Pine Script v5 User Manual

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

60 pages of text and code. The 518 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. Backgrounds (concepts/backgrounds)
3. Bar coloring (concepts/bar-coloring)
4. Bar plotting (concepts/bar-plotting)
5. Bar states (concepts/bar-states)
6. Chart information (concepts/chart-information)
7. Colors (concepts/colors)
8. Fills (concepts/fills)
9. Inputs (concepts/inputs)
10. Levels (concepts/levels)
11. Libraries (concepts/libraries)
12. Lines and boxes (concepts/lines-and-boxes)
13. Non-standard charts data (concepts/non-standard-charts-data)
14. Other timeframes and data (concepts/other-timeframes-and-data)
15. Plots (concepts/plots)
16. Repainting (concepts/repainting)
17. Sessions (concepts/sessions)
18. Strategies (concepts/strategies)
19. Tables (concepts/tables)
20. Text and shapes (concepts/text-and-shapes)
21. Time (concepts/time)
22. Timeframes (concepts/timeframes)
23. Error messages (error-messages)
24. Alerts (faq/alerts)
25. Data structures (faq/data-structures)
26. Functions (faq/functions)
27. Indicators (faq/indicators)
28. Other data and timeframes (faq/other-data-and-timeframes)
29. Programming (faq/programming)
30. Arrays (language/arrays)
31. Built-ins (language/built-ins)
32. Conditional structures (language/conditional-structures)
33. Enums (language/enums)

34. Execution model (language/execution-model)
35. Identifiers (language/identifiers)
36. Loops (language/loops)
37. Maps (language/maps)
38. Matrices (language/matrices)
39. Methods (language/methods)
40. Objects (language/objects)
41. Operators (language/operators)
42. Script structure (language/script-structure)
43. Time series (language/time-series)
44. Type system (language/type-system)
45. User-defined functions (language/user-defined-functions)
46. Variable declarations (language/variable-declarations)
47. To Pine Script® version 3 (migration-guides/to-pine-version-3)
48. To Pine Script® version 4 (migration-guides/to-pine-version-4)
49. To Pine Script® version 5 (migration-guides/to-pine-version-5)
50. First indicator (primer/first-indicator)
51. First steps (primer/first-steps)
52. Next steps (primer/next-steps)
53. Release notes (release-notes)
54. Welcome to Pine Script® v5 (welcome)
55. Where can I get more information? (where-can-i-get-more-information)
56. Debugging (writing/debugging)
57. Limitations (writing/limitations)
58. Profiling and optimization (writing/profiling-and-optimization)
59. Publishing scripts (writing/publishing)
60. Style guide (writing/style-guide)

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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=5 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=5 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=5 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=5 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=5 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=5 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=5 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.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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=5 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=5 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.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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=5 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.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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=5 indicator("Single-color candles") plotcandle(open, high, low, close)

To color them green or red, we can use the following code:

//@version=5 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=5 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=5 indicator("Daily bars") // Use gaps to only return data when the 1D timeframe completes, `na` otherwise. [o, h, l, c] = request.security(syminfo.tickerid, "D", [open, high, low, close], gaps = barmerge.gaps_on) var color UP_COLOR = color.silver var 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) // Only plot candles on intraday timeframes, // and when non `na` values are returned by `request.security()` because a HTF has completed. plotcandle(timeframe.isintraday ? o : na, h, l, c, color = bodyColor, wickcolor = wickColor)

Note that:

- We show the script's plot after having used “Visual Order/Bring to Front” from the script's “More” menu. This causes our script's candles to appear on top of the chart's candles.
- The script will only display candles 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 use var to declare our `UP_COLOR` and `DN_COLOR` color constants on bar zero only. We use constants because those colors are used in more than one place in our code. This way, if we need to change them, we need only do so in one place.
- 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=5 indicator("Dual-color bars") paletteColor = close >= open ? color.lime : color.red plotbar(open, high, low, close, color = paletteColor)

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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=5 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=5 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=5 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=5 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=5 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:

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Chart information

Introduction

The way scripts can obtain information about the chart and symbol they are currently running on is through a subset of Pine Script®'s built-in variables. The ones we cover here allow scripts to access information relating to:

- The chart's prices and volume
- The chart's symbol
- The chart's timeframe
- The session (or time period) the symbol trades on

Prices and volume

The built-in variables for OHLCV values are:

- open: the bar's opening price.
- high: the bar's highest price, or the highest price reached during the realtime bar's elapsed time.
- low: the bar's lowest price, or the lowest price reached during the realtime bar's elapsed time.
- close: the bar's closing price, or the **current price** in the realtime bar.
- volume: the volume traded during the bar, or the volume traded during the realtime bar's elapsed time. The unit of volume information varies with the instrument. It is in shares for stocks, in lots for forex, in contracts for futures, in the base currency for crypto, etc.

Other values are available through:

- hl2: the average of the bar's high and low values.
- hlc3: the average of the bar's high, low and close values.
- ohlc4: the average of the bar's open, high, low and close values.

On historical bars, the values of the above variables do not vary during the bar because only OHLCV information is available on them. When running on historical bars, scripts execute on the bar's close, when all the bar's information is known and cannot change during the script's execution on the bar.

Realtime bars are another story altogether. When indicators (or strategies using `calc_on_every_tick = true`) run in realtime, the values of the above variables (except open) will vary between successive iterations of the script on the realtime bar, because they represent their **current** value at one point in time during the progress of the realtime bar. This may lead to one form of repainting. See the page on Pine Script's execution model for more details.

The [] history-referencing operator can be used to refer to past values of the built-in variables, e.g., `close[1]` refers to the value of close on the previous bar, relative to the particular bar the script is executing on.

Symbol information

Built-in variables in the `syminfo` namespace provide scripts with information on the symbol of the chart the script is running on. This information changes every time a script user changes the chart's symbol. The script then re-executes on all the chart's bars using the new values of the built-in variables:

- syminfo.basecurrency: the base currency, e.g., “BTC” in “BTCUSD”, or “EUR” in “EURUSD”.
- syminfo.currency: the quote currency, e.g., “USD” in “BTCUSD”, or “CAD” in “USDCAD”.
- syminfo.description: The long description of the symbol.
- syminfo.mintick: The symbol's tick value, or the minimum increment price can move in. Not to be confused with *pips* or *points*. On “ES1!” (“S&P 500 E-Mini”) the tick size is 0.25 because that is the minimal increment the price moves in.
- syminfo.pointvalue: The point value is the multiple of the underlying asset determining a contract's value. On “ES1!” (“S&P 500 E-Mini”) the point value is 50, so a contract is worth 50 times the price of the instrument.
- syminfo.prefix: The prefix is the exchange or broker's identifier: “NASDAQ” or “BATS” for “AAPL”, “CME_MINI_DL” for “ES1!”.
- syminfo.root: It is the ticker's prefix for structured tickers like those of futures. It is “ES” for “ES1!”, “ZW” for “ZW1!”.
- syminfo.session: It reflects the session setting on the chart for that symbol. If the “Chart settings/Symbol/Session” field is set to “Extended”, it will only return “extended” if the symbol and the user's feed allow for extended sessions. It is rarely displayed and used mostly as an argument to the `session` parameter in ticker.new().
- syminfo.ticker: It is the symbol's name, without the exchange part (syminfo.prefix): “BTCUSD”, “AAPL”, “ES1!”, “USDCAD”.
- syminfo.tickerid: This string is rarely displayed. It is mostly used as an argument for request.security()'s `symbol` parameter. It includes session, prefix and ticker information.
- syminfo.timezone: The timezone the symbol is traded in. The string is an IANA time zone database name (e.g., “America/New_York”).
- syminfo.type: The type of market the symbol belongs to. The values are “stock”, “futures”, “index”, “forex”, “crypto”, “fund”, “dr”, “cfd”, “bond”, “warrant”, “structured” and “right”.

This script will display the values of those built-in variables on the chart:

```
//@version=5 indicator("`syminfo.*` built-ins", "", true) printTable(txtLeft, txtRight) => var table t = table.new(position.middle_right, 2, 1) table.cell(t, 0, 0, txtLeft, bgcolor = color.yellow, text_halign = text.align_right) table.cell(t, 1, 0, txtRight, bgcolor = color.yellow, text_halign = text.align_left) nl = "\n" left = "syminfo.basecurrency: " + nl + "syminfo.currency: " + nl + "syminfo.description: " + nl + "syminfo.mintick: " + nl + "syminfo.pointvalue: " + nl + "syminfo.prefix: " + nl + "syminfo.root: " + nl + "syminfo.session: " + nl + "syminfo.ticker: " + nl + "syminfo.tickerid: " + nl + "syminfo.timezone: " + nl + "syminfo.type: " right = syminfo.basecurrency + nl + syminfo.currency + nl + syminfo.description + nl + str.tostring(syminfo.mintick) + nl + str.tostring(syminfo.pointvalue) + nl + syminfo.prefix + nl + syminfo.root + nl + syminfo.session + nl + syminfo.ticker + nl + syminfo.tickerid + nl + syminfo.timezone + nl + syminfo.type printTable(left, right)
```

Chart timeframe

A script can obtain information on the type of timeframe used on the chart using these built-ins, which all return a “simple bool” result:

- timeframe.isseconds
- timeframe.isminutes
- timeframe.isintraday
- timeframe.isdaily
- timeframe.isweekly

- timeframe.ismonthly
- timeframe.isdwm

Two additional built-ins return more specific timeframe information:

- timeframe.multiplier returns a “simple int” containing the multiplier of the timeframe unit. A chart timeframe of one hour will return `60` because intraday timeframes are expressed in minutes. A 30sec timeframe will return `30` (seconds), a daily chart will return `1` (day), a quarterly chart will return `3` (months), and a yearly chart will return `12` (months). The value of this variable cannot be used as an argument to `timeframe` parameters in built-in functions, as they expect a string in timeframe specifications format.
- timeframe.period returns a string in Pine Script's timeframe specification format.

See the page on Timeframes for more information.

Session information

Session information is available in different forms:

- The syminfo.session built-in variable returns a value that is either session.regular or session.extended. It reflects the session setting on the chart for that symbol. If the “Chart settings/Symbol/Session” field is set to “Extended”, it will only return “extended” if the symbol and the user's feed allow for extended sessions. It is used when a session type is expected, for example as the argument for the `session` parameter in ticker.new().
- Session state built-ins provide information on the trading session a bar belongs to.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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.

Z-index

When you place elements in a script's visual space, they have relative depth on the *z* axis; some will appear on top of others. The *z-index* is a value that represents the position of elements on the *z* axis. Elements with the highest z-index appear on top.

Elements drawn in Pine Script are divided in groups. Each group has its own position in the *z* space, and **within the same group**, elements created last in the script's logic will appear on top of other elements from the same group. An element of one group cannot be placed outside the region of the *z* space attributed to its group, so a plot can never appear on top of a table, for example, because tables have the highest z-index.

This list contains the groups of visual elements, ordered by increasing z-index, so background colors are always at the bottom of *z* space, and tables will always appear on top of all other elements:

- Background colors
- Fills
- Plots
- Hlines
- LineFills
- Lines
- Boxes
- Labels
- Tables

Note that by using `explicit_plot_zorder = true` in indicator() or strategy(), you can control the relative z-index of `plot*()`, hline() and fill() visuals using their sequential order in the script.

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)
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	#FF5252	color.rgb(255, 82, 82)
color.silver	#B2B5BE	color.rgb(178, 181, 190)
color.teal	#00897B	color.rgb(0, 137, 123)
color.white	#FFFFFF	color.rgb(255, 255, 255)
color.yellow	#FFEB3B	color.rgb(255, 235, 59)

In the following script, all plots use the same color.olive color with a transparency of 40, but expressed in different ways. All five methods are functionally equivalent:

//@version=5 indicator("", "", true) // ———— Transparency (#99) is included in the hex value. plot(ta.sma(close, 10), "10", #80800099) // ———— Transparency is included in the color-generating function's arguments. plot(ta.sma(close, 30), "30", color.new(color.olive, 40)) plot(ta.sma(close, 50), "50",

color.rgb(128, 128, 0, 40)) // ———— Use `transp` parameter (deprecated and advised against) plot(ta.sma(close, 70), "70", color.olive, transp = 40) plot(ta.sma(close, 90), "90", #808000, transp = 40)

> Notice!The last two plot() calls specify transparency using the `transp` parameter. This use should be avoided as the `transp` is deprecated in Pine Script v5. Using the `transp` parameter to define transparency is not as flexible because it requires an argument of *input integer* type, which entails it must be known before the script is executed, and so cannot be calculated dynamically, as your script executes bar to bar. Additionally, if you use a `color` argument that already includes transparency information, as is done in the next three plot() calls, any argument used for the `transp` parameter would have no effect. This is also true for other functions with a `transp` parameter.

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=5 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 c_ma = maRising ? maBullColorInput : maBearColorInput plot(ma, "MA", c_ma, 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 a `c_ma` color variable that is assigned one of our two colors, depending on the value of the `maRising` boolean. We use the ? : ternary operator to write our conditional statement.

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=5 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) ? na : pHiColorInput, 2, plot.style_line) plot(pLo, "Low", ta.change(pLo) ? 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(color, transp) to use to create different transparencies for volume columns using one of two bull/bear base colors:

//@version=5 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(red, green, blue, transp) 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=5 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 = c_holiday, wickcolor = holidayColor, bordercolor = c_holiday)

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(value, bottom_value, top_value, bottom_color, top_color). 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=5 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=5 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 `c_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, `c_endColor` color 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=5 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)) + 1)) / 255), 60, 90) float loTransp = clamp(100 - (100 * math.max(1, nz(ta.barssince(ta.change(loBot)) + 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, PineScript can automatically display all of the colors used in script's plots in the "Settings/Styles" 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, PineScript automatically displays the colors in the "Settings/Styles" menu and allows the user to change them, along with the style of the plot:

//@version=5 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 the plot via the "Styles" tab, include `editable = false` in the plot() call.

The colors in the above script can be automatically displayed in this way because they are already known *during compilation*. Colors that are known during compilation are of type "const" or "input" only.

However, if *even a single calculated color* is of type "simple color" or "series color", *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 most often due to using color.new() and color.rgb() functions. The qualifier of the color that they return is the strongest qualifier of the values passed to these functions. If each call to these functions passes only "const" or "input" values, the resulting colors are also "const" or "input", and the script *does* display them in the "Styles" menu.

> WarningThe 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 that 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=5 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 teal and 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 n the “Style” tab:

//@version=5 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=5 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.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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=5 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 a 10-bar SMA, 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=5 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=5 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=5 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=5 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.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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-period simple moving average (SMA) using ta.sma(close, 20). 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=5 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=5 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.price()
- input.source()
- input.session()
- input.time()
- 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=5 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`, and `display`. Some input functions also include other parameters: `options`, `minval`, `maxval`, `step` and `confirm`.

All these parameters expect “const” arguments. The only exceptions are for the `defval` and `options` parameters of the source and enum inputs, as input.source() uses “series float” value, and input.enum() uses *members* of an enum type.

Since `input.*()` parameters accept “const” arguments in most cases and the “input” and other qualifiers are stronger than “const”, it follows that one cannot use the result from one `input.*()` call as an argument in another `input.*()` call.

Let's go over each parameter:

- `defval` is the first parameter of all input functions. It is the default value that appears in the input widget. It requires an argument of the type of input value that the function applies to.
- `title` requires a “const string” argument. It is the field's label.
- `tooltip` requires a “const string” argument. When the parameter is used, a question mark icon will appear to the right of the field. When users hover over it, the tooltip's text will appear. Note that if multiple input fields are grouped on one line using `inline`, the tooltip will always appear to the right of the rightmost field, and display the text of the last `tooltip` argument used in the line. Newlines (`\n`) are supported in the argument string.

- `inline` requires a “const string” argument. Using the same argument for the parameter in multiple `input.*()` calls will group their input widgets on the same line. There is a limit to the width the “Inputs” tab will expand, so a limited quantity of input fields can be fitted on one line. Using one `input.*()` call with a unique argument for `inline` has the effect of bringing the input field left, immediately after the label, foregoing the default left-alignment of all input fields used when no `inline` argument is used.
- `group` requires a “const string” argument. Use it to group any number of inputs in an organized section. The string used as the `group` argument becomes the section's heading. All `input.*()` calls to be grouped together must use the same string for their `group` argument.
- `options` requires a comma-separated list of elements enclosed in square brackets (e.g., `["ON", "OFF"]`, `[1, 2, 3]`, `[myEnum.On, myEnum.Off]`). The input uses the specified elements as menu selections in its resulting dropdown widget. Users can only select one menu item at a time. When supplying an `options` list, the `defval` value must be one of the list's elements. Inputs that allow `minval`, `maxval`, or `step` parameters cannot use those parameters and the `options` parameter simultaneously.
- `minval` requires a “const int/float” argument, depending on the type of the `defval` value. It is the minimum valid value for the input field.
- `maxval` requires a “const int/float” argument, depending on the type of the `defval` value. It is the maximum valid value for the input field.
- `step` is the increment by which the field's value will move when the widget's up/down arrows are used.
- `confirm` requires a “const bool” (`true` or `false`) argument. This parameter affects the behavior of the script when it is added to a chart. `input.*()` calls using `confirm = true` will cause the “Settings/Inputs” tab to popup when the script is added to the chart. `confirm` is useful to ensure that users configure a particular field.

The `minval`, `maxval` and `step` parameters are only present in the signature of the input.int() and input.float() functions.

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) → input int/float/bool/color/strin
```

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=5 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) → input int
```

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

//@version=5 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=5 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, confirm
input.float(defval, title, options, tooltip, inline, group, confirm) → input int
```

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

//@version=5 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=5 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 ligther 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=5 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.

Timeframe input

Timeframe inputs can be useful when you want to be able to change the timeframe used to calculate values in your scripts.

Let's do away with our BBs from the previous sections and add a timeframe input to a simple MA script:

//@version=5 indicator("MA", "", true) tfInput = input.timeframe("D", "Timeframe") ma = ta.sma(close, 20) securityNoRepaint(sym, tf, src) => request.security(sym, tf, src[barstate.isrealtime ? 1 : 0]) [barstate.isrealtime ? 0 : 1] maHTF = securityNoRepaint(syminfo.tickerid, tfInput, ma) plot(maHTF, "MA", color.aqua)

Note that:

- We use the input.timeframe() function to receive the timeframe input.
- The function creates a dropdown widget where some standard timeframes are proposed. The list of timeframes also includes any you have favorated in the chart user interface.
- We use the `tfInput` in our request.security() call. We also use `gaps = barmerge.gaps_on` in the call, so the function only returns data when the higher timeframe has completed.

Symbol input

The input.symbol() function creates a widget that allows users to search and select symbols like they would from the chart's user interface.

Let's add a symbol input to our script:

//@version=5 indicator("MA", "", true) tfInput = input.timeframe("D", "Timeframe") symbolInput = input.symbol("", "Symbol") ma = ta.sma(close, 20) securityNoRepaint(sym, tf, src) => request.security(sym, tf, src[barstate.isrealtime ? 1 : 0])[barstate.isrealtime ? 0 : 1] maHTF = securityNoRepaint(symbolInput, tfInput, ma) plot(maHTF, "MA", color.aqua)

Note that:

- The `defval` argument we use is an empty string. This causes request.security(), where we use the `symbolInput` variable containing that input, to use the chart's symbol by default. If the user selects another symbol and wants to return to the default value using the chart's symbol, he will need to use the “Reset Settings” selection from the “Inputs” tab's “Defaults” menu.
- We use the `securityNoRepaint()` user-defined function to use request.security() in such a way that it does not repaint; it only returns values when the higher timeframe has completed.

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=5 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=5 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

Time inputs use the input.time() function. The function returns a Unix time in milliseconds (see the Time page for more information). This type of data also contains date information, so the input.time() function returns a time **and** a date. That is the reason why its widget allows for the selection of both.

Here, we test the bar's time against an input value, and we plot an arrow when it is greater:

//@version=5 indicator("Time input", "T", true) timeAndDateInput = input.time(timestamp("1 Aug 2021 00:00 +0300"), "Date and time") barIsLater = time > timeAndDateInput plotchar(barIsLater, "barIsLater", "□", location.top, size = size.tiny)

Note that the `defval` value we use is a call to the timestamp() function.

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=5 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=5 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 multliple `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.

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

//@version=5 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.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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) → 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 int” 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=5 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=5 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.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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 Script's typing nomenclature, scopes and user-defined functions. If you need to brush up on qualified types, see the User Manual's page on the Type system. For more information on user-defined functions and scopes, see the User-defined functions page.

You can browse the library scripts published publicly by members in TradingView's Community Scripts.

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=5 // @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=5 // @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 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 cannot call `request.*()` functions in their local scopes unless the script includes `dynamic_requests = true` in the library() declaration statement. See this section of the Other timeframes and data page to learn more about *dynamic requests*.

Library functions always return "simple" or "series" results. 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" qualified type.

Qualified type control

The qualified types of arguments supplied in calls to library functions are autodetected based on how each argument is used inside the function. If the argument can be used as a "series", it is qualified as such. If it cannot, an attempt is made with the "simple" type qualifier. This explains why this code:

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

will work when called using `myCustomLibrary.myEma(20)`, even though ta.ema()'s `length` parameter requires a "simple int" argument. When the Pine Script compiler detects that a "series" length cannot be used with ta.ema(), it tries the "simple" qualifier, which in this case is allowed.

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 arguments are qualified as “series” by default, using the series modifier is redundant.

User-defined types and objects

Libraries can export user-defined types (UDTs), and library functions can return objects of these types.

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

//@version=5 library("Point") export type point int x float y bool isHi bool wasBreached = false

A script importing that library and creating an object of its `point` UDT would look somewhat like this:

//@version=5 indicator("") import userName/Point/1 as pt newPoint = pt.point.new()

Note that:

- This code won't compile because no “Point” library is published, and the script doesn't display anything.
- `userName` would need to be replaced by the TradingView user name of the library's publisher.
- We use the built-in `new()` method to create an object from the `point` UDT.
- We prefix the reference to the library's `point` UDT with the `pt` alias defined in the import statement, just like we would when using a function from an imported library.

A library **must** export a UDT if any exported functions or methods accept or return an object of that type, or if the fields of another exported UDT accept an instance of the type.

When a library only uses a UDT internally, it does not need to export the type. The following library uses the `point` type internally, but it only exports the `drawPivots()` function, which does not have a parameter of the `point` type or return a `point` object:

//@version=5 library("PivotLabels", true) // We use this `point` UDT in the library, but it does NOT require exporting because // 1. The exported function's parameters do not use the UDT. // 2. The exported function does not return a UDT result. 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 pivotHi => point.new(time[rightLegs], pivotHi, true) 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. // @returns Nothing. 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=5 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 values 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=5 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=5 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=5 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.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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.

> Notice!Using the types we discuss on this page often involves arrays, especially when working with polylines, which *require* an array of chart.point instances. We therefore recommend you become familiar with arrays 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.

> Notice!On TradingView charts, a complete set of *Drawing Tools* allows users to create and modify drawings using mouse actions. While they may sometimes resemble drawing objects created with Pine Script code, they are **unrelated** entities. Pine scripts cannot interact with drawing tools from the chart user interface, and mouse actions do not directly affect Pine 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 - 9999`. 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=5 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=5 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 the `periodLine` and the variables that reference chart.point values (`openPoint` and `closePoint`) only on the *first* chart bar, and it assigns new values to these variables over its execution. After detecting a change on the `timeframe`, it sets the `color` of the existing `periodLine` using line.set_color(), creates new values for the `openPoint` and `closePoint` using chart.point.now(), then assigns a new line using those points to the `periodLine`.

On other bars where the `periodLine` value is not na, the script assigns a new chart.point to the `closePoint`, then uses line.set_second_point() and line.set_color() as methods to update the line's properties:

//@version=5 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 ID assigned to the `directionLine` persists over subsequent bars until the `newDirection` condition occurs, in which case the script assigns a new line 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 `directionLine` and calculate its `slope`, which it uses to determine the color of each drawing and plot. It retrieves extended values of the `directionLine` from *beyond* its second point using line.get_price() and plots them on the chart:

//@version=5 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=5 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 `id` 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 draws a horizontal line with the extend.right property whenever an RSI crosses its EMA.

The script stores all line IDs in a `lines` array that it uses as a queue to only display the last `numberOfLines` on the chart. When the size of the array exceeds the specified `numberOfLines`, the script removes the array's oldest line ID using array.shift() and deletes it with line.delete():

//@version=5 //@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=5 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 - 9999`. 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 one of the following values: size.tiny, size.small, size.normal, size.large, size.huge, or size.auto (default).

`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`.

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=5 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_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(), and box.set_text_font_family() 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.

Note that, unlike lines, the `box.*` namespace does not contain a setter function to modify a box's `xloc`. Users must create a new box with the desired `xloc` setting for such cases.

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 change in the `timeframe`, it assigns new boxes to its `upBox` and `downBox` variables, resets its `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 `timeframe`:

//@version=5 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` by retrieving the last element from the read-only box.all array, then calls box.get_top() and box.get_bottom() to get its y-coordinates. It uses these values to calculate the coordinates for a new box that's `scaleFactor` times taller than the previous:

//@version=5 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 to it once every `length` bars. On other bars, it uses box.copy() to create a `copiedBox` and calls `box.set_*()` functions to modify its properties. As shown on the chart below, these changes do not modify the `originalBox`:

//@version=5 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 specified `numberOfBoxes`, 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` 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 based on its box.get_top() value relative to the `highestTop`:

//@version=5 //@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 value 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 pushes a new chart.point 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, the script draws a new polyline on the chart, connecting the coordinates from each chart point in the array, starting from the first:

//@version=5 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=5 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 the `yValue` by a random value in the chart.point.now() calls.

To visualize the behavior, this script also creates a horizontal line at the `price` value from each chart.point 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=5 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`, we 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, it clears the `points` array, calculates the `numberOfSides` and `rotationOffset` of the new polygon drawing based on math.random() values, then uses a `for` loop to push `numberOfSides` new chart.points into the array that contain stepped coordinates from an elliptical path with `xScale` and `yScale` semi-axes. The script draws the polygon by connecting each chart.point from the `points` array using a *closed polyline* with straight line segments:

//@version=5 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 `id`, 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 IDs 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`. The script then creates a randomly colored *curved polyline* connecting the coordinates from the `points` and pushes its ID into the `polylines` array.

When the array's size exceeds the specified `numberOfSpirals`, the script removes the oldest polyline using array.shift() and deletes the object using polyline.delete():

```
//@version=5 //@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 ID 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 and assigns it to the `currentDrawing`.

When the `newPeriod` condition is `false` (i.e., the current period is not complete), the script deletes the polyline referenced by the `currentDrawing` before creating a new one, resulting in a dynamic drawing that changes over the developing period:

//@version=5 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=5 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=5 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.

Future references with `xloc.bar_index`

Objects positioned using xloc.bar_index can contain x-coordinates no further than 500 bars into the future.

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=5 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=5 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.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Non-standard charts data

Introduction

These functions allow scripts to fetch information from non-standard bars or chart types, regardless of the type of chart the script is running on. They are: ticker.heikinashi(), ticker.renko(), ticker.linebreak(), ticker.kagi() and ticker.pointfigure(). All of them work in the same manner; they create a special ticker identifier to be used as the first argument in a request.security() function call.

```
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=5 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=5 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=5 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[1]. 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=5 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[1]. 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=5 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[1] 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=5 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[1], 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=5 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)

Footnotes

1. On TradingView, Renko, Line Break, Kagi and PnF chart types are generated from OHLC values from a lower timeframe. These chart types thus represent only an approximation of what they would be like if they were generated from tick data. ↩ ↩2 ↩3 ↩4

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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.quandl() retrieves information from NASDAQ Data Link (formerly Quandl).
- request.financial() retrieves financial data from FactSet.
- request.economic() retrieves economic and industry data.
- request.seed() retrieves data from a *user-maintained* GitHub repository.

> Notice!Throughout 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 (price adjustments, session settings, non-standard chart types, etc.) 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.quandl(ticker, gaps, index, ignore_invalid_symbol) → series float

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

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

```
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.

> Notice!Users can also allow compatible scripts to evaluate their scopes in other contexts without requiring `request.*()` functions by using 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=5 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=5 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. Only unique `request.*()` calls count toward this limit because they are the only calls that fetch *new data*. Redundant calls to the same `request.*()` function with identical arguments *do not* retrieve new data. Instead, they *reuse* the data from the first executed call. See this section of the Limitations page for more information.

`gaps`

When using a `request.*()` function to retrieve data from another context, the data may 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.

> Notice!When using the timeframe parameter of the `indicator()` function to evaluate the entire script in a different timeframe, the `timeframe_gaps` parameter specifies how the script handles nonexistent values. This parameter is similar to the `gaps` parameter for `request.*()` functions.

Suppose we have a script executing on an 1-minute chart that requests hourly data for the chart's symbol using request.security(). The function call returns new values only on the 1-minute bars that cover the opening and 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.

When 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, when 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=5 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 price and currency. 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 to visualize the `marketCap` history:

```
//@version=5 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.
- This script creates a table and initializes its cell on the first chart bar, then updates the cell's text on the last chart bar. To learn more about working with tables, see the Tables page.

`currency`

The `currency` parameter of a `request.*()` function allows users to specify the currency of the requested data. When this parameter's value differs from the syminfo.currency of the symbol, the function converts the requested values to express them in the specified `currency`. This parameter accepts a built-in variable from the `currency.*` namespace, such as currency.JPY, or a "string" representing a valid currency code (e.g., "JPY"). If a script allows dynamic requests, this parameter allows a "series" argument that can change across executions. Otherwise, it accepts a value with a "simple" or weaker qualifier.

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.

> Notice!Not all `request.*()` function calls return values expressed as a currency amount. Therefore, currency conversion is *not* always necessary. For example, some series the request.financial() function returns use units other than currency, such as the "PIOTROSKI_F_SCORE" and "NUMBER_OF_EMPLOYEES" metrics. It is up to programmers to determine when currency conversion is appropriate in 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, namely 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` in request.security() references a series with 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 specified `timeframe` in the call is the same as the chart the script executes on, i.e., timeframe.period.
- The function call requests data from an intrabar timeframe, i.e., a timeframe smaller than the timeframe.period. See this section for more information.

> Notice!Using request.security() to leak future data into the past is **misleading** and **not allowed** in script publications. While your script's results on historical bars may look great due to its seemingly "magical" acquisition of prescience (which it will not be able to reproduce on realtime bars), you will mislead yourself and the users of your script. If you publish your script to share it with others, ensure you **do not mislead users** by accessing future information from a dataset 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=5 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 restarting the script's executions, as realtime bars become historical bars.
- The series that uses lookahead and a historical offset (aqua) *does not* repaint its values, as it always references the last *confirmed* value from the higher timeframe. See the Avoiding repainting section of this page for more information.

> Notice!In Pine Script v1 and v2, the `security()` function did not include a `lookahead` parameter, but it behaved as it does in later versions of Pine with `lookahead = barmerge.lookahead_on`, meaning that it systematically used data from the future HTF context on historical bars. Therefore, users should *exercise caution* with Pine v1 or v2 scripts that use HTF `security()` calls unless the function calls contain historical offsets.

Dynamic requests

A script can define all its `request.*()` functions as *dynamic* by specifying `dynamic_requests = true` in the indicator(), strategy(), or library() declaration statement. By default, dynamic requests are off.

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.

"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=5 indicator("'series' arguments demo", dynamic_requests = true) //@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.
- Removing `dynamic_requests = true` from this script's declaration statement causes a *compilation error*. Scripts without dynamic requests enabled cannot use "series string" `symbol` arguments in request.security() calls.

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=5 indicator("Invalid realtime request demo", dynamic_requests = true) //@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 and plots those values on the chart:

//@version=5 indicator("In local scopes demo", dynamic_requests = true, 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=5 indicator("Loop-dependent expression demo", dynamic_requests = true, 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=5 library("DynamicRequests", dynamic_requests = true) //@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 raises a custom runtime error when the `timeframe` argument is not higher 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 that include `dynamic_requests = true` can 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=5 indicator("Nested requests demo", dynamic_requests = true) //@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 we included `dynamic_requests = true` 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=5 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 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.

> Notice!As 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.

> Notice!When using the value from an input.source() call in the `expression` argument and the input references a series from another indicator, `request.*()` functions calculate that value's results using the **chart's symbol**, regardless of the `symbol` argument supplied, since they cannot evaluate the scopes required by an external series. We therefore do not recommend attempting to request external source input data from other contexts.

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=5 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 that raises a runtime error when the `higherTimeframe` input represents a timeframe smaller than the chart timeframe, effectively preventing the script from requesting LTF data:

//@version=5 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 can only return the results from 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=5 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. See this section of the Limitations page.

> Notice!While scripts can use request.security() to retrieve the values from a *single* intrabar on each chart bar, which might provide utility in some unique cases, we recommend using the request.security_lower_tf() function for intrabar analysis when possible, as it returns an array containing data from *all* available intrabars within a chart bar. See this section 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 calculated by the script
- Tuples
- Calls to user-defined functions
- Chart points
- Collections
- User-defined types

> Notice!The request.security() function duplicates the scopes and operations required by the `expression` to calculate its requested values in another context, which elevates runtime memory consumption. Additionally, the extra scopes produced by each call to request.security() count toward the script's *compilation limits*. See the Scope count section of the Limitations page for more information.

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 price from the daily timeframe while on an intraday chart. We can accomplish this 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 from the daily timeframe.

It's important to note that newcomers to Pine may 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 will return 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 ta.sma(close, length) as its `expression` argument to return a `length`-bar SMA from the specified timeframe:

//@version=5 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.

Calculated variables

The `expression` parameter of request.security() accepts variables declared in the global scope, allowing scripts to evaluate their variables' calculations from other contexts without redundantly listing the operations in each function call.

For example, one can declare the following variable:

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

and execute the variable's calculation from another context with request.security():

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

The function call in the line above will return the result of the `priceReturn` calculation on another `symbol`'s data as a series adapted to the current chart, which the script can display directly on the chart or utilize in additional operations.

The following example compares the price returns of the current chart's symbol and another specified `symbol`. The script declares the `priceReturn` variable from the chart's context, then uses that variable in request.security() to evaluate its calculation on another `symbol`. It then calculates the correlation between the `priceReturn` and `requestedReturn` and plots the result on the chart:

//@version=5 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, except it uses the close values fetched from the input `symbol`.
- The script colors the plot with a gradient based on the `correlation` value. To learn more about color gradients in Pine, see this section of our User Manual's page on colors.

Tuples

Tuples in Pine Script are comma-separated sets of expressions enclosed in brackets that can hold multiple values of any available type. We use tuples when creating functions or other local blocks that return more than one value.

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.

> Notice!The combined size of all tuples returned by `request.*()` calls in a script cannot exceed 127 elements. See this section of the Limitations page for more information.

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

For example, this script calculates the percent rank of the close price over `length` bars and assigns the expression to the `rank` variable. It then calls request.security() to request a tuple containing the `rank`, ta.crossover(rank, 50), and ta.crossunder(rank, 50) values from the specified `timeframe`. The script plots the `requestedRank` and uses the `crossOver` and `crossUnder` "bool" values within bgcolor() to conditionally highlight the chart pane's background:

//@version=5 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=5 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 reference types that represent coordinates on the chart. Lines, boxes, polylines, and labels use chart.point 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=5 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:

- Since this example is designed specifically for higher timeframes, we've included a custom runtime error that the script raises when the timeframe.in_seconds() of the `higherTimeframe` is smaller than that of the chart's timeframe.

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

The script creates a `data` map with "string" keys and "float" values to hold high, low, highest, and lowest price values on each bar, which it uses as the `expression` in request.security() to calculate an `otherData` map representing the `data` from the specified context. It uses the values associated with the "High", "Low", "Highest", and "Lowest" keys of the `otherData` map to calculate the `ratio` that it plots in the chart pane:

//@version=5 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 ID.

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=5 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 100,000 intrabars from a lower timeframe. See this section of our Limitations page for more information.

> Notice!Working with request.security_lower_tf() involves frequent usage of arrays since it always returns array results. We therefore recommend you familiarize yourself with arrays to make the most of this function in your scripts.

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

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

This function **only** requests data from timeframes less than or equal to the chart's timeframe. If the `timeframe` of the request represents a higher timeframe than the chart's timeframe, the function will either raise a runtime error or return na values depending on the `ignore_invalid_timeframe` argument in the call. The default value for this parameter is `false`, meaning it raises an error and halt the script's executions when attempting to request HTF data.

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 declared in the script's main scope. 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.

> Notice!As with request.security(), request.security_lower_tf() duplicates the scopes and operations required to calculate the `expression` from another context. The scopes from request.security_lower_tf() increase runtime memory consumption and count toward the script's compilation limits. See the Scope count section of the Limitations page to learn more.

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 returns its results as arrays. The elements in the returned arrays represent the `expression` values retrieved from the lower timeframe sorted in ascending order based on each intrabar's timestamp.

The type template assigned to the returned arrays corresponds to the value 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 of ta.change(close) values from the `lowerTimeframe` 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`, then plots the results on the chart alongside the `priceChange` for comparison:

//@version=5 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 100,000 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 several arrays of intrabar data with a single request.security_lower_tf() function call.

> Notice!The combined size of all tuples returned by `request.*()` calls in a script is limited to 127 elements. See this 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=5 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 may need to request the IDs of collections from an intrabar context. However, unlike request.security(), one cannot pass collections or calls to functions that return them as the `expression` argument in a request.security_lower_tf() call, as arrays cannot directly reference other collections.

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

> Notice!The use case described below is **advanced** and **not** recommended for beginners. Before exploring this approach, we recommend understanding how user-defined types and collections work in Pine Script. When possible, we recommend using *simpler* methods to manage LTF requests, and only using this approach when *nothing else* will 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 to hold an `array<float>` instance. 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 call the built-in `*.new()` function as the `expression`. For example, this line of code calls `Wrapper.new()` with array.from(close) as its `collection` within request.security_lower_tf():

//@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 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 arrays of intrabar data from a `lowerTimeframe` and uses them to display data from a specific intrabar. 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 an array of `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 assigned to that instance and calls plotcandle() to display its values on the chart:

//@version=5 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 hold an array of price `data`. type Prices array<float> data //@function Returns 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` 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 from the `intrabarPrices`, or 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 only references a `Prices` ID if the size of the `requestedPrices` array is greater than the `intrabarIndex`, as attempting to use array.get() to get an element that doesn't exist will result in an out of bounds error.
- The `intrabarData` variable only references the `data` field from `intrabarPrices` if a valid `Prices` ID exists since a script cannot reference fields of an na value.
- The process used in this example is *not* necessary to achieve the intended result. We could instead avoid using UDTs and pass an `[open, high, low, close]` tuple to the `expression` parameter to retrieve a tuple of arrays for further operations, as explained in the previous section.

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 plots the result as candles on the chart. It also highlights the background when 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=5 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=5 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=5 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=5 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.

> Notice!It's often helpful to distinguish historical bars from realtime bars when working with `request.*()` functions. Scripts can determine whether bars have historical or realtime states via 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=5 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=5 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=5 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=5 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 100,000 intrabars from an LTF context. 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=5 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.

> Notice!Scripts 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=5 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.quandl()`

TradingView forms partnerships with many fintech companies to provide users access to extensive information on financial instruments, economic data, and more. One of our many partners is Nasdaq Data Link (formerly Quandl), which provides multiple *external* data feeds that scripts can access via the request.quandl() function.

Here is the function's signature:

