST1 = "No reset; cumulate since the beginning of the chart" string RST2 = "On a stepped higher timeframe (HTF)" string RST3 = "On a fixed HTF" string RST4 = "At a fixed time" string RST5 = "At the beginning of the regular session" string RST6 = "At the first visible chart bar" string RST7 = "Fixed rolling period" string LTF1 = "Least precise, covering many chart bars" string LTF2 = "Less precise, covering some chart bars" string LTF3 = "More precise, covering less chart bars" string LTF4 = "Most precise, 1min intrabars" string TT_TOTVOL = "The 'Bodies' value is the transparency of the total volume candle bodies. Zero is opaque, 100 is transparent." string TT_RST_HTF = "This value is used when '" + RST3 +"' is selected." string TT_RST_TIME = "These values are used when '" + RST4 +"' is selected. A reset will occur when the time is greater or equal to the

bar's open time, and less than its close time.\nHour: 0-23\nMinute: 0-59" string TT_RST_PERIOD = "This value is used when "' + RST7 +"' is selected."

In this example:

- The `RST*` and `LTF*` constants will be used as tuple elements in the `options` argument of `input.*()` calls.
- The `TT_*` constants will be used as `tooltip` arguments in `input.*()` calls. Note how we use a line continuation for long string literals.
- We do not use var to initialize constants. The Pine Script runtime is optimized to handle declarations on each bar, but using var to initialize a variable only the first time it is declared incurs a minor penalty on script performance because of the maintenance that var variables require on further bars.

Note that:

- Literals used in more than one place in a script should always be declared as a constant. Using the constant rather than the literal makes it more readable if it is given a meaningful name, and the practice makes code easier to maintain. Even though the quantity of milliseconds in a day is unlikely to change in the future, `MS_IN_DAY` is more meaningful than `1000 * 60 * 60 * 24`.
- Constants only used in the local block of a function or if, while, etc., statement for example, can be declared in that local block.

<inputs>

It is **much** easier to read scripts when all their inputs are in the same code section. Placing that section at the beginning of the script also reflects how they are processed at runtime, i.e., before the rest of the script is executed.

Suffixing input variable names with `input` makes them more readily identifiable when they are used later in the script: `maLengthInput`, `bearColorInput`, `showAvgInput`, etc.

// ———— Inputs string resetInput = input.string(RST2, "CVD Resets", inline = "00", options = [RST1, RST2, RST3, RST4, RST5, RST6, RST7]) string fixedTfInput = input.timeframe("D", " Fixed HTF: ", tooltip = TT_RST_HTF) int hourInput = input.int(9, " Fixed time hour: ", inline = "01", minval = 0, maxval = 23) int minuteInput = input.int(30, "minute", inline = "01", minval = 0, maxval = 59, tooltip = TT_RST_TIME) int fixedPeriodInput = input.int(20, " Fixed period: ", inline = "02", minval = 1, tooltip = TT_RST_PERIOD) string ltfModeInput = input.string(LTF3, "Intrabar precision", inline = "03", options = [LTF1, LTF2, LTF3, LTF4])

<function_declarations>

All user-defined functions must be defined in the script's global scope; nested function definitions are not allowed in Pine Script.

Optimal function design should minimize the use of global variables in the function's scope, as they undermine function portability. When it can't be avoided, those functions must follow the global variable declarations in the code, which entails they can't always be placed in the <function_declarations> section. Such dependencies on global variables should ideally be documented in the function's comments.

It will also help readers if you document the function's objective, parameters and result. The same syntax used in libraries can be used to document your functions. This can make it easier to port your functions to a

library should you ever decide to do so:

//@version=6 indicator("<function_declarations>", "", true) string SIZE_LARGE = "Large" string SIZE_NORMAL = "Normal" string SIZE_SMALL = "Small" string sizeInput = input.string(SIZE_NORMAL, "Size", options = [SIZE_LARGE, SIZE_NORMAL, SIZE_SMALL]) // @function Used to produce an argument for the `size` parameter in built-in functions. // @param userSize (simple string) User-selected size. // @returns One of the `size.*` built-in constants. // Dependencies SIZE_LARGE, SIZE_NORMAL, SIZE_SMALL getSize(simple string userSize) => result = switch userSize SIZE_LARGE => size.large SIZE_NORMAL => size.normal SIZE_SMALL => size.small => size.auto if ta.rising(close, 3) label.new(bar_index, na, yloc = yloc.abovebar, style = label.style_arrowup, size = getSize(sizeInput))

<calculations>

This is where the script's core calculations and logic should be placed. Code can be easier to read when variable declarations are placed near the code segment using the variables. Some programmers prefer to place all their non-constant variable declarations at the beginning of this section, which is not always possible for all variables, as some may require some calculations to have been executed before their declaration.

<strategy_calls>

Strategies are easier to read when strategy calls are grouped in the same section of the script.

<visuals>

This section should ideally include all the statements producing the script's visuals, whether they be plots, drawings, background colors, candle-plotting, etc. See the Pine Script user manual's section on Z-index for more information on how the relative depth of visuals is determined.

<alerts>

Alert code will usually require the script's calculations to have executed before it, so it makes sense to put it at the end of the script.

