Pine Script v4 User Manual

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

42 pages of text and code. The 194 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. Background coloring with bgcolor (annotations/background-coloring-with-bgcolor)
2. Barcoloring a series with barcolor (annotations/barcoloring-a-series-with-barcolor)
3. Custom OHLC bars and candles (annotations/custom-ohlc-bars-and-candles)
4. Filling the background with fill (annotations/filling-in-the-background-with-fill)
5. Plot annotation (annotations/plot-annotation)
6. Plotting shapes, chars and arrows (annotations/plotting-shapes-chars-and-arrows)
7. Price levels, hline (annotations/price-levels-hline)
8. Script inputs (annotations/script-inputs)
9. Study annotation (annotations/study-annotation)
10. HOWTOs (appendix/howtos)
11. Pine compilation and execution errors (appendix/pine-compilation-errors)
12. Pine version 3 migration guide (appendix/pine-version-3-migration-guide)
13. Debugging (debugging)
14. Alerts (essential/alerts)
15. Arrays (essential/arrays)
16. Bar states. Built-in variables barstate.* (essential/bar-states-built-in-variables-barstate)
17. Colors (essential/colors)
18. Context switching and the security function (essential/context-switching-the-security-function)
19. Drawings (essential/drawings)
20. Extended and regular sessions (essential/extended-and-regular-sessions)
21. Indicator repainting (essential/indicator-repainting)
22. Non-standard chart types data (essential/non-standard-chart-types-data)
23. Session and time information (essential/sessions-and-time-functions)
24. Strategies (essential/strategies)
25. Tables (essential/tables)
26. Comments (language/comments)
27. Declaring functions (language/declaring-functions)
28. Execution model (language/execution-model)
29. Expressions, declarations and statements (language/expressions-declarations-and-statements)
30. Functions and annotations (language/functions-and-annotations)
31. Identifiers (language/identifiers)
32. Line wrapping (language/line-wrapping)
33. Operators (language/operators)

34. Structure of the script (language/structure-of-the-script)
35. Type system (language/type-system)
36. Versions (language/versions)
37. Public Library (public-library)
38. Publishing scripts (publishing-scripts)
39. Quickstart guide (quickstart-guide)
40. Release notes (release-notes)
41. Introduction (welcome)
42. Where can I get more information? (where-can-i-get-more-information)

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Background coloring with bgcolor

The bgcolor annotation 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 color used in `bgcolor` can be calculated in an expression and an optional `transp` parameter (0--100, default 90) allows control over the transparency level.

Here is a script that colors the background of trading sessions (try it on EURUSD, at *30 minutes* resolution):

//@version=4 study("bgcolor example", overlay=true) timeinrange(res, sess) => time(res, sess) != 0 premarket = #0050FF regular = #0000FF postmarket = #5000FF color notrading = na sessioncolor = timeinrange("30", "0400-0930") ? premarket : timeinrange("30", "0930-1600") ? regular : timeinrange("30", "1600-2000") ? postmarket : notrading bgcolor(sessioncolor, transp=75)

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Barcoloring a series with barcolor

The barcolor annotation function lets you specify the color of bars. The coloring can be conditional. The following script renders *inside* and *outside* bars in different colors:

//@version=4 study("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)

As you can see, the `na` value leaves bars as is.

`barcolor` works whether the script is running in `overlay=true` mode or not.

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Custom OHLC bars and candles

You may create your own custom *bars* and *candles* in Pine scripts by using the plotbar and plotcandle annotation functions:

//@version=4 study("Example 1") plotbar(open, high, low, close)

Example 1 simply replicates bars of the current symbol. To color them green or red, we can use the following code:

//@version=4 study("Example 2") palette = close >= open ? color.lime : color.red plotbar(open, high, low, close, color=palette)

Example 2 illustrates using the `color` argument, which can be given constant values such as `red`, `lime`, `"#FF9090"`, as well as expressions that calculate colors conditionally at runtime (see the `palette` variable in the example above).

The `plotcandle` annotation function is similar to `plotbar`, but it plots candles instead of bars and has an optional argument: `wickcolor`.