```
request.quandl(ticker, gaps, index, ignore_invalid_symbol) → series float
```

The `ticker` parameter accepts a "string" value representing the ID of the database published on Nasdaq Data Link and its time series code, separated by the "/" delimiter. For example, the code "FRED/DFF" represents the "Effective Federal Funds Rate" time series from the "Federal Reserve Economic Data" database.

The `index` parameter accepts an "int" value representing the *column index* of the requested data, where 0 is the first available column. Consult the database's documentaion on Nasdaq Data Link's website to see available columns.

For details on the `gaps` and `ignore_invalid_symbol` parameters, see the Common characteristics section of this page.

> Notice!The request.quandl() function can only request **free** data from Nasdaq Data Link. No data that requires a paid subscription to their services is accessible with this function. Nasdaq Data Link may change the data it provides over time, and they may not update available datasets regularly. Therefore, it's up to programmers to research the supported data available for request and review the documentation provided for each dataset. You can search for free data here.

This script requests Bitcoin hash rate ("HRATE") information from the "Bitcoin Data Insights" ("BCHAIN") database and plots the retrieved time series data on the chart. It uses color.from_gradient() to color the area plot based on the distance from the current hash rate to its all-time high:

//@version=5 indicator("Quandl demo", "BTC hash rate") //@variable The estimated hash rate for the Bitcoin network. float hashRate = request.quandl("BCHAIN/HRATE", barmerge.gaps_off, 0) //@variable The percentage threshold from the all-time highest `hashRate`. float dropThreshold = input.int(40, "Drop threshold", 0, 100) //@variable The all-time highest `hashRate`. float maxHashRate = ta.max(hashRate) //@variable The value `dropThreshold` percent below the `maxHashRate`. float minHashRate = maxHashRate * (100 - dropThreshold) / 100 //@variable The color of the plot based on the `minHashRate` and `maxHashRate`. color plotColor = color.from_gradient(hashRate, minHashRate, maxHashRate, color.orange, color.blue) // Plot the `hashRate`. plot(hashRate, "Hash Rate Estimate", plotColor, style = plot.style_area)

`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=5 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=5 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().

> Notice!The tables in these sections are quite lengthy, as 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
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

Financial	period	financial_id
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

Financial	`period`	`financial_id`
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
Operating lease liabilities	FQ, FH, FY	OPERATING_LEASE_LIABILITIES
Other common equity	FQ, FH, FY	OTHER_COMMON_EQUITY

Financial	period	financial_id
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
Purchase of investments	FQ, FH, FY, TTM	PURCHASE_OF_INVESTMENTS

Financial	period	financial_id
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=5 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.

> Notice!The tables in the sections below are rather large, as 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.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.

> Notice!The creation of *new* Pine Seeds repositories is currently **unavailable**. However, the data feeds from *existing* repositories are still accessible from charts and Pine scripts. The documentation here 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.

> Notice!As with request.security() and request.security_lower_tf(), request.seed() duplicates the scopes necessary to evaluate its `expression` in another context, which contributes toward compilation limits and script memory demands. See the Limitations page’s section on scope count limits for more information.

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=5 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.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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=5 indicator("`plot()`", "", true) plot(high, "Blue `high` line") 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 1-pixel blue line across the bar highs.
- The second plots crosses at the mid-point of bodies. The crosses are colored lime when the bar is up and purple when it is down. The argument used for `linewidth` is `6` but it is not a pixel value; just a relative size.
- The third call plots a 3-pixel wide step line following the low point of bodies.
- The fourth call plot 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 one of our conditions causes the color to change, the line will be silver. The conditions that change the color of the line require it to be higher/lower than its value two bars ago. This makes for less noisy color transitions than if we merely looked for a higher/lower value than the previous one.

This script shows other uses of plot() in a pane:

//@version=5 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 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 have calculated earlier 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 will be 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 will be plotted below the zero line.
- Finally, we plot a zero line. We could just as well have used `hline(0)` there.
- 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.

plot() calls must always be placed in a line's first position, which entails they are always in the script's global scope. They can't be placed in user-defined functions or structures like if, for, 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.

A script can only plot in its own visual space, whether it is in a pane or on the chart as an overlay. Scripts running in a pane can only color bars in the chart area.

`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; it must be converted to an “int” or a “float” for use as an argument. For example, if `newDay` is of “bool” type, then `newDay ? 1 : 0` can be used to plot 1 when the variable is `true`, and zero 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. na values will not plot as a line, but they will be bridged when a value that is not na comes in. Non-na values are only bridged if they are visible on the chart.
- plot.style_linebr: Allows the plotting of discontinuous lines by not plotting on na values, and not joining gaps, i.e., bridging over na values.
- 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`

Allows control over how many of the last bars the plotted values are visible. An “input int” argument is required, so it cannot be calculated dynamically.

`display`

The default is display.all. When it is set to display.none, plotted values will not affect the scale of the script's visual space. The plot will be invisible and will not appear in indicator values or the Data Window. It can be useful in plots destined for use as external inputs for other scripts, or for plots used with the `{{plot("[plot_title]")}}` placeholder in alertcondition() calls, e.g.:

//@version=5 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")}}')

`force_overlay`

If `true`, the plotted results will display on the main chart pane, even when the script occupies a separate pane. Optional. The default is `false`.

Plotting conditionally

plot() calls cannot be used in conditional structures such as if, but they can be controlled 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=5 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. See

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=5 indicator("", "", true) startInput = input.time(timestamp("2021-01-01")) plot(time > startInput ? close : na)

Color control

The Conditional coloring section of the page on colors 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=5 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=5 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=5 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=5 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=5 indicator("Pivot plots", "", true) pivotHigh = fixnan(ta.pivothigh(3,3)) plot(pivotHigh, "High pivot", ta.change(pivotHigh) ? 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

Pine Script has an hline() function to plot horizontal lines (see the page on Levels). hline() is useful because it has some line styles unavailable with plot(), but it also has some limitations, namely that it does not accept

“series color”, and that its `price` parameter requires an “input int/float”, so cannot vary during the script's execution.

You can plot levels with plot() in a few different ways. This shows a CCI indicator with levels plotted using plot():

//@version=5 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 shift used when the line is plotted (negative values shift in the past, positive values shift into the future). For example:

//@version=5 indicator("", "", true) plot(close, color = color.red, offset = -5) plot(close, color = color.lime, offset = 5)

As can be seen in the screenshot, the *red* series has been shifted to the left (since the argument's value is negative), while the *green* series has been shifted to the right (its value is positive).

Plot count limit

Each script is limited to a maximum plot count of 64. All `plot*()` calls and alertcondition() calls count in the plot count of a script. Some types of calls count for more than one in the total plot count.

plot() calls count for one in the total plot count if they use a “const color” argument for the `color` parameter, which means it is known at compile time, e.g.:

plot(close, color = color.green)

When they use another qualified type, such as any one of these, they will count for two in the total plot count:

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"

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=5 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=5 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.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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.

> Notice!We will not discuss the perils of using strategies on non-standard charts, as this problem is not related to repainting. See the Backtesting on Non-Standard Charts: Caution! script for a discussion of the subject.

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=5 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=5 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=5 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=5 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=5 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` contains values that fluctuate without confirmation from the `higherTimeframe`, meaning it will *repaint* when the script restarts its execution. 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=5 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:

- We used the plotshape() function to mark the chart when there's a change on the `higherTimeframe`.
- This script produces a runtime error if the `higherTimeframe` is lower than the chart's timeframe.
- On historical bars, the `repaintingClose` 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. One **cannot** be removed without compromising the integrity of the function.
- Unlike a plain request.security() call, this wrapper function cannot accept tuple `expression` arguments. For multi-element use cases, one can pass a user-defined type whose fields contain the desired elements to request.

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=5 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 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=5 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=5 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.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Sessions

Introduction

Session information is usable in three different ways in Pine Script®:

1. **Session strings** containing from-to start times and day information that can be used in functions such as time() and time_close() to detect when bars are in a particular time period, with the option of limiting valid sessions to specific days. The input.session() function provides a way to allow script users to define session values through a script's "Inputs" tab (see the Session input section for more information).
2. **Session states** built-in variables such as session.ismarket can identify which session a bar belongs to.
3. When fetching data with request.security() you can also choose to return data from *regular* sessions only or *extended* sessions. In this case, the definition of **regular and extended sessions** is that of the exchange. It is part of the instrument's properties — not user-defined, as in point #1. This notion of *regular* and *extended* sessions is the same one used in the chart's interface, in the "Chart Settings/Symbol/Session" field, for example.

The following sections cover both methods of using session information in Pine Script.

Note that:

- Not all user accounts on TradingView have access to extended session information.
- There is no special "session" type in Pine Script. Instead, session strings are of "string" type but must conform to the session string syntax.

Session strings

Session string specifications

Session strings used with time() and time_close() must have a specific format. Their syntax is:

```
<time_period>:<days>
```

Where:

- <time_period> uses times in "hhmm" format, with "hh" in 24-hour format, so `1700` for 5PM.

The time periods are in the "hhmm-hhmm" format, and a comma can separate multiple time periods to specify combinations of discrete periods.

For example, - <days> is a set of digits from 1 to 7 that specifies on which days the session is valid.

1 is Sunday, 7 is Saturday.

> Notice!**The default days are**: `1234567`, which is different in Pine Script v5 than in earlier versions where `23456` (weekdays) is used. For v5 code to reproduce the behavior from previous versions, it should explicitly mention weekdays, as in `"0930-1700:23456"`.

These are examples of session strings:

`"24x7"`

A 7-day, 24-hour session beginning at midnight.

`"0000-0000:1234567"`

Equivalent to the previous example.

`"0000-0000"`

Equivalent to the previous two examples because the default days are `1234567`.

`"0000-0000:23456"`

The same as the previous example, 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 is valid on all days of the week.

`"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 is valid Monday through Friday.

`"1000-1001:26"`

A weird 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.

Using session strings

Session properties defined with session strings are independent of the exchange-defined sessions determining when an instrument can be traded. Programmers have complete liberty in creating whatever session definitions suit their purpose, which is usually to detect when bars belong to specific time periods. This is accomplished in Pine Script by using one of the following two signatures of the time() function:

```
time(timeframe, session, timezone) → series int
```

```
time(timeframe, session) → series int
```

Here, we use time() with a `session` argument to display the market's opening high and low values on an intraday chart:

//@version=5 indicator("Opening high/low", overlay = true) sessionInput = input.session("0930-0935") sessionBegins(sess) => t = time("", sess) timeframe.isintraday and (not barstate.isfirst) and na(t[1]) and not na(t) var float hi = na var float lo = na if sessionBegins(sessionInput) hi := high lo := low plot(lo, "lo", color.fuchsia, 2, plot.style_circles) plot(hi, "hi", color.lime, 2, plot.style_circles)

Note that:

- We use a session input to allow users to specify the time they want to detect. We are only looking for the session's beginning time on bars, so we use a five-minute gap between the beginning and end time of our `"0930-0935"` default value.
- We create a `sessionBegins()` function to detect the beginning of a session. Its `time("", sess)` call uses an empty string for the function's `timeframe` parameter, which means it uses the chart's timeframe, whatever that is. The function returns `true` when:
    - The chart uses an intraday timeframe (seconds or minutes).
    - The script isn't on the chart's first bar, which we ensure with `(not barstate.isfirst)`. This check prevents the code from always detecting a session beginning on the first bar because `na(t[1]) and not na(t)` is always `true` there.
    - The time() call has returned na on the previous bar because it wasn't in the session's time period, and it has returned a value that is not na on the current bar, which means the bar is **in** the session's time period.

Session states

Three built-in variables allow you to distinguish the type of session the current bar belongs to. They are only helpful on intraday timeframes:

- session.ismarket returns `true` when the bar belongs to regular trading hours.
- session.ispremarket returns `true` when the bar belongs to the extended session preceding regular trading hours.
- session.ispostmarket returns `true` when the bar belongs to the extended session following regular trading hours.

Using sessions with `request.security()`

When your TradingView account provides access to extended sessions, you can choose to see their bars with the “Settings/Symbol/Session” field. There are two types of sessions:

- **regular** (which does not include pre- and post-market data), and
- **extended** (which includes pre- and post-market data).

Scripts using the request.security() function to access data can return extended session data or not. This is an example where only regular session data is fetched:

//@version=5 indicator("Example 1: Regular Session Data") regularSessionData = request.security("NASDAQ:AAPL", timeframe.period, close, barmerge.gaps_on) plot(regularSessionData, style = plot.style_linebr)

If you want the request.security() call to return extended session data, you must first use the ticker.new() function to build the first argument of the request.security() call:

//@version=5 indicator("Example 2: Extended Session Data") t = ticker.new("NASDAQ", "AAPL", session.extended) extendedSessionData = request.security(t, timeframe.period, close, barmerge.gaps_on) plot(extendedSessionData, style = plot.style_linebr)

Note that the previous chart's gaps in the script's plot are now filled. Also, keep in mind that our example scripts do not produce the background coloring on the chart; it is due to the chart's settings showing extended hours.

The ticker.new() function has the following signature:

```
ticker.new(prefix, ticker, session, adjustment) → simple string
```

Where:

- `prefix` is the exchange prefix, e.g., `"NASDAQ"`
- `ticker` is a symbol name, e.g., `"AAPL"`
- `session` can be `session.extended` or `session.regular`. Note that this is **not** a session string.
- `adjustment` adjusts prices using different criteria: `adjustment.none`, `adjustment.splits`, `adjustment.dividends`.

Our first example could be rewritten as:

//@version=5 indicator("Example 1: Regular Session Data") t = ticker.new("NASDAQ", "AAPL", session.regular) regularSessionData = request.security(t, timeframe.period, close, barmerge.gaps_on) plot(regularSessionData, style = plot.style_linebr)

If you want to use the same session specifications used for the chart's main symbol, omit the third argument in ticker.new(); it is optional. If you want your code to declare your intention explicitly, use the syminfo.session built-in variable. It holds the session type of the chart's main symbol:

//@version=5 indicator("Example 1: Regular Session Data") t = ticker.new("NASDAQ", "AAPL", syminfo.session) regularSessionData = request.security(t, timeframe.period, close, barmerge.gaps_on) plot(regularSessionData, style = plot.style_linebr)

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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. It also displays relevant information and simulated performance results in the dedicated Strategy Tester tab.

A simple strategy example

The following script is a simple strategy that simulates entering a long or short position when two moving averages cross. When the `fastMA` crosses above the `slowMA`, it places a “buy” market order to enter a long position. When the `fastMA` crosses below the `slowMA`, it places a “sell” market order to enter a short position:

//@version=5 strategy("Simple strategy demo", overlay = true, margin_long = 100, margin_short = 100) //@variable The length of the `fastMA` and half the length of the `slowMA`. int lengthInput = input.int(14, "Base length", 2) // Calculate two moving averages with different lengths. float fastMA = ta.sma(close, lengthInput) float slowMA = ta.sma(close, lengthInput * 2) // Place an order to enter a long position when `fastMA` crosses over `slowMA`. if ta.crossover(fastMA, slowMA) strategy.entry("buy", strategy.long) // Place an order to enter a short position when `fastMA` crosses under `slowMA`. if ta.crossunder(fastMA, slowMA) strategy.entry("sell", strategy.short) // Plot the moving averages. 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 `margin_long` and `margin_short` arguments in the strategy() call specify that the strategy must have 100% of a long or short trade’s amount available to allow the trade. See this section for more information.
- 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 or published strategy from the “Indicators, Metrics & Strategies” menu, or write a custom strategy in the Pine Editor and click the “Add to chart” option in the top-right corner:

The script plots trade markers on the main chart pane and displays simulated performance results inside the Strategy Tester 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 uses the chart's **synthetic** prices in its simulation, which do not typically represent real-world market prices, leading to unrealistic strategy 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 enabling the *"Fill orders on standard OHLC"* option in the strategy's properties or including `fill_orders_on_standard_ohlc = true` in the strategy() declaration statement.

Strategy Tester

The *Strategy Tester* visualizes the hypothetical performance of a strategy script and displays its properties. To use it, add a script declared with the strategy() function to the chart, then open the "Strategy Tester" tab. If two or more strategies are on the chart, specify which one to analyze by selecting its name in the top-left corner.

After the selected script executes across the chart's data, the Strategy Tester populates the following four tabs with relevant strategy information:

- Overview
- Performance Summary
- List of Trades
- Properties

Overview

The Overview tab provides a quick look into a strategy's performance over a sequence of simulated trades. This tab displays essential performance metrics and a chart with three helpful plots:

- The Equity baseline plot visualizes the strategy's simulated equity across closed trades.
- The Drawdown column plot shows how far the strategy's equity fell below its peak across trades.
- The Buy & hold equity plot shows the equity growth of a strategy that enters a single long position and holds that position throughout the testing range.

Note that:

- The chart has two separate vertical scales. The "Equity" and "Buy & hold equity" plots use the scale on the left, and the "Drawdown" plot uses the scale on the right. Users can toggle the plots and choose between absolute or percentage scales using the options at the bottom.
- When a user clicks on a point in this chart, the main chart scrolls to the corresponding bar where the trade closed and displays a tooltip containing the closing time.

Performance Summary

The Performance Summary tab presents an in-depth summary of a strategy's key performance metrics, organized into separate columns. The "All" column shows performance information for all simulated trades,

and the “Long” and “Short” columns show relevant metrics separately for long and short trades. This view provides more detailed insights into a strategy’s overall and directional trading performance:

List of Trades

The List of Trades tab chronologically lists a strategy’s simulated trades. Each item in the list displays vital information about a trade, including the dates and times of entry and exit orders, the names of the orders, the order prices, and the number of contracts/shares/lots/units. In addition, each item shows the trade’s profit or loss and the strategy’s cumulative profit, run-up, and drawdown:

Note that:

- Hovering the mouse over a list item’s entry or exit information reveals a “Scroll to bar” button. Clicking that button navigates the main chart to the bar where the entry or exit occurred.
- The list shows each trade in *descending* order by default, with the latest trade at the top. Users can reverse this order by clicking the “Trade #” button above the list.

Properties

The “Properties” tab provides detailed information about a strategy’s configuration and the dataset that it executes across, organized into four collapsible sections:

- The “Date Range” section shows the range of dates that had simulated trades, 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 only appears 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 strategy’s properties, including the initial capital, account currency, order size, margin, pyramiding, commission, slippage, and other settings.

Broker emulator

TradingView uses a *broker emulator* to simulate trades while running a strategy script. Unlike in real-world trading, the emulator fills a strategy’s orders exclusively using available *chart data* by default. Consequently, it executes orders on historical bars *after a bar closes*. Similarly, the earliest point that it can fill orders on realtime bars is after a new price tick. For more information about this behavior, see the Execution model page.

Because the broker emulator only uses price data from the chart by default, it makes *assumptions* about intrabar price movement when filling orders. The emulator analyzes the opening, high, low, and closing prices of chart bars to infer intrabar activity using the following logic:

- If the opening price of a bar is closer to the high than 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 the high, the emulator assumes that the market price moved in this order: **open** → **low** → **high** → **close**.
- The emulator assumes *no gaps* exist between intrabars inside each chart bar, meaning it considers *any* value within a bar’s high-low range as a valid price for order execution.

- When filling *price-based orders* (all orders except market orders), the emulator assumes intrabars **do not** exist within the gap between the previous bar's close and the current bar's open. If the market price crosses an order's price during the gap between two bars, the emulator fills the order at the current bar's *open* and not at the specified price.

Bar magnifier

Users with Premium and higher-tier plans can override the broker emulator's default assumptions about intrabar prices by enabling the Bar Magnifier backtesting mode. In this mode, the emulator uses data from *lower timeframes* to obtain more granular information about price action within bars, allowing more precise order fills in the strategy's simulation.

To enable the Bar Magnifier mode, include `use_bar_magnifier = true` in the strategy() declaration statement, or select the "Using bar magnifier" option in the "Fill orders" section of the strategy's "Settings/Properties" tab.

The following example script illustrates how the Bar Magnifier can enhance order-fill behavior. When the time value crosses the defined `orderTime`, it creates "Buy" and "Exit" limit orders at the calculated `entryPrice` and `exitPrice`. For visual reference, the script colors the background orange when it places the orders, and it draws two horizontal lines at the order prices:

//@version=5 strategy("Bar Magnifier Demo", overlay = true, use_bar_magnifier = false) //@variable The UNIX timestamp to place the order at. int orderTime = timestamp("UTC", 2023, 3, 22, 18) //@variable Is `color.orange` when `time` crosses the `orderTime`, false otherwise. color orderColor = na // Entry and exit prices. float entryPrice = hl2 - (high - low) float exitPrice = entryPrice + (high - low) * 0.25 // Entry and exit lines. var line entryLine = na var line exitLine = na if ta.cross(time, orderTime) // Draw new entry and exit lines. entryLine := line.new(bar_index, entryPrice, bar_index + 1, entryPrice, color = color.green, width = 2) exitLine := line.new(bar_index, exitPrice, bar_index + 1, exitPrice, color = color.red, width = 2) // Update order highlight color. orderColor := color.new(color.orange, 80) // Place limit orders at the `entryPrice` and `exitPrice`. strategy.entry("Buy", strategy.long, limit = entryPrice) strategy.exit("Exit", "Buy", limit = exitPrice) // Update lines while the position is open. else if strategy.position_size > 0.0 entryLine.set_x2(bar_index + 1) exitLine.set_x2(bar_index + 1) bgcolor(orderColor)

Because the script does not include a `use_bar_magnifier` argument in the strategy() function, the broker emulator uses the default assumptions when filling the orders: that the bar's price moved from open to high, high to low, and then low to close. Therefore, after filling the "Buy" order at the price indicated by the green line, the broker emulator inferred that the market price did not go back up to touch the red line and trigger the "Exit" order. In other words, the strategy *could not* enter and exit the position on the same bar according to the broker emulator's assumptions.

After we enable the Bar Magnifier mode, the broker emulator can access *10-minute* data on the 60-minute chart instead of relying on its assumptions about hourly bars. On this timeframe, the market price *did* move back up to the "Exit" order's price after reaching the "Buy" order's price in the same hour. Therefore, with the Bar Magnifier enabled in this scenario, both orders execute on the same hourly bar:

> Notice!Scripts can request a maximum of 200,000 bars from a lower timeframe. Due to this limitation, some symbols with lengthier history might *not* have intrabar coverage for their initial chart bars. Enabling the Bar Magnifier mode **does not** affect the trades on chart bars that do not have available intrabar data.

Orders and trades

Pine Script strategies use orders to make 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 after the emulator fills an order.

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=5 strategy("Order execution demo", "My strategy", true, margin_long = 100, margin_short = 100) //@function Displays the specified `txt` in a label at the `high` of the current bar. debugLabel(string txt) => label.new(bar_index, high, text = txt, color=color.lime, style = label.style_label_lower_right, textcolor = color.black, size = size.large) //@variable Is `true` on every 20th bar, `false` otherwise. bool longCondition = bar_index % 20 == 0 // Draw a label and place a long market order when `longCondition` occurs. if longCondition debugLabel("Long entry order created") strategy.entry("My Long Entry Id", strategy.long) // Place a closing market order whenever there is an open position. strategy.close_all()

Note that:

- Although the script calls strategy.close_all() on every bar, the function only creates a new exit order when 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 the broker emulator fills an order is on the next available price tick, because creating and filling an order on the same tick is unrealistic. Since strategies recalculate after each bar closes by default, the next available tick where the emulator fills a generated order is at the *open* of the *following bar*. For example, when the `longCondition` occurs on bar 20, the script places an entry order to fill on the next tick, which is at the open of bar 21. When the strategy recalculates its values after bar 21 closes, it places an order to close the current position on the next tick, which is at the open of bar 22.

Order types

Pine Script strategies can simulate different order types to suit specific trading system needs. The main notable order types include market, limit, stop, and stop-limit.

Market orders

A *market order* is the simplest type of order, which most order placement commands generate by default. A market order is an instruction to buy or sell a security as soon as possible, irrespective of the price. As such, the broker emulator always executes a market order on the next available tick.

The example below alternates between placing a long and short market order once every `lengthInput` bars. When the bar_index is divisible by `2 * lengthInput`, the strategy generates a long market order. Otherwise, it places a short market order when the bar_index is divisible by the `lengthInput`:

```
//@version=5 strategy("Market order demo", overlay = true, margin_long = 100, margin_short = 100)
//@variable Number of bars between long and short entries. int lengthInput = input.int(10, "Cycle length", 1)
//@function Displays the specified `txt` in a label on the current bar. 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, `false`
otherwise. longCondition = bar_index % (2 * lengthInput) == 0 //@variable Is `true` every `lengthInput` bars,
`false` otherwise. shortCondition = bar_index % lengthInput == 0 // Generate a long market order with a
`color.green` label on `longCondition`. 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 `color.red` label on `shortCondition`. else if shortCondition debugLabel("Short market order created",
color.red) strategy.entry("My Short Entry Id", strategy.short)
```

Note that:

- The labels indicate the bars where the script generates the market orders. The broker emulator fills each order at the open of the following bar.
- The strategy.entry() command can automatically *reverse* an open position in the opposite direction. See this section below for more information.

Limit orders

A *limit order* is an instruction to buy or sell a security 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 without waiting for the market price to reach that value.

For example, the following script generates a long limit order 800 ticks below the close of the bar 100 bars before the last chart bar using the strategy.entry() command. It draws a label to signify the bar where the strategy created the order and a line to visualize the order's price:

```
//@version=5 strategy("Limit order demo", overlay = true, margin_long = 100, margin_short = 100)
//@function Displays text passed to `txt` and a horizontal line at `price` when called. 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 ) // Generate a long limit order with a label and
line 100 bars before the `last_bar_index`. if last_bar_index - bar_index == 100 limitPrice = close -
syminfo.mintick * 800 debugLabel(limitPrice, "Long Limit order created") strategy.entry("Long", strategy.long,
limit = limitPrice)
```

Notice that in the chart above, the label and the start of the line occurred several bars before the "Long" entry marker. The broker emulator could not fill the order while the market price remained *above* the `limitPrice` because such a price is a *worse* value for the long trade. After the price fell and reached the `limitPrice`, the emulator filled the order mid-bar at that value.

If we set the `limitPrice` to a value *above* the bar's close rather than *below*, the broker emulator fills the order at the open of the following bar because the closing price is already a more *favorable* value for the

long trade. Here, we set the `limitPrice` in the script to 800 ticks above the bar's close to demonstrate this effect:

```
//@version=5 strategy("Limit order demo", overlay = true, margin_long = 100, margin_short = 100) //@function Displays text passed to `txt` and a horizontal line at `price` when called. 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 ) // Generate a long limit order with a label and line 100 bars before the `last_bar_index`. if last_bar_index - bar_index == 100 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.

When 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 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 created the order and a line to display the stop price. As we see in the chart below, the strategy entered a long position immediately after the price crossed the stop level:

```
//@version=5 strategy("Stop order demo", overlay = true, margin_long = 100, margin_short = 100) //@function Displays text passed to `txt` when called and shows the `price` level on the chart. debugLabel(price, 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 ) // Generate a long stop order with a label and lines 100 bars before the last bar. if last_bar_index - bar_index == 100 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 previous section for an example.

When 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 includes additional drawings to show where the strategy activated the subsequent limit order and to visualize the limit price.

In this example chart, notice how the market price reached the limit level on the next bar after the stop-limit order was created, but the strategy did not enter a position because the limit order was not yet active. After

price later reached the stop level, the strategy placed the limit order, and then the broker emulator filled it after the market price dropped back down to the limit level:

```
//@version=5 strategy("Stop-Limit order demo", overlay = true, margin_long = 100, margin_short = 100) //@function Displays text passed to `txt` when called and shows the `price` level on the chart. 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 ) var float stopPrice = na var float limitPrice = na // Generate a long stop-limit order with 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 a 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 segments 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 with 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 strategy.entry() executes while there is an open position 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 and 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. When the `buyCondition` occurs once every 100 bars, the script calls strategy.entry() with `qty = 15` to open a long position of 15 shares. Otherwise, when the `sellCondition` occurs on every 50th bar, the script calls strategy.entry() with `qty = 5` to enter a new short

position of five shares. The script also highlights the chart's background on the bars where the `buyCondition` and `sellCondition` occurs:

```
//@version=5 strategy("Reversing positions demo", overlay = true) //@variable Is `true` on every 100th bar, `false` otherwise. bool buyCondition = bar_index % 100 == 0 //@variable Is `true` on every 50th bar, `false` otherwise. bool sellCondition = bar_index % 50 == 0 if buyCondition // Place a "buy" market order to close the 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 the 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` occurs. bgcolor(buyCondition ? color.new(color.blue, 90) : sellCondition ? color.new(color.red, 90) : na)
```

The trade markers on the chart show the *transaction size*, not the size of the resulting position. The markers above show that the transaction size was *20 shares* on each order fill rather than 15 for long orders and five for short orders. Since strategy.entry() reverses a position in the opposite direction by default, each call *adds* the open position's size (e.g., 15 for long entries) to the new order's size (e.g., 5 for short entries), resulting in a quantity of 20 shares on each entry after the first. Although each of these *transactions* is 20 shares in size, the resulting positions are 5 shares for each short entry and 15 for each long entry.

Note that:

- The strategy.risk.allow_entry_in() function *overrides* the allowed direction for the strategy.entry() command. When a script specifies a trade direction with this risk management command, orders from strategy.entry() in the opposite direction *close* the open position without allowing a reversal.

Pyramiding

Another unique characteristic of the strategy.entry() command is its connection to a strategy's *pyramiding* property. Pyramiding specifies the maximum number of *successive entries* a strategy allows in the same direction. Users can set this property by including a `pyramiding` argument in the strategy() declaration statement or by adjusting the "Pyramiding" input in the script's "Settings/Properties" tab. The default value is 1, meaning the strategy can open new positions but cannot add to them using orders from strategy.entry() calls.

The following example uses strategy.entry() to place a market order when the `entryCondition` occurs on every 25th bar. The direction of the orders changes once every 100 bars, meaning every 100-bar cycle includes *four* strategy.entry() calls with the same direction. For visual reference of the conditions, the script highlights the chart's background based on the current direction each time the `entryCondition` occurs:

```
//@version=5 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, `false` otherwise. bool entryCondition = bar_index % 25 == 0 // Change the `direction` on every 100th bar. if bar_index % 100 == 0 direction *= -1 // Place a market order based on the current `direction` when the `entryCondition` occurs. if entryCondition strategy.entry("Entry", direction == 1 ? strategy.long : strategy.short) //@variable When the `entryCondition` 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`. bgcolor(bgColor, title = "Background highlight")
```

Notice that although the script calls strategy.entry() with the same direction four times within each 100-bar cycle, the strategy *does not* execute an order after every call. It cannot open more than one trade per position with strategy.entry() because it uses the default pyramiding value of 1.

Below, we modified the script by including `pyramiding = 4` in the strategy() declaration statement to allow up to four successive trades in the same direction. Now, an order fill occurs after every strategy.entry() call:

//@version=5 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, `false` otherwise. bool entryCondition = bar_index % 25 == 0 // Change the `direction` on every 100th bar. if bar_index % 100 == 0 direction *= -1 // Place a market order based on the current `direction` when the `entryCondition` occurs. if entryCondition strategy.entry("Entry", direction == 1 ? strategy.long : strategy.short) //@variable When the `entryCondition` 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`. bgcolor(bgColor, title = "Background highlight")

`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 with the `limit` and `stop` parameters. Orders from strategy.order() 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 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 highlights the background to signify where the strategy places a "buy" or "sell" order:

//@version=5 strategy("`strategy.order()` demo", overlay = true) //@variable Is `true` on every 100th bar, `false` otherwise. bool buyCondition = bar_index % 100 == 0 //@variable Is `true` on every 25th bar, `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` occurs. bgcolor(buyCondition ? color.new(color.blue, 90) : sellCondition ? color.new(color.red, 90) : na)

This particular strategy never simulates a *short position*. Unlike the strategy.entry() command, strategy.order() *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 strategy.entry() instead of the strategy.order() command in this example, the strategy would alternate between entering long and short positions 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` value 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* 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.

The absolute parameters of the strategy.exit() command *supersede* the relative ones. If a strategy.exit() call contains `profit` and `limit` arguments, it creates a take-profit order at the `limit` price and ignores the `profit` value. Likewise, if the call includes `loss` and `stop` arguments, the command ignores the `loss` value and places a stop-loss order at the `stop` price.

> Notice!The 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` occurs, the script calls strategy.entry() to place a "buy" market order. It also calls strategy.exit() with `limit` and `stop` arguments to create a take-profit order at the `limitPrice` and a stop-loss order at the `stopPrice`. The script plots the `limitPrice` and `stopPrice` values on the chart to visualize the exit order prices:

//@version=5 strategy("Take-profit and stop-loss demo", overlay = true) //@variable Is `true` on every 100th bar. bool buyCondition = bar_index % 100 == 0 //@variable The current take-profit order price. var float takeProfit = na //@variable 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` price and a stop-loss order at the `stopLoss` price. strategy.exit("exit", "buy", limit = takeProfit, stop = stopLoss) // Set `takeProfit` and `stopLoss` to `na` when the position closes. if ta.change(strategy.closedtrades) > 0 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)

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.

When a strategy.exit() call includes a `from_entry` argument, the resulting exit orders only apply to existing entry orders that have a matching ID. If the specified `from_entry` value does not match the ID of any entry 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", which means it creates exit orders only for open trades with the "buy2" entry ID. This version does not place *any* exit orders because it does not create any entry orders with the "buy2" ID:

//@version=5 strategy("Invalid `from_entry` ID demo", overlay = true) //@variable Is `true` on every 100th bar. bool buyCondition = bar_index % 100 == 0 //@variable The current take-profit order price. var float takeProfit = na //@variable The current stop-loss order price. var float stopLoss = na if buyCondition // Update the `takeProfit` and `stopLoss` values before entering the trade. 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 `takeProfit` and `stopLoss` to `na` when the position closes. if ta.change(strategy.closedtrades) > 0 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)

Note that:

- When a strategy.exit() call *does not* include a `from_entry` argument, it creates exit orders for *all* the position's open trades, 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 strategy.exit() to create successive *partial* exit orders for the same entry ID, helping to simplify the formation of multi-level exit strategies. To use multiple strategy.exit() calls to exit from an open trade, include a `qty` or `qty_percent` argument in each call to specify how much 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 position.

Note that:

- When 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.

This example demonstrates a simple strategy that creates two partial exit order brackets for an entry ID. When the `buyCondition` occurs, the script places a “buy” market order for two shares with strategy.entry(), and it creates “exit1” and “exit2” brackets using two calls to strategy.exit(). The first call uses a `qty` of 1, and the second uses a `qty` of 3:

//@version=5 strategy("Multi-level exit demo", "test", overlay = true) //@variable Is `true` on every 100th bar. 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 before entering the trade. 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` of 2. strategy.entry("buy", strategy.long, qty = 2) // Place an "exit1" bracket with a `qty` of 1 at the `takeProfit1` and `stopLoss1` prices. strategy.exit("exit1", "buy", limit = takeProfit1, stop = stopLoss1, qty = 1) // Place an "exit2" bracket with a `qty` of 3 at the `takeProfit1` and `stopLoss1` prices. // The size of the resulting orders decreases to match the open position. strategy.exit("exit2", "buy", limit = takeProfit2, stop = stopLoss2, qty = 3) // Set `takeProfit1` and `stopLoss1` to `na` when the price touches either value. if high >= takeProfit1 or low <= stopLoss1 takeProfit1 := na stopLoss1 := na // Set `takeProfit2` and `stopLoss2` to `na` when the price touches either value. if high >= takeProfit2 or low <= stopLoss2 takeProfit2 := na stopLoss2 := na // Plot the `takeProfit*` and `stopLoss*` values. 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 one remaining share in the position. However, we specified a size of *three shares* for the “exit2” order bracket, 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:

- This strategy only fills **one** exit order from the “exit1” bracket, **not both**. When 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 reduced the “exit2” orders because all orders from the strategy.exit() calls automatically belong to the same strategy.oca.reduce group by default. Learn more about OCA groups below.

When creating multiple exit orders with *different* strategy.exit() calls, it’s crucial to note that the orders from each call *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, this script generates a “buy” entry order for 20 shares with a strategy.entry() call and “limit” and “stop” exit orders with two separate calls to strategy.exit() 100 bars before the last chart bar. We specified a quantity of 19 shares for the “limit” order and 20 for the “stop” order:

//@version=5 strategy("Reserved exit demo", "test", overlay = true) //@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. bool longCondition = last_bar_index - bar_index == 100 if longCondition // Update the `limitPrice` and `stopPrice`. 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`. strategy.exit("limit", limit = limitPrice, qty = 19) // Create a stop-loss order at the `stopPrice`. Although this call specifies a `qty` of 20, the previous // `strategy.exit()` call reserved 19, meaning this call creates an exit order for only 1 share. strategy.exit("stop", stop = stopPrice, qty = 20) //@variable Is `true` when the strategy has an open position, `false` otherwise. bool showPlot = strategy.opentrades == 1 // Plot the `limitPrice` and `stopPrice` when `showPlot` 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 unfamiliar with the strategy.exit() command's unique behaviors might expect this strategy to close the entire market position if it fills the "stop" order before the "limit" order. However, the trade markers in the chart below show that the "stop" order only reduces the position by **one share**. The strategy.exit() call for the "limit" order executes first in the code, reserving 19 shares of the open position for closure with that order. 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 better value in the favorable direction (upward for long positions and downward for short positions).

This type of exit order has two components: an *activation level* and a *trail offset*. The activation level is the value the market price must cross to activate the trailing stop calculation, and the trail offset is the distance the activated stop follows behind the price as it reaches successively better values.

Three strategy.exit() parameters determine the activation level and 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 include 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 `trail_price` for the activation level and ignores the `trail_points` value.

The example below demonstrates how a trailing stop order works. The strategy places a "Long" market order with the strategy.entry() command 100 bars before the last chart bar, and it calls strategy.exit() 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 the market price must reach to activate the trailing stop order. After the price crosses this level, 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 distance of `trailOffsetInput` ticks from the best value. The exit order *does not* change its price level when the market price decreases or does not reach a new high. Eventually, the market price crosses below the trailing stop, triggering an exit:

//@version=5 strategy("Trailing stop order demo", overlay = true, margin_long = 100, margin_short = 100) //@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` 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`. debugDrawings(float price, string txt, color drawingColor, bool drawLine = false) => // Draw a label showing the `txt` at the `price` on the current bar. label.new(bar_index, price, text = txt, color = drawingColor, textcolor = color.white, style = label.style_label_lower_right, size = size.large) // Draw a horizontal line at the `price` starting from the current bar when `drawLine` is `true`. 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`. activationLevel := open + syminfo.mintick * activationOffsetInput // Create the trailing stop order that activates at the `activationLevel` and trails behind the `high` by // `trailOffsetInput` ticks. strategy.exit("Trailing Stop", from_entry = "Long", trail_price = activationLevel, trail_offset = trailOffsetInput) // Create drawings to signify 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` for the first time to show 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` reaches a new high. else if theoreticalStopPrice > trailingStop trailingStop := theoreticalStopPrice // Plot the `trailingStop` value to visualize the trailing 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 call to strategy.exit() with that ID as the `from_entry` argument places exit orders for each corresponding entry created before or on the bar where the call occurs.

For example, this script periodically calls strategy.entry() 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, the script calls strategy.exit() once with "buy" as its `from_entry` argument to generate separate exit orders for each entry with that ID. When the market price reaches the `takeProfit` or `stopLoss` value, the broker emulator fills *two* exit orders and closes the position:

//@version=5 strategy("Exits for entries with the same ID demo", overlay = true, pyramiding = 2) //@variable Take-profit price for exit commands. var float takeProfit = na //@variable Stop-loss price for exit commands. var float stopLoss = na //@variable Is `true` on two consecutive bars in 100-bar cycles. 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 on the second order. if strategy.opentrades == 1 // Update the `takeProfit` and `stopLoss`. takeProfit := close * 1.01 stopLoss :=

close * 0.99 // Place exit orders for both "buy" entries. strategy.exit("exit", "buy", limit = takeProfit, stop = stopLoss) // Set `takeProfit` and `stopLoss` to `na` when 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, when it does not include a `from_entry` argument.

Here, 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. Since this version does not specify which entries the exit orders apply to, the strategy.exit() call creates orders for *every* entry in the position:

//@version=5 strategy("Exits for entries with different IDs demo", overlay = true, pyramiding = 2) //@variable Take-profit price for exit commands. var float takeProfit = na //@variable Stop-loss price for exit commands. var float stopLoss = na //@variable Is `true` on two consecutive bars in 100-bar cycles. 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 on the second order. if strategy.opentrades == 1 // Update the `takeProfit` and `stopLoss`. 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 `takeProfit` and `stopLoss` to `na` when 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. When 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 creates a long entry order with strategy.entry() on each bar within a user-specified time range, and it calls strategy.exit() without a `from_entry` argument on *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 an exit order fills each time the market price is not above an entry order's price.

The script prompts users 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=5 strategy("Exit persist demo", overlay = true, margin_long = 100, margin_short = 100, 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, which allows the position to have up to 100 open entries from strategy.entry().
- The script uses labels and bgcolor() 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 position.

We can observe the unique strategy.exit() behavior in this example by comparing the code itself with the script's chart outputs. The script calls strategy.exit() *one time*, only on the bar with the blue label. However, this single call placed exit orders for every entry **before** or on that bar and continued placing exit orders for all entries **after** that bar. This behavior occurs because strategy.exit() has no way to 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 after the position fully closes.

The above script would exhibit different behavior if we included a `from_entry` argument in the strategy.exit() call. When a call to this command specifies a `from_entry` ID, it only applies to entries with that ID which the strategy created *before* or *on* the bar of the call. The command does not place exit orders for subsequent entries created *after* that bar in that case, even ones with the same ID.

Here, we added `from_entry = "Entry"` to our script's strategy.exit() call, meaning it only produces exit orders for entries with the "Entry" ID. Only 17 exits occur this time, each corresponding to an entry order created before or on the bar with the blue label. The call does not affect any entries that the strategy creates *after* that bar:

```
//@version=5 strategy("Exit persist demo", overlay = true, margin_long = 100, margin_short = 100, 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 below demonstrates a simple strategy that places a “buy” entry order with strategy.entry() once every 50 bars and a market order to close the long position with strategy.close() 25 bars afterward:

```
//@version=5 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 exit from all of them.

The following script creates a “buy” order with strategy.entry() once every 25 bars, and it calls strategy.close() with “buy” as its `id` argument to close all open trades with that entry ID once every 100 bars. The market order from strategy.close() closes the entire position in this case because every open trade has the same “buy” entry ID:

//@version=5 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 with strategy.entry() calls.

The strategy.close_all() command generates a market order to exit from the open position that *does not* link to any specific entry ID. This command is helpful when a strategy needs to 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, and then it calls strategy.close_all() to create a single order that closes the entire position on the next bar:

//@version=5 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() and strategy.order() calls that use `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 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 we see 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 it reaches that level:

//@version=5 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 affects *all* unfilled orders with a specified ID. It does nothing if the specified `id` represents the ID of an order that does not exist. When there is more than one unfilled order with the specified ID, the command cancels *all* of them at once.

Below, we've 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 strategy.cancel() with "buy" as the `id` argument, resulting in nothing happening when the market price reaches any of the order prices (horizontal lines):

//@version=5 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() per position. The script would also achieve the same result without this setting if it called strategy.order() instead because pyramiding *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. If the call happens after that point, it has *no effect* because the broker emulator fills market orders on the *next available tick*.

This example places a "buy" market order 100 bars before the last chart bar with strategy.entry(), then it attempts to cancel the order on the next bar with strategy.cancel_all(). The cancellation command *does not* affect the "buy" order because the broker emulator fills the order on the next bar's *opening tick*, which occurs *before* the script evaluates the strategy.cancel_all() call:

//@version=5 strategy("Cancel market demo", "test", 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 manage 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 values with the “Order size” inputs in the “Settings/Properties” tab.
- Include a *non-na* `qty` argument in the strategy.entry() or strategy.order() call. When 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` values 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=5 strategy("Buy low, sell high", 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 because the specified `qty` in the entry commands takes precedence. If we want to use the default size, we must *remove* the `qty` arguments from the strategy.entry() calls or set their values to na.

Here, we edited the previous script by including ternary expressions for the `qty` arguments in both strategy.entry() calls that replace input values of 0 with na. If the specified `longAmount` or `shortAmount` is 0, which is what we set as the new default, the corresponding entry orders use the strategy’s default order size instead, as we see below:

//@version=5 strategy("Buy low, sell high", 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, which means that 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 places “Buy1” and “Buy2” entry orders sequentially, starting 100 bars before the latest chart bar. When the position size is 0, it calls strategy.entry() to place the “Buy1” order for five units. After the strategy’s position size matches the size of that order, it uses strategy.entry() to place 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 plots the strategy.position_size value in a separate pane:

//@version=5 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 == 0 ? na : 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() 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 strategy.close() with "Buy2" as its `id` argument, and it includes "Buy1" as the `from_entry` argument in the strategy.exit() call. The market order from strategy.close() executes on the next available tick, meaning the broker emulator fills it *before* the take-profit and stop-loss orders from strategy.exit():

//@version=5 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 == 0 ? na : 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 "List of Trades" tab shows that five units of the order go toward closing the "Buy1" trade *first* because it is the oldest, and the remaining five units close *half* of the "Buy2" trade. After that, the "bracket" orders from the strategy.exit() call close the rest of the position:

Note that:

- If we included `close_entries_rule = "ANY"` in the strategy() declaration statement, the market order from strategy.close() would close the open trade with the "Buy2" entry ID *first*, and then the "bracket" orders from strategy.exit() 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.