Spacing

A space should be used on both sides of all operators, except unary operators (`-1`). A space is also recommended after all commas and when using named function arguments, as in `plot(series = close)`:

int a = close > open ? 1 : -1 var int newLen = 2 newLen := min(20, newlen + 1) float a = -b float c = d > e ? d - e : d int index = bar_index % 2 == 0 ? 1 : 2 plot(close, color = color.red)

Line wrapping

Line wrapping can make long lines of code easier to read by defining a *single line* of code across *multiple* lines in the script. Generally, scripts can wrap lines using any indentation length that is *not* a multiple of four, because the four-space or tab indentation defines local blocks in Pine.

However, if part of a wrapped line is enclosed in *parentheses* `( )` or square brackets `[ ]`, that code can use *any* number of spaces for indentation without restriction, including zero or a multiple of four. For example:

```
//@version=6 indicator("Line wrapping demo") // A wrapped line that is *not* enclosed in parentheses or brackets can use any indentation *except* a multiple of four. float closeDiff =   close // Indented by two spaces. - close[1] // Indented by two spaces. // A wrapped line that *is* enclosed in parentheses *can* use multiples of four spaces for indentation. float percentChange = (     closeDiff) / close[         // Indented by four spaces.     1] * 100                  // Indented by four spaces. // Within the same expression or statement, each wrapped line can use different indentation lengths. // The parentheses or brackets enclosing wrapped lines can also be placed on separate lines. color plotColor = percentChange > percentChange[     1     // Indented by four spaces. ] ? color.rgb(                 // No indentation.   76, 175, 80                 // Indented by two spaces.  ) : color.rgb(                 // Indented by one space.         255, 82, 82             // Indented by eight spaces.     )                         // Indented by four spaces. plot(  percentChange, title = "Percent change", // Indented by one space.    color = plotColor,                    // Indented by three spaces.     linewidth = 2,                 // Indented by four spaces.         style = plot.style_histogram, // Indented by eight spaces. format = format.percent   // Indented by six spaces. )                                 // No indentation.
```

Line wrapping is also useful when working with long single-line strings. For example, instead of defining a lengthy string on a single line of code, programmers can split that string into smaller parts and concatenate them using the + operator to wrap the expression across multiple lines for readability:

```
//@version=6 indicator("Defining a long string across multiple lines demo") //@variable A single, long string created by concatenating three smaller single-line strings. // These wrapped lines are indented by five spaces. If enclosed in parentheses, they can use four spaces instead. var string newString = "This is one long string result that is defined " + "across multiple lines of code by concatenating smaller strings. " + "When output, the text appears without line breaks until we include the \n newline escape sequence." if barstate.isfirst // Output the `newString` result in the Pine Logs pane. log.info(newString)
```

It is possible to use multiple line wrapping styles within the same script and even within the same statement, as shown in the first example above. However, to keep the code organized and easy to read, we recommend maintaining a *consistent* line wrapping style within the same script where possible. For instance, programmers can choose to align wrapped lines to their nearest tab space, or to apply minimal wrapping only when the length of a line exceeds the Pine Editor's line length guide.

This example script shows a consistent line wrapping style that lists each argument in a function call on a wrapped line indented by four spaces. It wraps each call's closing parenthesis on a separate line without indentation to align it vertically with the beginning of the expression and signify the end of the wrapped code:

```
//@version=6 indicator("Consistent line wrapping style demo") //@variable The number of values to show in the plot, counting backwards from the last bar. int lengthInput = input.int( defval = 10, title = "Show last", minval = 1 ) // Plot the last `lengthInput` values of the `close` series. plot( series = close, title = "Close", color = color.blue, linewidth = 3, show_last = lengthInput )
```

Similar line-wrapping behaviors apply to multiline strings, which can span multiple visible lines in the code. All parts of the code between a multiline string's """ or ''' delimiters, including leading spaces and line breaks, represent *literal text*. Pine treats expressions that use multiline strings as part of a single code line, similar to other types of line-wrapped code. See the Line wrapping section of the Script structure page for an example.

For readability, it is often helpful to define multiline strings separately in the *global scope*, or in line-wrapped expressions with *no indentation*. For example:

```
//@version=6 indicator("Multiline string style demo") //@variable A string created using multiline syntax. string multilineStr = """ This string spans multiple lines in the editor, but it's considered part of a single line of code. All parts of the code here, including line breaks, represent literal text. Starting the definition in non-indented code can help readability, because any indentation, <- such as this <- or this is automatically included in the string. """ //@variable References an array of strings created using multiline syntax. var array<string> multiStrArray = array.from( """When defining multiline strings inside expressions, such as function calls, """, """starting each string on a separate, non-indented line in the editor can help promote readability.""" ) // Draw labels to display text from the multiline string and array of strings on the last historical bar. if barstate.islastconfirmedhistory label.new(bar_index, 0, multilineStr, textalign = text.align_left) label.new(bar_index, 0, multiStrArray.join(), style = label.style_label_up, textalign = text.align_left)
```

Vertical alignment

Vertical alignment using tabs or spaces can be useful in code sections containing many similar lines such as constant declarations or inputs. They can make mass edits much easier using the Pine Editor's multi-cursor feature (`ctrl` + `alt` +):

```
// Colors used as defaults in inputs. color COLOR_AQUA = #0080FFff color COLOR_BLACK = #000000ff color COLOR_BLUE = #013BCAff color COLOR_CORAL = #FF8080ff color COLOR_GOLD = #CCCC00ff
```

Explicit typing

Including the type of variables when declaring them is not required. However, it helps make scripts easier to read, navigate, and understand. It can help clarify the expected types at each point in a script's execution and distinguish a variable's declaration (using `=`) from its reassignments (using `:=`). Using explicit typing can also make scripts easier to debug.