Both `plotbar` and `plotcandle` need four series as the arguments that will be used for new bar/candle OHLC prices. If one of the arguments for a bar has a `na` value, then the bar is not plotted. Example:

//@version=4 study("Example 3") c = close > open ? na : close plotcandle(open, high, low, c)

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:

//@version=4 study("Example 4") len = input(9) smooth(x) => sma(x, len) o = smooth(open) h = smooth(high) l = smooth(low) c = smooth(close) plotcandle(o, h, l, c)

You may find it useful to plot OHLC values taken from a higher timeframe. You can, for example, plot daily bars on a *60 minutes* chart:

// NOTE: add this script on intraday chart //@version=4 study("Example 5") higherRes = input("D", type=input.resolution) is_newbar(res) => t = time(res) not na(t) and (na(t[1]) or t > t[1]) o = security(syminfo.tickerid, higherRes, open) h = security(syminfo.tickerid, higherRes, high) l = security(syminfo.tickerid, higherRes, low) c = security(syminfo.tickerid, higherRes, close) plotbar(is_newbar(higherRes) ? o : na, h, l, c, color=c >= o ? color.lime : color.red)

The `plotbar` and `plotcandle` annotation functions also have a `title` argument, so users can distinguish them in the *Style* tab of the *Settings* dialog box.

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Filling the background with fill

The fill annotation function lets you color the background between two plotted series or two horizontal lines created with `hline`. The following example illustrates how `fill` works:

//@version=4 study("fill Example") p1 = plot(sin(high)) p2 = plot(cos(low)) p3 = plot(sin(close)) fill(p1, p3, color=color.red) fill(p2, p3, color=color.blue) h1 = hline(0) h2 = hline(1.0) h3 = hline(0.5) h4 = hline(1.5) fill(h1, h2, color=color.yellow) fill(h3, h4, color=color.lime)

Please note that the `fill` annotation cannot be used with `plot` and `hline` arguments simultaneously. As a workaround you may replace `hline` calls with a `plot(<constant number>)`. For example:

//@version=4 study("Fill example 2") src = close, len = 10 ma = sma(src, len) osc = 100 * (ma - src) / ma p = plot(osc) // NOTE: fill(p, hline(0)) wouldn't work, instead use this: fill(p, plot(0))

You can set the fill color by using constants like `color=color.red` or `color=#ff001a`, as well as complex expressions like `color = close >= open ? color.green : color.red`. Example:

//@version=4 study(title="Colored fill") line1=sma(close, 5) line2=sma(close, 20) p1 = plot(line1) p2 = plot(line2) fill(p1, p2, color = line1 > line2 ? color.green : color.red)

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Plot annotation

The plot annotation function has one mandatory parameter: a value of *series* type, which it displays as a line. A basic call looks like this:

plot(close)

Pine's automatic type conversions makes it possible to also use any numeric value as an argument. For example:

plot(125.2)

In this case, the value 125.2 will automatically be converted to a series type value which will be the same number on every bar. The plot will be represented as a horizontal line.

The `plot` annotation has many optional parameters, in particular those which set the line's display style: `style`, `color`, `linewidth`, `transparency`, and others.

The value of the `color` parameter can be defined in different ways. If it is a color constant, for example `color.red`, then the whole line will be plotted using a *red* color:

plot(close, color=color.red)

The value of `color` can also be an expression of a *series* type of color values. This series of colors will be used to color the rendered line. For example:

c = close >= open ? color.lime : color.red plot(close, color = c)

The `offset` parameter specifies the shift used when the line is plotted (negative values shift to the left while positive values shift to the right)[1]. For example:

//@version=4 study("My Script 12", overlay=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).

Footnotes

1. In Pine there is a built-in offset function which shifts the values of a series to the right while discarding 'out of range' values. The advantage of the `offset` function lies in the fact that its result can be used in other expressions to execute complex calculations. In the case of `plot` function's `offset` parameter, the shift is only cosmetic; the actual values in the series are not moved. ↩

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Plotting shapes, chars and arrows

There are situations where you need to mark or highlight specific bars on a chart, and where a usual line plot using `plot` may not be optimal. Although it may be possible to do so using `plot` with `style=plot.style_circles` or `style=plot.style_cross`, it is often easier to use the `plotshape`, `plotchar` and `plotarrow` annotation functions.

plotshape

The plotshape function can display a variety of shapes. The script below will draw an "X" above all green bars:

study('plotshape example 1', overlay=true) data = close >= open plotshape(data, style=shape.xcross)

The first parameter, `data`, is used as a series of logical values. The crosses are drawn on each true value. Nothing is drawn on false or `na` values. You may use a series of logical values or numbers as the first argument of `plotshape`. A 0 or `na` is considered a false value, any other value is considered true.