> Notice!All 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` 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`

When 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 demonstrate 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 is 0 when the cross occurs, the strategy places two stop orders with 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 strategy.close_all() to close the position with a market order:

//@version=5 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 evaluating strategy.close_all() because both orders have the same size. We see this behavior in the chart below, where the strategy alternated between executing "Long" and "Short" orders a few times without executing an order from strategy.close_all():

To eliminate scenarios where 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 it executes 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 strategy executes either the "Long" or "Short" order, it cancels the other order and waits for strategy.close_all() to close the position:

//@version=5 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`

When 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, which 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 strategy.entry() with `qty = 6` to create an entry order, and then it uses three strategy.order() calls to create a stop order at the `stop` price and two limit orders at the `limit1` and `limit2` prices. The strategy.order() call for the "Stop" order uses `qty = 6`, and the two calls for the "Limit 1" and "Limit 2" orders both use `qty = 3`:

```
//@version=5 strategy("Multiple TP Demo", overlay = true) 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 it does not work as initially intended. The problem with this script is that the orders from strategy.order() **do not** belong to an OCA group by default (unlike strategy.exit(), whose orders automatically belong to a strategy.oca.reduce OCA group). Since the strategy does not assign the strategy.order() calls to any OCA group, it does not reduce any unfilled stop or limit orders after executing an order. 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` and strategy.oca.reduce as the `oca_type` 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=5 strategy("Multiple TP Demo", overlay = true) 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 its amount by three units when it executes the “Limit 1” order. Keeping the `qty` value of 3 would cause the second limit order’s size to drop to 0 after the strategy fills the first limit order, meaning it would never execute.

`strategy.oca.none`

When 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` for the strategy.order() and strategy.entry() commands.

Currency

Pine Script strategies can use different currencies in their calculations than the instruments they simulate trades on. Programmers can specify a strategy's account currency by including a `currency.*` variable as the `currency` argument in the strategy() declaration statement. The default value is currency.NONE, meaning the strategy uses the same currency as the current chart (syminfo.currency). Script users can change the account currency using the "Base currency" input in the script's "Settings/Properties" tab.

When 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. If no exchange provides the rate directly, it derives the rate using a spread symbol. The strategy multiplies all monetary values, including simulated profits/losses, by the determined cross rate to express them in the account currency. To retrieve the rate that a strategy uses to convert monetary values, call request.currency_rate() with syminfo.currency as the `from` argument and strategy.account_currency as the `to` argument.

Note that:

- Programmers can directly convert values expressed in a strategy's account currency to the chart's currency and vice versa via 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 a strategy's cross-rate calculations match those that `request.*()` functions use.

On each of the latest 500 bars, the strategy places an entry order with strategy.entry(), and it places a take-profit and stop-loss order one tick away from the entry price with strategy.exit(). The size of each entry order is `1.0 / syminfo.mintick`, rounded to the nearest tick, which means that the profit/loss of each closed trade is equal to *one point* in the chart's *quote currency*. We specified currency.EUR as the account currency in the strategy() declaration statement, meaning the strategy multiplies all monetary values by a cross rate to express them in Euros.

The script calculates the absolute change in the ratio of the strategy's net profit to the symbol's point value to determine the value of *one unit* of the chart's currency in Euros. It plots this value alongside the result from a request.currency_rate() call that uses syminfo.currency and strategy.account_currency as the `from` and `to` arguments. As we see below, both plots align, confirming that strategies and `request.*()` functions use the *same* daily cross-rate calculations:

//@version=5 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:

- When 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 for its cross-rate calculations. For example, on a "1W" chart,

the strategy bases its cross rate on the previous Thursday's closing values. However, it still uses the latest confirmed daily rate on *realtime* bars.

Altering calculation behavior

Strategy scripts execute across all available historical chart bars and continue to execute on realtime bars as new data comes in. However, by default, strategies only recalculate their values after a bar *closes*, even on realtime bars, and the earliest point that the broker emulator fills the orders a strategy places on the close one bar is at the *open* of the following bar.

Users can change these behaviors with the `calc_on_every_tick`, `calc_on_order_fills`, and `process_orders_on_close` parameters of the strategy() declaration statement or the corresponding inputs in the "Recalculate" and "Fill orders" sections of the script's "Settings/Properties" tab. The sections below explain how these settings affect a strategy's calculations.

`calc_on_every_tick`

The `calc_on_every_tick` parameter of the strategy() function determines the frequency of a strategy's calculations on *realtime bars*. When this parameter's value is `true`, the script recalculates on each *new tick* in the realtime data feed. Its default value is `false`, meaning the script only executes on a realtime bar after it closes. Users can also toggle this recalculation behavior with the "On every tick" input in the script's "Settings/Properties" tab.

Enabling this setting can be useful in forward testing because it allows a strategy to use realtime price updates in its calculations. However, it *does not* affect the calculations on historical bars because historical data feeds *do not* contain complete tick data: the broker emulator considers each historical bar to have only four ticks (open, high, low, and close). Therefore, users should exercise caution and understand the limitations of this setting. If enabling calculation on every tick causes a strategy to behave *differently* on historical and realtime bars, the strategy will **repaint** after the user reloads it.

The following example demonstrates how recalculation on every tick can cause strategy repainting. The script uses strategy.entry() calls to place a long entry order each time the close reaches its `highest` value and a short entry order each time the close reaches its `lowest` value. The strategy() declaration statement includes `calc_on_every_tick = true`, meaning that on realtime bars, it can recalculate and place orders on new price updates *before* a bar closes:

//@version=5 strategy("Donchian Channel Break", overlay = true, calc_on_every_tick = true, pyramiding = 20) int length = input.int(15, "Length") float highest = ta.highest(close, length) float lowest = ta.lowest(close, length) if close == highest strategy.entry("Buy", strategy.long) if close == lowest strategy.entry("Sell", strategy.short) // Highlight the background of realtime bars. bgcolor(barstate.isrealtime ? color.new(color.orange, 80) : na) plot(highest, "Highest", color = color.lime) plot(lowest, "Lowest", color = color.red)

Note that:

- The script uses a pyramiding value of 20, allowing it to simulate up to 20 entries per position with the strategy.entry() command.
- The script highlights the chart's background orange when barstate.isrealtime is `true` to indicate realtime bars.

After applying the script to our chart and letting it run on several realtime bars, we see the following output:

The script placed a “Buy” order on *each tick* where the close was at the `highest` value, which happened *more than once* on each realtime bar. Additionally, the broker emulator filled each market order at the current realtime price rather than strictly at the open of the following chart bar.

After we reload the chart, we see that the strategy *changed* its behavior and *repainted* its results on those bars. This time, the strategy placed only *one* “Buy” order for each *closed bar* where the condition was valid, and the broker emulator filled each order at the open of the following bar. It did not generate multiple entries per bar because what were previously realtime bars became *historical* bars, which **do not** hold complete tick data:

`calc_on_order_fills`

The `calc_on_order_fills` parameter of the strategy() function enables a strategy to recalculate immediately after an *order fills*, allowing it to use more granular information and place additional orders without waiting for a bar to close. Its default value is `false`, meaning the strategy does not allow recalculation immediately after every order fill. Users can also toggle this behavior with the “After order is filled” input in the script’s “Settings/Properties” tab.

Enabling this setting can provide a strategy script with additional data that would otherwise not be available until after a bar closes, such as the current average price of a simulated position on an open bar.

The example below shows a simple strategy that creates a “Buy” order with strategy.entry() whenever the strategy.position_size is 0. The script uses strategy.position_avg_price to calculate price levels for the strategy.exit() call’s stop-loss and take-profit orders that close the position.

We’ve included `calc_on_order_fills = true` in the strategy() declaration statement, meaning that the strategy recalculates each time the broker emulator fills a “Buy” or “Exit” order. Each time an “Exit” order fills, the strategy.position_size reverts to 0, triggering a new “Buy” order. The broker emulator fills the “Buy” order on the next tick at one of the bar’s OHLC values, and then the strategy uses the recalculated strategy.position_avg_price value to determine new “Exit” order prices:

//@version=5 strategy("Intrabar exit", 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 if strategy.position_size == 0.0 strategy.entry("Buy", strategy.long) 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:

- Without enabling recalculation on order fills, this strategy would not place new orders *before* a bar closes. After an exit, the strategy would wait for the bar to close before placing a new “Buy” order, which the broker emulator would fill on the *next tick* after that, i.e., the open of the following bar.

It’s important to note that enabling `calc_on_order_fills` can produce unrealistic strategy results in some cases because the broker emulator may assume order-fill prices that are *not* obtainable in real-world trading. Therefore, users should exercise caution and carefully examine their strategy logic when allowing recalculation on order fills.

For example, the following script places a “Buy” order after each new order fill and bar close over the most recent 25 historical bars. The strategy simulates *four* entries per bar because the broker emulator considers each historical bar to have *four ticks* (open, high, low, and close). This behavior is unrealistic because it is not typically possible to fill an order at a bar’s *exact* high or low price:

//@version=5 strategy("buy on every fill", overlay = true, calc_on_order_fills = true, pyramiding = 100) if last_bar_index - bar_index <= 25 strategy.entry("Buy", strategy.long)

`process_orders_on_close`

By default, strategies simulate orders at the close of each bar, meaning that the earliest opportunity to fill the orders and execute strategy calculations and alerts is on the opening of the following bar. Programmers can change this behavior to process orders on the *closing tick* of each bar by setting `process_orders_on_close` to `true` in the strategy() declaration statement. Users can set this behavior by changing the “Fill Orders/On Bar Close” setting in the “Settings/Properties” tab.

This behavior is most useful when backtesting manual strategies in which traders exit from a position before a bar closes, or in scenarios where algorithmic traders in non-24x7 markets set up after-hours trading capability so that alerts sent after close still have hope of filling before the following day.

Note that:

- Using strategies with `process_orders_on_close` enabled to send alerts to a third-party service might cause unintended results. Alerts on the close of a bar still occur after the market closes, and real-world orders based on such alerts might not fill until after the market opens again.
- The strategy.close() and strategy.close_all() commands feature an `immediately` parameter that, if `true`, allows the resulting market order to fill on the same tick where the strategy created it. This parameter provides an alternative way for programmers to selectively apply `process_orders_on_close` behavior to closing market orders without affecting the behavior of other order placement commands.

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 a broker/exchange charges when executing trades. Commission can be a flat fee per trade or contract/share/lot/unit, or a percentage of the total transaction value. Users can set the commission properties of their strategies by including `commission_type` and `commission_value` arguments in the strategy() function, or by setting the “Commission” inputs in the “Properties” tab of the strategy settings.

The following script is a simple strategy that simulates a “Long” position of 2% of equity when `close` equals the `highest` value over the `length`, and closes the trade when it equals the `lowest` value:

//@version=5 strategy("Commission Demo", overlay=true, default_qty_value = 2, default_qty_type = strategy.percent_of_equity) length = input.int(10, "Length") float highest = ta.highest(close, length) float lowest = ta.lowest(close, length) switch close highest => strategy.entry("Long", strategy.long) lowest =>

strategy.close("Long") plot(highest, color = color.new(color.lime, 50)) plot(lowest, color = color.new(color.red, 50))

The results in the Strategy Tester show that the strategy had a positive equity growth of 17.61% over the testing range. However, the backtest results do not account for fees the broker/exchange may charge. Let's see what happens to these results when we include a small commission on every trade in the strategy simulation. In this example, we've included `commission_type = strategy.commission.percent` and `commission_value = 1` in the strategy() declaration, meaning it will simulate a commission of 1% on all executed orders:

//@version=5 strategy("Commission Demo", overlay=true, default_qty_value = 2, default_qty_type = strategy.percent_of_equity, commission_type = strategy.commission.percent, commission_value = 1) length = input.int(10, "Length") float highest = ta.highest(close, length) float lowest = ta.lowest(close, length) switch close highest => strategy.entry("Long", strategy.long) lowest => strategy.close("Long") plot(highest, color = color.new(color.lime, 50)) plot(lowest, color = color.new(color.red, 50))

As we can see in the example above, after applying a 1% commission to the backtest, the strategy simulated a significantly reduced net profit of only 1.42% and a more volatile equity curve with an elevated max drawdown. These results highlight the impact that commission can have on a strategy's hypothetical performance.

Slippage and unfilled limits

In real-life trading, a broker/exchange may fill orders at slightly different prices than a trader intended, due to volatility, liquidity, order size, and other market factors, which can profoundly impact a strategy's performance. The disparity between expected prices and the actual prices at which the broker/exchange executes trades is what we refer to as *slippage*. Slippage is dynamic and unpredictable, making it impossible to simulate precisely. However, factoring in a small amount of slippage on each trade during a backtest or forward test might help the results better align with reality. Users can model slippage in their strategy results, sized as a fixed number of *ticks*, by including a `slippage` argument in the strategy() declaration statement or by setting the "Slippage" input in the "Settings/Properties" tab.

The following example demonstrates how simulating slippage affects 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 and closes the position when the price dips below the EMA while it's falling. We've included `slippage = 20` in the strategy() function, which declares that the price of each simulated order will slip 20 ticks in the direction of the trade.

The script uses strategy.opentrades.entry_bar_index() and strategy.closedtrades.exit_bar_index() to get the `entryIndex` and `exitIndex`, which it uses to obtain the `fillPrice` of the order. When the bar index is at the `entryIndex`, the `fillPrice` is the first strategy.opentrades.entry_price() value. At the `exitIndex`, `fillPrice` is the strategy.closedtrades.exit_price() value from the last closed trade. The script plots the expected fill price along with the simulated fill price after slippage to visually compare the difference:

//@version=5 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 = 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(fillPrice ? open : na, "Expected fill price", "—", location.absolute, expectedColor) plotchar(fillPrice, "Fill price after slippage", "—", location.absolute, filledColor)

Note that:

- Since the strategy applies constant slippage to all order fills, some orders can fill *outside* the candle range in the simulation. Exercise 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, as unlike market orders, they cannot execute at a worse price than the specified value. However, even if the market price reaches an order's price, there's a chance that a limit order might not fill, depending on the state of the real-life market, because limit orders can only fill if a security has sufficient liquidity and price action around their values. To account for the possibility of *unfilled* orders in a backtest, users can specify the `backtest_fill_limits_assumption` value in the declaration statement or use the "Verify price for limit orders" input in the "Settings/Properties" tab. This setting instructs the strategy to fill limit orders only after the market price moves a defined number of ticks past the order prices.

The following example places a limit order of 2% equity at a bar's hlcc4 price when the high is the `highest` value over the past `length` bars and there are no pending entries. The strategy closes the market position and cancels all orders after the low is the `lowest` value. Each time the strategy triggers an order, it draws a horizontal line at the `limitPrice`, which it updates on each bar until closing the position or canceling the order:

//@version=5 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 script assumes that all limit orders are guaranteed to fill when the market price reaches their values, which is often not the case in real-life trading. Let's add price verification to our limit orders to account for potentially unfilled ones. In this example, we've included `backtest_fill_limits_assumption = 3` in the strategy() function call. As we can see, using limit verification omits some simulated order fills and

changes the times of others, because the entry orders can now only fill after the price exceeds the limit price by *three ticks*:

> Notice!Limit verification can change the *times* of some order fills. However, strategies still execute verified limit orders at the same *prices*. This “time-warping” effect is a compromise that preserves the prices of limit orders, but it can cause a strategy to fill the orders at times that wouldn’t necessarily be possible in the real world. Therefore, users should exercise caution with this setting and understand its limitations when analyzing strategy results.

Risk management

Designing a strategy that performs well, especially in a broad class of markets, is a challenging task. Most strategies are designed for specific market patterns/conditions and can produce uncontrolled losses when applied to other data. Therefore, a strategy’s risk management behavior can be critical to its performance. Programmers can set risk management criteria in their strategy scripts 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, it *always* applies to the strategy unless the programmer removes the call from the code.

strategy.risk.allow_entry_in()

This command overrides the market direction allowed for all strategy.entry() commands in the script. When a user specifies the trade direction with the strategy.risk.allow_entry_in() function (e.g., long) the strategy enters trades only in that direction. If a script calls an entry command in the opposite direction while there’s an open market position, the strategy simulates a market order to *close* the position.

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 timeframe is higher than daily). If the strategy creates more orders than the maximum, 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 loss the strategy tolerates per trading day (or per chart bar if the timeframe is higher than daily). When the strategy's losses reach this threshold, it 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 using strategy.entry() commands. If the quantity of an entry command results in a market position that exceeds this threshold, the strategy reduces the order quantity so that the resulting position does not exceed the limit.

Margin

Margin is the minimum percentage of a market position that a trader must hold in their account as collateral to receive and sustain a loan from their broker to achieve their desired *leverage*. The `margin_long` and `margin_short` parameters of the strategy() declaration statement and the "Margin for long/short positions" inputs in the "Properties" tab of the script settings specify margin percentages for long and short positions. For example, if a trader sets the margin for long positions to 25%, they must have enough funds to cover 25% of an open long position. This margin percentage also means the trader can potentially spend up to 400% of their equity on their trades.

If a strategy's simulated funds cannot cover the losses from a margin trade, the broker emulator triggers a *margin call*, which forcibly liquidates all or part of the open position. The exact number of contracts/shares/lots/units that the emulator liquidates is *four times* the amount required to cover the loss, which helps prevent constant margin calls on subsequent bars. The emulator determines liquidated quantity using the following algorithm:

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 symbol: `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`

To examine this calculation in detail, let's add the built-in Supertrend Strategy to the NASDAQ:TSLA chart on the "1D" timeframe and set the "Order size" to 300% of equity and the "Margin for long positions" to 25% in the "Properties" tab of the strategy settings:

The first entry happened at the bar's opening price on 16 Sep 2010. The strategy bought 682,438 shares (Position Size) at 4.43 USD (Entry Price). Then, on 23 Sep 2010, when the price dipped to 3.9 (Current

Price), the emulator forcibly liquidated 111,052 shares with a margin call. The calculations below show how the broker emulator determined this amount for the margin call 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
```

Note that:

- The strategy.margin_liquidation_price variable's value represents the price level that will cause a margin call if the market price reaches it. For more information about how margin works and the formula for calculating a position's margin call price, see this page 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 Tester, directly within their code's logic and calculations.

Several `strategy.*` variables hold fundamental 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 hold 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` crosses above 10 and the strategy.opentrades value is 0, the script calls strategy.entry() to place a “Buy” market order. On the following bar, where that order fills, it calls strategy.exit() to create a stop-loss order at a user-specified percentage below the strategy.position_avg_price. If the `rank` crosses above 80 during the open trade, the script uses strategy.close() to exit the position on the next bar.

The script draws a table on the main chart pane displaying formatted strings containing 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 profit to the average loss, and the profit factor (the ratio of the gross profit to the gross loss). It also plots the total equity in a separate pane and highlights the pane’s background based on the strategy’s open profit:

//@version=5 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 only draws the table 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.
- We included `force_overlay = true` in the table.new() call 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 its `median` 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 it in a single-cell table:

//@version=5 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 because we included `pyramiding = 5` in the strategy() declaration statement. See the pyramiding section for more information.
- The strategy.exit() instance in this script persists and generates exit orders for every entry in the open position because we did not specify a `from_entry` ID. 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 tracks one specific condition per function call, and the alert() function, which tracks all its calls simultaneously, 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.

When 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 can only execute alert() function calls on the close of realtime bars, as if they used alert.freq_once_per_bar_close, regardless of the `freq` argument in the call. Users can change the alert frequency by including `calc_on_every_tick = true` in the strategy() call or selecting the "Recalculate/On every tick" option in the "Settings/Properties" tab before creating the alert. However, depending on the script, this setting can adversely impact the strategy's behavior. See the `calc_on_every_tick` section for more information.

Order fill alert triggers do not suffer the same limitations as the triggers from alert() calls, which makes them more suitable for sending alerts to third parties for automation. Alerts from order fill events execute *immediately*, unaffected by a script's `calc_on_every_tick` setting. Users can set the default message for order fill alerts via the `//@strategy_alert_message` compiler annotation. The text provided with this 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 `@strategy_alert_message` with *placeholders* for the trade action, current position size, ticker name, and fill price values in the message text:

//@version=5 //@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” dropdown tab:

When the alert fires, the strategy populates 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 aware 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 possible solutions to mitigate their effects.

> Notice!Although testing strategies on existing data might give traders helpful information about a strategy’s qualities, it’s important to note that neither the past nor the present guarantees the future. Financial markets can change rapidly and unpredictably, which can cause a strategy to sustain uncontrollable losses. Additionally, simulated results may not fully account for other real-world factors that can impact trading performance. Therefore, we recommend that traders thoroughly understand the limitations and risks of backtests and forward tests and consider them “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 historical performance of a trading strategy or model by simulating and analyzing its past 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. However, traders should always understand the risks and limitations when basing their 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 its performance. While forward testing has the distinct advantage of not being affected by certain 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, which is known as *lookahead bias*. Not only is this bias a common cause of

unrealistic strategy results, since the future is never actually knowable beforehand, but it is also one of the typical 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. Since there is no future data to leak into the past on realtime bars, the strategy will behave differently on historical and realtime bars if its results have lookahead bias.

To eliminate lookahead bias in a strategy:

- Do not use repainting variables that leak future values into the past in the order placement or cancellation logic.
- Do not include barmerge.lookahead_on in `request.*()` calls without offsetting the data series, as described in this section of the Repainting page.
- Use realistic strategy calculation behavior.

Selection bias

Selection bias occurs when a trader analyzes only results on specific instruments or timeframes while ignoring others. This bias can distort the perspective of the strategy's robustness, which can impact trading decisions and performance optimizations. Traders can reduce the effects of selection bias by evaluating their strategies on multiple, ideally diverse, symbols and timeframes, and ensuring not to ignore poor performance results or "cherry-pick" testing ranges.

Overfitting

A common problem when optimizing a strategy based on backtest results is overfitting ("curve fitting"), which means tailoring the strategy for specific data. An overfitted strategy often fails to generalize well on new, unseen data. One widely-used approach to help reduce the potential for 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, otherwise known as "in-sample" (IS) and "out-of-sample" (OOS) backtesting.

In this approach, traders optimize strategy parameters on the IS data, and they test the optimized configuration on the OOS data without additional fine-tuning. Although this 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 unknowable.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Tables

Introduction

Tables are objects that can be used to position information in specific and fixed locations in a script's visual space. Contrary to all other plots or objects drawn in Pine Script®, tables are not anchored to specific bars; they *float* in a script's space, whether in overlay or pane mode, in studies or strategies, independently of the chart bars being viewed or the zoom factor used.

Tables contain cells arranged in columns and rows, much like a spreadsheet. They are created and populated in two distincts steps:

1. A table's structure and key attributes are defined using table.new(), which returns a table ID that acts like a pointer to the table, just like label, line, or array IDs do. The table.new() call will create the table object but does not display it.
2. Once created, and for it to display, the table must be populated using one table.cell() call for each cell. Table cells can contain text, or not. This second step is when the width and height of cells are defined.

Most attributes of a previously created table can be changed using `table.set_*()` setter functions. Attributes of previously populated cells can be modified using `table.cell_set_*()` functions.

A table is positioned in an indicator's space by anchoring it to one of nine references: the four corners or midpoints, including the center. Tables are positioned by expanding the table from its anchor, so a table anchored to the position.middle_right reference will be drawn by expanding up, down and left from that anchor.

Two modes are available to determine the width/height of table cells:

- A default automatic mode calculates the width/height of cells in a column/row using the widest/highest text in them.
- An explicit mode allows programmers to define the width/height of cells using a percentage of the indicator's available x/y space.

Displayed table contents always represent the last state of the table, as it was drawn on the script's last execution, on the dataset's last bar. Contrary to values displayed in the Data Window or in indicator values, variable contents displayed in tables will thus not change as a script user moves his cursor over specific chart bars. For this reason, it is strongly recommended to always restrict execution of all `table.*()` calls to either the first or last bars of the dataset. Accordingly:

- Use the var keyword to declare tables.
- Enclose all other calls inside an if barstate.islast block.

Multiple tables can be used in one script, as long as they are each anchored to a different position. Each table object is identified by its own ID. Limits on the quantity of cells in all tables are determined by the total number of cells used in one script.

Creating tables

When creating a table using table.new(), three parameters are mandatory: the table's position and its number of columns and rows. Five other parameters are optional: the table's background color, the color and width of the table's outer frame, and the color and width of the borders around all cells, excluding the outer frame. All table attributes except its number of columns and rows can be modified using setter functions: table.set_position(), table.set_bgcolor(), table.set_frame_color(), table.set_frame_width(), table.set_border_color() and table.set_border_width().

Tables can be deleted using table.delete(), and their content can be selectively removed using table.clear().

When populating cells using table.cell(), you must supply an argument for four mandatory parameters: the table id the cell belongs to, its column and row index using indices that start at zero, and the text string the cell contains, which can be null. Seven other parameters are optional: the width and height of the cell, the text's attributes (color, horizontal and vertical alignment, size), and the cell's background color. All cell attributes can be modified using setter functions: table.cell_set_text(), table.cell_set_width(), table.cell_set_height(), table.cell_set_text_color(), table.cell_set_text_halign(), table.cell_set_text_valign(), table.cell_set_text_size() and table.cell_set_bgcolor().

Keep in mind that each successive call to table.cell() redefines **all** the cell's properties, deleting any properties set by previous table.cell() calls on the same cell.

Placing a single value in a fixed position

Let's create our first table, which will place the value of ATR in the upper-right corner of the chart. We first create a one-cell table, then populate that cell:

//@version=5 indicator("ATR", "", true) // We use `var` to only initialize the table on the first bar. var table atrDisplay = table.new(position.top_right, 1, 1) // We call `ta.atr()` outside the `if` block so it executes on each bar. myAtr = ta.atr(14) if barstate.islast // We only populate the table on the last bar. table.cell(atrDisplay, 0, 0, str.tostring(myAtr))

Note that:

- We use the var keyword when creating the table with table.new().
- We populate the cell inside an if barstate.islast block using table.cell().
- When populating the cell, we do not specify the `width` or `height`. The width and height of our cell will thus adjust automatically to the text it contains.
- We call `ta.atr(14)` prior to entry in our if block so that it evaluates on each bar. Had we used `str.tostring(ta.atr(14))` inside the if block, the function would not have evaluated correctly because it would be called on the dataset's last bar without having calculated the necessary values from the previous bars.

Let's improve the usability and aesthethics of our script:

//@version=5 indicator("ATR", "", true) atrPeriodInput = input.int(14, "ATR period", minval = 1, tooltip = "Using a period of 1 yields True Range.") var table atrDisplay = table.new(position.top_right, 1, 1, bgcolor = color.gray, frame_width = 2, frame_color = color.black) myAtr = ta.atr(atrPeriodInput) if barstate.islast table.cell(atrDisplay, 0, 0, str.tostring(myAtr, format.mintick), text_color = color.white)

Note that:

- We used table.new() to define a background color, a frame color and its width.

- When populating the cell with table.cell(), we set the text to display in white.
- We pass format.mintick as a second argument to the str.tostring() function to restrict the precision of ATR to the chart's tick precision.
- We now use an input to allow the script user to specify the period of ATR. The input also includes a tooltip, which the user can see when he hovers over the “i” icon in the script's “Settings/Inputs” tab.

Coloring the chart's background

This example uses a one-cell table to color the chart's background on the bull/bear state of RSI:

//@version=5 indicator("Chart background", "", true) bullColorInput = input.color(color.new(color.green, 95), "Bull", inline = "1") bearColorInput = input.color(color.new(color.red, 95), "Bear", inline = "1") // ———— Function colors chart bg on RSI bull/bear state. colorChartBg(bullColor, bearColor) => var table bgTable = table.new(position.middle_center, 1, 1) float r = ta.rsi(close, 20) color bgColor = r > 50 ? bullColor : r < 50 ? bearColor : na if barstate.islast table.cell(bgTable, 0, 0, width = 100, height = 100, bgcolor = bgColor) colorChartBg(bullColorInput, bearColorInput)

Note that:

- We provide users with inputs allowing them to specify the bull/bear colors to use for the background, and send those input colors as arguments to our `colorChartBg()` function.
- We create a new table only once, using the var keyword to declare the table.
- We use table.cell() on the last bar only, to specify the cell's properties. We make the cell the width and height of the indicator's space, so it covers the whole chart.

Creating a display panel

Tables are ideal to create sophisticated display panels. Not only do they make it possible for display panels to always be visible in a constant position, they provide more flexible formatting because each cell's properties are controlled separately: background, text color, size and alignment, etc.

Here, we create a basic display panel showing a user-selected quantity of MAs values. We display their period in the first column, then their value with a green/red/gray background that varies with price's position with regards to each MA. When price is above/below the MA, the cell's background is colored with the bull/bear color. When the MA falls between the current bar's open and close, the cell's background is of the neutral color:

//@version=5 indicator("Price vs MA", "", true) var string GP1 = "Moving averages" int masQtyInput = input.int(20, "Quantity", minval = 1, maxval = 40, group = GP1, tooltip = "1-40") int masStartInput = input.int(20, "Periods begin at", minval = 2, maxval = 200, group = GP1, tooltip = "2-200") int masStepInput = input.int(20, "Periods increase by", minval = 1, maxval = 100, group = GP1, tooltip = "1-100") var string GP2 = "Display" string tableYposInput = input.string("top", "Panel position", inline = "11", options = ["top", "middle", "bottom"], group = GP2) string tableXposInput = input.string("right", "", inline = "11", options = ["left", "center", "right"], group = GP2) color bullColorInput = input.color(color.new(color.green, 30), "Bull", inline = "12", group = GP2) color bearColorInput = input.color(color.new(color.red, 30), "Bear", inline = "12", group = GP2) color neutColorInput = input.color(color.new(color.gray, 30), "Neutral", inline = "12", group = GP2) var table panel = table.new(tableYposInput + "_" + tableXposInput, 2, masQtyInput + 1) if barstate.islast // Table header. table.cell(panel, 0, 0, "MA", bgcolor = neutColorInput) table.cell(panel, 1, 0, "Value", bgcolor = neutColorInput) int period = masStartInput for i = 1 to masQtyInput // ———— Call MAs on each bar. float ma = ta.sma(close, period) // ———— Only execute table code on last bar. if

barstate.islast // Period in left column. table.cell(panel, 0, i, str.tostring(period), bgcolor = neutColorInput) // If MA is between the open and close, use neutral color. If close is lower/higher than MA, use bull/bear color. bgColor = close > ma ? open < ma ? neutColorInput : bullColorInput : open > ma ? neutColorInput : bearColorInput // MA value in right column. table.cell(panel, 1, i, str.tostring(ma, format.mintick), text_color = color.black, bgcolor = bgColor) period += masStepInput

Note that:

- Users can select the table's position from the inputs, as well as the bull/bear/neutral colors to be used for the background of the right column's cells.
- The table's quantity of rows is determined using the number of MAs the user chooses to display. We add one row for the column headers.
- Even though we populate the table cells on the last bar only, we need to execute the calls to ta.sma() on every bar so they produce the correct results. The compiler warning that appears when you compile the code can be safely ignored.
- We separate our inputs in two sections using `group`, and join the relevant ones on the same line using `inline`. We supply tooltips to document the limits of certain fields using `tooltip`.

Displaying a heatmap

Our next project is a heatmap, which will indicate the bull/bear relationship of the current price relative to its past values. To do so, we will use a table positioned at the bottom of the chart. We will display colors only, so our table will contain no text; we will simply color the background of its cells to produce our heatmap. The heatmap uses a user-selectable lookback period. It loops across that period to determine if price is above/below each bar in that past, and displays a progressively lighter intensity of the bull/bear color as we go further in the past:

//@version=5 indicator("Price vs Past", "", true) var int MAX_LOOKBACK = 300 int lookBackInput = input.int(150, minval = 1, maxval = MAX_LOOKBACK, step = 10) color bullColorInput = input.color(#00FF00ff, "Bull", inline = "11") color bearColorInput = input.color(#FF0080ff, "Bear", inline = "11") // ———— Function draws a heatmap showing the position of the current `_src` relative to its past `_lookBack` values. drawHeatmap(src, lookBack) => // float src : evaluated price series. // int lookBack: number of past bars evaluated. // Dependency: MAX_LOOKBACK // Force historical buffer to a sufficient size. max_bars_back(src, MAX_LOOKBACK) // Only run table code on last bar. if barstate.islast var heatmap = table.new(position.bottom_center, lookBack, 1) for i = 1 to lookBackInput float transp = 100. * i / lookBack if src > src[i] table.cell(heatmap, lookBack - i, 0, bgcolor = color.new(bullColorInput, transp)) else table.cell(heatmap, lookBack - i, 0, bgcolor = color.new(bearColorInput, transp)) drawHeatmap(high, lookBackInput)

Note that:

- We define a maximum lookback period as a `MAX_LOOKBACK` constant. This is an important value and we use it for two purposes: to specify the number of columns we will create in our one-row table, and to specify the lookback period required for the `_src` argument in our function, so that we force Pine Script to create a historical buffer size that will allow us to refer to the required quantity of past values of `_src` in our for loop.
- We offer users the possibility of configuring the bull/bear colors in the inputs and we use `inline` to place the color selections on the same line.

- Inside our function, we enclose our table-creation code in an if barstate.islast construct so that it only runs on the last bar of the chart.
- The initialization of the table is done inside the if statement. Because of that, and the fact that it uses the var keyword, initialization only occurs the first time the script executes on a last bar. Note that this behavior is different from the usual var declarations in the script's global scope, where initialization occurs on the first bar of the dataset, at bar_index zero.
- We do not specify an argument to the `text` parameter in our table.cell() calls, so an empty string is used.
- We calculate our transparency in such a way that the intensity of the colors decreases as we go further in history.
- We use dynamic color generation to create different transparencies of our base colors as needed.
- Contrary to other objects displayed in Pine scripts, this heatmap's cells are not linked to chart bars. The configured lookback period determines how many table cells the heatmap contains, and the heatmap will not change as the chart is panned horizontally, or scaled.
- The maximum number of cells that can be displayed in the scritp's visual space will depend on your viewing device's resolution and the portion of the display used by your chart. Higher resolution screens and wider windows will allow more table cells to be displayed.

Tips

- When creating tables in strategy scripts, keep in mind that unless the strategy uses `calc_on_every_tick = true`, table code enclosed in if barstate.islast blocks will not execute on each realtime update, so the table will not display as you expect.
- Keep in mind that successive calls to table.cell() overwrite the cell's properties specified by previous table.cell() calls. Use the setter functions to modify a cell's properties.
- Remember to control the execution of your table code wisely by restricting it to the necessary bars only. This saves server resources and your charts will display faster, so everybody wins.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Text and shapes

Introduction

You may display text or shapes using five different ways with Pine Script®:

- 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 charts that will not move as users scroll of zoom 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 function include are able to 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.
- The color or size of text can sometimes be controlled using function parameters, but no inline formatting (bold, italics, monospace, etc.) is possible.
- Text from Pine scripts always displays on the chart in the Trebuchet MS font, which is used in many TradingView texts, including this one.

This script displays text using the four methods available in Pine Script:

//@version=5 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 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=5 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=5 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=5 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=5 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 will need to use `\n` followed by a special non-printing character that doesn’t get stripped out to preserve the newline's functionality. Here we’re using a Unicode Zero-width space (U+200E). While you don’t see it in the following code’s strings, it is there and can be copy/pasted. The special Unicode character 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=5 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=5 indicator("", "", true) body = close - open plotarrow(body, colorup = color.teal, colordown = color.orange)

Note how the heigth 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=5 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.

> Notice!On TradingView charts, a complete set of *Drawing Tools* allows users to create and modify drawings using mouse actions. While they may sometimes look similar to drawing objects created with Pine Script code, they are unrelated entities. Drawing objects created using Pine code cannot be modified with mouse actions, and hand-drawn drawings from the chart user interface are not visible from Pine scripts.

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. It has the following signature:

```
label.new(x, y, text, xloc, yloc, color, style, textcolor, size, textalign, too
```

The *setter* functions allowing you to change a label's properties are:

- label.set_x()
- label.set_y()
- label.set_xy()
- 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()

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=5 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=5 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) // 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=5 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 (maximum of 500 bars in the future). 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=5 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=5 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=5 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=5 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 in the realtime bar. See the page on Pine Script's Execution model.

This script demonstrates the effect of rollback when running in the realtime bar:

//@version=5 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.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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=5 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, #}\n ISO 8601 representation (Exchange time zone): {1}\n Custom date and time representation (UTC+0 time zone): {2}\n Custom 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.

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 UNIX timestamps of the last historical bar's opening time and closing time 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 displays all four representations for comparison within a table in the bottom-right corner of the chart:

```
//@version=5 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=5 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=5 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 opening time crosses the `unixTimestamp`. If the timestamp represents a future time, the label uses 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=5 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=5 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 *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=5 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 realtime bars of tick-based and price-based charts.

`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=5 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.

> Notice!Because timenow only updates 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=5 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 unconfirmed (open), the label is blue to signify that additional updates can occur. After the bar is confirmed (closed), the label turns 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=5 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=5 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 in a single-cell table created only on that bar. When the script executes on the last available bar, it draws a label showing 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=5 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=5 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=5 indicator("`syminfo.timezone` demo", overlay = true, dynamic_requests = 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.
- The indicator() declaration statement includes `dynamic_requests = true`, 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 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, bars_back) → series int
functionName(timeframe, session, bars_back) → series int
functionName(timeframe, session, timezone, bars_back) → series int
```

Where:

- `functionName` is the function's identifier.
- The `timeframe` parameter accepts a timeframe string. The function uses the script's main timeframe if the argument is timeframe.period or an empty "string".
- The `session` parameter accepts a session string defining 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* weekdays 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 `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 `bars_back` parameter accepts an "int" value specifying which bar's time the returned timestamp represents. If the value is positive, the function returns the timestamp from that number of bars back relative to the current bar. If the value is negative and greater than or equal to -500, the function returns the *expected* timestamp of a future bar. The default value is 0.

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 times of realtime bars on non-time-based charts, so it returns na on those bars. Similarly, the time() function with a negative `bars_back` value 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.

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.

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=5 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=5 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 calls time("1M") to get the opening UNIX timestamp of the current bar on the "1M" timeframe. It detects when bars on that timeframe open by checking when the ta.change() of the timestamp returns a value greater than 0. On each occurrence of the condition, the script detects whether the HTF bar opened after a gap by checking if the opening time is greater than the previous bar's time_close("1M") value.

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=5 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`.

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=5 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=5 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 uses plotted shapes, labels, and background colors to visualize the conditions on the chart:

//@version=5 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 first and third, leading to different formatted strings in the last column:


```
//@version=5 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 valid 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.
`"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.

Token	Represents	Remarks and examples
`"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 (**acronym 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=5 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=5 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=5 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=5 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, 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`.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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=5 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 and the input.timeframe() function 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 by the script's user through the call to input.timeframe().
- 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, we generate a runtime error.
- We finally print the two timeframe values converted to minutes.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Error messages

The if statement is too long

This error occurs when the indented code inside an if statement is too large for the compiler. Because of how the compiler works, you won't receive a message telling you exactly how many lines of code you are over the limit. The only solution now is to break up your if statement into smaller parts (functions or smaller if statements). The example below shows a reasonably lengthy if statement; theoretically, this would throw `line 4: if statement is too long`:

//@version=5 indicator("My script") var e = 0 if barstate.islast a = 1 b = 2 c = 3 d = 4 e := a + b + c + d plot(e)

To fix this code, you could move these lines into their own function:

//@version=5 indicator("My script") var e = 0 doSomeWork() => a = 1 b = 2 c = 3 d = 4 result = a + b + c + d if barstate.islast e := doSomeWork() plot(e)

Script requesting too many securities

The maximum number of securities in script is limited to 40. If you declare a variable as a `request.security` function call and then use that variable as input for other variables and calculations, it will not result in multiple `request.security` calls. But if you will declare a function that calls `request.security` --- every call to this function will count as a `request.security` call.

It is not easy to say how many securities will be called looking at the source code. Following example have exactly 3 calls to `request.security` after compilation:

//@version=5 indicator("Securities count") a = request.security(syminfo.tickerid, '42', close) // (1) first unique security call b = request.security(syminfo.tickerid, '42', close) // same call as above, will not produce new security call after optimizations plot(a) plot(a + 2) plot(b) sym(p) => // no security call on this line request.security(syminfo.tickerid, p, close) plot(sym('D')) // (2) one indirect call to security plot(sym('W')) // (3) another indirect call to security request.security(syminfo.tickerid, timeframe.period, open) // result of this line is never used, and will be optimized out

Script could not be translated from: null

study($)

Usually this error occurs in version 1 Pine scripts, and means that code is incorrect. Pine Script® of version 2 (and higher) is better at explaining errors of this kind. So you can try to switch to version 2 by adding a special attribute in the first line. You'll get `line 2: no viable alternative at character '$'`:

// @version=2 study($)

line 2: no viable alternative at character '$'

This error message gives a hint on what is wrong. `$` stands in place of string with script title. For example:

// @version=2 study("title")

Mismatched input <...> expecting <???>

Same as `no viable alternative`, but it is known what should be at that place. Example:

//@version=5 indicator("My Script") plot(1)

```
line 3: mismatched input 'plot' expecting 'end of line without line continuation'
```

To fix this you should start line with `plot` on a new line without an indent:

//@version=5 indicator("My Script") plot(1)

Loop is too long (> 500 ms)

We limit the computation time of loop on every historical bar and realtime tick to protect our servers from infinite or very long loops. This limit also fail-fast indicators that will take too long to compute. For example, if you'll have 5000 bars, and indicator takes 500 milliseconds to compute on each of bars, it would have result in more than 16 minutes of loading:

//@version=5 indicator("Loop is too long", max_bars_back = 101) s = 0 for i = 1 to 1e3 // to make it longer for j = 0 to 100 if timestamp(2017, 02, 23, 00, 00) <= time[j] and time[j] < timestamp(2017, 02, 23, 23, 59) s := s + 1 plot(s)

It might be possible to optimize algorithm to overcome this error. In this case, algorithm may be optimized like this:

//@version=5 indicator("Loop is too long", max_bars_back = 101) bar_back_at(t) => i = 0 step = 51 for j = 1 to 100 if i < 0 i := 0 break if step == 0 break if time[i] >= t i := i + step i else i := i - step i step := step / 2 step i s = 0 for i = 1 to 1e3 // to make it longer s := s - bar_back_at(timestamp(2017, 02, 23, 23, 59)) + bar_back_at(timestamp(2017, 02, 23, 00, 00)) s plot(s)

Script has too many local variables

This error appears if the script is too large to be compiled. A statement `var=expression` creates a local variable for `var`. Apart from this, it is important to note, that auxiliary variables can be implicitly created during the process of a script compilation. The limit applies to variables created both explicitly and implicitly. The limitation of 1000 variables is applied to each function individually. In fact, the code placed in a *global* scope of a script also implicitly wrapped up into the main function and the limit of 1000 variables becomes applicable to it. There are few refactorings you can try to avoid this issue:

var1 = expr1 var2 = expr2 var3 = var1 + var2

can be converted into:

var3 = expr1 + expr2

Pine Script cannot determine the referencing length of a series. Try using max_bars_back in the indicator or strategy function

The error appears in cases where Pine Script wrongly autodetects the required maximum length of series used in a script. This happens when a script's flow of execution does not allow Pine Script to inspect the use

of series in branches of conditional statements (`if`, `iff` or `?`), and Pine Script cannot automatically detect how far back the series is referenced. Here is an example of a script causing this problem:

```
//@version=5 indicator("Requires max_bars_back") test = 0.0 if bar_index > 1000 test := ta.roc(close, 20) plot(test)
```

In order to help Pine Script with detection, you should add the `max_bars_back` parameter to the script's `indicator` or `strategy` function:

```
//@version=5 indicator("Requires max_bars_back", max_bars_back = 20) test = 0.0 if bar_index > 1000 test := ta.roc(close, 20) plot(test)
```

You may also resolve the issue by taking the problematic expression out of the conditional branch, in which case the `max_bars_back` parameter is not required:

```
//@version=5 indicator("My Script") test = 0.0 roc20 = ta.roc(close, 20) if bar_index > 1000 test := roc20 plot(test)
```

In cases where the problem is caused by a **variable** rather than a built-in **function** (`vwma` in our example), you may use the `max_bars_back` function to explicitly define the referencing length for that variable only. This has the advantage of requiring less runtime resources, but entails that you identify the problematic variable, e.g., variable `s` in the following example:

```
//@version=5 indicator("My Script") f(off) => t = 0.0 s = close if bar_index > 242 t := s[off] t plot(f(301))
```

This situation can be resolved using the `max_bars_back` **function** to define the referencing length of variable `s` only, rather than for all the script's variables:

```
//@version=5 indicator("My Script") f(off) => t = 0.0 s = close max_bars_back(s, 301) if bar_index > 242 t := s[off] t plot(f(301))
```

When using drawings that refer to previous bars through `bar_index[n]` and `xloc = xloc.bar_index`, the time series received from this bar will be used to position the drawings on the time axis. Therefore, if it is impossible to determine the correct size of the buffer, this error may occur. To avoid this, you need to use `max_bars_back(time, n)`. This behavior is described in more detail in the section about drawings.

Memory limits exceeded. The study allocates X times more than allowed

The most common cause for this error is returning objects and collections from request.*() functions. Other possible causes include unnecessary drawing updates, excess historical buffer capacity, or inefficient use of max_bars_back.

Returning collections from `request.*()` functions

A common source of the "*Memory limits exceeded*" error is returning objects or collections from another chart symbol or timeframe using request.*() functions.

When requesting data from other contexts, the data for *each bar* is copied and stored in memory to allow the script to reference it later in the main context. This can use a lot of memory, depending on the data. Requesting large collections can easily lead to excessive memory consumption.

Let’s look at an example script where we request data to calculate the balance of power (BOP) for the symbol at a higher timeframe. Here, the *request expression* is a custom function that populates a persistent array with our calculated BOP values, returning the *full array* to the main context on *each* bar. We intend to use these stored array values to calculate and plot the average BOP in the main context. However, returning every array instance consumes a lot of memory, and so this script can throw a memory error on charts with a sufficiently long history:

//@version=5 indicator("BOP array in higher timeframe context", "Memory limit demo") //@variable User-input length for calculating average of BOP values. int avgLength = input.int(5, "Average BOP Length", minval = 1) //Returns a copy of the `dataArray` on every bar, which uses a lot of memory. dataFunction() => //@variable Persistent array containing the "balance of power" (BOP) values for all bars from the higher timeframe. var array<float> dataArray = array.new_float(0) //@variable The "balance of power" percentage calculated for the current bar. float bop = (close - open) / (high - low) * 100 dataArray.push(bop) //Return the full collection. dataArray // Request the full BOP array from the 1D timeframe. array<float> reqData = request.security(syminfo.tickerid, "1D", dataFunction()) // Plot zero line. hline(0, "Zero line", color.gray, hline.style_dotted) // Latest BOP value and average BOP are calculated in the main context if `reqData` is not `na`. //@variable The latest BOP value from the `reqData` array. float latestValue = na //@variable The average of the last `avgLength` BOP values. float avgBOP = na if not na(reqData) // Retrieve BOP value for the current main context bar. latestValue := reqData.last() // Calculate the average BOP for the most-recent values from the higher timeframe array. //@variable Size of the `reqData` array returned from the higher timeframe. int dataSize = reqData.size() //@variable A subset of the latest values from the `reqData` array. Its size is determined by the `avgLength` set. array<float> lastValues = dataSize >= avgLength ? reqData.slice(dataSize - avgLength, dataSize): reqData avgBOP := lastValues.avg() // Plot the BOP value and average line. color plotColor = latestValue >= 0 ? color.aqua : color.orange plot(latestValue, "BOP", plotColor, style = plot.style_columns) plot(avgBOP, "Avg", color.purple, linewidth = 3)

How do I fix this?

Optimize requests and limit the data returned to the main context to ensure that only the *minimum necessary* data is stored in memory.

If possible, try to return *calculated results* directly rather than returning the collections themselves, or only return collections *conditionally*, when they are necessary in the main context.

Let’s consider a few common scenarios where scripts need specific data in the main context.

Return last state only

If a script needs only the *last state* of a requested collection in the main context: use an if barstate.islast condition to return a copy of the *last bar’s collection* only.

Here, we modified our script to display only the *latest* average BOP (a single value), rather than plotting an average line. The updated request function now returns the calculated BOP values directly for each bar, and returns the higher timeframe’s array only on the last bar:

//@version=5 indicator("BOP array on last bar", "Memory limit demo") //@variable User-input length for calculating average of BOP values. int avgLength = input.int(5, "Average BOP Length", minval = 1) // Returns the calculated `bop` each bar, and a copy of the `dataArray` on the last bar or `na` otherwise. dataFunction() => //@variable Persistent array containing the "balance of power" (BOP) values for all higher timeframe bars. var array<float> dataArray = array.new_float(0) //@variable The "balance of power"

percentage calculated for the current higher timeframe bar. float bop = (close - open) / (high - low) * 100 dataArray.push(bop) // Return the collection on the last bar only. if barstate.islast [bop, dataArray] else [bop, na] // Request calculated BOP value, and BOPs array if on last bar, from the higher timeframe. [reqValue, reqData] = request.security(syminfo.tickerid, "1D", dataFunction()) // Plot zero line. hline(0, "Zero line", color.gray, hline.style_dotted) // Plot the BOP value for each main context bar. color plotColor = reqValue >= 0 ? color.aqua : color.orange plot(reqValue, "BOP", plotColor, style = plot.style_columns) // Calculate the average BOP for most-recent values from the higher timeframe array, and display result in a table cell. if not na(reqData) //@variable Size of the `reqData` array returned from the higher timeframe. int dataSize = reqData.size() //@variable A subset of the latest values from the `reqData` array. Its size is determined by the `avgLength` set. array<float> lastValues = dataSize >= avgLength ? reqData.slice(dataSize - avgLength, dataSize): reqData //@variable The average of the last `avgLength` BOP values. 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(avgLength) + " BOPs: " + str.tostring(avgBOP, "##.##") + "%", text_color = color.white)

Return calculated results

If a script needs the *result* of a calculation on a collection, but does not need the collection itself in the main context, use a user-defined function as the request expression. The function can calculate on the collection in the *requested* context and return only the result to the main context.

For example, we can calculate the average BOP directly within our request function. Therefore, only the calculated values are stored in memory, and the request expression returns a tuple (current BOP and average BOP) to plot the results in the main context:

//@version=5 indicator("Return BOP results only", "Memory limit demo") //@variable User-input length for calculating average of BOP values. int avgLength = input.int(5, "Average BOP Length", minval = 1) // Returns the calculated `bop` and `avgBOP` values directly. dataFunction() => //@variable Persistent array containing the "balance of power" (BOP) values for all higher timeframe bars. var array<float> dataArray = array.new_float(0) //@variable The "balance of power" percentage calculated for the current higher timeframe bar. float bop = (close - open) / (high - low) * 100 dataArray.push(bop) // Calculate the average BOP for the `avgLength` most-recent values. //@variable Size of the `dataArray`. int dataSize = dataArray.size() //@variable A subset of the latest values from the `dataArray`. Its size is determined by the `avgLength` set. array<float> lastValues = dataSize >= avgLength ? dataArray.slice(dataSize - avgLength, dataSize): dataArray //@variable The average of the last `avgLength` BOP values. float avgBOP = lastValues.avg() //Return the calculated results. [bop, avgBOP] // Request BOP and average BOP values from the higher timeframe. [reqValue, reqAverage] = request.security(syminfo.tickerid, "1D", dataFunction()) // Plot zero line. hline(0, "Zero line", color.gray, hline.style_dotted) // Plot the BOP value and average line. color plotColor = reqValue >= 0 ? color.aqua : color.orange plot(reqValue, "BOP", plotColor, style = plot.style_columns) plot(reqAverage, "Avg", color.purple, linewidth = 3)

Return the collection on some bars

If a script needs the *collection itself* in the main context, but *not for every bar*, use conditional expressions to return only the necessary collections to the main context, returning na otherwise. The logic in the main context can then address the na gaps in the series and perform its desired actions on the reduced collections.

For example, if we want to calculate the average BOP across each *month* instead of using a user-input length, we can return the array from the requested context only when there is a change to a new month, returning na otherwise. We then maintain the previous month's values in the main context to keep a valid array for all intra-month bars:

//@version=5 indicator("Monthly BOP array", "Memory limit demo") // Returns the calculated `bop`, and a copy of the `dataArray` on a month's first trading day only, or `na` otherwise. dataFunction() => //@variable Persistent array containing the "balance of power" (BOP) values for all higher timeframe bars. var array<float> dataArray = array.new_float(0) // When a new month starts, return monthly data array to calculate average BOP for completed month. //@variable Array is `na` except on first trading day of each month, when it contains completed month's BOP values. array<float> returnArray = na //@variable Is `true` on the first bar of each month, `false` otherwise. bool isNewMonth = timeframe.change("1M") if isNewMonth returnArray := dataArray //Clear persistent array to start storing new month's data. if isNewMonth[1] dataArray.clear() //@variable The "balance of power" percentage calculated for the current higher timeframe bar. float bop = (close - open) / (high - low) * 100 dataArray.push(bop) //Return the calculated result and the `returnArray`. [bop, returnArray] // Request BOP data from the higher timeframe. (Returns calculated BOP and array of BOP values if new month starts) [reqValue, reqData] = request.security(syminfo.tickerid, "1D", dataFunction()) // Calculate the average BOP for the most-recent completed month. //@variable Persistent array that holds the BOP values for the most-recent completed month. var array<float> completedMonthBOPs = array.new_float(0) // If new month starts (i.e., `reqData` is not returned as `na`), then `completedMonthBOPs` is updated with new values. // Otherwise, it persists the last valid values for the rest of the month to adjust for `na` gaps. completedMonthBOPs := na(reqData) ? completedMonthBOPs : reqData //@variable The average BOP for the most-recent completed month. float avgBOP = completedMonthBOPs.avg() // Plot the BOP value and average line. color plotColor = reqValue >= 0 ? color.aqua : color.orange plot(reqValue, "BOP", plotColor, style = plot.style_columns) plot(avgBOP, "Avg", color.purple, linewidth = 3)

Other possible error sources and their fixes

There are a few other ways to optimize scripts to consume less memory.

Minimize `request.*()` calls

The request.*() function calls can be computationally expensive, because they retrieve data from other contexts, which can often require significant resource usage. Excessive or inefficient requests can easily cause scripts to reach the memory limit.

This memory consumption is especially substantial for scripts requesting data from *lower timeframes*, where the request function returns an array of multiple lower timeframe bars for *each* main context bar. For example, requesting "1" data on a "1D" chart returns hundreds of "1" bars for each "1D" bar that executes the request. In the process, the script must allocate memory to store all the requested data arrays so that it can access them later in the main context, which quickly increases the memory consumption.

Programmers can reduce the number of requested expressions by:

- Removing unnecessary `request.*()` function calls.
- Changing the requested timeframe to a higher resolution.
- Condensing multiple requests to the *same* context into a single `request.*()` call.

- Adjusting the `request.*()` function's calc_bars_count parameter to restrict the historical data points in the requested context.

See this section in the User Manual for more information on optimizing `request.*()` calls.

Refrain from using `max_bars_back` unless necessary

The `max_bars_back` parameter of an indicator or strategy sets the size of the *history buffer* for all series variables in a script. The history buffer determines the number of historical references *stored in memory* for the script's built-in and user-defined variables.

By default, the Pine Script runtime automatically allocates an appropriate buffer for each variable. Therefore, the `max_bars_back` parameter and function are only necessary when Pine cannot determine the referencing length of a series.

If you encounter this referencing length error, ensure that you set the `max_bars_back` value appropriately to your script's needs. Setting a value that's too large can lead to excessive memory consumption, as it stores unnecessary historical data that the script ultimately doesn't use. Read up on how to optimize using `max_bars_back` in our Help Center.

Minimize historical buffer calculations

The Pine Script runtime automatically creates historical buffers for all variables and function calls in a script. It determines the size of a buffer based on the *historical references* needed in the code (the references made using the [] history-referencing operator).

As the script runs across the dataset, referencing distant points in bar history can cause the script to restart its execution on previous bars to adjust its historical buffer size (see this User Manual article to learn more). Larger buffers in turn lead to an increase in memory consumption and can result in a runtime error. Ensure that scripts are referencing *necessary* historical values only, and avoid referencing very distant points in history when possible.

You can use the indicator() function's `calc_bars_count` parameter or the max_bars_back() function to *manually restrict* the historical data capacity on a script-wide or variable-specific scale. However, be aware that these methods can also cause memory consumption issues of their own if used improperly.

Reduce drawing updates for tables

Tables only display their *last state* on a chart. Any updates to a table on historical bars are redundant, because they are not visible. To use the least memory, draw the table *once*, and fill it on the last bar.

Use the var keyword to declare table objects once. Enclose all other setter function calls in a conditional if barstate.islast block for better performance. For more about tables, see this User Manual article.

Do not update drawings on historical bars

Similar to tables, any updates to drawing objects such as lines and labels that are made on historical bars are never seen by the user. The user only sees updates on *realtime* bars.

Eliminate updates to historical drawings during historical bars wherever possible. For more information, see this User Manual section.

Minimize 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 from point `x1` to `x2`, then needs to update the line's endpoint (`x2`), it's more computationally expensive to delete the existing line and redraw a new line from `x1` to `x3`. Instead, using the *setter* function line.set_x2() to update the existing line's endpoint is more efficient.

Look for ways to optimize drawing objects in a script:

- Reduce the number of redrawn objects by initializing drawing object *identifiers* and using their setter functions to modify properties.
- Remove unnecessary chart drawings using the `delete()` functions (e.g., line.delete() and label.delete()).
- Reduce an indicator's maximum drawings limit using the `max_lines_count` or `max_labels_count` parameters.

Filter dates in strategies

The total number of trades or orders in a strategy can impact the memory consumption of a script. For large datasets, reduce the number of unnecessary historical orders stored in memory by limiting the *starting point* of your strategy.

You can filter the strategy's date by adding a conditional expression that compares the bar time to a specified timestamp to only place entry/exit orders beyond a certain date.

See an example of date filtering in strategies here.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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=5 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=5 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=5 // 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=5 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 unique contexts. 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=5 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) // Continue with additional symbols up to a maximum of 40...

Note that:

- A script can execute up to 40 *unique* `request.*()` calls. A `request.*()` call is not unique if a script already calls the same function with the same arguments. See this 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 more strict 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 may 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=5 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=5 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=5 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=5 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.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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=5 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 the original matrix, its transpose, and the resulting sum matrix in a table on the chart.

//@version=5 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=5 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 when accessing data directly without searching through each element, as you need to do with 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=5 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 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, store complex data such as User-Defined Types, manipulate objects 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.

> Notice!An array is itself a series. Historical values of an array can be accessed by using the history-referencing operator.

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.
- 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.
- 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. If you want 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=5 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=5 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=5 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=5 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=5 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=5 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=5 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=5 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`.

> Notice!The array must be of type “int” or “float” and must be sorted in ascending order for array.binary_search() to work.

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=5 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=5 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=5 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 debugging with labels section 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=5 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=5 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. Pine Logs can display any script values or variables at any part of your code, including local scopes, functions, 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 console messages 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=5 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=5 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)

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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=5 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)

> Notice!When scripts pass variable lengths to built-in functions, an error can occur if the specified number of bars back is excessively large and increases dynamically and unpredictably. The error message "*Pine cannot determine the referencing length of a series. Try using max_bars_back*" indicates this issue. For more information on resolving this problem, refer to the Help Center article and User Manual entry on the topic.

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=5 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=5 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?

Use the custom function `calcBaseUnit()` in the following example script to return the correct pip value for Forex symbols, or the base unit of change for non-forex symbols:

//@version=5 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])

> Notice!This function might not address all potential scenarios, so confirm the results with your broker to ensure the function's output matches the broker's values.

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(number0, number1, …). 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(series,n).

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=5 indicator("Average session opening volume") import PineCoders/ConditionalAverages/1 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(src = volume, cond = session.isfirstbar_regular), "avg. When", #FF00FF) // Average volume over the last five opening bars. plot(PCca.avgWhenLast(src = volume, cond = session.isfirstbar_regular, cnt = 5), "avgWhenInLast()", #00FF00)

> Notice!The ConditionalAverages and getSeries libraries by PineCoders offer additional functions for conditional averaging and the creation of custom datasets.

How can I generate a random number?

Use the math.random() function to generate random numbers. This example script creates a circle plot with random RGB color values and a random y value between 0 and 1:

//@version=5 indicator("Random demo", overlay = false) // Generate a random price value (the default range is 0 to 1). float y = math.random() // Generate a color with red, green, and blue values as separate random 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=5 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))
```

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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`

Volume, or another series, plotted as columns, 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.

> Notice!The 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, the user can add a bottom margin in the script settings.

```
//@version=5 indicator("Dynamically scaled volume", overlay=true, max_lines_count=500) // Import the PineCoders' VisibleChart library import PineCoders/VisibleChart/4 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. 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. Note that using several `request.security()` calls can cause scripts compile and run more slowly than expected.

> Notice!Scripts can make a maximum of 40 distinct `request.*()` function calls. Therefore, scripts can retrive data from a maximum of 40 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.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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=5 indicator("HTF ATR") float higherTfAtr = request.security(symbol = syminfo.tickerid, timeframe = "1D", expression = ta.atr(14)) plot(higherTfAtr)

> Notice!This script repaints elapsed realtime bars when the daily value is finalized. See the How to avoid repainting when using the `request.security()` function? FAQ section to find out how to prevent repainting.

Which `security.*` 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 data from lower timeframes (LTF) 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=5 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=5 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)

> Notice!While easier to work with, this approach is more computationally expensive, because the script has to commit to memory an array of several values instead of a single value. If the logic of the script allows for it – as it does in our case, where both approaches return identical data – it's often better to use the first approach in big scripts with a lot of intrabar computations, due to the performance benefits it brings.

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.

> Notice!Failing to offset the `expression` argument results in *lookahead bias*, where the script appears to have access to historical values before they would actually have occurred. See the lookahead and future leak sections for more information.

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 triggers a runtime error if the chart's timeframe exceeds or matches the daily timeframe, to prevent the return of inaccurate values.

//@version=5 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=5 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=5 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=5 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=5 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=5 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=5 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.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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=5 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=5 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=5 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.

For a definition of the *access types* of published scripts: open, protected, and Invite-Only, see this article in the Help Center.

For an explanation on the *visibility* (public/private) that a script can have, see the Visibility section of the Publishing scripts page in the User Manual.

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=5 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 "*Open*" menu and choose the "*Built-in script*" 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 scripts displays a string in a label on the last bar of the chart using a custom function.

//@version=5 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)

For more techniques, see the debugging strings section of the User Manual.

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 Compound and nested 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.

ADVANCED

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Arrays

> Notice!This page contains advanced material. If you are a beginning Pine Script® programmer, we recommend you become familiar with other, more accessible Pine Script features before you venture here.

Introduction

Pine Script Arrays are one-dimensional collections that can hold multiple value references. Think of them as a better way to handle cases where one would otherwise need to explicitly declare 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.

Scripts reference arrays using array IDs similar to the IDs of lines, labels, and other *special types*. Pine Script does not use an indexing operator to reference individual array elements. Instead, functions including array.get() and array.set() read and write the values of array elements.

Scripts reference the elements of an array using an *index*, which starts at 0 and extends to the number of elements in the array minus one. Arrays in Pine Script can have a dynamic size that varies across bars, as one can change the number of elements in an array on each iteration of a script. Scripts can contain multiple array instances. The size of arrays is limited to 100,000 elements.

> Notice!We will use *beginning* of an array to designate index 0, and *end* of an array to designate the array's element with the highest index value. We will also extend the meaning of *array* to include array IDs, for the sake of brevity.

Declaring arrays

Pine Script uses the following syntax to declare arrays:

```
[var/varip ][array<type>/<type[]> ]<identifier> = <expression>
```

Where `<type>` is a type template for the array that declares the type of values it will contain, and the `<expression>` returns either an array of the specified type or `na`.

When declaring a variable as an array, we can use the array keyword followed by a type template. Alternatively, we can use the `type` name followed by the `[]` modifier (not to be confused with the [] *history-referencing operator*).

Since Pine always uses type-specific functions to create arrays, the `array<type>/type[]` part of the declaration is redundant, except when declaring an array variable assigned to `na`. Even when not required, explicitly declaring the array type helps clearly state the intention to readers.

This line of code declares an array variable named `prices` that points to `na`. In this case, we must specify the type to declare that the variable can reference arrays containing “float” values:

array<float> prices = na

We can also write the above example in this form:

float[] prices = na

When declaring an array and the `<expression>` is not `na`, use one of the following functions: array.new<type>(size, initial_value), array.from(), or array.copy(). For `array.new<type>(size, initial_value)` functions, the arguments of the `size` and `initial_value` parameters can be “series” to allow dynamic sizing and initialization of array elements. The following example creates an array containing zero “float” elements, and this time, the array ID returned by the array.new<float>() function call is assigned to `prices`:

prices = array.new<float>(0)

> Notice!The `array.*` namespace also contains type-specific functions for creating arrays, including 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(). The array.new<type>() function can create an array of any type, including user-defined types.

The `initial_value` parameter of `array.new*` functions allows users to set all elements in the array to a specified value. If no argument is provided for `initial_value`, the array is filled with `na` values.

This line declares an array ID named `prices` pointing to an array containing two elements, each assigned to the bar's `close` value:

prices = array.new<float>(2, close)

To create an array and initialize its elements with different values, use array.from(). This function infers the array's size and the type of elements it will hold from the arguments in the function call. As with `array.new*` functions, it accepts “series” arguments. All values supplied to the function must be of the same type.

For example, all three of these lines of code will create identical “bool” arrays with the same two elements:

statesArray = array.from(close > open, high != close) bool[] statesArray = array.from(close > open, high != close) array<bool> statesArray = array.from(close > open, high != close)

Using `var` and `varip` keywords

Users can utilize var and varip keywords to instruct a script to declare an array variable only once on the first iteration of the script on the first chart bar. Array variables declared using these keywords point to the same array instances until explicitly reassigned, allowing an array and its element references 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=5 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 re-declare the array on each bar. In this case, after execution of the array.push() call, the a.size() call would return a value of 1.

> Notice!Array variables declared using varip behave as ones using var on historical data, but they update their values for realtime bars (i.e., the bars since the script's last compilation) on each new price tick. Arrays assigned to varip variables can only hold int, float, bool, color, or string types or user-defined types that exclusively contain within their fields these types or collections (arrays, matrices) of these types.

Reading and writing array elements

Scripts can write values to existing individual array elements using array.set(id, index, value), and read using array.get(id, index). 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(id) 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=5 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=5 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(id, value, index_from, index_to) 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 array.fill()'s last parameter, `index_to`, must be one greater than the last index the function will fill. The remaining elements will hold `na` values, as the array.new() 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=5 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=5 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=5 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=5 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

Pine Script's 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 the get() method 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=5 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=5 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=5 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=5 indicator("Lows from new highs", "", true) var lows = array.new<float>(0) flushLows = false // Remove last element from the 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=5 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 `i_pivotLegs` bars back. 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))

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=5 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)

Copying

You can copy an array using array.copy(). Here we copy the array `a` to a new array named `_b`:

//@version=5 indicator("`array.copy()`") a = array.new<float>(0) array.push(a, 0) array.push(a, 1) if barstate.islast b = array.copy(a) array.push(b, 2) label.new(bar_index, 0, "a: " + str.tostring(a) + "\nb: " + str.tostring(b), size = size.large)

Note that simply using `_b = a` in the previous example would not have copied the array, but only its ID. From thereon, both variables would point to the same array, so using either one would affect the same array.

Joining

Use array.join() to concatenate all of the elements in the array into a string and separate these elements with the specified separator:

//@version=5 indicator("") v1 = array.new<string>(10, "test") v2 = array.new<string>(10, "test") array.push(v2, "test1") v3 = array.new_float(5, 5) v4 = array.new_int(5, 5) l1 = label.new(bar_index, close, array.join(v1)) l2 = label.new(bar_index, close, array.join(v2, ",")) l3 = label.new(bar_index, close, array.join(v3, ",")) l4 = label.new(bar_index, close, array.join(v4, ","))

Sorting

Arrays containing “int” or “float” elements can be sorted in either ascending or descending order using array.sort(). The `order` parameter is optional and defaults to order.ascending. As all `array.*()` function arguments, it is qualified as “series”, so can be determined at runtime, as is done here. Note that in the example, which array is sorted is also determined at runtime:

```
//@version=5 indicator("`array.sort()`") a = array.new<float>(0) b = array.new<float>(0) array.push(a, 2) array.push(a, 0) array.push(a, 1) array.push(b, 4) array.push(b, 3) array.push(b, 5) if barstate.islast barUp = close > open array.sort(barUp ? a : b, barUp ? order.ascending : order.descending) label.new(bar_index, 0, "a " + (barUp ? "is sorted ▲: " : "is not sorted: ") + str.tostring(a) + "\n\n" + "b " + (barUp ? "is not sorted: " : "is sorted ▼: ") + str.tostring(b), size = size.large)
```

Another useful option for sorting arrays is to use the array.sort_indices() function, which takes a reference to the original array and returns an array containing the indices from the original array. Please note that this function won't modify the original array. The `order` parameter is optional and defaults to order.ascending.

Reversing

Use array.reverse() to reverse an array:

```
//@version=5 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))
```

Slicing

Slicing an array using array.slice() creates a shallow copy of a subset of the parent array. You determine the size of the subset to slice using the `index_from` and `index_to` parameters. The `index_to` argument must be one greater than the end of the subset you want to slice.

The shallow copy created by the slice acts like a window on the parent array's content. The indices used for the slice define the window's position and size over the parent array. If, as in the example below, a slice is created from the first three elements of an array (indices 0 to 2), then regardless of changes made to the parent array, and as long as it contains at least three elements, the shallow copy will always contain the parent array's first three elements.

Additionally, once the shallow copy is created, operations on the copy are mirrored on the parent array. Adding an element to the end of the shallow copy, as is done in the following example, will widen the window by one element and also insert that element in the parent array at index 3. In this example, to slice the subset from index 0 to index 2 of array `a`, we must use `_sliceOfA = array.slice(a, 0, 3)`:

```
//@version=5 indicator("`array.slice()`") a = array.new<float>(0) array.push(a, 0) array.push(a, 1) array.push(a, 2) array.push(a, 3) if barstate.islast // Create a shadow of elements at index 1 and 2 from array `a`. sliceOfA = array.slice(a, 0, 3) label.new(bar_index, 0, "BEFORE\na: " + str.tostring(a) + "\nsliceOfA: " + str.tostring(sliceOfA)) // Remove first element of parent array `a`. array.remove(a, 0) // Add a new element at the end of the shallow copy, thus also affecting the original array `a`. array.push(sliceOfA, 4) label.new(bar_index, 0, "AFTER\na: " + str.tostring(a) + "\nsliceOfA: " + str.tostring(sliceOfA), style = label.style_label_up)
```

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 occurence is the one with the lowest index. We can also find the last occurrence of a value with array.lastindexof():

//@version=5 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.

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 v5 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 those runtime errors in this section.

Index xx is out of bounds. Array size is yy

This will most probably be the most frequent error you encounter. It will happen when you reference an nonexistent array index. The “xx” value will be the value of the faulty index you tried to use, and “yy” will be the size of the array. Recall that array indices start at zero — not one — and end at the array's size, minus one. An array of size 3's last valid index is thus `2`.

To avoid this error, you must make provisions in your code logic to prevent using an index lying outside of the array's index boundaries. This code will generate the error because the last index we use in the loop is outside the valid index range for the array:

//@version=5 indicator("Out of bounds index") a = array.new<float>(3) for i = 1 to 3 array.set(a, i, i) plot(array.pop(a))

The correct `for` statement is:

for i = 0 to 2

To loop on all array elements in an array of unknown size, use:

```
//@version=5 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 you size arrays dynamically using a field in your script's *Settings/Inputs* tab, protect the boundaries of that value using input.int()'s `minval` and `maxval` parameters:

```
//@version=5 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'

When an array ID is initialized to `na`, operations on it are not allowed, since no array exists. All that exists at that point is an array variable containing the `na` value rather that a valid array ID pointing to an existing array. Note that an array created with no elements in it, as you do when you use `a = array.new_int(0)`, has a valid ID nonetheless. This code will throw the error we are discussing:

```
//@version=5 indicator("Out of bounds index") array<int> a = na array.push(a, 111) label.new(bar_index, 0, "a: " + str.tostring(a))
```

To avoid it, create an array with size zero using:

```
array<int> a = array.new_int(0)
```

or:

```
a = array.new_int(0)
```

Array is too large. Maximum size is 100000

This error will appear 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 will occur if array.shift() is called to remove the first element of an empty array.

Cannot use pop() if array is empty.

This error will occur 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 poiting to the “window” at the parent array's indices 3 to 4, is pointing out of the parent array's boundaries:

```
//@version=5 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)
```

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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.

In this page we present an overview of some of Pine Script'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 v5 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.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, 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.new(), color.rgb(), 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 v5 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.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Conditional structures

Introduction

The conditional structures in Pine Script® are if and switch. They can be used:

- For their side effects, i.e., when they don't return a value but do things, like reassign values to variables or call functions.
- To return a value or a tuple which can then be assigned to one (or more, in the case of tuples) variable.

Conditional structures, like the for and while structures, can be embedded; you can use an if or switch inside another structure.

Some Pine Script built-in functions cannot be called from within the local blocks of conditional structures. They are: alertcondition(), barcolor(), fill(), hline(), indicator(), library(), plot(), plotbar(), plotcandle(), plotchar(), plotshape(), strategy(). This 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.

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 ican be done using if structures, as we do here to restrict updates to our label to the chart's last bar:

//@version=5 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.

This is an example:

//@version=5 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 exectuted, na is returned.

`switch` with an expression

Let's look at an example of a switch using an expression:

//@version=5 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=5 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 to be executed on each bar, which would result in a compiler warning and erratic behavior:

//@version=5 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)

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"
ADVANCED

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Enums

> Notice!This page contains advanced material. If you are a beginning Pine Script® programmer, we recommend you become familiar with other, more accessible Pine Script features before you venture here.

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. 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 “simple” fields representing the specific, *predefined values* of the same *unique* type that variables, expressions, and collections can accept.

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:

//@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* the fields belong to.
- We used the `//@enum` and `//@field` annotations to document the meaning of the enum and its fields.
- Unlike the `buy` and `sell` fields, the `neutral` field does not include a specified title. As such, its title is the "string" representation of its *name* ("neutral").

To retrieve a member of an enum, reference its field name using *dot notation* syntax, i.e.:

enumName.fieldName

As with other types, scripts can assign enum members to variables, function parameters, and UDT fields, allowing strict control over their allowed values.

For instance, this line of code 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 declaring the variable's *type* as `Signal` because the compiler can automatically infer that information from the assigned value. If we use na as the initial value instead, we must use `Signal` as the type keyword to specify that `mySignal` will accept a `Signal` member:

Signal mySignal = na

Using enums

Scripts can compare enum members with the == and != operators and use them 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 uses `OscType` members within a switch structure to determine which oscillator it calculates. The script calls this function using the value from an enum input as the `selection` argument and plots the resulting oscillator:

//@version=5 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. OscType oscInput = input.enum(OscType.rsi, "Oscillator type") //@function Calculates one of three oscillators based on a specified `selection`. //@param source The series of values to process. //@param length The number of bars in the calculation. //@param selection Determines which oscillator to calculate. 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`. plot(calcOscillator(close, 20, oscInput))

Note that:

- The `selection` parameter of the `calcOscillator()` function can only take on one of *four* 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. See this section to learn more about enum inputs.

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 raises a *compilation error* because it can't use a member of the `OscType2` enum as the `selection` argument in the `calcOscillator()` call:

//@version=5 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. OscType2 oscInput = input.enum(OscType2.rsi, "Oscillator type") //@function Calculates one of three oscillators based on a specified `selection`. //@param source The series of values to process. //@param length The number of bars in the calculation. //@param selection Determines which oscillator to calculate. 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`. // Raises a compilation error because only members of `OscType` are allowed. plot(calcOscillator(close, 20, 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 in the script's “Settings/Inputs” tab when calling the input.enum() function.

Scripts can also utilize enum field titles in their calculations and logic. Use the string conversion function (str.tostring()) on an enum field to access its title.

The following example combines different enum field titles to construct a ticker ID for requesting data from another context. The script declares two enums, `Exchange` and `Pair`, whose respective fields represent *exchange* and *currency pair* names. It uses input.enum() to assign user-specified enum members to the `exchangeInput` and `pairInput` variables, then retrieves the “string” titles from those variables with str.tostring() and concatenates them to form an “Exchange:Symbol” pair for use in a request.security() call:

//@version=5 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 declare a collection of enum members, include the enum’s *name* in the collection’s type template.

For example, this code block creates an empty array to hold members of the `FooBar` enum. The only values this array can allow as elements 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 only accept `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=5 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` and initializes its cell on the *first bar*, then updates the cell's text on the *latest bar*. See this section of the Profiling and optimization page to learn more.

Shadowing

To avoid potential conflicts where namespaces added to Pine Script in the future would conflict with the names of enums in existing scripts, enum names can *shadow* some of Pine's namespaces.

For example, one can declare an enum like the following, whose name shadows the `syminfo.*` namespace:

//@version=5 indicator("Shadowing demo") enum syminfo abcd log.info(str.tostring(syminfo.abcd))

However, using such a name for an enum is only allowed if the enum's fields **do not** have names matching any of the namespace's built-ins. Otherwise, Pine will not be able to determine which value the script is supposed to use, resulting in a compilation error:

//@version=5 indicator("Name conflict demo") enum syminfo abcd tickerid // This matches the built-in `syminfo.tickerid` variable, causing a compilation error. log.info(str.tostring(syminfo.tickerid))

Additionally, one cannot use any of Pine's built-in type names as the name of an enum.

> Notice!While it is possible for some enum names to shadow language namespaces, as shown above, we recommend choosing **unique** names for enums wherever possible for more readable code that's easier to maintain.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Execution model

The execution model of the Pine Script® runtime is intimately linked to Pine Script's time series and type system. Understanding all three is key to making the most of the power of Pine Script.

The execution model determines how your script is executed on charts, and thus how the code you write in scripts works. Your code would do nothing were it not for Pine Script's runtime, which kicks in after your code has compiled and it is executed on your chart because one of the events triggering the execution of a script has occurred.

When a Pine script is loaded on a chart it executes once on each historical bar using the available OHLCV (open, high, low, close, volume) values for each bar. Once the script's execution reaches the rightmost bar in the dataset, if trading is currently active on the chart's symbol, then Pine Script *indicators* will execute once every time an *update* occurs, i.e., price or volume changes. Pine Script *strategies* will by default only execute when the rightmost bar closes, but they can also be configured to execute on every update, like indicators do.

All symbol/timeframe pairs have a dataset comprising a limited number of bars. When you scroll a chart to the left to see the dataset's earlier bars, the corresponding bars are loaded on the chart. The loading process stops when there are no more bars for that particular symbol/timeframe pair or the maximum number of bars your account type permits has been loaded. You can scroll the chart to the left until the very first bar of the dataset, which has an index value of 0 (see bar_index).

When the script first runs on a chart, all bars in a dataset are *historical bars*, except the rightmost one if a trading session is active. When trading is active on the rightmost bar, it is called the *realtime bar*. The realtime bar updates when a price or volume change is detected. When the realtime bar closes, it becomes an *elapsed realtime bar* and a new realtime bar opens.

Calculation based on historical bars

Let's take a simple script and follow its execution on historical bars:

//@version=5 indicator("My Script", overlay = true) src = close a = ta.sma(src, 5) b = ta.sma(src, 50) c = ta.cross(a, b) plot(a, color = color.blue) plot(b, color = color.black) plotshape(c, color = color.red)

On historical bars, a script executes at the equivalent of the bar's close, when the OHLCV values are all known for that bar. Prior to execution of the script on a bar, the built-in variables such as `open`, `high`, `low`, `close`, `volume` and `time` are set to values corresponding to those from that bar. A script executes **once per historical bar**.

Our example script is first executed on the very first bar of the dataset at index 0. Each statement is executed using the values for the current bar. Accordingly, on the first bar of the dataset, the following statement:

src = close

initializes the variable `src` with the `close` value for that first bar, and each of the next lines is executed in turn. Because the script only executes once for each historical bar, the script will always calculate using the same `close` value for a specific historical bar.

The execution of each line in the script produces calculations which in turn generate the indicator's output values, which can then be plotted on the chart. Our example uses the `plot` and `plotshape` calls at the end of the script to output some values. In the case of a strategy, the outcome of the calculations can be used to plot values or dictate the orders to be placed.

After execution and plotting on the first bar, the script is executed on the dataset's second bar, which has an index of 1. The process then repeats until all historical bars in the dataset are processed and the script reaches the rightmost bar on the chart.

Calculation based on realtime bars

The behavior of a Pine script on the realtime bar is very different than on historical bars. Recall that the realtime bar is the rightmost bar on the chart when trading is active on the chart's symbol. Also, recall that strategies can behave in two different ways in the realtime bar. By default, they only execute when the realtime bar closes, but the `calc_on_every_tick` parameter of the `strategy` declaration statement can be set to true to modify the strategy's behavior so that it executes each time the realtime bar updates, as indicators do. The behavior described here for indicators will thus only apply to strategies using `calc_on_every_tick=true`.

The most important difference between execution of scripts on historical and realtime bars is that while they execute only once on historical bars, scripts execute every time an update occurs during a realtime bar. This entails that built-in variables such as `high`, `low` and `close` which never change on a historical bar, **can** change at each of a script's iteration in the realtime bar. Changes in the built-in variables used in the script's calculations will, in turn, induce changes in the results of those calculations. This is required for the script to follow the realtime price action. As a result, the same script may produce different results every time it executes during the realtime bar.

Note: In the realtime bar, the `close` variable always represents the **current price**. Similarly, the `high` and `low` built-in variables represent the highest high and lowest low reached since the realtime bar's beginning. Pine Script's built-in variables will only represent the realtime bar's final values on the bar's last update.

Let's follow our script example in the realtime bar.

When the script arrives on the realtime bar it executes a first time. It uses the current values of the built-in variables to produce a set of results and plots them if required. Before the script executes another time when the next update happens, its user-defined variables are reset to a known state corresponding to that of the last *commit* at the close of the previous bar. If no commit was made on the variables because they are initialized every bar, then they are reinitialized. In both cases their last calculated state is lost. The state of plotted labels and lines is also reset. This resetting of the script's user-defined variables and drawings prior to each new iteration of the script in the realtime bar is called *rollback*. Its effect is to reset the script to the same known state it was in when the realtime bar opened, so calculations in the realtime bar are always performed from a clean state.

The constant recalculation of a script's values as price or volume changes in the realtime bar can lead to a situation where variable `c` in our example becomes true because a cross has occurred, and so the red marker plotted by the script's last line would appear on the chart. If on the next price update the price has moved in such a way that the `close` value no longer produces calculations making `c` true because there is no longer a cross, then the marker previously plotted will disappear.

When the realtime bar closes, the script executes a last time. As usual, variables are rolled back prior to execution. However, since this iteration is the last one on the realtime bar, variables are committed to their final values for the bar when calculations are completed.

To summarize the realtime bar process:

- A script executes **at the open of the realtime bar and then once per update**.
- Variables are rolled back **before every realtime update**.
- Variables are committed **once at the closing bar update**.

Events triggering the execution of a script

A script is executed on the complete set of bars on the chart when one of the following events occurs:

- A new symbol or timeframe is loaded on a chart.
- A script is saved or added to the chart, from the Pine Script Editor or the chart's “Indicators & strategies” dialog box.
- A value is modified in the script's “Settings/Inputs” dialog box.
- A value is modified in a strategy's “Settings/Properties” dialog box.
- A browser refresh event is detected.

A script is executed on the realtime bar when trading is active and:

- One of the above conditions occurs, causing the script to execute on the open of the realtime bar, or
- The realtime bar updates because a price or volume change was detected.

Note that when a chart is left untouched when the market is active, a succession of realtime bars which have been opened and then closed will trail the current realtime bar. While these *elapsed realtime bars* will have been *confirmed* because their variables have all been committed, the script will not yet have executed on them in their *historical* state, since they did not exist when the script was last run on the chart's dataset.

When an event triggers the execution of the script on the chart and causes it to run on those bars which have now become historical bars, the script's calculation can sometimes vary from what they were when calculated on the last closing update of the same bars when they were realtime bars. This can be caused by slight variations between the OHLCV values saved at the close of realtime bars and those fetched from data feeds when the same bars have become historical bars. This behavior is one of the possible causes of *repainting*.

More information

- The built-in `barstate.*` variables provide information on the type of bar or the event where the script is executing. The page where they are documented also contains a script that allows you to visualize the difference between elapsed realtime and historical bars, for example.
- The Strategies page explains the details of strategy calculations, which are not identical to those of indicators.

Historical values of functions

Every function call in Pine leaves a trail of historical values that a script can access on subsequent bars using the [] operator. The historical series of functions depend on successive calls to record the output on

every bar. When a script does not call functions on each bar, it can produce an inconsistent history that may impact calculations and results, namely when it depends on the continuity of their historical series to operate as expected. The compiler warns users in these cases to make them aware that the values from a function, whether built-in or user-defined, might be misleading.

To demonstrate, let's write a script that calculates the index of the current bar and outputs that value on every second bar. In the following script, we've defined a `calcBarIndex()` function that adds 1 to the previous value of its internal `index` variable on every bar. The script calls the function on each bar that the `condition` returns `true` on (every other bar) to update the `customIndex` value. It plots this value alongside the built-in `bar_index` to validate the output:

//@version=5 indicator("My script") //@function Calculates the index of the current bar by adding 1 to its own value from the previous bar. // The first bar will have an index of 0. calcBarIndex() => int index = na index := nz(index[1], replacement = -1) + 1 //@variable Returns `true` on every other bar. condition = bar_index % 2 == 0 int customIndex = na // Call `calcBarIndex()` when the `condition` is `true`. This prompts the compiler to raise a warning. if condition customIndex := calcBarIndex() plot(bar_index, "Bar index", color = color.green) plot(customIndex, "Custom index", color = color.red, style = plot.style_cross)

Note that:

- The nz() function replaces na values with a specified `replacement` value (0 by default). On the first bar of the script, when the `index` series has no history, the na value is replaced with -1 before adding 1 to return an initial value of 0.

Upon inspecting the chart, we see that the two plots differ wildly. The reason for this behavior is that the script called `calcBarIndex()` within the scope of an if structure on every other bar, resulting in a historical output inconsistent with the `bar_index` series. When calling the function once every two bars, internally referencing the previous value of `index` gets the value from two bars ago, i.e., the last bar the function executed on. This behavior results in a `customIndex` value of half that of the built-in `bar_index`.

To align the `calcBarIndex()` output with the `bar_index`, we can move the function call to the script's global scope. That way, the function will execute on every bar, allowing its entire history to be recorded and referenced rather than only the results from every other bar. In the code below, we've defined a `globalScopeBarIndex` variable in the global scope and assigned it to the return from `calcBarIndex()` rather than calling the function locally. The script sets the `customIndex` to the value of `globalScopeBarIndex` on the occurrence of the `condition`:

//@version=5 indicator("My script") //@function Calculates the index of the current bar by adding 1 to its own value from the previous bar. // The first bar will have an index of 0. calcBarIndex() => int index = na index := nz(index[1], replacement = -1) + 1 //@variable Returns `true` on every second bar. condition = bar_index % 2 == 0 globalScopeBarIndex = calcBarIndex() int customIndex = na // Assign `customIndex` to `globalScopeBarIndex` when the `condition` is `true`. This won't produce a warning. if condition customIndex := globalScopeBarIndex plot(bar_index, "Bar index", color = color.green) plot(customIndex, "Custom index", color = color.red, style = plot.style_cross)

This behavior can also radically impact built-in functions that reference history internally. For example, the ta.sma() function references its past values “under the hood”. If a script calls this function conditionally rather than on every bar, the values within the calculation can change significantly. We can ensure calculation consistency by assigning ta.sma() to a variable in the global scope and referencing that variable's history as needed.

The following example calculates three SMA series: `controlSMA`, `localSMA`, and `globalSMA`. The script calculates `controlSMA` in the global scope and `localSMA` within the local scope of an if structure. Within the if structure, it also updates the value of `globalSMA` using the `controlSMA` value. As we can see, the values from the `globalSMA` and `controlSMA` series align, whereas the `localSMA` series diverges from the other two because it uses an incomplete history, which affects its calculations:

//@version=5 indicator("My script") //@variable Returns `true` on every second bar. condition = bar_index % 2 == 0 controlSMA = ta.sma(close, 20) float globalSMA = na float localSMA = na // Update `globalSMA` and `localSMA` when `condition` is `true`. if condition globalSMA := controlSMA // No warning. localSMA := ta.sma(close, 20) // Raises warning. This function depends on its history to work as intended. plot(controlSMA, "Control SMA", color = color.green) plot(globalSMA, "Global SMA", color = color.blue, style = plot.style_cross) plot(localSMA, "Local SMA", color = color.red, style = plot.style_cross)

Why this behavior?

This behavior is required because forcing the execution of functions on each bar would lead to unexpected results in those functions that produce side effects, i.e., the ones that do something aside from returning the value. For example, the label.new() function creates a label on the chart, so forcing it to be called on every bar even when it is inside of an if structure would create labels where they should not logically appear.

Exceptions

Not all built-in functions use their previous values in their calculations, meaning not all require execution on every bar. For example, math.max() compares all arguments passed into it to return the highest value. Such functions that do not interact with their history in any way do not require special treatment.

If the usage of a function within a conditional block does not cause a compiler warning, it's safe to use without impacting calculations. Otherwise, move the function call to the global scope to force consistent execution. When keeping a function call within a conditional block despite the warning, ensure the output is correct at the very least to avoid unexpected results.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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=5 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 leverage 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=5 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=5 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 `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:

```
[var_declaration =] loop_header
    statements | continue | break
    return_expression
```

Where:

- `loop_header` refers to the loop structure's header statement, which defines the criteria that control its iterations.
- `statements` refers to the code statements and expressions within the loop's body, i.e., the *indented* block of code beneath the loop header. All statements within the body belong 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 for more information.
- `return_expression` represents the last code line or block within the loop's body. The loop returns the results from this code after the final iteration. To use the values from a loop's return expression, *assign* the results to a variable or tuple.
- `var_declaration` is an optional variable or tuple declaration. If a loop statement includes a declared variable or tuple, the `return_expression` determines the assigned values. If the header's conditions do not allow the loop to iterate, the assigned values are na.

Scope

All code lines that a script will execute 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=5 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=5 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* a result. The values a loop returns come from the *latest* execution of the *last* expression or nested structure within the loop body as of the *final* iteration. Loops return na values when no iterations occur. A loop's returned results are *usable* only if they are not of the void type and the script assigns a variable or tuple declaration to the loop statement. The declared variables hold the values from the return expression for use in additional calculations outside the loop's scope.

The values a loop returns may 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 stops *prematurely* due to a `continue` or `break` keyword, it returns the values 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.

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=5 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 executions of its local code block over a *defined number* of iterations.

Pine Script uses the following syntax to define a for loop:

```
[var_declaration =] 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 *variable* whose value the loop increments after each iteration.
- `from_num` determines the `counter` variable's *initial value*, i.e., the value on the first iteration.
- `to_num` determines the `counter` variable's *final value*, i.e., the maximum value of the loop counter that the header allows a new iteration for. The loop increments the `counter` value by a fixed amount until it *reaches or passes* this value.
- `step_num` is the amount by which the `counter` value increases or decreases after each iteration until it reaches or passes the `to_num`. Specifying this value is optional. The default value is 1.

See the Common characteristics section above for detailed information about the `var_declaration`, `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:

//@version=5 indicator("Simple `for` loop demo") if barstate.islastconfirmedhistory // Define a `for` loop that iterates from `i == 0` to `i == 10`. for i = 0 to 10 // Draw a new `label` on the current bar at the `i` level. 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 would behave the same if we named the counter `level` 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 loop's header specifies that the counter starts at 0 when the script enters the loop, and its value increases by 1 after each iteration until it reaches 10, at which point the last iteration occurs. In other words, the loop executes *11 iterations* in total.

A for loop's local block repeatedly executes for the number of times required for its counter variable's value to reach the specified `to_num` boundary. The counter's defined boundaries and step size directly control the number of loop iterations. Therefore, a script establishes the expected number of iterations *before* the loop starts. As such, for loops are best suited for iterative tasks where the maximum number of iterations is knowable *in advance*.

The example below calculates and plots the volume-weighted moving average (VWMA) of open prices across `maLengthInput` chart bars, and it analyzes the differences between the current `vwmaOpen` and past `vwmaOpen` values within a for loop on the last historical chart bar. On each loop iteration, the script retrieves a previous bar's `vwmaOpen` value, takes the difference between that value and the current `vwmaOpen`, and displays the result in a label at the corresponding historical bar's opening price:

//@version=5 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) //On last bar, loop through `lengthInput` most recent historical bars and output `label` of differences in `vwmaOpen`. if barstate.islastconfirmedhistory 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 the `vwmaDifference` with sign specification and rounding. 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`. 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 counter's value starts at the `lookbackInput` and *decreases* by 1 until `i == 1`.

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:

for i = 0 to array.size(myArray) – 1 element = array.get(i)

Note that:

- Array *indexing* starts from 0, but the array.size() function *counts* the elements (i.e., starting from 1). Therefore, we have to subtract one from the array's size to get the maximum index value. This way, the loop counter avoids representing an array index that is out of bounds 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 calculates the RSI and momentum of close prices over three different lengths (10, 20, and 50) and displays their values within 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` to access its elements, convert their values to strings, and populate the remaining table cells:

//@version=5 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, which allows us 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 this section of the Profiling and optimization page for more information.

`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 conditional expression remains `true`.

Pine Script uses the following syntax to define a while loop:

```
[var_declaration =] 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.

See the Common characteristics section above for detailed information about the `var_declaration`, `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=5 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 may not change on a specific iteration, the *precise* number of expected iterations may **not** be knowable *before* the loop begins, unlike a for loop. Therefore, while loops are advantageous 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.

A while loop is necessary in this case because the current bar's channel values are *not* knowable in advance, and we **cannot** predict the precise number of iterations required to encapsulate all the consecutive bars within the channel's range:

//@version=5 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*). As such, similar to a for loop, a script establishes the expected number of for…in loop iterations *before* the loop starts.

Pine Script features *two* general forms of the for…in loop statement. The *first form* uses the following syntax:

```
[var_declaration =] for item in collection_id
    statements | continue | break
    return_expression
```

Where `item` is a *variable* that holds sequential values from the specified `collection_id`. The variable's value 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*:

```
[var_declaration =] 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 for more information.

See the Common characteristics section above for detailed information about the `var_declaration`, `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.

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, one can use a for loop to access an array's elements using the counter variable as the lookup index in functions such as array.get(). However, to prevent out-of-bounds errors, programmers must ensure the counter always represents a *valid index*. 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 0 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=5 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=5 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", as array indexing 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 calculated pivot high levels, 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. The script accesses each line within the array using a for…in loop, analyzing and modifying the properties of the `pivotLine` retrieved 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 for visual reference:

```
//@version=5 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) //@variable The average active `highPivotPrice`. float avgHiPivot = ta.sma(highPivotPrice, maxRecentLines) 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 each active `pivotLine`'s price with the current high value, and it uses the prices to calculate the `avgActivePivot` on each bar.
- 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 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 *one less* than the row count, which represents the *last* row index:

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 displays a custom "string" representing the rows of a matrix with extra information, which it displays within a label. When the script executes on the last historical bar, it creates a 3x3 `randomMatrix` populated with random values. Then, using the first form of the for…in loop, the script iterates through each `row` in the `randomMatrix` to create a "string" representing the row's contents, its average, and whether the average is above 0.5, and it concatenates that "string" with the `labelText`. After the loop ends, the script creates a label displaying the `labelText` value:

//@version=5 indicator("`for row in matrix` demo", "Custom matrix label") //@variable Generates a random 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=5 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 with an assigned map of "string" keys and "float" values. The script puts the keys from the `newKeys` array into the collection with corresponding random values. It then uses a for…in loop to iterate through the key-value pairs from the `simpleMap` and construct the `displayText`. After the loop ends, the script shows the `displayText` within a label to visualize the result:

//@version=5 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 utilizes 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 `simpleMap`, and the second iterates

directly through the map's key-value pairs to construct the custom `displayText`.

> Notice!
>
> Scripts can modify the sizes of arrays and matrices while iterating through them with a for…in loop. However, it's crucial to note that while a script iterates through a map with a for…in loop, that map's size **cannot** change. Attempting to *add* or *remove* key-value pairs while looping directly through a map will cause a *runtime error*.
>
> To 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

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Maps

> Notice!This page contains advanced material. If you are a beginning Pine Script® programmer, we recommend you become familiar with other, more accessible Pine Script features before you venture here.

Introduction

Pine Script Maps are collections that store elements in *key-value pairs*. They allow scripts to collect multiple value references associated with unique identifiers (keys).

Unlike arrays and matrices, maps are *unordered* collections. Scripts quickly access a map's values by referencing the keys from the key-value pairs put into them rather than traversing an internal index.

A map's keys can be of any fundamental type or enum type, and its values can be of any available type. Maps cannot directly use other collections (maps, arrays, or matrices) as values, but they can hold UDT instances containing these data structures within their fields. See this section for more information.

As with other collections, maps can contain up to 100,000 elements in total. Since each key-value pair in a map consists of two elements (a unique *key* and its associated *value*), the maximum number of key-value pairs a map can hold is 50,000.

Declaring a map

Pine Script uses the following syntax to declare maps:

```
[var/varip ][map<keyType, valueType> ]<identifier> = <expression>
```

Where `<keyType, valueType>` is the map's type template that declares the types of keys and values it will contain, and the `<expression>` returns either a map instance or `na`.

When declaring a map variable assigned to `na`, users must include the map keyword followed by a type template to tell the compiler that the variable can accept maps with `keyType` keys and `valueType` values.

For example, this line of code declares a new `myMap` variable that can accept map instances holding pairs of string keys and float values:

map<string, float> myMap = na

When the `<expression>` is not `na`, the compiler does not require explicit type declaration, as it will infer the type information from the assigned map object.

This line declares a `myMap` variable assigned to an empty map with string keys and float values. Any maps assigned to this variable later must have the same key and value types:

myMap = map.new<string, float>()

Using `var` and `varip` keywords

Users can include the var or varip keywords to instruct their scripts to declare map variables only on the first chart bar. Variables that use these keywords point to the same map instances on each script iteration until explicitly reassigned.

For example, this script declares a `colorMap` variable assigned to a map that holds pairs of string keys and color values on the first chart bar. The script displays an `oscillator` on the chart and uses the values it put into the `colorMap` on the *first* bar to color the plots on *all* bars:

//@version=5 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 as ones using var on historical data, but they update their key-value pairs for realtime bars (i.e., the bars since the script's last compilation) on each new price tick. Maps assigned to varip variables can only hold values of int, float, bool, color, or string types or user-defined types that exclusively contain within their fields these types or collections (arrays, matrices, or maps) of 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 map `id`.

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 its value on the chart:

//@version=5 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 data.put() 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 a value 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` points to that same object without copying it. Modifying the value referenced by a key-value pair will also affect 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 with the “B” key affects the one referenced by the “A” key, as shown by the candle plot of the “A” object's fields:

//@version=5 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 myMap.put() call. For more information, see this section of our User Manual's page on objects.

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 key/value references within a map `id` 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=5 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 keys() and 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 to put 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 pair into the map *after* the one with the “Third” key, the pair's key and value are still second in the `keys` and `values` arrays since the key was already present in `m` *before* the change:

//@version=5 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)

> Notice!The elements in a map.values() array point to the same values as the map `id`. Consequently, when the map's values are of *reference types*, including line, linefill, box, polyline, label, table, chart.point,

> or UDTs, modifying the instances referenced by the map.values() array will also affect those referenced by the map `id` since the contents of both collections point to identical objects.

`map.contains()`

To check if a specific `key` exists within a map `id`, 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=5 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 a map `id`, 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 `key`. Otherwise, it returns na.

To remove all key-value pairs from a map `id` at once, use map.clear().

The following script creates a new `m` map, puts key-value pairs into the map, uses m.remove() within a loop to remove each valid `key` listed in the `removeKeys` array, then calls m.clear() to remove all remaining key-value pairs. It uses a custom `debugLabel()` method to display the size, keys, and values of `m` after each change:

```
//@version=5 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 were present in the keys of `m`. 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 uses mapA.put_all(mapB) 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=5 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 a map's keys() array and 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=5 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 a map `id` 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 `m` and updates the values associated with its keys. The script displays the key-value pairs in `m` and `mCopy` on the chart using our custom `debugLabel()` method:

//@version=5 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 random `float` values. m = map.new<string, float>() // Assign random 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 holding values of a *special 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 will affect the instances 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 of string keys and label values and puts a key-value pair into it. The script copies the map to a `shallow` variable via the built-in 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=5 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 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 values() array and the value from `globalData.get(50)` (i.e., the 50-bar EMA):

//@version=5 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 use other maps, arrays, or matrices as values, but they can hold values of a user-defined type that contains collections within its fields.

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 use other collections as values, we will first create a *wrapper type* with a field to hold 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 of string keys and `Wrapper` values, where the `data` field of each value in the map points to a `map<string, float>` instance:

mapOfMaps = map.new<string, Wrapper>()

The script below uses this concept to construct a map containing 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 `mapOfMaps`, then creates a string representation of the nested maps with a user-defined `toString()` method, which it displays on the chart in a

label:

//@version=5 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

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Matrices

> Notice!This page contains advanced material. If you are a beginning Pine Script® programmer, we recommend you become familiar with other, more accessible Pine Script features before you venture here.

Introduction

Pine Script Matrices are collections that store value references in a rectangular format. They are the equivalent of two-dimensional array objects with functions and methods for inspection, modification, and specialized calculations. As with arrays, all matrix elements must be of the same type, user-defined type, or enum type.

Matrices reference their elements using *two* indices: one index for their *rows* and the other for their *columns*. Each index starts at 0 and extends to the number of rows/columns in the matrix minus one. 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 (e.g., a 5x5 matrix has a total of 25). Like arrays, the total number of elements in a matrix cannot exceed 100,000.

Declaring a matrix

Pine Script uses the following syntax for matrix declaration:

```
[var/varip ][matrix<type> ]<identifier> = <expression>
```

Where `<type>` is a type template for the matrix that declares the type of values it will contain, and the `<expression>` returns either a matrix instance of the type or `na`.

When declaring a matrix variable as `na`, users must specify that the identifier will reference matrices of a specific type by including the matrix keyword followed by a type template.

This line declares a new `myMatrix` variable with a value of `na`. It explicitly declares the variable as `matrix<float>`, which tells the compiler that the variable can only accept matrix objects containing float values:

matrix<float> myMatrix = na

When a matrix variable is not assigned to `na`, the matrix keyword and its type template are optional, as the compiler will use the type information from the object the variable references.

Here, we declare a `myMatrix` variable referencing a new `matrix<float>` instance with two rows, two columns, and an `initial_value` of 0. The variable gets its type information from the new object in this case, so it doesn't require an explicit type declaration:

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 only once rather than on every bar. A matrix variable declared with this keyword will point to the same instance throughout the span of the chart unless the script explicitly assigns another matrix to it, allowing a matrix and its element references to persist between script iterations.

This script declares an `m` variable assigned to 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 on the chart. As we see from the plot, the value of m.get(0, 0) persists between bars, never returning to the initial value of 0:

//@version=5 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 as ones using var on historical data, but they update their values for realtime bars (i.e., the bars since the script's last compilation) on each new price tick. Matrices assigned to varip variables can only hold int, float, bool, color, or string types or user-defined types that exclusively contain within their fields these types or collections (arrays, matrices, or maps) of 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 the m.get() and m.set() methods. 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=5 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 a random value:

myMatrix = matrix.new<float>(4, 4) myMatrix.fill(math.random())

Note when using matrix.fill() with matrices containing special 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 object on the first bar. On each bar, the script sets the `x` attribute of the label referenced at row 0, column 0 to bar_index, and the `text` attribute 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 points to the *same* instance, resulting in only one label on the chart that updates its `x` and `text` attributes on each bar:

//@version=5 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

Matrices facilitate the retrieval of all values from a specific row or column via the matrix.row() and matrix.col() functions. These functions return the values as an array object sized according to the other dimension of the matrix, i.e., the size of a matrix.row() array equals the number of columns and the size of a matrix.col() array equals the number of rows.

The script below populates a 3x2 `m` matrix with the values 1 - 6 on the first chart bar. It calls the m.row() and m.col() methods 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=5 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) //@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 rows() and columns() methods rather than array.size() to demonstrate that the size of the `row0` array equals the number of columns and the size of the `column0` array equals the number of rows.

matrix.row() and matrix.col() copy the references in 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=5 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 returned by 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 point to 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 since both elements reference the same object:

//@version=5 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 value 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_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 1.

The script below declares an empty `m` matrix and inserts rows and columns using the m.add_row() and m.add_col() 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 `m` after each row/column insertion with a user-defined `debugLabel()` function to visualize the process:

//@version=5 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")

> Notice!Just as the row or column arrays retrieved from a matrix of line, linefill, box, polyline, label, table, chart.point, or UDT instances behave as shallow copies, the elements of matrices containing such types reference the same objects as the arrays inserted into them. Modifications to the element values in either object affect the other 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 1.

For this example, we've added these lines of code to our “Rows and columns demo” script from the 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 the m.remove_row() and m.remove_col() methods and displays the rows in a label at `bar_index + 30`. As we can see, `m` has a 3x3 shape after executing 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 locations of the elements at the `row1/column1` and `row2/column2` indices.

Let's add the following lines to the previous example, which swap the first and last rows of `m` 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 the new array 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 add_row() method to insert the new `values` at the `row` index and uses the 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=5 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=5 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 a `row` array for each row in the `m` matrix:

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() of each `row`.

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 via the m.add_col() and m.remove_col() methods on each subsequent bar. It uses `m.rowWiseAvg()` to calculate the array of row-wise `averages`, then it plots the element values on the chart:

//@version=5 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:

- `for...in` loops can also reference the index value of each row. For example, `for [i, row] in m` creates a tuple containing the `i` row index and 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 references.

For example, this script assigns a new matrix to the `myMatrix` variable and adds two columns. It creates a new `myCopy` matrix from `myMatrix` using the myMatrix.copy() method, then adds a new row. It displays the rows of both matrices in labels via the user-defined `debugLabel()` function:

//@version=5 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 point to the same values as the original matrix. When matrices contain 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.

This script declares a `myMatrix` variable with a `newLabel` as the initial value. It then copies `myMatrix` to a `myCopy` variable via 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 element values in `myCopy` affect the values in both matrices:

//@version=5 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 point to copies of the original values) by explicitly copying each object the matrix references.

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 the built-in copy(), we see that there are now two labels on the chart, and any changes to the label from `myCopy` do not affect the one from `myMatrix`:

//@version=5 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 m.submatrix() method, then calls our user-defined `debugLabel()` function to display the rows of both matrices in labels:

//@version=5 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 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 `globalMatrix`, and the `calcAvg()` function references `previous` matrices assigned to `globalMatrix` using the [] operator to calculate a matrix of averages:

//@version=5 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.
- `calcAvg()` calculates the average 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 our

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=5 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 m.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=5 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 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=5 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 using the m.transpose() method, and displays 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:

//@version=5 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 the contents of a matrix via matrix.sort(). Unlike array.sort(), which sorts *elements*, this function organizes all *rows* in a matrix in a specified `order` (order.ascending by default) based on the values in a specified `column`.

This script declares a 3x3 `m` matrix, sorts the rows of the `m1` copy in ascending order based on the first column, then sorts the rows of the `m2` copy in descending order based on the second column. It displays the original matrix and sorted copies in labels using our `debugLabel()` method:

//@version=5 indicator("Sorting rows 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. matrix<int> m = matrix.new<int>() if bar_index == last_bar_index - 1 // Add rows to `m`. m.add_row(0, array.from(3, 2, 4)) m.add_row(1, array.from(1, 9, 6)) m.add_row(2, array.from(7, 8, 9)) m.debugLabel(note = "Original") // Copy `m` and sort rows in ascending order based on the first column (default). matrix<int> m1 = m.copy() m1.sort() m1.debugLabel(bar_index + 10, color.green, note = "Sorted using col 0\n(Ascending)") // Copy `m` and sort rows in descending order based on the second column. matrix<int> m2 = m.copy() m2.sort(1, order.descending) m2.debugLabel(bar_index + 20, color.red, note = "Sorted using col 1\n(Descending)")

It's important to note that matrix.sort() does not sort the columns of a matrix. However, one *can* use this function to sort matrix columns with the help of matrix.transpose().

As an example, this script contains a `sortColumns()` method that uses the sort() method to sort the transpose of a matrix using the column corresponding to the `row` of the original matrix. The script uses this method to sort the `m` matrix based on the contents of its first row:

//@version=5 indicator("Sorting columns 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) //@function Sorts the columns of `this` matrix based on the values in the specified `row`. method sortColumns(matrix<int> this, int row = 0, bool ascending = true) => //@variable The transpose of `this` matrix. matrix<int> thisT = this.transpose() //@variable Is `order.ascending` when `ascending` is `true`, `order.descending` otherwise. order = ascending ? order.ascending : order.descending // Sort the rows of `thisT` using the `row` column. thisT.sort(row, order) //@variable A copy of `this` matrix with sorted columns. result = thisT.transpose() //@variable A 3x3 matrix. matrix<int> m = matrix.new<int>() if bar_index == last_bar_index - 1 // Add rows to `m`. m.add_row(0, array.from(3, 2, 4)) m.add_row(1, array.from(1, 9, 6)) m.add_row(2, array.from(7, 8, 9)) m.debugLabel(note = "Original") // Sort the columns of `m` based on the first row and display the result. m.sortColumns(0).debugLabel(bar_index + 10, note = "Sorted using row 0\n(Ascending)")

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=5 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 get() and set() matrix 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 col() 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=5 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 matrix.avg() by the 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() of the two matrices. This example displays the original matrix, its transpose, and the resulting sum and difference matrices in labels on the chart:

//@version=5 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 = "Aᵀ") // Calculate the sum of the two matrices. The resulting matrix is symmetric. matrix.sum(m, t).debugLabel(bar_index + 20, color.green, note = "A + Aᵀ") // Calculate the difference between the two matrices. The resulting matrix is antisymmetric. matrix.diff(m, t).debugLabel(bar_index + 30, color.red, note = "A - Aᵀ")

Note that:

- In this example, we've labeled the original matrix as “A” and the transpose as “Aᵀ”.
- Adding “A” and “Aᵀ” produces a symmetric matrix, and subtracting them produces an antisymmetric matrix.

`matrix.mult()`

Scripts can multiply two matrices via the matrix.mult() function. This function also facilitates the multiplication of 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 `id1` and the same number of columns as `id2`. 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 `id1` and column in `id2`:

//@version=5 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 `id1` by a single-column matrix, but it returns an array with the same number of elements as the number of rows in `id1`. When matrix.mult() passes a scalar as its `id2` value, the function returns a new matrix whose elements are the elements in `id1` 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 utilizes 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 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=5 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 utility 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 <_PageMatrices_MatrixCalculations_SpecialCalculations_MatrixDet>`, 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 uses the m.inv() and m.pinv() methods to calculate the inverse or pseudoinverse of `m`. The script displays the original matrix, its inverse or pseudoinverse, and their product in labels on the chart:

//@version=5 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() //@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 will only call m.inv() 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`) 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=5 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=5 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/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=5 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=5 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=5 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=5 indicator("Invalid from_row, to_row demo") //@variable A 4x4 matrix filled with a random 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=5 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=5 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=5 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

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Methods

> Notice!This page contains advanced material. If you are a beginning Pine Script® programmer, we recommend you become familiar with other, more accessible Pine Script features before you venture here.

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 includes built-in methods for all *special types*, including array, matrix, map, line, linefill, box, polyline, label, and table. 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=5 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 deviaiton 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.* functions in the script with equivalent methods:

//@version=5 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 deviaiton 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 deviaiton 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 deviaiton 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 deviaiton 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=5 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 deviaiton 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) => na(this) ? "bool(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=5 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=5 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

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Objects

> Notice!This page contains advanced material. If you are a beginning Pine Script® programmer, we recommend you become familiar with other, more accessible Pine Script features before you venture here.

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=5 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=5 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=5 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 objects, allowing users to add virtual dimensions to their data structures. To declare a collection of objects, pass a UDT name into its type template.

This example declares an empty array that will hold 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, use the collection's type keyword followed by its type template. For example:

var array<pivotPoint> pivotHighArray = na pivotHighArray := array.new<pivotPoint>()

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=5 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=5 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=5 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 has fields with *special types* (array, matrix, map, line, linefill, box, polyline, label, table, or chart.point), 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=5 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=5 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 UDTs or object names in existing scripts; as a rule, UDTs and object names shadow the language's namespaces. For example, a UDT or object can use the name of built-in types, such as line or table.

Only the language's five primitive types cannot be used to name UDTs or objects: int, float, string, bool, and color.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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=5 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 (means they need two *operands* — or values — to work on, like in `1 + 2`). The `+` and `-` also serve as unary operators (means they work on one operand, like `-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.

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=5 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. If both operands have a numerical value, the result will be of type *bool*, i.e., `true`, `false` or na.

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 `valueWhenConditionIsTrue` is returned. If `condition` is `false` or na, then `valueWhenConditionIsFalse` is returned.

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 the return values on each side of the `:` are expressions — not local blocks, so they will not affect the limit of 500 local blocks per scope.

`[]` 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 moves on sucessive bars, the offset used with the operator will refer 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 provide for handling these special cases, they can introduce invalid results in your script's calculations which can ripple through all the way to the realtime bar. The na and nz functions are designed to allow for handling such cases.

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.

`=` assignement operator

The `=` operator is used to assign a variable when it is initialized — or declared —, i.e., the first time you use it. It says *this is a new variable that I will be using, and I want it to start on each bar with this value*.

These are all valid variable declarations:

i = 1 MS_IN_ONE_MINUTE = 1000 * 60 showPlotInput = input.bool(true, "Show plots") pHi = 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=5 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 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 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.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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=5

- The version number can be 1 to 5.
- 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 type of script has distinct requirements:

- Indicators must contain at least one function call which produces output on the chart (e.g., plot(), plotshape(), barcolor(), line.new(), etc.).
- Strategies must contain at least one `strategy.*()` call, e.g., strategy.entry().
- Libraries must contain at least one exported function or user-defined type.

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 declaration
- 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 multi-line function declarations 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 v5 indicator can be generated in the Pine Script Editor by using the “Open” button and choosing “New blank indicator”:

//@version=5 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=5 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 v5 strategy by selecting “New blank strategy” instead:

//@version=5 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=5 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

Long lines can be split on multiple lines, or “wrapped”. Wrapped lines must be indented with any number of spaces, provided it's not a multiple of four (those boundaries are used to indent local blocks):

a = open + high + low + close

may be wrapped as:

a = open + high + low + close

A long plot() call may be wrapped as:

plot(ta.correlation(src, ovr, length), color = color.new(color.purple, 40), style = plot.style_area, trackprice = true)

Statements inside user-defined function declarations can also be wrapped. However, since a local block must syntactically begin with an indentation (4 spaces or 1 tab), when splitting it onto the following line, the continuation of the statement must start with more than one indentation (not equal to a multiple of four spaces). For example:

updown(s) => isEqual = s == s[1] isGrowing = s > s[1] ud = isEqual ? 0 : isGrowing ? (nz(ud[1]) <= 0 ? 1 : nz(ud[1])+1) : (nz(ud[1]) >= 0 ? -1 : nz(ud[1])-1)

You can use comments in wrapped lines:

//@version=5 indicator("") c = open > close ? color.red : high > high[1] ? color.lime : // A comment low < low[1] ? color.blue : color.black bgcolor(c)

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=5 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
```

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Time series

Much of the power of Pine Script® stems from the fact that it is designed to process *time series* efficiently. Time series are not a qualified type; they are the fundamental structure Pine Script uses to store the successive values of a variable over time, where each value is tethered to a point in time. Since charts are composed of bars, each representing a particular point in time, time series are the ideal data structure to work with values that may change with time.

The notion of time series is intimately linked to Pine Script's execution model and type system concepts. Understanding all three is key to making the most of the power of Pine Script.

Take the built-in open variable, which contains the “open” price of each bar in the dataset, the *dataset* being all the bars on any given chart. If your script is running on a 5min chart, then each value in the open time series is the “open” price of the consecutive 5min chart bars. When your script refers to open, it is referring to the “open” price of the bar the script is executing on. To refer to past values in a time series, we use the [] history-referencing operator. When a script is executing on a given bar, `open[1]` refers to the value of the open time series on the previous bar.

While time series may remind programmers of arrays, they are totally different. Pine Script does use an array data structure, but it is a completely different concept than a time series.

Time series in Pine Script, combined with its special type of runtime engine and built-in functions, are what makes it easy to compute the cumulative total of close values without using a for loop, with only `ta.cum(close)`. This is possible because although `ta.cum(close)` appears rather static in a script, it is in fact executed on each bar, so its value becomes increasingly larger as the close value of each new bar is added to it. When the script reaches the rightmost bar of the chart, `ta.cum(close)` returns the sum of the close value from all bars on the chart.

Similarly, the mean of the difference between the last 14 high and low values can be expressed as `ta.sma(high - low, 14)`, or the distance in bars since the last time the chart made five consecutive higher highs as `barssince(rising(high, 5))`.

Even the result of function calls on successive bars leaves a trace of values in a time series that can be referenced using the [] history-referencing operator. This can be useful, for example, when testing the close of the current bar for a breach of the highest high in the last 10 bars, but excluding the current bar, which we could write as `breach = close > highest(close, 10)[1]`. The same statement could also be written as `breach = close > highest(close[1], 10)`.

The same looping logic on all bars is applied to function calls such as `plot(open)` which will repeat on each bar, successively plotting on the chart the value of open for each bar.

Do not confuse “time series” with the “series” qualifier. The *time series* concept explains how consecutive values of variables are stored in Pine Script; the “series” qualifier denotes variables whose values can change bar to bar. Consider, for example, the timeframe.period built-in variable which has the “simple” qualifier and “string” type, meaning it is of the “simple string” qualified type. The “simple” qualifier entails that the variable's value is established on bar zero (the first bar where the script executes) and will not change during the script's execution on any of the chart's bars. The variable's value is the chart's timeframe in string format, so `"D"` for a 1D chart, for example. Even though its value cannot change during the script, it would be syntactically correct in Pine Script (though not very useful) to refer to its value 10 bars ago using

`timeframe.period[10]`. This is possible because the successive values of timeframe.period for each bar are stored in a time series, even though all the values in that particular time series are the same. Note, however, that when the [] operator is used to access past values of a variable, it yields a “series” qualified value, even when the variable without an offset uses a different qualifier, such as “simple” in the case of timeframe.period.

When you grasp how time series can be efficiently handled using Pine Script's syntax and its execution model, you can define complex calculations using little code.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Type system

Introduction

The Pine Script® type system determines the compatibility of a script's values with various functions and operations. While it's possible to write simple scripts without knowing anything about the type system, a reasonable understanding of it is necessary to achieve any degree of proficiency with the language, and an in-depth knowledge of its subtleties allows programmers to harness its full potential.

Pine Script uses types to classify all values, and it uses qualifiers to determine whether values and references are constant, established on the first script execution, or dynamic across executions. This system applies to all Pine values and references, including literals, variables, expressions, function returns, and function arguments.

The type system closely intertwines with Pine's execution model and time series concepts. Understanding all three is essential for making the most of the power of Pine Script.

> Notice!For the sake of brevity, we often use “type” to refer to a “qualified type”.

Qualifiers

Pine Script *qualifiers* identify when values are accessible to a script:

- Values and references qualified as const are established at compile time (i.e., when saving the script in the Pine Editor or adding it to the chart).
- Values qualified as input are established at input time (i.e., when confirming values based on user input, primarily from the “Settings/Inputs” tab).
- Values qualified as simple are established at bar zero (i.e., the first script execution).
- Values qualified as series can change throughout the script's executions.

Pine Script bases the dominance of type qualifiers on the following hierarchy: **const < input < simple < series**, where “const” is the *weakest* qualifier and “series” is the *strongest*. The qualifier hierarchy translates to this rule: whenever a variable, function, or operation is compatible with a specific qualified type, values with *weaker* qualifiers are also allowed.

Scripts always qualify their expressions' returned types based on the *dominant qualifier* in their calculations. For example, evaluating an expression that involves “input” and “series” values will return a value qualified as “series”. Furthermore, scripts **cannot** change a value's qualifier to one that's *lower* on the hierarchy. If a value acquires a *stronger* qualifier (e.g., a value initially inferred as “simple” becomes “series” later in the script's executions), that state is irreversible.

It's important to note that “series” values are the **only** ones that can change across script executions, including those from various built-ins, such as close and volume, as well as the results of expressions involving “series” values. All values qualified as “const”, “input”, or “simple” remain consistent across all script executions.

const

Values or references qualified as “const” are established at *compile time*, before the script starts its executions. Compilation initially occurs when saving a script in the Pine Editor, which does not require it to run on a chart. Values or references with the “const” qualifier *never change* between script executions, not even on the first execution.

All *literal* values and the results returned by expressions involving only values qualified as “const” automatically adopt the “const” qualifier.

These are some examples of literal values:

- *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"'`

Our Style guide recommends using uppercase SNAKE_CASE to name “const” variables for readability. While not a requirement, one can also use the var keyword when declaring “const” variables so the script only initializes them on the *first bar* of the dataset. See this section of our User Manual for more information.

Below is an example that uses “const” values within the indicator() and plot() functions, which both require a value of the “const string” qualified type as their `title` argument:

//@version=5 // The following global variables are all of the "const string" qualified type: //@variable The title of the indicator. INDICATOR_TITLE = "const demo" //@variable The title of the first plot. var PLOT1_TITLE = "High" //@variable The title of the second plot. const string PLOT2_TITLE = "Low" //@variable The title of the third plot. PLOT3_TITLE = "Midpoint between " + PLOT1_TITLE + " and " + PLOT2_TITLE indicator(INDICATOR_TITLE, overlay = true) plot(high, PLOT1_TITLE) plot(low, PLOT2_TITLE) plot(hl2, PLOT3_TITLE)

The following example will raise a compilation error since it uses syminfo.ticker, which returns a “simple” value because it depends on chart information that's only accessible after the script's first execution:

//@version=5 //@variable The title in the `indicator()` call. var NAME = "My indicator for " + syminfo.ticker indicator(NAME, "", true) // Causes an error because `NAME` is qualified as a "simple string". plot(close)

The const keyword allows the declaration of variables and parameters with constant *value assignments*. Declaring a variable with this keyword instructs the script to forbid using *reassignment* and *compound assignment* operations on it. For example, this script declares the `myVar` variable with the keyword, then attempts to assign a new “float” value to the variable with the addition assignment operator (+=), resulting in a compilation error:

//@version=5 indicator("Cannot reassign const demo") //@variable A "float" variable declared as `const`, preventing reassignment. const float myVar = 0.0 myVar += 1.0 // Causes an error. Reassignment and compound assignments are not allowed on `const` variables. plot(myVar)

It's crucial to note that declaring a variable with the const keyword forces it to maintain a constant reference to the value returned by a specific expression, but that *does not* necessarily define the nature of the assigned value. For example, a script can declare a const variable that maintains a constant reference to an expression returning the *ID* of a *special type*. Although the script cannot *reassign* the variable, the assigned ID is a “series” value:

//@version=5 indicator("Constant reference to 'series' ID demo") //@variable A `label` variable declared as `const`, preventing reassignment. // Although the reference is constant, the ID of the `label` is a "series" value. const label myVar = label.new(bar_index, close)

input

Most values qualified as “input” are established after initialization via the `input.*()` functions. These functions produce values that users can modify within the “Inputs” tab of the script's settings. When one changes any of the values in this tab, the script *restarts* from the beginning of the chart's history to ensure its inputs are consistent throughout its executions. Some of Pine's built-in variables, such as chart.bg_color also use the “input” qualifier, even though `input.*()` functions do not return them, since the script receives their values at *input time*.

> Notice!The input.source() and input.enum() functions are exceptions in the `input.*()` namespace, as they *do not* return values qualified as “input”. The input.source() function returns “series” values since built-in variables such as open and close, as well as the values from another script's plots, have the “series” qualifier. The input.enum() function returns a “simple” result because all values belonging to enums are available on the first script execution. See this manual's Inputs page for more information.

The following script plots the value of a `sourceInput` from the `symbolInput` and `timeframeInput` context. The request.security() call is valid in this script since its `symbol` and `timeframe` parameters allow “simple string” arguments, meaning they can also accept “input string” values because the “input” qualifier is *lower* on the hierarchy:

//@version=5 indicator("input demo", overlay = true) //@variable The symbol to request data from. Qualified as "input string". symbolInput = input.symbol("AAPL", "Symbol") //@variable The timeframe of the data request. Qualified as "input string". timeframeInput = input.timeframe("D", "Timeframe") //@variable The source of the calculation. Qualified as "series float". sourceInput = input.source(close, "Source") //@variable The `sourceInput` value from the requested context. Qualified as "series float". requestedSource = request.security(symbolInput, timeframeInput, sourceInput) plot(requestedSource)

simple

Values qualified as “simple” are available on the first script execution, and they remain consistent across subsequent executions.

Users can explicitly define variables and parameters that accept “simple” values by including the `simple` keyword in their declaration.

Many built-in variables return “simple” qualified values because they depend on information that a script can only obtain once it starts running on the chart. Additionally, many built-in functions require “simple” arguments that do not change over time. Wherever a script allows “simple” values, it can also accept values qualified as “input” or “const”.

This script highlights the background to warn users that they're using a non-standard chart type. It uses the value of chart.is_standard to calculate the `isNonStandard` variable, then uses that variable's value to calculate a `warningColor` that also references a “simple” value. The `color` parameter of bgcolor() allows a

“series color” argument, meaning it can also accept a “simple color” value since “simple” is lower on the hierarchy:

//@version=5 indicator("simple demo", overlay = true) //@variable Is `true` when the current chart is non-standard. Qualified as "simple bool". isNonStandard = not chart.is_standard //@variable Is orange when the the current chart is non-standard. Qualified as "simple color". simple color warningColor = isNonStandard ? color.new(color.orange, 70) : na // Colors the chart's background to warn that it's a non-standard chart type. bgcolor(warningColor, title = "Non-standard chart color")

series

Values qualified as “series” provide the most flexibility in scripts since they can change across executions.

Users can explicitly define variables and parameters that accept “series” values by including the `series` keyword in their declarations.

Built-in variables such as open, high, low, close, volume, time, and bar_index, and the result from any expression using such built-ins, are qualified as “series”. The result of any function or operation that returns a dynamic value will always be a “series”, as will the results from using the history-referencing operator [] to access historical values. Wherever a script allows “series” values, it will also accept values with any other qualifier, as “series” is the *highest* qualifier on the hierarchy.

This script displays the highest and lowest value of a `sourceInput` over `lengthInput` bars. The values assigned to the `highest` and `lowest` variables are of the “series float” qualified type, as they can change throughout the script's execution:

//@version=5 indicator("series demo", overlay = true) //@variable The source value to calculate on. Qualified as "series float". series float sourceInput = input.source(close, "Source") //@variable The number of bars in the calculation. Qualified as "input int". lengthInput = input.int(20, "Length") //@variable The highest `sourceInput` value over `lengthInput` bars. Qualified as "series float". series float highest = ta.highest(sourceInput, lengthInput) //@variable The lowest `sourceInput` value over `lengthInput` bars. Qualified as "series float". lowest = ta.lowest(sourceInput, lengthInput) plot(highest, "Highest source", color.green) plot(lowest, "Lowest source", color.red)

Types

Pine Script *types* classify values and determine the functions and operations they're compatible with. They include:

- The fundamental types: int, float, bool, color, and string
- The special types: plot, hline, line, linefill, box, polyline, label, table, chart.point, array, matrix, and map
- User-defined types (UDTs)
- Enums
- void

Fundamental types refer to the underlying nature of a value, e.g., a value of 1 is of the “int” type, 1.0 is of the “float” type, “AAPL” is of the “string” type, etc. Special types and user-defined types utilize *IDs* that refer to objects of a specific type. For example, a value of the “label” type contains an ID that acts as a *pointer* referring to a “label” object. The “void” type refers to the output from a function or method that does not return a usable value.

Pine Script can automatically convert values from some types into others. The auto-casting rules are: **int** → **float** → **bool**. See the Type casting section of this page for more information.

In most cases, Pine Script can automatically determine a value's type. However, we can also use type keywords to *explicitly* specify types for readability and for code that requires explicit definitions (e.g., declaring a variable assigned to na). For example:

//@version=5 indicator("Types demo", overlay = true) //@variable A value of the "const string" type for the `ma` plot's title. string MA_TITLE = "MA" //@variable A value of the "input int" type. Controls the length of the average. int lengthInput = input.int(100, "Length", minval = 2) //@variable A "series float" value representing the last `close` that crossed over the `ma`. var float crossValue = na //@variable A "series float" value representing the moving average of `close`. float ma = ta.sma(close, lengthInput) //@variable A "series bool" value that's `true` when the `close` crosses over the `ma`. bool crossUp = ta.crossover(close, ma) //@variable A "series color" value based on whether `close` is above or below its `ma`. color maColor = close > ma ? color.lime : color.fuchsia // Update the `crossValue`. 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)

int

Values of the “int” type represent integers, i.e., whole numbers without any fractional quantities.

Integer literals are numeric values written in *decimal* notation. For example:

1 -1 750

Built-in variables such as bar_index, time, timenow, dayofmonth, and strategy.wintrades all return values of the “int” type.

float

Values of the “float” type represent floating-point numbers, i.e., numbers that can contain whole and fractional quantities.

Floating-point literals are numeric values written with a `.` delimiter. They may also contain the symbol `e` or `E` (which means “10 raised to the power of X”, where X is the number after the `e` or `E` symbol). For example:

3.14159 // Rounded value of Pi (π) - 3.0 6.02e23 // 6.02 * 10^23 (a very large value) 1.6e-19 // 1.6 * 10^-19 (a very small value)

The internal precision of “float” values in Pine Script is 1e-16.

Built-in variables such as close, hlcc4, volume, ta.vwap, and strategy.position_size all return values of the “float” type.

bool

Values of the “bool” type represent the truth value of a comparison or condition, which scripts can use in conditional structures and other expressions.

There are only two literals that represent boolean values:

true // true value false // false value

When an expression of the “bool” type returns na, scripts treat its value as `false` when evaluating conditional statements and operators.

Built-in variables such as barstate.isfirst, chart.is_heikinashi, session.ismarket, and timeframe.isdaily all return values of the “bool” type.

color

Color literals have the following format: `#RRGGBB` or `#RRGGBBAA`. The letter pairs represent *hexadecimal* values between `00` and `FF` (0 to 255 in decimal) where:

- `RR`, `GG` and `BB` pairs respectively represent the values for the color's red, green and blue components.
- `AA` is an optional value for the color's opacity (or *alpha* component) where `00` is invisible and `FF` opaque. When the literal does not include an `AA` pair, the script treats it as fully opaque (the same as using `FF`).
- The hexadecimal letters in the literals can be uppercase or lowercase.

These are examples of “color” literals:

#000000 // black color #FF0000 // red color #00FF00 // green color #0000FF // blue color #FFFFFF // white color #808080 // gray color #3ff7a0 // some custom color #FF000080 // 50% transparent red color #FF0000ff // same as #FF0000, fully opaque red color #FF000000 // completely transparent red color

Pine Script also has built-in color constants, including color.green, color.red, color.orange, color.blue (the default color in `plot*()` functions and many of the default color-related properties in drawing types), etc.

When using built-in color constants, it is possible to add transparency information to them via the color.new() function.

Note that when specifying red, green or blue components in `color.*()` functions, we use “int” or “float” arguments with values between 0 and 255. When specifying transparency, we use a value between 0 and 100, where 0 means fully opaque and 100 means completely transparent. For example:

//@version=5 indicator("Shading the chart's background", overlay = true) //@variable A "const color" value representing the base for each day's color. color BASE_COLOR = color.rgb(0, 99, 165) //@variable A "series int" value that modifies the transparency of the `BASE_COLOR` in `color.new()`. int transparency = 50 + int(40 * dayofweek / 7) // Color the background using the modified `BASE_COLOR`. bgcolor(color.new(BASE_COLOR, transparency))

See the User Manual's page on colors for more information on using colors in scripts.

string

Values of the “string” type represent sequences of letters, numbers, symbols, spaces, and other characters.

String literals in Pine are characters enclosed in single or double quotation marks. For example:

"This is a string literal using double quotes." 'This is a string literal using single quotes.'

Single and double quotation marks are functionally equivalent in Pine Script. A “string” enclosed within double quotation marks can contain any number of single quotation marks and vice versa:

"It's an example" 'The "Star" indicator'

Scripts can *escape* the enclosing delimiter in a “string” using the backslash character (`\`). For example:

'It\'s an example' "The \"Star\" indicator"

We can create “string” values containing the new line escape character (`\n`) for displaying multi-line text with `plot*()` and `log.*()` functions and objects of drawing types. For example:

"This\nString\nHas\nOne\nWord\nPer\nLine"

We can use the + operator to concatenate “string” values:

"This is a " + "concatenated string."

The built-ins in the `str.*()` namespace create “string” values using specialized operations. For instance, this script creates a *formatted string* to represent “float” price values and displays the result using a label:

//@version=5 indicator("Formatted string demo", overlay = true) //@variable A "series string" value representing the bar's OHLC data. string ohlcString = str.format("Open: {0}\nHigh: {1}\nLow: {2}\nClose: {3}", open, high, low, close) // Draw a label containing the `ohlcString`. label.new(bar_index, high, ohlcString, textcolor = color.white)

See our User Manual's page on Text and shapes for more information about displaying “string” values from a script.

Built-in variables such as syminfo.tickerid, syminfo.currency, and timeframe.period return values of the “string” type.

plot and hline

Pine Script's plot() and hline() functions return IDs that respectively reference instances of the “plot” and “hline” types. These types display calculated values and horizontal levels on the chart, and one can assign their IDs to variables for use with the built-in fill() function.

For example, this script plots two EMAs on the chart and fills the space between them using a fill() call:

//@version=5 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 The plot of the `emaFast` value. emaFastPlot = plot(emaFast, "Fast EMA", color.orange, 3) //@variable The plot of the `emaSlow` value. emaSlowPlot = plot(emaSlow, "Slow EMA", color.gray, 3) // Fill the space between the `emaFastPlot` and `emaSlowPlot`. fill(emaFastPlot, emaSlowPlot, color.new(color.purple, 50), "EMA Fill")

It's important to note that unlike other special types, there is no `plot` or `hline` keyword in Pine to explicitly declare a variable's type as “plot” or “hline”.

Users can control where their scripts' plots display via the variables in the `display.*` namespace and a `plot*()` function's `force_overlay` parameter. Additionally, one script can use the values from another script's plots as *external inputs* via the input.source() function (see our User Manual's section on source inputs).

Drawing types

Pine Script drawing types allow scripts to create custom drawings on charts. They include the following: line, linefill, box, polyline, label, and table.

Each type also has a namespace containing all the built-ins that create and manage drawing instances. For example, the following `*.new()` constructors create new objects of these types in a script: line.new(), linefill.new(), box.new(), polyline.new(), label.new(), and table.new().

Each of these functions returns an *ID* which is a reference that uniquely identifies a drawing object. IDs are always qualified as "series", meaning their qualified types are "series line", "series label", etc. Drawing IDs act like pointers, as each ID references a specific instance of a drawing in all the functions from that drawing's namespace. For instance, the ID of a line returned by a line.new() call is used later to refer to that specific object once it's time to delete it with line.delete().

Chart points

Chart points are special types that represent coordinates on the chart. Scripts use the information from chart.point objects to determine the chart locations of lines, boxes, polylines, and labels.

Objects of this type contain three *fields*: `time`, `index`, and `price`. Whether a drawing instance uses the `time` or `price` field from a chart.point as an x-coordinate depends on the drawing's `xloc` property.

We can use any of the following functions to create chart points in a script:

- chart.point.new() - Creates a new chart.point with a specified `time`, `index`, and `price`.
- chart.point.now() - Creates a new chart.point with a specified `price` y-coordinate. The `time` and `index` fields contain the time and bar_index of the bar the function executes on.
- chart.point.from_index() - Creates a new chart.point with an `index` x-coordinate and `price` y-coordinate. The `time` field of the resulting instance is na, meaning it will not work with drawing objects that use an `xloc` value of xloc.bar_time.
- chart.point.from_time() - Creates a new chart.point with a `time` x-coordinate and `price` y-coordinate. The `index` field of the resulting instance is na, meaning it will not work with drawing objects that use an `xloc` value of xloc.bar_index.
- chart.point.copy() - Creates a new chart.point containing the same `time`, `index`, and `price` information as the `id` in the function call.

This example draws lines connecting the previous bar's high to the current bar's low on each chart bar. It also displays labels at both points of each line. The line and labels get their information from the `firstPoint` and `secondPoint` variables, which reference chart points created using chart.point.from_index() and chart.point.now():

//@version=5 indicator("Chart points demo", overlay = true) //@variable A new `chart.point` at the previous `bar_index` and `high`. firstPoint = chart.point.from_index(bar_index - 1, high[1]) //@variable A new `chart.point` at the current bar's `low`. secondPoint = chart.point.now(low) // Draw a new line connecting coordinates from the `firstPoint` and `secondPoint`. // This line uses the `index` fields from the points as x-coordinates. line.new(firstPoint, secondPoint, color = color.purple, width = 3) // Draw a label at the `firstPoint`. Uses the point's `index` field as its x-coordinate. label.new(firstPoint, str.tostring(firstPoint.price), color = color.green, style = label.style_label_down, textcolor = color.white) // Draw a label at the `secondPoint`. Uses the point's `index` field as its x-coordinate. label.new(secondPoint, str.tostring(secondPoint.price), color = color.red, style = label.style_label_up, textcolor = color.white)

Collections

Collections in Pine Script (arrays, matrices, and maps) utilize reference IDs, much like other special types (e.g., labels). The type of the ID defines the type of *elements* the collection will contain. In Pine, we specify array, matrix, and map types by appending a type template to the array, matrix, or map keywords:

- `array<int>` defines an array containing "int" elements.
- `array<label>` defines an array containing "label" IDs.
- `array<UDT>` defines an array containing IDs referencing objects of a user-defined type (UDT).
- `matrix<float>` defines a matrix containing "float" elements.
- `matrix<UDT>` defines a matrix containing IDs referencing objects of a user-defined type (UDT).
- `map<string, float>` defines a map containing "string" keys and "float" values.
- `map<int, UDT>` defines a map containing "int" keys and IDs of user-defined type (UDT) instances as values.

For example, one can declare an "int" array with a single element value of 10 in any of the following, equivalent ways:

a1 = array.new<int>(1, 10) array<int> a2 = array.new<int>(1, 10) a3 = array.from(10) array<int> a4 = array.from(10)

Note that:

- The `int[]` syntax can also specify an array of "int" elements, but its use is discouraged. No equivalent exists to specify the types of matrices or maps in that way.
- Type-specific built-ins exist for arrays, such as array.new_int(), but the more generic array.new<type> form is preferred, which would be `array.new<int>()` to create an array of "int" elements.

User-defined types

The type keyword allows the creation of *user-defined types* (UDTs) from which scripts can create objects. UDTs are composite types; they contain an arbitrary number of *fields* that can be of any type, including other user-defined types.

The syntax to declare a user-defined type is:

```
[export ]type <UDT_identifier>
    <field_type> <field_name>[ = <value>]
    ...
```

where:

- export is the keyword that a library script uses to export the user-defined type. To learn more about exporting UDTs, see our User Manual's Libraries page.
- `<UDT_identifier>` is the name of the user-defined type.
- `<field_type>` is the type of the field.
- `<field_name>` is the name of the field.
- `<value>` is an optional default value for the field, which the script will assign to it when creating new objects of that UDT. If one does not provide a value, the field's default is na. The same rules as those

governing the default values of parameters in function signatures apply to the default values of fields. For example, a UDT's default values cannot use results from the history-referencing operator [] or expressions.

This example declares a `pivotPoint` UDT with an “int” `pivotTime` field and a “float” `priceLevel` field that will respectively hold time and price information about a calculated pivot:

//@type A user-defined type containing pivot information. //@field pivotTime Contains time information about the pivot. //@field priceLevel Contains price information about the pivot. type pivotPoint int pivotTime float priceLevel

User-defined types support *type recursion*, i.e., the fields of a UDT can reference objects of the same UDT. Here, we've added a `nextPivot` field to our previous `pivotPoint` type that references another `pivotPoint` instance:

//@type A user-defined type containing pivot information. //@field pivotTime Contains time information about the pivot. //@field priceLevel Contains price information about the pivot. //@field nextPivot A `pivotPoint` instance containing additional pivot information. type pivotPoint int pivotTime float priceLevel pivotPoint nextPivot

Scripts can use two built-in methods to create and copy UDTs: `new()` and `copy()`. See our User Manual's page on Objects to learn more about working with UDTs.

Enum types

The enum keyword allows the creation of an *enum*, otherwise known as an *enumeration*, *enumerated type*, or *enum type*. An enum is a unique type construct containing distinct, named fields representing *members* (i.e., possible values) of the type. Enums allow programmers to control the values accepted by variables, conditional expressions, and collections, and they facilitate convenient dropdown input creation with the input.enum() function.

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 allowing a library to export the enum for use in other scripts. See this section to learn more about exporting enum types.
- `<enumName>` is the name of the enum type. Scripts can use the enum's name as the type keyword in variable declarations and type templates.
- `<field_*>` is the name of an enum field, representing 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 field in the enum. To retrieve an enum member, reference its field name using dot notation syntax (i.e., `enumName.fieldName`).

- `<title_*>` is a “const string” title assigned to a field. If one does not specify a title, the field's title is the “string” representation of its name. The input.enum() function displays field titles within its dropdown in the script's “Settings/Inputs” tab. Users can also retrieve a field's title with the str.tostring() function. As with field names, each field's title must not match the name or title of any other field in the enum.

This example declares an `maChoice` enum. Each field within this declaration represents a distinct member of the `maChoice` enum type:

//@enum An enumeration of named values for moving average selection. //@field sma Selects a Simple Moving Average. //@field ema Selects an Exponential Moving Average. //@field wma Selects a Weighted Moving Average. //@field hma Selects a Hull Moving Average. enum maChoice sma = "Simple Moving Average" ema = "Exponential Moving Average" wma = "Weighted Moving Average" hma = "Hull Moving Average"

Note that:

- All the enum's possible values are available upon the *first* script execution and do not change across subsequent executions. Hence, they automatically adopt the simple qualifier.

The script below uses the `maChoice` enum within an input.enum() call to create a *dropdown* input in the “Settings/Inputs” tab that displays all the field titles. The `maInput` value represents the member of the enum that corresponds to the user-selected title. The script uses the selected member within a switch structure to determine the built-in moving average it calculates:

//@version=5 indicator("Enum types demo", overlay = true) //@enum An enumeration of named values for moving average selection. //@field sma Selects a Simple Moving Average. //@field ema Selects an Exponential Moving Average. //@field wma Selects a Weighted Moving Average. //@field hma Selects 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. 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 selected by the `maInput`. 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`. 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.

void

There is a “void” type in Pine Script. Functions having only side-effects and returning no usable result return the “void” type. An example of such a function is alert(); it does something (triggers an alert event), but it returns no usable value.

Scripts cannot use “void” results in expressions or assign them to variables. No `void` keyword exists in Pine Script since one cannot declare a variable of the “void” type.

`na` value

There is a special value in Pine Script called na, which is an acronym for *not available*. We use na to represent an undefined value from a variable or expression. It is similar to `null` in Java and `None` in Python.

Scripts can automatically cast na values to almost any type. However, in some cases, the compiler cannot infer the type associated with an na value because more than one type-casting rule may apply. For example:

// Compilation error! myVar = na

The above line of code causes a compilation error because the compiler cannot determine the nature of the `myVar` variable, i.e., whether the variable will reference numeric values for plotting, string values for setting text in a label, or other values for some other purpose later in the script's execution.

To resolve such errors, we must explicitly declare the type associated with the variable. Suppose the `myVar` variable will reference “float” values in subsequent script iterations. We can resolve the error by declaring the variable with the float keyword:

float myVar = na

or by explicitly casting the na value to the “float” type via the float() function:

myVar = float(na)

To test if the value from a variable or expression is na, we call the na() function, which returns `true` if the value is undefined. For example:

//@variable Is 0 if the `myVar` is `na`, `close` otherwise. float myClose = na(myVar) ? 0 : close

Do not use the == comparison operator to test for na values, as scripts cannot determine the equality of an undefined value:

//@variable Returns the `close` value. The script cannot compare the equality of `na` values, as they're undefined. float myClose = myVar == na ? 0 : close

Best coding practices often involve handling na values to prevent undefined values in calculations.

We can ensure the expression also returns an actionable value on the first bar by replacing the undefined past value with a value from the current bar. This line of code uses the nz() function to replace the past bar's close with the current bar's open when the value is na:

//@variable Is `true` when the `close` exceeds the last bar's `close` (or the current `open` if the value is `na`). bool risingClose = close > nz(close[1], open)

Protecting scripts against na instances helps to prevent undefined values from propagating in a calculation's results. For example, this script declares an `allTimeHigh` variable on the first bar. It then uses the math.max() between the `allTimeHigh` and the bar's high to update the `allTimeHigh` throughout its execution:

//@version=5 indicator("na protection demo", overlay = true) //@variable The result of calculating the all-time high price with an initial value of `na`. var float allTimeHigh = na // Reassign the value of the `allTimeHigh`. // Returns `na` on all bars because `math.max()` can't compare the `high` to an undefined value. allTimeHigh := math.max(allTimeHigh, high) plot(allTimeHigh) // Plots `na` on all bars.

This script plots a value of na on all bars, as we have not included any na protection in the code. To fix the behavior and plot the intended result (i.e., the all-time high of the chart's prices), we can use nz() to replace

na values in the `allTimeHigh` series:

//@version=5 indicator("na protection demo", overlay = true) //@variable The result of calculating the all-time high price with an initial value of `na`. var float allTimeHigh = na // Reassign the value of the `allTimeHigh`. // We've used `nz()` to prevent the initial `na` value from persisting throughout the calculation. allTimeHigh := math.max(nz(allTimeHigh), high) plot(allTimeHigh)

Type templates

Type templates specify the data types that collections (arrays, matrices, and maps) can contain.

Templates for arrays and matrices consist of a single type identifier surrounded by angle brackets, e.g., `<int>`, `<label>`, and `<PivotPoint>` (where `PivotPoint` is a user-defined type (UDT)).

Templates for maps consist of two type identifiers enclosed in angle brackets, where the first specifies the type of *keys* in each key-value pair, and the second specifies the *value* type. For example, `<string, float>` is a type template for a map that holds `string` keys and `float` values.

Users can construct type templates from:

- Fundamental types: int, float, bool, color, and string
- The following special types: line, linefill, box, polyline, label, table, and chart.point
- User-defined types (UDTs)
- Enum types

Note that:

- Maps can use any of these types as *values*, but they can only accept fundamental types or enum types as *keys*.

Scripts use type templates to declare variables that reference collections, and when creating new collection instances. For example:

//@version=5 indicator("Type templates demo") //@variable A variable initially assigned to `na` that accepts arrays of "int" values. array<int> intArray = na //@variable An empty matrix that holds "float" values. floatMatrix = matrix.new<float>() //@variable An empty map that holds "string" keys and "color" values. stringColorMap = map.new<string, color>()

Type casting

Pine Script includes an automatic type-casting mechanism that *casts* (converts) **“int”** values to **“float”** when necessary. Variables or expressions requiring “float” values can also use “int” values because any integer can be represented as a floating point number with its fractional part equal to 0.

For the sake of backward compatibility, Pine Script also automatically casts **“int”** and **“float”** values to **“bool”** when necessary. When passing numeric values to the parameters of functions and operations that expect “bool” types, Pine auto-casts them to “bool”. However, we do not recommend relying on this behavior. Most scripts that automatically cast numeric values to the “bool” type will produce a *compiler warning*. One can avoid the compiler warning and promote code readability by using the bool() function, which explicitly casts a numeric value to the “bool” type.

When casting an “int” or “float” to “bool”, a value of 0 converts to `false` and any other numeric value always converts to `true`.

This code below demonstrates deprecated auto-casting behavior in Pine. It creates a `randomValue` variable with a “series float” value on every bar, which it passes to the `condition` parameter in an if structure and the `series` parameter in a plotchar() function call. Since both parameters accept “bool” values, the script automatically casts the `randomValue` to “bool” when evaluating them:

//@version=5 indicator("Auto-casting demo", overlay = true) //@variable A random rounded value between -1 and 1. float randomValue = math.round(math.random(-1, 1)) //@variable The color of the chart background. color bgColor = na // This raises a compiler warning since `randomValue` is a "float", but `if` expects a "bool". if randomValue bgColor := color.new(color.blue, 60) // This does not raise a warning, as the `bool()` function explicitly casts the `randomValue` to "bool". if bool(randomValue) bgColor := color.new(color.blue, 60) // Display unicode characters on the chart based on the `randomValue`. // Whenever `math.random()` returns 0, no character will appear on the chart because 0 converts to `false`. plotchar(randomValue) // We recommend explicitly casting the number with the `bool()` function to make the type transformation more obvious. plotchar(bool(randomValue)) // Highlight the background with the `bgColor`. bgcolor(bgColor)

It's sometimes necessary to cast one type to another when auto-casting rules do not suffice. For such cases, the following type-casting functions are available: int(), float(), bool(), color(), string(), line(), linefill(), label(), box(), and table().

The example below shows a code that tries to use a “const float” value as the `length` argument in the ta.sma() function call. The script will fail to compile, as it cannot automatically convert the “float” value to the required “int” type:

//@version=5 indicator("Explicit casting demo", overlay = true) //@variable The length of the SMA calculation. Qualified as "const float". float LENGTH = 10.0 float sma = ta.sma(close, LENGTH) // Compilation error. The `length` parameter requires an "int" value. plot(sma)

The code raises the following error: *“Cannot call 'ta.sma' with argument 'length'='LENGTH'. An argument of 'const float' type was used but a 'series int' is expected.”*

The compiler is telling us that the code is using a “float” value where an “int” is required. There is no auto-casting rule to cast a “float” to an “int”, so we must do the job ourselves. In this version of the code, we've used the int() function to explicitly convert our “float” `LENGTH` value to the “int” type within the ta.sma() call:

//@version=5 indicator("explicit casting demo") //@variable The length of the SMA calculation. Qualified as "const float". float LENGTH = 10.0 float sma = ta.sma(close, int(LENGTH)) // Compiles successfully since we've converted the `LENGTH` to "int". plot(sma)

Explicit type casting is also handy when declaring variables assigned to na, as explained in the previous section.

For example, once could explicitly declare a variable with a value of na as a “label” type in either of the following, equivalent ways:

// Explicitly specify that the variable references "label" objects: label myLabel = na // Explicitly cast the `na` value to the "label" type: myLabel = label(na)

Tuples

A *tuple* is a comma-separated set of expressions enclosed in brackets. When a function, method, or other local block returns more than one value, scripts return those values in the form of a tuple.

For example, the following user-defined function returns the sum and product of two “float” values:

//@function Calculates the sum and product of two 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`. [sum, product]

When we call this function later in the script, we use a *tuple declaration* to declare multiple variables corresponding to the values returned by the function call:

// Declare a tuple containing the sum and product of the `high` and `low`, respectively. [hlSum, hlProduct] = calcSumAndProduct(high, low)

Keep in mind that unlike declaring single variables, we cannot explicitly define the types the tuple's variables (`hlSum` and `hlProduct` in this case), will contain. The compiler automatically infers the types associated with the variables in a tuple.

In the above example, the resulting tuple contains values of the same type (“float”). However, it's important to note that tuples can contain values of *multiple types*. For example, the `chartInfo()` function below returns a 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` when using 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]

Tuples are especially handy for requesting multiple values in one request.security() call.

For instance, this `roundedOHLC()` function returns a tuple containing OHLC values rounded to the nearest prices that are divisible by the symbol's minimum tick value. We call this function as the `expression` argument in request.security() to request a tuple containing daily OHLC values:

//@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, "D", roundedOHLC())

We can also achieve the same result by directly passing a tuple of rounded values as the `expression` in the request.security() call:

[op, hi, lo, cl] = request.security(syminfo.tickerid, "D", [math.round_to_mintick(open), math.round_to_mintick(high), math.round_to_mintick(low), math.round_to_mintick(close)])

Local blocks of conditional structures, including if and switch statements, can return tuples. For example:

[v1, v2] = if close > open [high, close] else [close, low]

and:

[v1, v2] = switch close > open => [high, close] => [close, low]

However, ternaries cannot contain tuples, as the return values in a ternary statement are not considered local blocks:

// Not allowed. [v1, v2] = close > open ? [high, close] : [close, low]

Note that all items within a tuple returned from a function are qualified as “simple” or “series”, depending on its contents. If a tuple contains a “series” value, all other elements within the tuple will also adopt the “series” qualifier. For example:

//@version=5 indicator("Qualified types in tuples demo") makeTicker(simple string prefix, simple string ticker) => tId = prefix + ":" + ticker // simple string source = close // series float [tId, source] // Both variables are series now. [tId, source] = makeTicker("BATS", "AAPL") // Error cannot call 'request.security' with 'series string' tId. r = request.security(tId, "", source) plot(r)

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

User-defined functions

Introduction

User-defined functions are functions that you write, as opposed to the built-in functions in Pine Script®. They are useful to define calculations that you must do repetitevely, or that you want to isolate from your script's main section of calculations. Think of user-defined functions as a way to extend the capabilities of Pine Script, when no built-in function will do what you need.

You can write your functions in two ways:

- In a single line, when they are simple, or
- On multiple lines

Functions can be located in two places:

- If a function is only used in one script, you can include it in the script where it is used. See our Style guide for recommendations on where to place functions in your script.
- You can create a Pine Script library to include your functions, which makes them reusable in other scripts without having to copy their code. Distinct requirements exist for library functions. They are explained in the page on libraries.

Whether they use one line or multiple lines, user-defined functions have the following characteristics:

- They cannot be embedded. All functions are defined in the script's global scope.
- They do not support recursion. It is **not allowed** for a function to call itself from within its own code.
- The type of the value returned by a function is determined automatically and depends on the type of arguments used in each particular function call.
- A function's returned value is that of the last value in the function's body.
- Each instance of a function call in a script maintains its own, independent history.

Single-line functions

Simple functions can often be written in one line. This is the formal definition of single-line functions:

```
<function_declaration>
    <identifier>(<parameter_list>) => <return_value>

<parameter_list>
    {<parameter_definition>{, <parameter_definition>}}

<parameter_definition>
    [<identifier> = <default_value>]

<return_value>
    <statement> | <expression> | <tuple>
```

Here is an example:

f(x, y) => x + y

After the function `f()` has been declared, it's possible to call it using different types of arguments:

a = f(open, close) b = f(2, 2) c = f(open, 2)

In the example above, the type of variable `a` is *series* because the arguments are both *series*. The type of variable `b` is *integer* because arguments are both *literal integers*. The type of variable `c` is *series* because the addition of a *series* and *literal integer* produces a *series* result.

Multi-line functions

Pine Script also supports multi-line functions with the following syntax:

```
<identifier>(<parameter_list>) =>
    <local_block>

<identifier>(<list of parameters>) =>
    <variable declaration>
    ...
    <variable declaration or expression>
```

where:

```
<parameter_list>
    {<parameter_definition>{, <parameter_definition>}}

<parameter_definition>
    [<identifier> = <default_value>]
```

The body of a multi-line function consists of several statements. Each statement is placed on a separate line and must be preceded by 1 indentation (4 spaces or 1 tab). The indentation before the statement indicates that it is a part of the body of the function and not part of the script's global scope. After the function's code, the first statement without an indent indicates the body of the function has ended.

Either an expression or a declared variable should be the last statement of the function's body. The result of this expression (or variable) will be the result of the function's call. For example:

geom_average(x, y) => a = x*x b = y*y math.sqrt(a + b)

The function `geom_average` has two arguments and creates two variables in the body: `a` and `b`. The last statement calls the function `math.sqrt` (an extraction of the square root). The `geom_average` call will return the value of the last expression: `(math.sqrt(a + b))`.

Scopes in the script

Variables declared outside the body of a function or of other local blocks belong to the *global* scope. User-declared and built-in functions, as well as built-in variables also belong to the global scope.

Each function has its own *local* scope. All the variables declared within the function, as well as the function's arguments, belong to the scope of that function, meaning that it is impossible to reference them from outside --- e.g., from the global scope or the local scope of another function.

On the other hand, since it is possible to refer to any variable or function declared in the global scope from the scope of a function (except for self-referencing recursive calls), one can say that the local scope is embedded into the global scope.

In Pine Script, nested functions are not allowed, i.e., one cannot declare a function inside another one. All user functions are declared in the global scope. Local scopes cannot intersect with each other.

Functions that return multiple results

In most cases a function returns only one result, but it is possible to return a list of results (a *tuple*-like result):

fun(x, y) => a = x+y b = x-y [a, b]

Special syntax is required for calling such functions:

[res0, res1] = fun(open, close) plot(res0) plot(res1)

Limitations

User-defined functions can use any of the Pine Script built-ins, except: barcolor(), fill(), hline(), indicator(), library(), plot(), plotbar(), plotcandle(), plotchar(), plotshape() and strategy().

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Variable declarations

Introduction

Variables are identifiers that hold values. They must be *declared* in your code before you use them. The syntax of variable declarations is:

```
[<declaration_mode>] [<type>] <identifier> = <expression> | <structure>
```

or

```
<tuple_declaration> = <function_call> | <structure>
```

where:

- `|` means “or”, and parts enclosed in square brackets (`[]`) can appear zero or one time.
- <declaration_mode> is the variable's declaration mode. It can be var or varip, or nothing.
- <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.
- <structure> can be an if, for, while or switch *structure*.
- <tuple_declaration> is a comma-separated list of variable names enclosed in square brackets (`[]`), e.g., `[ma, upperBand, lowerBand]`.

These are all valid variable declarations. The last one requires four lines:

BULL_COLOR = color.lime i = 1 len = input(20, "Length") float f = 10.5 closeRoundedToTick = math.round_to_mintick(close) st = ta.supertrend(4, 14) var barRange = float(na) var firstBarOpen = open varip float lastClose = na [macdLine, signalLine, histLine] = ta.macd(close, 12, 26, 9) plotColor = if close > open color.green else color.red

> Notice!The above statements all contain the `=` assignment operator because they are **variable declarations**. When you see similar lines using the `:=` reassignment operator, the code is **reassigning** a value to a variable that was **already declared**. Those are **variable reassignments**. Be sure you understand the distinction as this is a common stumbling block for newcomers to Pine Script. See the next Variable reassignment section for details.

The formal syntax of a variable declaration is:

```
<variable_declaration>
    [<declaration_mode>] [<type>] <identifier> = <expression> | <structure>
    |
    <tuple_declaration> = <function_call> | <structure>
```

```
<declaration_mode>
    var | varip

<type>
    int | float | bool | color | string | line | linefill | label | box | table
```

Initialization with `na`

In most cases, an explicit type declaration is redundant because type is automatically inferred from the value on the right of the `=` at compile time, so the decision to use them is often a matter of preference. For example:

```
baseLine0 = na // compile time error! float baseLine1 = na // OK baseLine2 = float(na) // OK
```

In the first line of the example, the compiler cannot determine the type of the `baseLine0` variable because na is a generic value of no particular type. The declaration of the `baseLine1` variable is correct because its float type is declared explicitly. The declaration of the `baseLine2` variable is also correct because its type can be derived from the expression `float(na)`, which is an explicit cast of the na value to the float type. The declarations of `baseLine1` and `baseLine2` are equivalent.

Tuple declarations

Function calls or structures are allowed to return multiple values. When we call them and want to store the values they return, a *tuple declaration* must be used, which is a comma-separated set of one or more values enclosed in brackets. This allows us to declare multiple variables simultaneously. As an example, the ta.bb() built-in function for Bollinger bands returns three values:

[bbMiddle, bbUpper, bbLower] = ta.bb(close, 5, 4)

Using an underscore (_) as an identifier

When declaring a variable, it is possible to use a single underscore (_) as its identifier. A value assigned to such a variable cannot be accessed. You can assign any number of values to a _ identifier anywhere in the script, even if the current scope already has such an assignment.

This is particularly useful when a tuple returns unneeded values. Let's write another Bollinger Bands script. Here, we only need the bands themselves, without the center line:

//@version=5 indicator("Underscore demo") // We do not need the middle Bollinger Bands value, and do not use it. // To make this clear, we assign it to the `_` identifier. [_, bbUpper, bbLower] = ta.bb(close, 5, 4) // We can continue to use `_` in the same code without causing compilation errors: [bbMiddleLong, _, _] = ta.bb(close, 20, 2) plot(bbUpper)

Variable reassignment

A variable reassignment is done using the := reassignment operator. It can only be done after a variable has been first declared and given an initial value. Reassigning a new value to a variable is often necessary in calculations, and it is always necessary when a variable from the global scope must be assigned a new value from within a structure's local block, e.g.:

//@version=5 indicator("", "", true) sensitivityInput = input.int(2, "Sensitivity", minval = 1, tooltip = "Higher values make color changes less sensitive.") ma = ta.sma(close, 20) maUp = ta.rising(ma, sensitivityInput) maDn = ta.falling(ma, sensitivityInput) // On first bar only, initialize color to gray var maColor = color.gray if maUp // MA has risen for two bars in a row; make it lime. maColor := color.lime else if maDn // MA has fallen for two bars in a row; make it fuchsia. maColor := color.fuchsia plot(ma, "MA", maColor, 2)

Note that:

- We initialize `maColor` on the first bar only, so it preserves its value across bars.
- On every bar, the if statement checks if the MA has been rising or falling for the user-specified number of bars (the default is 2). When that happens, the value of `maColor` must be reassigned a new value from within the if local blocks. To do this, we use the := reassignment operator.
- If we did not use the := reassignment operator, the effect would be to initialize a new `maColor` local variable which would have the same name as that of the global scope, but actually be a very confusing independent entity that would persist only for the length of the local block, and then disappear without a trace.

All user-defined variables in Pine Script are *mutable*, which means their value can be changed using the := reassignment operator. Assigning a new value to a variable may change its *type qualifier* (see the page on Pine Script's type system for more information). A variable can be assigned a new value as many times as needed during the script's execution on one bar, so a script can contain any number of reassignments of one variable. A variable's declaration mode determines how new values assigned to a variable will be saved.

Declaration modes

Understanding the impact that declaration modes have on the behavior of variables requires prior knowledge of Pine Script's execution model.

When you declare a variable, if a declaration mode is specified, it must come first. Three modes can be used:

- “On each bar”, when none is specified
- var
- varip

On each bar

When no explicit declaration mode is specified, i.e. no var or varip keyword is used, the variable is declared and initialized on each bar, e.g., the following declarations from our first set of examples in this page's introduction:

BULL_COLOR = color.lime i = 1 len = input(20, "Length") float f = 10.5 closeRoundedToTick = math.round_to_mintick(close) st = ta.supertrend(4, 14) [macdLine, signalLine, histLine] = ta.macd(close, 12, 26, 9) plotColor = if close > open color.green else color.red

`var`

When the var keyword is used, the variable is only initialized once, on the first bar if the declaration is in the global scope, or the first time the local block is executed if the declaration is inside a local block. After that, it will preserve its last value on successive bars, until we reassign a new value to it. This behavior is very

useful in many cases where a variable's value must persist through the iterations of a script across successive bars. For example, suppose we'd like to count the number of green bars on the chart:

//@version=5 indicator("Green Bars Count") var count = 0 isGreen = close >= open if isGreen count := count + 1 plot(count)

Without the `var` modifier, variable `count` would be reset to zero (thus losing its value) every time a new bar update triggered a script recalculation.

Declaring variables on the first bar only is often useful to manage drawings more efficiently. Suppose we want to extend the last bar's close line to the right of the right chart. We could write:

//@version=5 indicator("Inefficient version", "", true) closeLine = line.new(bar_index - 1, close, bar_index, close, extend = extend.right, width = 3) line.delete(closeLine[1])

but this is inefficient because we are creating and deleting the line on each historical bar and on each update in the realtime bar. It is more efficient to use:

//@version=5 indicator("Efficient version", "", true) var closeLine = line.new(bar_index - 1, close, bar_index, close, extend = extend.right, width = 3) if barstate.islast line.set_xy1(closeLine, bar_index - 1, close) line.set_xy2(closeLine, bar_index, close)

Note that:

- We initialize `closeLine` on the first bar only, using the var declaration mode
- We restrict the execution of the rest of our code to the chart's last bar by enclosing our code that updates the line in an if barstate.islast structure.

There is a very slight penalty performance for using the var declaration mode. For that reason, when declaring constants, it is preferable not to use var if performance is a concern, unless the initialization involves calculations that take longer than the maintenance penalty, e.g., functions with complex code or string manipulations.

`varip`

Understanding the behavior of variables using the varip declaration mode requires prior knowledge of Pine Script's execution model and bar states.

The varip keyword can be used to declare variables that escape the *rollback process*, which is explained in the page on Pine Script's execution model.

Whereas scripts only execute once at the close of historical bars, when a script is running in realtime, it executes every time the chart's feed detects a price or volume update. At every realtime update, Pine Script's runtime normally resets the values of a script's variables to their last committed value, i.e., the value they held when the previous bar closed. This is generally handy, as each realtime script execution starts from a known state, which simplifies script logic.

Sometimes, however, script logic requires code to be able to save variable values **between different executions** in the realtime bar. Declaring variables with varip makes that possible. The “ip” in varip stands for *intrabar persist*.

Let's look at the following code, which does not use varip:

```
//@version=5 indicator("") int updateNo = na if barstate.isnew updateNo := 1 else updateNo := updateNo + 1 plot(updateNo, style = plot.style_circles)
```

On historical bars, barstate.isnew is always true, so the plot shows a value of “1” because the `else` part of the if structure is never executed. On realtime bars, barstate.isnew is only true when the script first executes on the bar's “open”. The plot will then briefly display “1” until subsequent executions occur. On the next executions during the realtime bar, the second branch of the if statement is executed because barstate.isnew is no longer true. Since `updateNo` is initialized to na at each execution, the `updateNo + 1` expression yields na, so nothing is plotted on further realtime executions of the script.

If we now use varip to declare the `updateNo` variable, the script behaves very differently:

```
//@version=5 indicator("") varip int updateNo = na if barstate.isnew updateNo := 1 else updateNo := updateNo + 1 plot(updateNo, style = plot.style_circles)
```

The difference now is that `updateNo` tracks the number of realtime updates that occur on each realtime bar. This can happen because the varip declaration allows the value of `updateNo` to be preserved between realtime updates; it is no longer rolled back at each realtime execution of the script. The test on barstate.isnew allows us to reset the update count when a new realtime bar comes in.

Because varip only affects the behavior of your code in the realtime bar, it follows that backtest results on strategies designed using logic based on varip variables will not be able to reproduce that behavior on historical bars, which will invalidate test results on them. This also entails that plots on historical bars will not be able to reproduce the script's behavior in realtime.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

To Pine Script® version 4

This is a guide to converting Pine Script® code from `@version=3` to `@version=4`.

Converter

The Pine Editor comes with a utility to automatically convert v3 indicators and strategies to v4. To access it, open a script with `//@version=3` in it and select the `Convert to v4` option in the `More` dropdown menu:

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)

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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 includes a utility to automatically convert v4 scripts to v5. To access it, open a script with `//@version=4` in it and select the “Convert to v5” option in the “More” menu identified by three dots at the top-right of the Editor's pane:

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)`

v4	v5
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()

v4	v5
supertrend()	ta.supertrend()
swma(x)	ta.swma(source)
tr	ta.tr
tr()	ta.tr()
tsi()	ta.tsi()
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()

v4	v5
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)`

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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 our 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 v5 Reference Manual popup when you `ctrl` / `cmd` + `click` on built-in Pine Script constructs, and opens the library publication page when doing the same with construsts imported from libraries.
- It provides an auto-complete feature that you can activate with `ctrl` + `space` or `cmd` + `I`, depending on your OS.
- It makes the write/compile/run cycle fast because saving a new version of a script loaded on the chart also executes it immediately.

To open the Editor, click on the “Pine Editor” tab at the bottom of your TradingView chart. This will open up the Editor’s pane.

First version

We will now create our first working Pine script, an implementation of the MACD indicator in Pine Script:

//@version=5 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)

- Start by bringing up the “Open” dropdown menu at the top right of the Editor and choose “New indicator”.
- Copy the example script above. The button on the top-right of the code widget allows you to do it with a single click.
- Select all the code already in the editor and replace it with the example script.
- Click “Save” and choose a name for your script. Your script is now saved in TradingView’s cloud, but under your account’s name. Nobody but you can use it.
- Click “Add to Chart” in the Editor’s menu bar. The MACD indicator appears in a separate *pane* under your chart.

Your first Pine script is running on your chart, which should look like this:

Let's look at our script's code, line by line:

Line 1: `//@version=5`

This is a compiler annotation telling the compiler the script will use version 5 of Pine Script.

Line 2: `indicator("MACD #1")`

Defines the name of the script that will appear on the chart as “MACD”.

Line 3: `fast = 12`

Defines a `fast` integer variable which will be the length of the fast EMA.

Line 4: `slow = 26`

Defines a `slow` integer variable which will be the length of the slow EMA.

Line 5: `fastMA = ta.ema(close, fast)`

Defines the variable `fastMA`, containing the result of the EMA calculation (Exponential Moving Average) with a length equal to `fast` (12), on the `close` series, i.e., the closing price of bars.

Line 6: `slowMA = ta.ema(close, slow)`

Defines the variable `slowMA`, containing the result of the EMA calculation with a length equal to `slow` (26), from `close`.

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 (Exponential Moving Average) 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 MACD “manually”, but because Pine Script is designed to write indicators and strategies, built-in Pine Script functions exist for many common indicators, including one for MACD: ta.macd().

This is the second version of our script:

//@version=5 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 have:

- Added inputs so we can change the lengths for the MAs
- We now use the ta.macd() built-in to calculate our MACD, which saves us three line and makes our code easier to read.

Let's repeat the same process as before to copy that code in a new indicator:

- Start by bringing up the “Open” dropdown menu at the top right of the Editor and choose “New indicator”.
- Then copy the example script above.
- Select all the code already in the editor and replace it with the second version of our script.
- Click “Save” and choose a name for your script different than the previous one.
- Click “Add to Chart” in the Editor's menu bar. The “MACD #2” indicator appears in a separate *pane* under the “MACD #1” indicator.

Your second Pine script is running on your chart. If you double-click on the indicator's name on your chart, you will bring up the script's “Settings/Inputs” tab, where you can now change the slow and fast lengths:

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 a variable, we have used the input() function so we can change the value in our script's “Settings/Inputs” tab. `12` will be the default value and the field's label will be `"Fast length"`. If the value is changed in the “Inputs” tab, the `fastInput` variable's content will contain the new value and the script will re-execute 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")`

We do the same for the slow length, taking care to use a different variable name, default value and text string for the field's label.

Line 5: `[macdLine, signalLine, histLine] = ta.macd(close, fastInput, slowInput, 9)`

This is where we call the ta.macd() built-in to perform all the first version's calculations in one line only. The function requires four parameters (the values after the function name, enclosed in parentheses). It returns three values into the three variables instead of only one, like the functions we used until now, which is why we need to enclose the list of three variables receiving the function's result in square brackets, 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`.

Line 6 and 7:

The variable names we are plotting there have changed, but the lines are doing the same thing as in our first version.

Our second version performs the same calculations as our first, but we can change the two lengths used to calculate it. Our code is also simpler and shorter by three lines. We have improved our script.

Next

We now recommend you go to our Next Steps page.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

First steps

Introduction

Welcome to the Pine Script® v5 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.

In 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 Script 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 & Strategies” button, or
- By browsing TradingView's Community Scripts, the largest repository of trading scripts in the world, with more than 100,000 scripts, most 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, use the “Indicators, Metrics & Strategies” button:

The dialog box 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 its name when you mouse over it.
- **Personal** displays the scipts you have written and saved in the Pine Editor. They are saved in TradingView's cloud.
- **Technicals** groups most TradingView built-ins 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 100,000+ published scripts written by TradingView users. The scripts can be sorted by one of the three different filters – Editor's Picks only shows open-source scripts hand-picked my our script moderatots, 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, the section containing the TradingView built-ins is selected:

When you click on one of the indicators or strategies (the ones marked with a symbol with two arrows), it loads on your chart.

Browsing Community Scripts

From TradingView's homepage, you can bring up the Community Scripts stream from the "Community" menu. Here, we are pointing to the "Editors' Picks" section, but there are many other categories you can choose from:

You can also search for scripts using the homepage's "Search" field, and filter scripts using different criteria. The Help Center has a page explaining the different types of scripts that are available.

The scripts stream shows script *widgets*, i.e., placeholders showing a miniature view of each publication's chart and description, and its author. By clicking on it you will open the *script's page*, where you can see the script on a chart, read the author's description, like the script, leave comments or read the script's source code if it was published open-source.

Once 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 reputation and open-source publications.

Reading code from Community Scripts is easy; if you don't see a grey or red “lock” icon in the upper-right corner of the script's widget, this indicates the script is open-source. By opening its script page, you will be able to see its source.

To see the code of TradingView built-ins, 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 will open and from there, you can see the script's code. If you want to play with it, you will need to press the “create a working copy” button. You will then be able to modify and save the code. Because you will have created a different version of the script, you will need to use the Editor's “Add to Chart” button to add that new copy to the chart.

This shows the Pine Editor having just opened after we selected the “View source” button from the indicator on our chart. We are about to create a working copy of its source because it is read-only for now, as indicated by the orange warning text:

You can also open TradingView built-in indicators from the Pine Editor (accessible from the “Pine Editor” tab at the bottom of the chart) by using the “Open” -> “Built-in script…” menu selection.

Writing scripts

We have built Pine Script to empower both budding and seasoned traders to create their own trading tools. We have designed it so it is relatively easy to learn for first-time programmers — although learning a first programming language, like trading, is rarely **very** easy for anyone — 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 oft-used functions that can be reused by other scripts.

The next step we recommend is to write your first indicator.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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 Pine Script's broker emulator, which can then simulate their execution. Strategies display backtest results in the “Strategy Tester” tab at the bottom of the chart, next to the “Pine Editor” tab.

Pine Script indicators also contain calculations, but cannot be used in backtesting. Because they do not require the broker emulator, they use less resources and will run faster. It is thus advantageous to use indicators whenever you can.

Both indicators and strategies can run in either overlay mode (over the chart's bars) or pane mode (in a separate section below or above the chart). Both can also plot information in their respective space, and both can generate alert events.

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 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 v5 User Manual
- Our Pine Script v5 Reference Manual

The Pine Script v5 User Manual, which is located on its separate page and in English only.

The Pine Script v5 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, which can be accessed from the Pine Editor, by either `ctrl` / `cmd` + `clicking` on a keyword, or by using the Editor's "More/Reference Manual…" menu. The Reference Manual is translated in other languages.

There are five different versions of Pine Script. Ensure the documentation you use corresponds to the Pine Script version you are coding with.

Where to go from here?

This Pine Script v5 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 v5 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!

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Release notes

This page contains release notes describing notable changes to the Pine Script® experience.

2024

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.
- ssyminfo.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

Matrix support 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 historic 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 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 pseudo-random 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.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Welcome to Pine Script® v5

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 100,000 Community Scripts.

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.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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 click the “Pine Editor” button on the toolbar. Once in the editor window, click the “Open” button, then select “Built-in script...” from the dropdown list 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. They can also be accessed from charts using the “Indicators & Strategies” button and the “Community Scripts” tab of the script searching dialog box.

External resources

- The PineCoders account on TradingView publishes useful information for Pine Script programmers. They also have content on their website.
- Kodify has TradingView tutorials on various topics for beginners and more experienced programmers alike. Topics include plotting, alerts, strategy orders, and complete example indicators and strategies.
- You can ask questions about programming in Pine Script in the `[pine-script]` tag on StackOverflow.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Debugging

Introduction

TradingView's close integration between the Pine Editor and the chart interface facilitates efficient, interactive debugging of Pine Script® code, as scripts can produce dynamic results in multiple locations, on and off the chart. Programmers can utilize such results to refine their script's behaviors and ensure everything works as expected.

When a programmer understands the appropriate techniques for inspecting the variety of behaviors one may encounter while writing a script, they can quickly and thoroughly identify and resolve potential problems in their code, which allows for a more seamless overall coding experience. This page demonstrates some of the handiest ways to debug code when working with Pine Script.

> Notice!Before venturing further on this page, we recommend familiarizing yourself with Pine's Execution model and Type system, as it's crucial to understand these details when debugging in the Pine Script environment.

The lay of the land

Pine scripts can output their results in multiple different ways, any of which programmers can utilize for debugging.

The `plot*()` functions can display results in a chart pane, the script's status line, the price (y-axis) scale, and the Data Window, providing simple, convenient ways to debug numeric and conditional values:

//@version=5 indicator("The lay of the land - Plots") // Plot the `bar_index` in all available locations. plot(bar_index, "bar_index", color.teal, 3)

Note that:

- A script's status line outputs will only show when enabling the “Values” checkbox within the “Indicators” section of the chart's “Status line” settings.
- Price scales will only show plot values or names when enabling the options from the “Indicators and financials” dropdown in the chart's “Scales and lines” settings.

The bgcolor() function displays colors in the script pane's background, and the barcolor() function changes the colors of the main chart's bars or candles. Both of these functions provide a simple way to visualize conditions:

//@version=5 indicator("The lay of the land - 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")

Pine's drawing types (line, box, polyline, label) produce drawings in the script's pane. While they don't return results in other locations, such as the status line or Data Window, they provide alternative, flexible solutions

for inspecting numeric values, conditions, and strings directly on the chart:

//@version=5 indicator("The lay of the land - Drawings", overlay = true) //@variable Is `true` when the time changes 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 //@variable Draws a line from the previous `closedIndex` and `closedPrice` to the current values. line debugLine = line.new(closedIndex[1], closedPrice[1], closedIndex, closedPrice, width = 2) //@variable Variable info 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.new(closedIndex, closedPrice, debugText, color = color.purple, textcolor = color.white)

The `log.*()` functions produce Pine Logs results. Every time a script calls any of these functions, the script logs a message in the Pine Logs pane, along with a timestamp and navigation options to identify the specific times, chart bars, and lines of code that triggered a log:

//@version=5 indicator("The lay of the land - 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))

One can apply any of the above, or a combination, to establish debugging routines to fit their needs and preferences, depending on the data types and structures they're working with. See the sections below for detailed explanations of various debugging techniques.

Numeric values

When creating code in Pine Script, working with numbers is inevitable. Therefore, to ensure a script works as intended, it's crucial to understand how to inspect the numeric (int and float) values it receives and calculates.

> Notice!This section discusses fundamental *chart-based* approaches for debugging numbers. Scripts can also convert numbers to strings, allowing one to inspect numbers using string-related techniques. For more information, see the Strings and Pine Logs sections.

Plotting numbers

One of the most straightforward ways to inspect a script's numeric values is to use `plot*()` functions, which can display results graphically on the chart and show formatted numbers in the script's status line, the price scale, and the Data Window. The locations where a `plot*()` function displays its results depend on the `display` parameter. By default, its value is display.all.

> Notice!Only a script's *global scope* can contain `plot*()` calls, meaning these functions can only accept global variables and literals. They cannot use variables declared from the local scopes of loops,

> conditional structures, or user-defined functions and methods.

The following example uses the plot() function to display the 1-bar change in the value of the built-in time variable measured in chart timeframes (e.g., a plotted value of 1 on the “1D” chart means there is a one-day difference between the opening times of the current and previous bars). Inspecting this series can help to identify time gaps in a chart's data, which is helpful information when designing time-based indicators.

Since we have not specified a `display` argument, the function uses display.all, meaning it will show data in *all* possible locations, as we see below:

//@version=5 indicator("Plotting numbers demo", "Time changes") //@variable The one-bar change in the chart symbol's `time` value, measured in units of the chart timeframe. float timeChange = ta.change(time) / (1000.0 * timeframe.in_seconds()) // Display the `timeChange` in all possible locations. plot(timeChange, "Time difference (in chart bar units)", color.purple, 3)

Note that:

- The numbers displayed in the script's status line and the Data Window reflect the plotted values at the location of the chart's cursor. These areas will show the latest bar's value when the mouse pointer isn't on the chart.
- The number in the price scale reflects the latest available value on the visible chart.

Without affecting the scale

When debugging multiple numeric values in a script, programmers may wish to inspect them without interfering with the price scales or cluttering the visual outputs in the chart's pane, as distorted scales and overlapping plots may make it harder to evaluate the results.

A simple way to inspect numbers without adding more visuals to the chart's pane is to change the `display` values in the script's `plot*()` calls to other `display.*` variables or expressions using them.

Let's look at a practical example. Here, we've drafted the following script that calculates a custom-weighted moving average by dividing the sum of `weight * close` values by the sum of the `weight` series:

//@version=5 indicator("Plotting without affecting the scale demo", "Weighted Average", true) //@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 `lengthInput`-bar weighted average. float average = numerator / denominator // Plot the `average`. plot(average, "Weighted Average", linewidth = 3)

Suppose we'd like to inspect the variables used in the `average` calculation to understand and fine-tune the result. If we were to use plot() to display the script's `weight`, `numerator`, and `denominator` in all locations, we can no longer easily identify our `average` line on the chart since each variable has a radically different scale:

//@version=5 indicator("Plotting without affecting the scale demo", "Weighted Average", true) //@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(close * weight, lengthInput) //@variable The denominator of the

average. float denominator = math.sum(weight, lengthInput) //@variable The `lengthInput`-bar weighted average. 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)

While we could hide individual plots from the “Style” tab of the script's settings, doing so also prevents us from inspecting the results in any other location. To simultaneously view the variables' values and preserve the scale of our chart, we can change the `display` values in our debug plots.

The version below includes a `debugLocations` variable in the debug plot() calls with a value of `display.all - display.pane` to specify that all locations *except* the chart pane will show the results. Now we can inspect the calculation's values without the extra clutter:

//@version=5 indicator("Plotting without affecting the scale demo", "Weighted Average", true) //@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(close * weight, lengthInput) //@variable The denominator of the average. float denominator = math.sum(weight, lengthInput) //@variable The `lengthInput`-bar weighted average. 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)

From local scopes

A script's *local scopes* are sections of indented code within conditional structures, functions, and methods. When working with variables declared within these scopes, using the `plot*()` functions to display their values directly *will not* work, as plots only work with literals and *global* variables.

To display a local variable's values using plots, one can assign its results to a global variable and pass that variable to the `plot*()` call.

> Notice!The approach described below works for local variables declared within conditional structures. Employing a similar process for functions and methods requires collections, user-defined types, or other built-in reference types. See the Debugging functions section for more information.

For example, this script calculates the all-time maximum and minimum change in the close price over a `lengthInput` period. It uses an if structure to declare a local `change` variable and update the global `maxChange` and `minChange` once every `lengthInput` bars:

//@version=5 indicator("Plotting numbers from local scopes demo", "Periodic changes") //@variable The number of chart bars in each period. int lengthInput = input.int(20, "Period length", 1) //@variable The maximum `close` change over each `lengthInput` period on the chart. var float maxChange = na //@variable The minimum `close` change over each `lengthInput` period on the chart. var float minChange = na //@variable Is `true` once every `lengthInput` bars. bool periodClose = bar_index % lengthInput == 0 if periodClose //@variable The change in `close` prices over `lengthInput` bars. float change = close - close[lengthInput] // Update the global `maxChange` and `minChange`. maxChange :=

math.max(nz(maxChange, change), change) minChange := math.min(nz(minChange, change), change) // Plot the `maxChange` and `minChange`. plot(maxChange, "Max periodic change", color.teal, 3) plot(minChange, "Min periodic change", color.maroon, 3) hline(0.0, color = color.gray, linestyle = hline.style_solid)

Suppose we want to inspect the history of the `change` variable using a plot. While we cannot plot the variable directly since the script declares it in a local scope, we can assign its value to another *global* variable for use in a `plot*()` function.

Below, we've added a `debugChange` variable with an initial value of na to the global scope, and the script reassigns its value within the if structure using the local `change` variable. Now, we can use plot() with the `debugChange` variable to view the history of available `change` values:

//@version=5 indicator("Plotting numbers from local scopes demo", "Periodic changes") //@variable The number of chart bars in each period. int lengthInput = input.int(20, "Period length", 1) //@variable The maximum `close` change over each `lengthInput` period on the chart. var float maxChange = na //@variable The minimum `close` change over each `lengthInput` period on the chart. var float minChange = na //@variable Is `true` once every `lengthInput` bars. bool periodClose = bar_index % lengthInput == 0 //@variable Tracks the history of the local `change` variable. float debugChange = na if periodClose //@variable The change in `close` prices over `lengthInput` bars. float change = close - close[lengthInput] // Update the global `maxChange` and `minChange`. maxChange := math.max(nz(maxChange, change), change) minChange := math.min(nz(minChange, change), change) // Assign the `change` value to the `debugChange` variable. debugChange := change // Plot the `maxChange` and `minChange`. plot(maxChange, "Max periodic change", color.teal, 3) plot(minChange, "Min periodic change", color.maroon, 3) hline(0.0, color = color.gray, linestyle = hline.style_solid) // Create a debug plot to visualize the `change` history. plot(debugChange, "Extracted change", color.purple, 15, plot.style_areabr)

Note that:

- The script uses plot.style_areabr in the debug plot, which doesn't bridge over na values as the default style does.
- When the rightmost visible bar's plotted value is na the number in the price scale represents the latest *non-na* value before that bar, if one exists.

With drawings

An alternative approach to graphically inspecting the history of a script's numeric values is to use Pine's drawing types, including lines, boxes, polylines, and labels.

While Pine drawings don't display results anywhere other than the chart pane, scripts can create them from within *local scopes*, including the scopes of functions and methods (see the Debugging functions section to learn more). Additionally, scripts can position drawings at *any* available chart location, irrespective of the current bar_index.

For example, let's revisit the “Periodic changes” script from the previous section. Suppose we'd like to inspect the history of the local `change` variable *without* using a plot. In this case, we can avoid declaring a separate global variable and instead create drawing objects directly from the if structure's local scope.

The script below is a modification of the previous script that uses boxes to visualize the `change` variable's behavior. Inside the scope of the if structure, it calls box.new() to create a box that spans from the bar

`lengthInput` bars ago to the current bar_index:

//@version=5 indicator("Drawing numbers from local scopes demo", "Periodic changes", max_boxes_count = 500) //@variable The number of chart bars in each period. int lengthInput = input.int(20, "Period length", 1) //@variable The maximum `close` change over each `lengthInput` period on the chart. var float maxChange = na //@variable The minimum `close` change over each `lengthInput` period on the chart. var float minChange = na //@variable Is `true` once every `lengthInput` bars. bool periodClose = bar_index % lengthInput == 0 if periodClose //@variable The change in `close` prices over `lengthInput` bars. float change = close - close[lengthInput] // Update the global `maxChange` and `minChange`. maxChange := math.max(nz(maxChange, change), change) minChange := math.min(nz(minChange, change), change) //@variable Draws a box on the chart to visualize the `change` value. box debugBox = box.new(bar_index - lengthInput, math.max(change, 0.0), bar_index, math.min(change, 0.0), color.purple, bgcolor = color.new(color.purple, 80), text = str.tostring(change)) // Plot the `maxChange` and `minChange`. plot(maxChange, "Max periodic change", color.teal, 3) plot(minChange, "Min periodic change", color.maroon, 3) hline(0.0, color = color.gray, linestyle = hline.style_solid)

Note that:

- The script includes `max_boxes_count = 500` in the indicator() function, which allows it to show up to 500 boxes on the chart.
- We used math.max(change, 0.0) and math.min(change, 0.0) in the box.new() function as the `top` and `bottom` values.
- The box.new() call includes str.tostring(change) as its `text` argument to display a *"string" representation* of the `change` variable's "float" value in each box drawing. See this portion of the Strings section below to learn more about representing data with strings.

For more information about using boxes and other related drawing types, see our User Manual's Lines and boxes page.

Conditions

Many scripts one will create in Pine involve declaring and evaluating *conditions* to dictate specific script actions, such as triggering different calculation patterns, visuals, signals, alerts, strategy orders, etc. As such, it's imperative to understand how to inspect the conditions a script uses to ensure proper execution.

> Notice!This section discusses debugging techniques based on chart visuals. To learn about *logging* conditions, see the Pine Logs section below.

As numbers

One possible way to debug a script's conditions is to define *numeric values* based on them, which allows programmers to inspect them using numeric approaches, such as those outlined in the previous section.

Let's look at a simple example. This script calculates the ratio between the ohlc4 price and the `lengthInput`-bar moving average. It assigns a condition to the `priceAbove` variable that returns `true` whenever the value of the ratio exceeds 1 (i.e., the price is above the average).

To inspect the occurrences of the condition, we created a `debugValue` variable assigned to the result of an expression that uses the ternary ?: operator to return 1 when `priceAbove` is `true` and 0 otherwise. The script plots the variable's value in all available locations:

//@version=5 indicator("Conditions as numbers demo", "MA signal") //@variable The number of bars in the moving average calculation. int lengthInput = input.int(20, "Length", 1) //@variable The ratio of the `ohlc4` price to its `lengthInput`-bar moving average. float ratio = ohlc4 / ta.sma(ohlc4, lengthInput) //@variable The condition to inspect. Is `true` when `ohlc4` is above its moving average, `false` otherwise. bool priceAbove = ratio > 1.0 //@variable Returns 1 when the `priceAbove` condition is `true`, 0 otherwise. int debugValue = priceAbove ? 1 : 0 // Plot the `debugValue. plot(debugValue, "Conditional number", color.teal, 3)

Note that:

- Representing “bool” values using numbers also allows scripts to display conditional shapes or characters at specific y-axis locations with plotshape() and plotchar(), and it facilitates conditional debugging with plotarrow(). See the next section to learn more.

Plotting conditional shapes

The plotshape() and plotchar() functions provide utility for debugging conditions, as they can plot shapes or characters at absolute or relative chart locations whenever they contain a `true` or non-na `series` argument.

These functions can also display *numeric* representations of the `series` in the script's status line and the Data Window, meaning they're also helpful for debugging numbers. We show a simple, practical way to debug numbers with these functions in the Tips section.

The chart locations of the plots depend on the `location` parameter, which is location.abovebar by default.

> Notice!When using location.abovebar or location.belowbar, the function positions the shapes/characters relative to the *main chart* prices. If the script plots its values in a separate chart pane, we recommend debugging with other `location` options to avoid affecting the pane's scale.

Let's inspect a condition using these functions. The following script calculates an RSI with a `lengthInput` length and a `crossBelow` variable whose value is the result of a condition that returns `true` when the RSI crosses below 30. It calls plotshape() to display a circle near the top of the pane each time the condition occurs:

//@version=5 indicator("Conditional shapes demo", "RSI cross under 30") //@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 the `crossBelow` condition as circles near the top of the pane. plotshape(crossBelow, "RSI crossed below 30", shape.circle, location.top, color.red, size = size.small)

Note that:

- The status line and Data Window show a value of 1 when `crossBelow` is `true` and 0 when it's `false`.

Suppose we'd like to display the shapes at *precise* locations rather than relative to the chart pane. We can achieve this by using conditional numbers and location.absolute in the plotshape() call.

In this example, we've modified the previous script by creating a `debugNumber` variable that returns the `rsi` value when `crossBelow` is `true` and na otherwise. The plotshape() function uses this new variable as its `series` argument and location.absolute as its `location` argument:

//@version=5 indicator("Conditional shapes demo", "RSI cross under 30") //@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) //@variable Returns the `rsi` when `crossBelow` is `true`, `na` otherwise. float debugNumber = crossBelow ? rsi : na // Plot the `rsi`. plot(rsi, "RSI", color.rgb(136, 76, 146), linewidth = 3) // Plot circles at the `debugNumber`. plotshape(debugNumber, "RSI when it crossed below 30", shape.circle, location.absolute, color.red, size = size.small)

Note that:

- Since we passed a *numeric* series to the function, our conditional plot now shows the values of the `debugNumber` in the status line and Data Window instead of 1 or 0.

Another handy way to debug conditions is to use plotarrow(). This function plots an arrow with a location relative to the *main chart prices* whenever the `series` argument is nonzero and not na. The length of each arrow varies with the `series` value supplied. As with plotshape() and plotchar(), plotarrow() can also display numeric results in the status line and the Data Window.

> Notice!Since this function always positions arrows relative to the main chart prices, we recommend only using it if the script occupies the main chart pane to avoid otherwise interfering with the scale.

This example shows an alternative way to inspect our `crossBelow` condition using plotarrow(). In this version, we've set `overlay` to `true` in the indicator() function and added a plotarrow() call to visualize the conditional values. The `debugNumber` in this example measures how far the `rsi` dropped below 30 each time the condition occurs:

//@version=5 indicator("Conditional shapes demo", "RSI cross under 30", true) //@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) //@variable Returns `rsi - 30.0` when `crossBelow` is `true`, `na` otherwise. float debugNumber = crossBelow ? rsi - 30.0 : na // Plot the `rsi`. plot(rsi, "RSI", color.rgb(136, 76, 146), display = display.data_window) // Plot circles at the `debugNumber`. plotarrow(debugNumber, "RSI cross below 30 distnce")

Note that:

- We set the `display` value in the plot() of the `rsi` to display.data_window to preserve the chart's scale.

To learn more about plotshape(), plotchar(), and plotarrow(), see this manual's Text and shapes page.

Conditional colors

An elegant way to visually represent conditions in Pine is to create expressions that return color values based on `true` or `false` states, as scripts can use them to control the appearance of drawing objects or the results of `plot*()`, fill(), bgcolor(), or barcolor() calls.

> Notice!As with `plot*()` functions, scripts can only call fill(), bgcolor() and barcolor() from the *global scope*, and the functions cannot accept any local variables.

For example, this script calculates the change in close prices over `lengthInput` bars and declares two “bool” variables to identify when the price change is positive or negative.

The script uses these “bool” values as conditions in ternary expressions to assign the values of three “color” variables, then uses those variables as the `color` arguments in plot(), bgcolor(), and barcolor() to debug the results:

//@version=5 indicator("Conditional colors demo", "Price change colors") //@variable The number of bars in the price change calculation. int lengthInput = input.int(10, "Length", 1) //@variable The change in `close` prices over `lengthInput` bars. float priceChange = ta.change(close, lengthInput) //@variable Is `true` when the `priceChange` is a positive value, `false` otherwise. bool isPositive = priceChange > 0 //@variable Is `true` when the `priceChange` is a negative value, `false` otherwise. bool isNegative = priceChange < 0 //@variable Returns a color for the `priceChange` plot to show when `isPositive`, `isNegative`, or neither occurs. color plotColor = isPositive ? color.teal : isNegative ? color.maroon : chart.fg_color //@variable Returns an 80% transparent color for the background when `isPositive` or `isNegative`, `na` otherwise. color bgColor = isPositive ? color.new(color.aqua, 80) : isNegative ? color.new(color.fuchsia, 80) : na //@variable Returns a color to emphasize chart bars when `isPositive` occurs. Otherwise, returns the `chart.bg_color`. color barColor = isPositive ? color.orange : chart.bg_color // Plot the `priceChange` and color it with the `plotColor`. plot(priceChange, "Price change", plotColor, style = plot.style_area) // Highlight the pane's background with the `bgColor`. bgcolor(bgColor, title = "Background highlight") // Emphasize the chart bars with positive price change using the `barColor`. barcolor(barColor, title = "Positive change bars")

Note that:

- The barcolor() function always colors the main chart's bars, regardless of whether the script occupies another chart pane, and the chart will only display the results if the bars are visible.

See the Colors, Fills, Backgrounds, and Bar coloring pages for more information about working with colors, filling plots, highlighting backgrounds, and coloring bars.

Using drawings

Pine Script's drawing types provide flexible ways to visualize conditions on the chart, especially when the conditions are within local scopes.

Consider the following script, which calculates a custom `filter` with a smoothing parameter (`alpha`) that changes its value within an if structure based on recent volume conditions:

//@version=5 indicator("Conditional drawings demo", "Volume-based filter", true) //@variable The number of bars in the volume average. int lengthInput = input.int(20, "Volume average length", 1) //@variable The average `volume` over `lengthInput` bars. float avgVolume = ta.sma(volume, lengthInput) //@variable A custom price filter based on volume activity. float filter = close //@variable The smoothing parameter of the

filter calculation. Its value depends on multiple volume conditions. float alpha = na // Set the `alpha` to 1 if `volume` exceeds its `lengthInput`-bar moving average. if volume > avgVolume alpha := 1.0 // Set the `alpha` to 0.5 if `volume` exceeds its previous value. else if volume > volume[1] alpha := 0.5 // Set the `alpha` to 0.01 otherwise. else alpha := 0.01 // Calculate the new `filter` value. filter := (1.0 - alpha) * nz(filter[1], filter) + alpha * close // Plot the `filter`. plot(filter, "Filter", linewidth = 3)

Suppose we'd like to inspect the conditions that control the `alpha` value. There are several ways we could approach the task with chart visuals. However, some approaches will involve more code and careful handling.

For example, to visualize the if structure's conditions using plotted shapes or background colors, we'd have to create additional variables or expressions in the global scope for the `plot*()` or bgcolor() functions to access.

Alternatively, we can use drawing types to visualize the conditions concisely without those extra steps.

The following is a modification of the previous script that calls label.new() within specific branches of the conditional structure to draw labels on the chart whenever those branches execute. These simple changes allow us to identify those conditions on the chart without much extra code:

//@version=5 indicator("Conditional drawings demo", "Volume-based filter", true, max_labels_count = 500) //@variable The number of bars in the volume average. int lengthInput = input.int(20, "Volume average length", 1) //@variable The average `volume` over `lengthInput` bars. float avgVolume = ta.sma(volume, lengthInput) //@variable A custom price filter based on volume activity. float filter = close //@variable The smoothing parameter of the filter calculation. Its value depends on multiple volume conditions. float alpha = na // Set the `alpha` to 1 if `volume` exceeds its `lengthInput`-bar moving average. if volume > avgVolume // Add debug label. label.new(chart.point.now(high), "alpha = 1", color = color.teal, textcolor = color.white) alpha := 1.0 // Set the `alpha` to 0.5 if `volume` exceeds its previous value. else if volume > volume[1] // Add debug label. label.new(chart.point.now(high), "alpha = 0.5", color = color.green, textcolor = color.white) alpha := 0.5 // Set the `alpha` to 0.01 otherwise. else alpha := 0.01 // Calculate the new `filter` value. filter := (1.0 - alpha) * nz(filter[1], filter) + alpha * close // Plot the `filter`. plot(filter, "Filter", linewidth = 3)

Note that:

- We added the label.new() calls *above* the `alpha` reassignment expressions, as the returned types of each branch in the if structure must match.
- The indicator() function includes `max_labels_count = 500` to specify that the script can show up to 500 labels on the chart.

Compound and nested conditions

When a programmer needs to identify situations where more than one condition can occur, they may construct *compound conditions* by aggregating individual conditions with logical operators (and, or).

For example, this line of code shows a `compoundCondition` variable that only returns `true` if `condition1` and either `condition2` or `condition3` occurs:

bool compoundCondition = condition1 and (condition2 or condition3)

One may alternatively create *nested conditions* using conditional structures or ternary expressions. For example, this if structure assigns `true` to the `nestedCondition` variable if `condition1` and `condition2` or

`condition3` occurs. However, unlike the logical expression above, the branches of this structure also allow the script to execute additional code before assigning the “bool” value:

bool nestedCondition = false if condition1 // [additional_code] if condition2 // [additional_code] nestedCondition := true else if condition3 // [additional_code] nestedCondition := true

In either case, whether working with compound or nested conditions in code, one will save many headaches and ensure they work as expected by validating the behaviors of the *individual conditions* that compose them.

For example, this script calculates an `rsi` and the `median` of the `rsi` over `lengthInput` bars. Then, it creates five variables to represent different singular conditions. The script uses these variables in a logical expression to assign a “bool” value to the `compoundCondition` variable, and it displays the results of the `compoundCondition` using a conditional background color:

//@version=5 indicator("Compound conditions demo") //@variable The length of the RSI and median RSI calculations. int lengthInput = input.int(14, "Length", 2) //@variable The `lengthInput`-bar RSI. float rsi = ta.rsi(close, lengthInput) //@variable The `lengthInput`-bar median of the `rsi`. 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.rgb(201, 109, 34), 3) plot(median, "RSI Median", color.rgb(180, 160, 102), 2) // Highlight the background red when the `compundCondition` occurs. bgcolor(compundCondition ? color.new(color.red, 60) : na, title = "compundCondition")

As we see above, it's not necessarily easy to understand the behavior of the `compoundCondition` by only visualizing its end result, as five underlying singular conditions determine the final value. To effectively debug the `compoundCondition` in this case, we must also inspect the conditions that compose it.

In the example below, we've added five plotchar() calls to display characters on the chart and numeric values in the status line and Data Window when each singular condition occurs. Inspecting each of these results provides us with more complete information about the `compoundCondition`'s behavior:

//@version=5 indicator("Compound conditions demo") //@variable The length of the RSI and median RSI calculations. int lengthInput = input.int(14, "Length", 2) //@variable The `lengthInput`-bar RSI. float rsi = ta.rsi(close, lengthInput) //@variable The `lengthInput`-bar median of the `rsi`. 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.rgb(201, 109, 34), 3) plot(median, "RSI Median", color.rgb(180, 160, 102), 2) // Highlight the background red when the `compundCondition` occurs. bgcolor(compundCondition ? color.new(color.red, 60) : na, title = "compundCondition") // Plot characters on the chart when conditions 1-5 occur. plotchar(changeNegative ? rsi : na, "changeNegative (1)", "1", location.absolute, chart.fg_color) plotchar(prevAbove70 ? 70.0 : na, "prevAbove70 (2)", "2", location.absolute, chart.fg_color) plotchar(closeBelow ? close : na, "closeBelow (3)", "3", location.bottom, chart.fg_color) plotchar(betweenLevels ? 60 : na, "betweenLevels (4)", "4", location.absolute, chart.fg_color) plotchar(aboveMedian ? median : na, "aboveMedian (5)", "5", location.absolute, chart.fg_color)

Note that:

- Each plotchar() call uses a conditional number as the `series` argument. The functions display the numeric values in the status line and Data Window.
- All the plotchar() calls, excluding the one for the `closeBelow` condition, use location.absolute as the `location` argument to display characters at precise locations whenever their `series` is not na (i.e., the condition occurs). The call for `closeBelow` uses location.bottom to display its characters near the bottom of the pane.
- In this section's examples, we assigned individual conditions to separate variables with straightforward names and annotations. While this format isn't required to create a compound condition since one can combine conditions directly within a logical expression, it makes for more readable code that's easier to debug, as explained in the Tips section.

Strings

Strings are sequences of alphanumeric, control, and other characters (e.g., Unicode). They provide utility when debugging scripts, as programmers can use them to represent a script's data types as human-readable text and inspect them with drawing types that have text-related properties, or by using Pine Logs.

> Notice!This section discusses “string” conversions and inspecting strings via labels and tables. Boxes can also display text. However, their utility for debugging strings is more limited than the techniques covered in this section and the Pine Logs section below.

Representing other types

Users can create “string” representations of virtually any data type, facilitating effective debugging when other approaches may not suffice. Before exploring “string” inspection techniques, let's briefly review ways to *represent* a script's data using strings.

Pine Script includes predefined logic to construct “string” representations of several other built-in types, such as int, float, bool, array, and matrix. Scripts can conveniently represent such types as strings via the str.tostring() and str.format() functions.

For example, this snippet creates strings to represent multiple values using these functions:

//@variable Returns: "1.25" string floatRepr = str.tostring(1.25) //@variable Returns: "1" string rounded0 = str.tostring(1.25, "#") //@variable Returns: "1.3" string rounded1 = str.tostring(1.25, "#.#") //@variable Returns: "1.2500" string trailingZeros = str.tostring(1.25, "#.0000") //@variable Returns: "true" string

trueRepr = str.tostring(true) //@variable Returns: "false" string falseRepr = str.tostring(5 == 3) //@variable Returns: "[1, 2, -3.14]" string floatArrayRepr = str.tostring(array.from(1, 2.0, -3.14)) //@variable Returns: "[2, 20, 0]" string roundedArrayRepr = str.tostring(array.from(2.22, 19.6, -0.43), "#") //@variable Returns: "[Hello, World, !]" string stringArrayRepr = str.tostring(array.from("Hello", "World", "!")) //@variable Returns: "Test: 2.718 ^ 2 > 5: true" string mixedTypeRepr = str.format("{0}{1, number, #.###} ^ 2 > {2}: {3}", "Test: ", math.e, 5, math.e * math.e > 5) //@variable Combines all the above strings into a multi-line string. string combined = str.format("{0}\n{1}\n{2}\n{3}\n{4}\n{5}\n{6}\n{7}\n{8}\n{9}", floatRepr, rounded0, rounded1, trailingZeros, trueRepr, falseRepr, floatArrayRepr, roundedArrayRepr, stringArrayRepr, mixedTypeRepr)

When working with “int” values that symbolize UNIX timestamps, such as those returned from time-related functions and variables, one can also use str.format() or str.format_time() to convert them to human-readable date strings. This code block demonstrates multiple ways to convert a timestamp using these functions:

//@variable A UNIX timestamp, in milliseconds. int unixTime = 1279411200000 //@variable Returns: "2010-07-18T00:00:00+0000" string default = str.format_time(unixTime) //@variable Returns: "2010-07-18" string ymdRepr = str.format_time(unixTime, "yyyy-MM-dd") //@variable Returns: "07-18-2010" string mdyRepr = str.format_time(unixTime, "MM-dd-yyyy") //@variable Returns: "20:00:00, 2010-07-17" string hmsymdRepr = str.format_time(unixTime, "HH:mm:ss, yyyy-MM-dd", "America/New_York") //@variable Returns: "Year: 2010, Month: 07, Day: 18, Time: 12:00:00" string customFormat = str.format("Year: {0, time, yyyy}, Month: {1, time, MM}, Day: {2, time, dd}, Time: {3, time, hh:mm:ss}", unixTime, unixTime, unixTime, unixTime)

When working with types that *don't* have built-in “string” representations, e.g., color, map, user-defined types, etc., programmers can use custom logic or formatting to construct representations. For example, this code calls str.format() to represent a “color” value using its r, g, b, and t components:

//@variable The built-in `color.maroon` value with 17% transparency. color myColor = color.new(color.maroon, 17) // Get the red, green, blue, and transparency components from `myColor`. float r = color.r(myColor) float g = color.g(myColor) float b = color.b(myColor) float t = color.t(myColor) //@variable Returns: "color (r = 136, g = 14, b = 79, t = 17)" string customRepr = str.format("color (r = {0}, g = {1}, b = {2}, t = {3})", r, g, b, t)

There are countless ways one can represent data using strings. When choosing string formats for debugging, ensure the results are **readable** and provide enough information for proper inspection. The following segments explain ways to validate strings by displaying them on the chart using labels, and the section after these segments explains how to display strings as messages in the Pine Logs pane.

Using labels

Labels allow scripts to display dynamic text (“series strings”) at any available location on the chart. Where to display such text on the chart depends on the information the programmer wants to inspect and their debugging preferences.

On successive bars

When inspecting the history of values that affect the chart's scale or working with multiple series that have different types, a simple, handy debugging approach is to draw labels that display string representations on successive bars.

For example, this script calculates four series: `highestClose`, `percentRank`, `barsSinceHigh`, and `isLow`. It uses str.format() to create a formatted "string" representing the series values and a timestamp, then it calls label.new() to draw a label that display the results at the high on each bar:

```
//@version=5 indicator("Labels on successive bars demo", "Inspecting multiple series", true, max_labels_count = 500) //@variable The number of bars in the calculation window. int lengthInput = input.int(50, "Length", 1) //@variable The highest `close` over `lengthInput` bars. float highestClose = ta.highest(close, lengthInput) //@variable The percent rank of the current `close` compared to previous values over `lengthInput` bars. float percentRank = ta.percentrank(close, lengthInput) //@variable The number of bars since the `close` was equal to the `highestClose`. int barsSinceHigh = ta.barssince(close == highestClose) //@variable Is `true` when the `percentRank` is 0, i.e., when the `close` is the lowest. bool isLow = percentRank == 0.0 //@variable A multi-line string representing the `time`, `highestClose`, `percentRank`, `barsSinceHigh`, and `isLow`. string debugString = str.format( "time (GMT): {0, time, yyyy-MM-dd'T'HH:mm:ss}\nhighestClose: {1, number, #.####} \npercentRank: {2, number, #.##}%\nbarsSinceHigh: {3, number, integer}\nisLow: {4}", time, highestClose, percentRank, barsSinceHigh, isLow ) //@variable Draws a label showing the `debugString` at each bar's `high`. label debugLabel = label.new(chart.point.now(high), debugString, textcolor = color.white)
```

While the above example allows one to inspect the results of the script's series on any bar with a label drawing, consecutive drawings like these can clutter the chart, especially when viewing longer strings.

An alternative, more visually compact way to inspect successive bars' values with labels is to utilize the `tooltip` property instead of the `text` property, as a label will only show its tooltip when the cursor *hovers* over it.

Below, we've modified the previous script by using the `debugString` as the `tooltip` argument instead of the `text` argument in the label.new() call. Now, we can view the results on specific bars without the extra noise:

```
//@version=5 indicator("Tooltips on successive bars demo", "Inspecting multiple series", true, max_labels_count = 500) //@variable The number of bars in the calculation window. int lengthInput = input.int(50, "Length", 1) //@variable The highest `close` over `lengthInput` bars. float highestClose = ta.highest(close, lengthInput) //@variable The percent rank of the current `close` compared to previous values over `lengthInput` bars. float percentRank = ta.percentrank(close, lengthInput) //@variable The number of bars since the `close` was equal to the `highestClose`. int barsSinceHigh = ta.barssince(close == highestClose) //@variable Is `true` when the `percentRank` is 0, i.e., when the `close` is the lowest. bool isLow = percentRank == 0.0 //@variable A multi-line string representing the `time`, `highestClose`, `percentRank`, `barsSinceHigh`, and `isLow`. string debugString = str.format( "time (GMT): {0, time, yyyy-MM-dd'T'HH:mm:ss}\nhighestClose: {1, number, #.####} \npercentRank: {2, number, #.##}%\nbarsSinceHigh: {3, number, integer}\nisLow: {4}", time, highestClose, percentRank, barsSinceHigh, isLow ) //@variable Draws a label showing the `debugString` in a tooltip at each bar's `high`. label debugLabel = label.new(chart.point.now(high), tooltip = debugString)
```

It's important to note that a script can display up to 500 label drawings, meaning the above examples will only allow users to inspect the strings from the most recent 500 chart bars.

If a programmer wants to see the results from *earlier* chart bars, one approach is to create conditional logic that only allows drawings within a specific time range, e.g.:

```
if time >= startTime and time <= endTime <create_drawing_id>
```

If we use this structure in our previous example with chart.left_visible_bar_time and chart.right_visible_bar_time as the `startTime` and `endTime` values, the script will only create labels on **visible chart bars** and avoid drawing on others. With this logic, we can scroll to view labels on *any* chart bar, as long as there are up to `max_labels_count` bars in the visible range:

//@version=5 indicator("Tooltips on visible bars demo", "Inspecting multiple series", true, max_labels_count = 500) //@variable The number of bars in the calculation window. int lengthInput = input.int(50, "Length", 1) //@variable The highest `close` over `lengthInput` bars. float highestClose = ta.highest(close, lengthInput) //@variable The percent rank of the current `close` compared to previous values over `lengthInput` bars. float percentRank = ta.percentrank(close, lengthInput) //@variable The number of bars since the `close` was equal to the `highestClose`. int barsSinceHigh = ta.barssince(close == highestClose) //@variable Is `true` when the `percentRank` is 0, i.e., when the `close` is the lowest. bool isLow = percentRank == 0.0 //@variable A multi-line string representing the `time`, `highestClose`, `percentRank`, `barsSinceHigh`, and `isLow`. string debugString = str.format("time (GMT): {0, time, yyyy-MM-dd'T'HH:mm:ss}\nhighestClose: {1, number, #.####} \npercentRank: {2, number, #.##}%\nbarsSinceHigh: {3, number, integer}\nisLow: {4}", time, highestClose, percentRank, barsSinceHigh, isLow) if time >= chart.left_visible_bar_time and time <= chart.right_visible_bar_time //@variable Draws a label showing the `debugString` in a tooltip at each visible bar's `high`. label debugLabel = label.new(chart.point.now(high), tooltip = debugString)

Note that:

- If the visible chart contains more bars than allowed drawings, the script will only show results on the latest bars in the visible range. For best results with this technique, zoom on the chart to keep the visible range limited to the allowed number of drawings.

At the end of the chart

A frequent approach to debugging a script's strings with labels is to display them at the *end* of the chart, namely when the strings do not change or when only a specific bar's values require analysis.

The script below contains a user-defined `printLabel()` function that draws a label at the last available time on the chart, regardless of when the script calls it. We've used the function in this example to display a “Hello world!” string, some basic chart information, and the data feed's current OHLCV values:

//@version=5 indicator("Labels at the end of the chart demo", "Chart info", true) //@function Draws a label to print the `txt` at the last available time on the chart. // When called from the global scope, the label updates its text using the specified `txt` on every bar. //@param txt The string to display on the chart. //@param price The optional y-axis location of the label. If not specified, draws the label above the last chart bar. //@returns The resulting label ID. printLabel(string txt, float price = na) => int labelTime = math.max(last_bar_time, chart.right_visible_bar_time) var label result = label.new(labelTime, na, txt, xloc.bar_time, na(price) ? yloc.abovebar : yloc.price, na, label.style_none, chart.fg_color, size.large) label.set_text(result, txt) label.set_y(result, price) result //@variable A formatted string containing information about the current chart. string chartInfo = str.format("Symbol: {0}:{1}\nTimeframe: {2}\nStandard chart: {3}\nReplay active: {4}", syminfo.prefix, syminfo.ticker, timeframe.period, chart.is_standard, str.contains(syminfo.tickerid, "replay")) //@variable A formatted string containing OHLCV values. string ohlcvInfo = 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)) // Print "Hello world!" and the `chartInfo` at the end of the chart on the first bar. if barstate.isfirst printLabel("Hello world!" + "\n\n\n\n\n\n\n") printLabel(chartInfo + "\n\n") //

Print current `ohlcvInfo` at the end of the chart, updating the displayed text as new data comes in.
printLabel(ohlcvInfo)

Note that:

- The `printLabel()` function sets the x-coordinate of the drawn label using the max of the last_bar_time and the chart.right_visible_bar_time to ensure it always shows the results at the last available bar.
- When called from the *global scope*, the function creates a label with `text` and `y` properties that update on every bar.
- We've made three calls to the function and added linefeed characters (`\n`) to demonstrate that users can superimpose the results from multiple labels at the end of the chart if the strings have adequate line spacing.

Using tables

Tables display strings within cells arranged in columns and rows at fixed locations relative to a chart pane's visual space. They can serve as versatile chart-based debugging tools, as unlike labels, they allow programmers to inspect one or *more* “series strings” in an organized visual structure agnostic to the chart's scale or bar index.

For example, this script calculates a custom `filter` whose result is the ratio of the EMA of weighted close prices to the EMA of the `weight` series. For inspection of the variables used in the calculation, it creates a table instance on the first bar, initializes the table's cells on the last historical bar, then updates necessary cells with “string” representations of the values from `barsBack` bars ago on the latest chart bar:

//@version=5 indicator("Debugging with tables demo", "History inspection", true) //@variable The number of bars back in the chart's history to inspect. int barsBack = input.int(10, "Bars back", 0, 4999) //@variable The percent rank of `volume` over 10 bars. float weight = ta.percentrank(volume, 10) //@variable The 10-bar EMA of `weight * close` values. float numerator = ta.ema(weight * close, 10) //@variable The 10-bar EMA of `weight` values. float denominator = ta.ema(weight, 10) //@variable The ratio of the `numerator` to the `denominator`. float filter = numerator / denominator // Plot the `filter`. plot(filter, "Custom filter") //@variable The color of the frame, border, and text in the `debugTable`. color tableColor = chart.fg_color //@variable A table that contains "string" representations of variable names and values on the latest chart bar. var table debugTable = table.new(position.top_right, 2, 5, frame_color = tableColor, frame_width = 1, border_color = tableColor, border_width = 1) // Initialize cells on the last confirmed historical bar. if barstate.islastconfirmedhistory table.cell(debugTable, 0, 0, "Variable", text_color = tableColor) table.cell(debugTable, 1, 0, str.format("Value {0, number, integer} bars ago", barsBack), text_color = tableColor) table.cell(debugTable, 0, 1, "weight", text_color = tableColor) table.cell(debugTable, 1, 1, "", text_color = tableColor) table.cell(debugTable, 0, 2, "numerator", text_color = tableColor) table.cell(debugTable, 1, 2, "", text_color = tableColor) table.cell(debugTable, 0, 3, "denominator", text_color = tableColor) table.cell(debugTable, 1, 3, "", text_color = tableColor) table.cell(debugTable, 0, 4, "filter", text_color = tableColor) table.cell(debugTable, 1, 4, "", text_color = tableColor) // Update value cells on the last available bar. if barstate.islast table.cell_set_text(debugTable, 1, 1, str.tostring(weight[barsBack], format.percent)) table.cell_set_text(debugTable, 1, 2, str.tostring(numerator[barsBack])) table.cell_set_text(debugTable, 1, 3, str.tostring(denominator[barsBack])) table.cell_set_text(debugTable, 1, 4, str.tostring(filter[barsBack]))

Note that:

- The script uses the var keyword to specify that the table assigned to the `debugTable` variable on the first bar persists throughout the script's execution.
- This script modifies the table within two if structures. The first structure initializes the cells with table.cell() only on the last confirmed historical bar (barstate.islastconfirmedhistory). The second structure updates the `text` properties of relevant cells with string representations of our variables' values using table.cell_set_text() calls on the latest available bar (barstate.islast).

It's important to note that although tables can provide debugging utility, namely when working with multiple series or creating on-chart logs, they carry a higher computational cost than other techniques discussed on this page and may require *more code*. Additionally, unlike labels, one can only view a table's state from the latest script execution. We therefore recommend using them *wisely* and *sparingly* while debugging, opting for *simplified* approaches where possible. For more information about using table objects, see the Tables page.

Pine Logs

Pine Logs are *interactive messages* that scripts can output at specific points in their execution. They provide a powerful way for programmers to inspect a script's data, conditions, and execution flow with minimal code.

Unlike the other tools discussed on this page, Pine Logs have a deliberate design for in-depth script debugging. Scripts do not display Pine Logs on the chart or in the Data Window. Instead, they print messages with timestamps in the dedicated *Pine Logs pane*, which provides specialized navigation features and filtering options.

To access the Pine Logs pane, select “Pine Logs...” from the Editor's “More” menu or from the “More” menu of a script loaded on the chart that uses `log.*()` functions:

> Notice!Only **personal scripts** can generate Pine Logs. A published script *cannot* create logs, even if it has `log.*()` function calls in its code. One must consider alternative approaches, such as those outlined in the sections above, when publishing scripts with debugging functionality.

Creating logs

Scripts can create logs by calling the functions in the `log.*()` namespace.

All `log.*()` functions have the following signatures:

```
log.*(message) → void

log.*(formatString, arg0, arg1, ...) → void
```

The first overload logs a specified `message` in the Pine Logs pane. The second overload is similar to str.format(), as it logs a formatted message based on the `formatString` and the additional arguments supplied in the call.

Each `log.*()` function has a different *debug level*, allowing programmers to categorize and filter results shown in the pane:

- The log.info() function logs an entry with the *“info”* level that appears in the pane with gray text.
- The log.warning() function logs an entry with the *“warning”* level that appears in the pane with orange text.
- The log.error() function logs an entry with the *“error”* level that appears in the pane with red text.

This code demonstrates the difference between all three `log.*()` functions. It calls log.info(), log.warning(), and log.error() on the first available bar:

//@version=5 indicator("Debug levels demo", overlay = true) if barstate.isfirst log.info("This is an 'info' message.") log.warning("This is a 'warning' message.") log.error("This is an 'error' message.")

Pine Logs can execute anywhere within a script's execution. They allow programmers to track information from historical bars and monitor how their scripts behave on realtime, *unconfirmed* bars. When executing on historical bars, scripts generate a new message once for each `log.*()` call on a bar. On realtime bars, calls to `log.*()` functions can create new entries on *each new tick*.

For example, this script calculates the average ratio between each bar's `close - open` value to its `high - low` range. When the `denominator` is nonzero, the script calls log.info() to print the values of the calculation's variables on confirmed bars and log.warning() to print the values on unconfirmed bars. Otherwise, it uses log.error() to indicate that division by zero occurred, as such cases can affect the `average` result:

//@version=5 indicator("Logging historical and realtime data 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 range 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 // Log a division by zero error if the `denominator` is 0. if denominator == 0.0 log.error("Division by 0 in confirmed results!") // Otherwise, log the confirmed values. else log.info("Values (confirmed):\nnumerator: {1, number, #.########}\ndenominator: {2, number, #.########} \nratio: {0, number, #.########}\naverage: {3, number, #.########}", ratio, numerator, denominator, average) else // Log a division by zero error if the `denominator` is 0. if denominator == 0.0 log.error("Division by 0 on unconfirmed bar.") // Otherwise, log the unconfirmed values. else log.warning("Values (unconfirmed):\nnumerator: {1, number, #.########}\ndenominator: {2, number, #.########} \nratio: {0, number, #.########}\naverage: {3, number, #.########}", ratio, numerator, denominator, average)

Note that:

- Pine Logs *do not roll back* on each tick in an unconfirmed bar, meaning the results for those ticks show in the pane until the script restarts its execution. To only log messages on *confirmed* bars, use barstate.isconfirmed in the conditions that trigger a `log.*()` call.
- When logging on unconfirmed bars, we recommend ensuring those logs contain *unique information* or use different *debug levels* so you can filter the results as needed.
- The Pine Logs pane will show up to the most recent 10,000 entries for historical bars. If a script generates more than 10,000 logs on historical bars and a programmer needs to view earlier entries, they can use conditional logic to limit `log.*()` calls to specific occurrences. See this section for an example that limits log generation to a user-specified time range.

Inspecting logs

Pine Logs include some helpful features that simplify the inspection process. Whenever a script generates a log, it automatically prefixes the message with a granular timestamp to signify where the log event occurred in the time series. Additionally, each entry contains **“Source code”** and **“Scroll to bar”** icons, which appear when hovering over it in the Pine Logs pane:

Clicking an entry's “Source code” icon opens the script in the Pine Editor and highlights the specific line of code that triggered the log:

Clicking an entry's “Scroll to bar” icon navigates the chart to the specific bar where the log occurred, then temporarily displays a tooltip containing time information for that bar:

Note that:

- The time information in the tooltip depends on the chart's timeframe, just like the x-axis label linked to the chart's cursor and drawing tools. For example, the tooltip on an EOD chart will only show the weekday and the date, whereas the tooltip on a 10-second chart will also contain the time of day, including seconds.

When a chart includes more than one script that generates logs, it's important to note that each script maintains its own *independent* message history. To inspect the messages from a specific script when multiple are on the chart, select its title from the dropdown at the top of the Pine Logs pane:

Filtering logs

A single script can generate numerous logs, depending on the conditions that trigger its `log.*()` calls. While directly scrolling through the log history to find specific entries may suffice when a script only generates a few, it can become unwieldy when searching through hundreds or thousands of messages.

The Pine Logs pane includes multiple options for filtering messages, which allows one to simplify their results by isolating specific *character sequences*, *start times*, and *debug levels*.

Clicking the “Search” icon at the top of the pane opens a search bar, which matches text to filter logged messages. The search filter also highlights the matched portion of each message in blue for visual reference. For example, here, we entered “confirmed” to match all results generated by our previous script with the word somewhere in their text:

Notice that the results from this search also considered messages with *“unconfirmed”* as matches since the word contains our query. We can omit these matches by selecting the “Whole Word” checkbox in the options at the right of the search bar:

This filter also supports regular expressions (regex), which allow users to perform advanced searches that match custom *character patterns* when selecting the “Regex” checkbox in the search options. For example, this regex matches all entries that contain “average” followed by a sequence representing a number greater than 0.5 and less than or equal to 1:

```
average:\s*(0\.[6-9]\d*|0\.5\d*[1-9]\d*|1\.0*)
```

Clicking the “Start date” icon opens a dialog that allows users to specify the date and time of the first log shown in the results:

After specifying the starting point, a tag containing the starting time will appear above the log history:

Users can filter results by *debug level* using the checkboxes available when selecting the rightmost icon in the filtering options. Here, we've deactivated the “info” and “warning” levels so the results will only contain “error” messages:

Using inputs

Another, more involved way to interactively filter a script's logged results is to create inputs linked to conditional logic that activates specific `log.*()` calls in the code.

Let's look at an example. This code calculates an RMA of close prices and declares a few unique conditions to form a compound condition. The script uses log.info() to display important debugging information in the Pine Logs pane, including the values of the `compoundCondition` variable and the “bool” variables that determine its result.

We declared the `filterLogsInput`, `logStartInput`, and `logEndInput` variables respectively assigned to an input.bool() and two input.time() calls for custom log filtering. When `filterLogsInput` is `true`, the script will only generate a new log if the bar's time is between the `logStartInput` and `logEndInput` values, allowing us to interactively isolate the entries that occurred within a specific time range:

```
//@version=5 indicator("Filtering logs using inputs demo", "Compound condition in input range", 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.*()` functions assigned to the `filterLogsInput`, `logStartInput`, and `logEndInput` variables include a `group` argument to oragnize and distinguish them in the script's settings.
- The input.time() calls include `confirm = true` so that we can interactively set the start and end times directly on the chart. To reset the inputs, select “Reset points...” from the options in the script's “More” menu.

- The condition that triggers each log.info() call includes barstate.isconfirmed to limit log generation to *confirmed* bars.

Debugging functions

User-defined functions and methods are custom functions written by users. They encapsulate sequences of operations that a script can invoke later in its execution.

Every user-defined function or method has a *local scope* that embeds into the script's global scope. The parameters in a function's signature and the variables declared within the function body belong to that function's local scope, and they are *not* directly accessible to a script's outer scope or the scopes of other functions.

The segments below explain a few ways programmers can debug the values from a function's local scope. We will use this script as the starting point for our subsequent examples. It contains a `customMA()` function that returns an exponential moving average whose smoothing parameter varies based on the `source` distance outside the 25th and 75th percentiles over `length` bars:

//@version=5 indicator("Debugging functions demo", "Custom MA", true) //@variable The number of bars in the `customMA()` calculation. int lengthInput = input.int(50, "Length", 2) //@function Calculates a moving average that only responds to values outside the first and third quartiles. //@param source The series of values to process. //@param length The number of bars in the calculation. //@returns The moving average value. customMA(float source, int length) => //@variable The custom moving average. var float result = na // Calculate the 25th and 75th `source` percentiles. float q1 = ta.percentile_linear_interpolation(source, length, 25) float q3 = ta.percentile_linear_interpolation(source, length, 75) // Calculate the range values. float outerRange = math.max(source - q3, q1 - source, 0.0) 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()` result over `lengthInput` bars. float maValue = customMA(close, lengthInput) // Plot the `maValue`. plot(maValue, "Custom MA", color.blue, 3)

Extracting local variables

When a programmer wants to inspect a user-defined function's local variables by plotting its values, coloring the background or chart bars, etc., they must *extract* the values to the *global scope*, as the built-in functions that produce such outputs can only accept global variables and literals.

Since the values returned by a function are available to the scope where a call occurs, one straightforward extraction approach is to have the function return a tuple containing all the values that need inspection.

Here, we've modified the `customMA()` function to return a tuple containing all the function's calculated variables. Now, we can call the function with a *tuple declaration* to make the values available in the global scope and inspect them with plots:

//@version=5 indicator("Extracting local variables with tuples demo", "Custom MA", true) //@variable The number of bars in the `customMA()` calculation. int lengthInput = input.int(50, "Length", 2) //@function Calculates a moving average that only responds to values outside the first and third quartiles. //@param source The series of values to process. //@param length The number of bars in the calculation. //@returns The moving average value. customMA(float source, int length) => //@variable The custom moving average.

var float result = na // Calculate the 25th and 75th `source` percentiles. float q1 = ta.percentile_linear_interpolation(source, length, 25) float q3 = ta.percentile_linear_interpolation(source, length, 75) // Calculate the range values. float outerRange = math.max(source - q3, q1 - source, 0.0) 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, outerRange, totalRange, alpha] // Declare a tuple containing all values returned by `customMA()`. [maValue, q1Debug, q3Debug, outerRangeDebug, totalRangeDebug, alphaDebug] = customMA(close, lengthInput) // Plot the `maValue`. plot(maValue, "Custom MA", color.blue, 3) //@variable Display location for plots with different scale. notOnPane = display.all - display.pane // Display the extracted `q1` and `q3` values in all plot locations. plot(q1Debug, "q1", color.new(color.maroon, 50)) plot(q3Debug, "q3", color.new(color.teal, 50)) // Display the other extracted values in the status line and Data Window to avoid impacting the scale. plot(outerRangeDebug, "outerRange", chart.fg_color, display = notOnPane) plot(totalRangeDebug, "totalRange", chart.fg_color, display = notOnPane) plot(alphaDebug, "alpha", chart.fg_color, display = notOnPane) // Highlight the chart when `alphaDebug` is 0, i.e., when the `maValue` does not change. bgcolor(alphaDebug == 0.0 ? color.new(color.orange, 90) : na, title = "`alpha == 0.0` highlight")

Note that:

- We used `display.all - display.pane` for the plots of the `outerRangeDebug`, `totalRangeDebug`, and `alphaDebug` variables to avoid impacting the chart's scale.
- The script also uses a conditional color to highlight the chart pane's background when `debugAlpha` is 0, indicating the `maValue` does not change.

Another, more *advanced* way to extract the values of a function's local variables is to pass them to a *reference type* variable declared in the global scope.

Function scopes can access global variables for their calculations. While a script cannot directly reassign the values of global variables from within a function's scope, it can update the *elements or properties* of those values if they are reference types, such as arrays, matrices, maps, and user-defined types.

This version declares a `debugData` variable in the global scope that references a map with "string" keys and "float" values. Within the local scope of the `customMA()` function, the script puts *key-value pairs* containing each local variable's name and value into the map. After calling the function, the script plots the stored `debugData` values:

//@version=5 indicator("Extracting local variables with reference types demo", "Custom MA", true) //@variable The number of bars in the `customMA()` calculation. int lengthInput = input.int(50, "Length", 2) //@variable A map with "string" keys and "float" values for debugging the `customMA()`. map<string, float> debugData = map.new<string, float>() //@function Calculates a moving average that only responds to values outside the first and third quartiles. //@param source The series of values to process. //@param length The number of bars in the calculation. //@returns The moving average value. customMA(float source, int length) => //@variable The custom moving average. var float result = na // Calculate the 25th and 75th `source` percentiles. float q1 = ta.percentile_linear_interpolation(source, length, 25), map.put(debugData, "q1", q1) float q3 = ta.percentile_linear_interpolation(source, length, 75), map.put(debugData, "q3", q3) // Calculate the range values. float outerRange = math.max(source - q3, q1 - source, 0.0), map.put(debugData, "outerRange", outerRange) float totalRange = ta.range(source, length), map.put(debugData, "totalRange", totalRange) //@variable Half the ratio of the `outerRange` to the

`totalRange`. float alpha = 0.5 * outerRange / totalRange, map.put(debugData, "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()` result over `lengthInput` bars. float maValue = customMA(close, lengthInput) // Plot the `maValue`. plot(maValue, "Custom MA", color.blue, 3) //@variable Display location for plots with different scale. notOnPane = display.all - display.pane // Display the extracted `q1` and `q3` values in all plot locations. plot(map.get(debugData, "q1"), "q1", color.new(color.maroon, 50)) plot(map.get(debugData, "q3"), "q3", color.new(color.teal, 50)) // Display the other extracted values in the status line and Data Window to avoid impacting the scale. plot(map.get(debugData, "outerRange"), "outerRange", chart.fg_color, display = notOnPane) plot(map.get(debugData, "totalRange"), "totalRange", chart.fg_color, display = notOnPane) plot(map.get(debugData, "alpha"), "alpha", chart.fg_color, display = notOnPane) // Highlight the chart when the extracted `alpha` is 0, i.e., when the `maValue` does not change. bgcolor(map.get(debugData, "alpha") == 0.0 ? color.new(color.orange, 90) : na, title = "`alpha == 0.0` highlight")

Note that:

- We placed each map.put() call on the same line as each variable declaration, separated by a comma, to keep things concise and avoid adding extra lines to the `customMA()` code.
- We used map.get() to retrieve each value for the debug plot() and bgcolor() calls.

Local drawings and logs

Unlike `plot.*()` functions and others that require values accessible to the global scope, scripts can generate drawing objects and Pine Logs from directly within a function, allowing programmers to flexibly debug its local variables *without* extracting values to the outer scope.

In this example, we used labels and Pine Logs to display string representations of the values within the `customMA()` scope. Inside the function, the script calls str.format() to create a formatted string representing the local scope's data, then calls label.new() and log.info() to respectively display the text on the chart in a tooltip and log an “info” message containing the text in the Pine Logs pane:

//@version=5 indicator("Local drawings and logs demo", "Custom MA", true, max_labels_count = 500) //@variable The number of bars in the `customMA()` calculation. int lengthInput = input.int(50, "Length", 2) //@function Calculates a moving average that only responds to values outside the first and third quartiles. //@param source The series of values to process. //@param length The number of bars in the calculation. //@returns The moving average value. customMA(float source, int length) => //@variable The custom moving average. var float result = na // Calculate the 25th and 75th `source` percentiles. float q1 = ta.percentile_linear_interpolation(source, length, 25) float q3 = ta.percentile_linear_interpolation(source, length, 75) // Calculate the range values. float outerRange = math.max(source - q3, q1 - source, 0.0) 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 //@variable A formatted string containing representations of all local variables. string debugText = str.format("\n`customMA()` data\n----------\nsource: {0, number, #.########}\nlength: {1}\nq1: {2, number, #.########} \nq3: {3, number, #.########}\nouterRange: {4, number, #.########}\ntotalRange: {5, number, #.########} \nalpha{6, number, #.########}\nresult: {7, number, #.########}", source, length, q1, q3, outerRange, totalRange, alpha, result) // Draw a label with a tooltip displaying the `debugText`. label.new(bar_index, high, color = color.new(chart.fg_color, 80), tooltip = debugText) // Print an "info" message in the Pine Logs pane when the bar is confirmed. if barstate.isconfirmed log.info(debugText) // Return the `result`. result //@variable The

`customMA()` result over `lengthInput` bars. float maValue = customMA(close, lengthInput) // Plot the `maValue`. plot(maValue, "Custom MA", color.blue, 3)

Note that:

- We included `max_labels_count = 500` in the indicator() function to display labels for the most recent 500 `customMA()` calls.
- The function uses barstate.isconfirmed in an if statement to only call log.info() on *confirmed* bars. It does not log a new message on each realtime tick.

Debugging loops

Loops are structures that repeatedly execute a code block based on a *counter* (for), the contents of a collection (for...in), or a *condition* (while). They allow scripts to perform repetitive tasks without the need for redundant lines of code.

Each loop instance maintains a separate local scope, which all outer scopes cannot access. All variables declared within a loop's scope are specific to that loop, meaning one cannot use them in an outer scope.

As with other structures in Pine, there are numerous possible ways to debug loops. This section explores a few helpful techniques, including extracting local values for plots, inspecting values with drawings, and tracing a loop's execution with Pine Logs.

We will use this script as a starting point for the examples in the following segments. It aggregates the close value's rates of change over 1 - `lookbackInput` bars and accumulates them in a for loop, then divides the result by the `lookbackInput` to calculate a final average value:

//@version=5 indicator("Debugging loops demo", "Aggregate ROC") //@variable The number of 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 `aroc`. aroc += roc // Divide `aroc` by the `lookbackInput`. aroc /= lookbackInput // Plot the `aroc`. plot(aroc, "aroc", color.blue, 3)

Note that:

- The `aroc` is a *global* variable modified within the loop, whereas `pastClose` and `roc` are *local* variables inaccessible to the outer scope.

Inspecting a single iteration

When a programmer needs to focus on a specific loop iteration, there are multiple techniques they can use, most of which entail using a *condition* inside the loop to trigger debugging actions, such as extracting values to outer variables, creating drawings, logging messages, etc.

This example inspects the local `roc` value from a single iteration of the loop in three different ways. When the loop counter's value equals the `debugCounterInput`, the script assigns the `roc` to an `rocDebug` variable from the global scope for plotting, draws a vertical line from 0 to the `roc` value using line.new(), and logs a message in the Pine Logs pane using log.info():

```
//@version=5 indicator("Inspecting a single iteration demo", "Aggregate ROC", max_lines_count = 500)
//@variable The number of bars in the calculation. int lookbackInput = input.int(20, "Lookback", 1)
//@variable The `length` value in the loop's execution where value extraction occurs. int debugCounterInput = input.int(1, "Loop counter value", 1, group = "Debugging") //@variable The `roc` value extracted from the loop. float rocDebug = na //@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 // Trigger debug actions when the `length` equals the `debugCounterInput`. if length == debugCounterInput // Assign `roc` to `rocDebug` so the script can plot its value. rocDebug := roc // Draw a vertical line from 0 to the `roc` at the `bar_index`. line.new(bar_index, 0.0, bar_index, roc, color = color.new(color.gray, 50), width = 4) // Log an "info" message in the Pine Logs pane. log.info("{0}-bar `roc`{1}: {2, number, #.########}", length, barstate.isconfirmed ? " (confirmed)" : "", roc) // Divide `aroc` by the `lookbackInput`. aroc /= lookbackInput // Plot the `aroc`. plot(aroc, "aroc", color.blue, 3) // Plot the `rocDebug`. plot(rocDebug, "Extracted roc", color.new(color.rgb(206, 55, 136), 40), 2)
```

Note that:

- The input.int() call assigned to the `debugCounterInput` includes a `group` argument to distinguish it in the script's settings.
- The log.info() call includes “(confirmed)” in the formatted message whenever barstate.isconfirmed is `true`. Searching this text in the Pine Logs pane will filter out the entries from unconfirmed bars. See the Filtering logs section above.

Inspecting multiple iterations

When inspecting the values from several loop iterations, it's often helpful to utilize collections or strings to gather the results for use in output functions after the loop terminates.

This version demonstrates a few ways to collect and display the loop's values from all iterations. It declares a `logText` string and a `debugValues` array in the global scope. Inside the local scope of the for loop, the script *concatenates* a string representation of the `length` and `roc` with the `logText` and calls array.push() to push the iteration's `roc` value into the `debugValues` array.

After the loop ends, the script plots the first and last value from the `debugValues` array, draws a label with a *tooltip* showing a string representation of the array, and displays the `logText` in the Pine Logs pane upon the bar's confirmation:

```
//@version=5 indicator("Inspecting multiple iterations demo", "Aggregate 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` 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) // Plot the `roc` values from the first
```

and last iteration. plot(array.first(debugValues), "First iteration roc", color.new(color.rgb(166, 84, 233), 50), 2) plot(array.last(debugValues), "Last iteration roc", color.new(color.rgb(115, 86, 218), 50), 2) // Draw a label with a tooltip containing a "string" representation of the `debugValues` array. label.new(bar_index, aroc, color = color.new(color.rgb(206, 55, 136), 70), tooltip = str.tostring(debugValues)) // Log the `logText` in the Pine Logs pane when the bar is confirmed. if barstate.isconfirmed log.info(logText)

Another way to inspect a loop over several iterations is to generate sequential Pine Logs or create/modify drawing objects within the loop's scope to trace its execution pattern with granular detail.

This example uses Pine Logs to trace the execution flow of our script's loop. It generates a new “info” message on each iteration to track the local scope's calculations as the loop progresses on each confirmed bar:

//@version=5 indicator("Inspecting multiple iterations demo", "Aggregate ROC") //@variable The number of bars in the calculation. int lookbackInput = input.int(20, "Lookback", 1) //@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 if barstate.isconfirmed log.info("{0}\nlength (counter): {1}\npastClose: {2, number, #.#####}\n distance to pastClose: {3, number, #.########}\nroc: {4, number, #.########}\n aroc (before division): {5, number, #.########}\n{6}", length == 1 ? "LOOP START" : "", length, pastClose, close - pastClose, roc, aroc, length == lookbackInput ? "LOOP END" : "") // Divide `aroc` by the `lookbackInput`. aroc /= lookbackInput // Plot the `aroc`. plot(aroc, "aroc", color.blue, 3)

Note that:

- When iteratively generating logs or drawings from inside a loop, make it a point to avoid unnecessary clutter and strive for easy navigation. More is not always better for debugging, especially when working within loops.

Tips

Organization and readability

When writing scripts, it's wise to prioritize organized, readable source codes. Code that's organized and easy to read helps streamline the debugging process. Additionally, well-written code is easier to maintain over time.

Here are a few quick tips based on our Style guide and the examples on this page:

- Aim to follow the general script organization recommendations. Organizing scripts using this structure makes things easier to locate and inspect.
- Choose variable and function names that make them easy to *identify* and *understand*. See the Naming conventions section for some examples.
- It's often helpful to temporarily assign important parts of expressions to variables with informative names while debugging. Breaking expressions down into reusable parts helps simplify inspection processes.
- Use *comments* and *annotations* (`//@function`, `//@variable`, etc.) to document your code. Annotations are particularly helpful, as the Pine Editor's autosuggest displays variable and function descriptions in a pop-up when hovering over their identifiers anywhere in the code.

- Remember that *less is more* in many cases. Don't overwhelm yourself with excessive script outputs or unnecessary information while debugging. Keep things simple, and only include as much information as you need.

Speeding up repetitive tasks

There are a few handy techniques we often utilize when debugging our code:

- We use plotchar() or plotshape() to quickly display the results of “int”, “float”, or “bool” variables and expressions in the script's status line and the Data Window.
- We often use bgcolor() to visualize the history of certain conditions on the chart.
- We use a one-line version of our `printLabel()` function from this section to print strings at the end of the chart.
- We use a label.new() call with a `tooltip` argument to display strings in tooltips on successive bars.
- We use the `log.*()` functions to quickly display data with string representations in the Pine Logs pane.

When one establishes their typical debugging processes, it's often helpful to create *keyboard macros* to speed up repetitive tasks and spend less time setting up debug outputs in each code.

The following is a simple *AutoHotkey* script (**not** Pine Script code) that includes hotstrings for the above five techniques. The script generates code snippets by entering a specified character sequence followed by a whitespace:

```
; ————— This is AHK code, not Pine Script. —————

; Specify that hotstrings trigger when they end with space, tab, linefeed, or ca
#Hotstring EndChars `t `n `r

:X:,,show::SendInput, plotchar(%Clipboard%, "%Clipboard%", "", color = chart.fg_
:X:,,highlight::SendInput, bgcolor(bool(%Clipboard%) ? color.new(color.orange, 8
:X:,,print::SendInput, printLabel(string txt, float price = na) => int labelTime
:X:,,tooltip::SendInput, label.new(bar_index, high, color = color.new(chart.fg_c
:X:,,log::SendInput, log.info(str.tostring(%Clipboard%)){Enter}
```

The “,,show” macro generates a plotchar() call that uses the clipboard's contents for the `series` and `title` arguments. Copying a `variableName` variable or the `close > open` expression and typing “,,show” followed by a space will respectively yield:

plotchar(variableName, "variableName", "", color = chart.fg_color, display = display.all - display.pane)
plotchar(close > open, "close > open", "", color = chart.fg_color, display = display.all - display.pane)

The “,,highlight” macro generates a bgcolor() call that highlights the chart pane's background with a conditional color based on the variable or expression copied to the clipboard. For example, copying the barstate.isrealtime variable and typing “,,highlight” followed by a space will yield:

bgcolor(bool(barstate.isrealtime) ? color.new(color.orange, 80) : na, title = "barstate.isrealtime highlight")

The “,,print” macro generates the one-line `printLabel()` function and creates an empty `printLabel()` call with the cursor placed inside it. All you need to do after typing “,,print” followed by a space is enter the text you want to display:

printLabel(string txt, float price = na) => int labelTime = math.max(last_bar_time, chart.right_visible_bar_time), var label result = label.new(labelTime, na, txt, xloc.bar_time, na(price) ? yloc.abovebar : yloc.price, na, label.style_none, chart.fg_color, size.large), label.set_text(result, txt), label.set_y(result, price), result printLabel()

The “,,tooltip” macro generates a label.new() call with a `tooltip` argument that uses str.tostring() on the clipboard's contents. Copying the `variableName` variable and typing “,,tooltip” followed by a space yields:

label.new(bar_index, high, color = color.new(chart.fg_color, 70), tooltip = str.tostring(variableName))

The “,,log” macro generates a log.info() call with a `message` argument that uses str.tostring() on the clipboard's contents to display string representations of variables and expressions in the Pine Logs pane. Copying the expression `bar_index % 2 == 0` and typing “,,log” followed by a space yields:

log.info(str.tostring(bar_index % 2 == 0))

Note that:

- AHK is available for *Windows* devices. Research other software to employ a similar process if your machine uses a different operating system.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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 consecutives 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 triggering the execution of a script 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()
- 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=5 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 = "I", 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=5 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=5 // 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=5 // 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. A subsequent call to the same `request.*()` function with the same arguments is not unique. This limitation applies when using any `request.*()` functions, including:

- request.security()
- request.security_lower_tf()
- request.currency_rate()
- request.dividends()
- request.splits()
- request.earnings()
- request.quandl()
- request.financial()
- request.economic()
- request.seed()

When a script executes two or more identical `request.*()` function calls, only the *first* call 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=5 indicator("`request.*()` call limit demo", dynamic_requests = true) //@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=5 indicator("`request.*()` call limit demo", dynamic_requests = true) //@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 we included `dynamic_requests = true` in the indicator() declaration statement. 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 `request.*()` functions limit requested data via the `calc_bars_count` parameter. If this parameter is not specified, the default is used, which is 100,000 bars. If the plan permits more, this limit can be increased by passing a greater value.

The number of bars on the chart's timeframe covered by 100,000 intrabars varies with the number of intrabars each chart bar contains. For example, requesting data from the 1-minute timeframe while running the script on a 60-minute chart means each chart bar can contain up to 60 intrabars. In this case, the minimum number of chart bars covered by the intrabar request is 1,666, as 100,000 / 60 = 1,666.67. It's important to note, however, that a provider may not report data for *every* minute within an hour. Therefore, such a request may 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=5 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

Before the execution of a script, the compiler translates it into a tokenized *Intermediate Language* (IL). Using an IL allows Pine Script to accommodate larger scripts by applying various memory and performance optimizations. The compiler determines the size of a script based on the *number of tokens* in its IL form, **not** the number of characters or lines in the code viewable in the Pine Editor.

The compiled form of each indicator, strategy, and library script is limited to 80,000 tokens. If a script imports libraries, the total number of tokens from all imported libraries cannot exceed 1 million. There is no way to inspect a script's compiled form, nor its IL token count. As such, you will only know your script exceeds the size limit when the compiler reaches it.

In most cases, a script's compiled size will likely not reach the limit. However, if a compiled script does reach the token limit, the most effective ways to decrease compiled tokens are to reduce repetitive code, encapsulate redundant calls within functions, and utilize libraries when possible.

It's important to note that the compilation process omits any *unused* variables, functions, types, etc. from the final IL form, where “unused” refers to anything that *does not* affect the script's outputs. This optimization prevents superfluous elements in the code from contributing to the script's IL token count.

For example, the script below declares a user-defined type and a user-defined method and defines a sequence of calls using them:

//@version=5 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())

Despite the inclusion of array.new<myType>(), `myType.new()`, and `arr.m()` calls in the script, the only thing actually **output** by the script is `plot(close)`. The rest of the code does not affect the output. Therefore, the compiled form of this script will have the *same* number of tokens as:

//@version=5 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.

Scope count

The total number of scopes in a script, including its global scope and each local scope from the user-defined functions, methods, conditional structures it uses, cannot exceed 550.

It's important to note that the request.security(), request.security_lower_tf(), and request.seed() functions *duplicate* the scopes required to evaluate the values of their `expression` argument in another context. The scopes produced by each call to these `request.*()` functions also count toward the script's scope limit.

For example, suppose we created a script with a global variable that depends on the local scopes of 300 if structures. The total scope count for this script is *301* (1 global scope + 300 local scopes):

//@version=5 indicator("Scopes demo") var x = 0 if close > 0 x += 0 if close > 1 x += 1 // ... Repeat this `if close > n` pattern until `n = 299`. if close > 299 x += 299 plot(x)

Since the total number of scopes is within the limit, it will compile successfully. Now, suppose we call request.security() to evaluate the value of `x` from another context and plot its value as well. In this case, it will effectively *double* the script's scope count since the value of `x` depends on *all* the script's scopes:

//@version=5 indicator("Scopes demo") var x = 0 if close > 0 x += 0 if close > 1 x += 1 // ... Repeat this `if close > n` pattern until `n = 299`. if close > 299 x += 299 plot(x) plot(request.security(syminfo.tickerid, "1D", x)) // Causes compilation error since the scope count is now 602.

We can resolve this issue by encapsulating the if blocks within a user-defined function, as the scope of a function counts as one embedded scope:

//@version=5 indicator("Scopes demo") f() => var x = 0 if close > 0 x += 0 if close > 1 x += 1 // ... Repeat this `if close > n` pattern until `n = 299`. if close > 299 x += 299 plot(f()) plot(request.security(syminfo.tickerid, "1D", f())) // No compilation error.

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. This Help Center page discusses the historical buffer and how to change its size using either the `max_bars_back` parameter or the max_bars_back() function.

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=5 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 maximum of 9000 orders can be placed when backtesting strategies. When using Deep Backtesting, the limit is 1,000,000.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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 v5. To profile a script, add it to the chart, open the source code in the Pine Editor, and select “Enable profiler mode” from the dropdown next to the “Add to chart/Update on chart” 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=5 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.

> Notice!
>
> As with profiling tools for other languages, the Pine Profiler *wraps* a script and its significant code with extra calculations required to collect performance data. As such, a script's resource usage **increases** while profiling, and the Profiler's results reflect the script's runtime with those calculations included. Therefore, one should interpret the results as **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 **will 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=5 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=5 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 expressions. 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=5 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=5 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:

> Notice!When 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 and evaluates them **separately**. Consequently, the Profiler's results for lines with calls to those user-defined functions **will 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=5 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=5 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=5 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=5 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=5 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=5 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=5 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=5 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=5 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=5 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=5 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.

> Notice!The 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 **will 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=5 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.

> Notice!It's often wise to profile each configuration *more than once* to reduce the impact of outliers while assessing how a script's performance varies with its inputs or data. See the 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's average 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=5 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
```

> Notice!Whether profiling a script over a single run or multiple, it's crucial to understand that **results will vary**. While averaging results across several profiled script runs can help derive more stable performance estimates, such estimates are **not** impervious to variance.

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.

> Notice!Before looking for ways to optimize a script, profile it to gauge its performance and identify the **critical code regions** that will 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=5 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=5 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=5 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 calls to the `request.*()` family of functions. However, users should strive to keep their scripts' `request.*()` calls well *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=5 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=5 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=5 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=5 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=5 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=5 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 only executes 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=5 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=5 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 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=5 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:

> Notice!Not all iterative calculations will necessarily have loop-free alternatives. In the case where a script can **only** achieve its results through iteration, programmers can identify possible ways to optimize loops to improve performance. See the next section 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.

> Notice!Before 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=5 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=5 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 and plots the difference between the rescaled.first() and rescaled.avg() values on the chart:

//@version=5 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=5 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=5 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=5 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 explicitly defining a buffer size for a problematic historical reference with max_bars_back(), it's imperative to ensure that the script **will not** use more data than specified later in its executions, as the script will still re-execute on historical bars and try to calculate the buffer if the user-specified size is insufficient.
>
> Another consideration when explicitly defining buffer sizes is that the larger the buffer, the larger the *memory cost*. As such, programmers should aim to keep the explicit buffer length limited to **only** the maximum number of historical values the script will reference and **not more**. For example, defining a 5000-bar buffer when a script only requires 500 historical values will result in an unnecessary waste of memory.

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=5 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=5 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")

> Notice!This process may require trial and error, as determining 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.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Publishing scripts

Programmers who wish to share their Pine scripts with other traders can publish them.

> Notice!If you write scripts for your personal use, there is no need to publish them; you can save them in the Pine Editor and use the “Add to Chart” button to add your script to your chart.

Script visibility and access

When you publish a script, you control its **visibility** and **access**:

- **Visibility** is controlled by choosing to publish **publicly** or **privately**. See How do private ideas and scripts differ from public ones? in the Help Center for more details. Publish publicly when you have written a script you think can be useful to TradingViewers. Public scripts are subject to moderation. To avoid moderation, ensure your publication complies with our House Rules and Script Publishing Rules. Publish privately when you don't want your script visible to all other users, but want to share it with a few friends.
- **Access** determines if users will see your source code, and how they will be able to use your script. There are three access types: *open*, *protected* (reserved to paid accounts) or *invite-only* (reserved to Premium accounts). See What are the different types of published scripts? in the Help Center for more details.

When you publish a script

- The publication's title is determined by the argument used for the `title` parameter in the script's indicator() or strategy() declaration statement. That title is also used when TradingViewers search for script names.
- The name of your script on the chart will be the argument used for the `shorttitle` parameter in the script's indicator() or strategy() declaration statement, or the [title] argument in library().
- Your script must have a description explaining what your script does and how to use it.
- The chart you are using when you publish will become visible in your publication, including any other scripts or drawings on it. Remove unrelated scripts or drawings from your chart before publishing your script.
- Your script's code can later be updated. Each update can include *release notes* which will appear, dated, under your original description.
- Scripts can be liked, shared, commented on or reported by other users.
- Your published scripts appear under the “SCRIPTS” tab of your user profile.
- A *script widget* and a *script page* are created for your script. The script widget is your script's placeholder showing in script feeds on the platform. It contains your script's title, chart and the first few lines of your description. When users click on your script **widget**, the script's **page** opens. It contains all the information relating to your script.

Visibility

Public

When you publish a public script:

- Your script will be inluded in our Community Scripts where it becomes visible to the millions of TradingViewers on all internationalized versions of the site.
- Your publication must comply with House Rules and Script Publishing Rules.
- If your script is an invite-only script, you must comply with our Vendor Requirements.
- It becomes accessible through the search functions for scripts.
- You will not be able to edit your original description or its title, nor change its public/private visibility, nor its access type (open-source, protected, invite-only).
- You will not be able to delete your publication.

Private

When you publish a private script:

- It will not be visible to other users unless you share its url with them.
- It is visible to you from your user profile's “SCRIPTS” tab.
- Private scripts are identifiable by the “X” and “lock” icons in the top-right of their widget. The “X” is used to delete it.
- It is not moderated, unless you sell access to it or make it available publicly, as it is then no longer “private”.
- You can update its original description and title.
- You cannot link to or mentioned it from any public TradingView content (ideas, script descriptions, comments, etc.).
- It is not accessible through the search functions for scripts.

Access

Public or private scripts can be published using one of three access types: open, protected or invite-only. The access type you can select from will vary with the type of account you hold.

Open

The Pine Script® code of scripts published **open** is visible to all users. Open-source scripts on TradingView use the Mozilla license by default, but you may choose any license you want. You can find information on licensing at GitHub.

Protected

The code of **protected** scripts is hidden from view and no one but its author can access it. While the script's code is not accessible, protected scripts can be used freely by any user. Only accounts with paid plans may publish public protected scripts.

Invite-only

The **invite-only** access type protects both the script's code and its use. The publisher of an invite-only script must explicitly grant access to individual users. Invite-only scripts are mostly used by script vendors providing paid access to their scripts. Only Premium accounts can publish invite-only scripts, and they must comply with our Vendor Requirements.

TradingView does not benefit from script sales. Transactions concerning invite-only scripts are strictly between users and vendors; they do not involve TradingView.

Public invite-only scripts are the only scripts for which vendors are allowed to ask for payment on TradingView.

On their invite-only script's page, authors will see a “Manage Access” button. The “Manage Access” window allows authors to control who has access to their script.

Preparing a publication

1. Even if you intend to publish publicly, it is always best to start with a private publication because you can use it to validate what your final publication will look like. You can edit the title, description, code or chart of private publications, and contrary to public scripts, you can delete private scripts when you don't need them anymore, so they are the perfect way to practice before sharing a script publicly. You can read more about preparing script descriptions in the How We Write and Format Script Descriptions publication.

2. Prepare your chart. Load your script on the chart and remove other scripts or drawings that won't help users understand your script. Your script's plots should be easy to identify on the chart that will be published with it.

3. Load your code in the Pine Editor if it isn't already. In the Editor, click the “Publish Script” button:

4. A popup appears to remind you that if you publish publicly, it's important that your publication comply with House Rules. Once you're through the popup, place your description in the field below the script's title. The default title proposed for your publication is the `title` field from your script's code. It is always best to use that title; it makes it easier for users to search for your script if it is public. Select the visibility of your publication. We want to publish a private publication, so we check the “Private Script” checkbox at the bottom-right of the “Publish Script” window:

5. Select the access type you want for your script: Open, Protected or Invite-only. We have selected “Open” for open-source.

6. Select the appropriate categories for your script (at least one is mandatory) and enter optional custom tags.

7. Click the “Publish Private Script” button in the lower-right of the window. When the publication is complete, your published script's page will appear. You are done! You can confirm the publication by going to your User Profile and viewing your “SCRIPTS” tab. From there, you will be able to open your script's page and edit your private publication by using the “Edit” button in the top-right of your script's page. Note that you can also update private publications, just like you can public ones. If you want to share your private publication with a friend, privately send her the url from your script's page. Remember you are not allowed to share links to private publications in public TradingView content.

Publishing a script

Whether you intend to publish privately or publicly, first follow the steps in the previous section. If you intend to publish privately, you will be done. If you intend to publish publicly and are satisfied with the preparatory

process of validating your private publication, follow the same steps as above but do not check the “Private Script” checkbox and click the “Publish Public Script” button at the bottom-right of the “Publish Script” page.

When you publish a new public script, you have a 15-minute window to make changes to your description or delete the publication. After that you will no longer be able to change your publication's title, description, visiblity or access type. If you make an error, send a message to the PineCoders moderator account; they moderate script publications and will help.

Updating a publication

You can update both public or private script publications. When you update a script, its code must be different than the previously published version's code. You can add release notes with your update. They will appear after your script's original description in the script's page.

By default, the chart used when you update will replace the previous chart in your script's page. You can choose not to update your script page's chart, however. Note that while you can update the chart displayed in the script's page, the chart from the script's widget will not update.

In the same way you can validate a public publication by first publishing a private script, you can also validate an update on a private publication before proceeding with it on your public one. The process of updating a published script is the same for public and private scripts.

If you intend to update both the code and chart of your published script, prepare your chart the same way you would for a new publication. In the following example, we will **not** be updating the publication's chart:

1. As you would for a new publication, load your script in the Editor and click the “Publish Script” button.
2. Once in the “Publish Script” window, select the “Update Existing Script” button. Then select the script to update from the “Choose script” dropdown menu:
3. Enter your release notes in the text field. The differences in your code are highlighted below your release notes.
4. We do not want to update the publication's chart, so we check the “Don't update the chart” checkbox:
5. Click the “Publish New Version” button. You're done.

Notice!This is documentation for v5, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

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 v5, use:

//@version=5

<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=5 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 here 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 easier to read. Line wraps are defined by using an indentation level that is not a multiple of four, as four spaces or a tab are used to define local blocks. Here we use two spaces:

plot(series = close, title = "Close", color = color.blue, show_last = 10)

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.