By changing the value of the `style` parameter, it is possible to vary the shape. The available styles are:

Shape Name	Shape	Shape with Text
`shape.xcross`		
`shape.cross`		
`shape.circle`		
`shape.triangleup`		
`shape.triangledown`		
`shape.flag`		
`shape.arrowup`		
`shape.arrowdown`		
`shape.square`		
`shape.diamond`		
`shape.labelup`		
`shape.labeldown`		

The default position where `plotshape` draws shapes is above bars. To use another position, use the `location` parameter. The following script draws a green `shape.triangleup` *above* the green bars and a red `shape.triangledown` *below* the red bars:

//@version=4 study('plotshape example 2', overlay=true) data = close >= open plotshape(data, style=shape.triangleup, location=location.abovebar, color=color.green) plotshape(not data, style=shape.triangledown, location=location.belowbar, color=color.red)

Possible values for the `location` parameter are:

- `location.abovebar` - above a bar.
- `location.belowbar` - below a bar.

- `location.top` - top of the script's *y* space.
- `location.bottom` - bottom of the script's *y* space.
- `location.absolute` - any position in the *y* space.

`location.absolute` can be used when the shapes need to be positioned more precisely in the script's *y* space. The first parameter of the function is then used as a *y* coordinate.

'plotshape example 2' illustrates how the shape's color can be defined using the `color` parameter and expressions which will calculate the shape's color depending on conditions at runtime. For example:

//@version=4 study('plotshape example 3', overlay=true) data = close >= open plotshape(true, style=shape.flag, color=data ? color.green : color.red)

In the given example, the first parameter of the function `plotshape` is `true`, which means that a shape is displayed at every bar. The color is calculated by the `color=data ? color.green : color.red` expression.

Other features of the `plotshape` function:

- Set the name of a displayed series of data using the `title` parameter.
- Shift a series of shapes to the left/right using the `offset` parameter.
- Set the transparency of shapes with the `transp` parameter.
- Use the `text` parameter to display a short text above/below the shape. You may use `\n` to separate text lines.

plotchar

The main difference between `plotshape` and plotchar is that with `plotchar`, the shape is an ASCII or Unicode symbol (provided it's supported by the TradingView standard font) defined with the `char` parameter. For example:

study('plotchar example', overlay=true) data = close >= open plotchar(data, char='a')

The default character is ★ (U+2605, the "BLACK STAR" character). It's possible to use any letter or digit and many symbols, for example: ❤, ☀, €, ⚑, ❄, ◆, ⬆, ⬇. The supported character codes are those of the *Trebuchet MS* font family.

The next example uses the "SNOWFLAKE" (❄, U+2744) character:

study('plotchar example', overlay=true) data = close >= open plotchar(data, char='❄')

Like `plotshape`, the `plotchar` function allows you to:

- Set a shape's color with a constant or complex arithmetic expression using the `color` parameter.
- Set a shape's location with the `location` parameter.
- Set the name of a displayed series of data using the `title` parameter.
- Shift a series of shapes left/right using the `offset` parameter.
- Set the transparency of shapes using the `transp` parameter.
- Use the `text` parameter to display a short text above/below the shape. You may use `\n` to separate text lines.

plotarrow

The plotarrow annotation function allows for up/down arrows to be displayed on the chart. The arrow length is not the same on each bar and is calculated from the first parameter's value.

The first `series` parameter of the `plotarrow` function is used to place arrows on the chart using the following logic:

- If the `series` value on the current bar is greater than 0, then an *up arrow* will be drawn, the length of which will be proportional to the relative value of the series on that bar in relation to other series values.
- If the `series` value on the current bar is less than 0, then a *down arrow* will be drawn, the length of which will be proportional to the relative value of the series on that bar in relation to other series values.
- If the `series` value on the current bar is equal to 0 or `na` then the arrow is not displayed.

Here is a simple script that illustrates how the `plotarrow` function works:

study("plotarrow example", overlay=true) codiff = close - open plotarrow(codiff, colorup=teal, colordown=orange, transp=40)

As you can see, the greater the relative value of the `close - open` difference, the longer the arrow. If `close - open` is greater than zero, then an up arrow is rendered. When `close - open` is less than zero, a down arrow is rendered.

In another example, we'll start from the *Chaikin Oscillator* script in the built-in scripts and display it as an overlay above a chart using arrows:

study("Chaikin Oscillator Arrows", overlay=true) short = input(3,minval=1), long = input(10,minval=1) osc = ema(accdist, short) - ema(accdist, long) plotarrow(osc)

This screenshot shows the original *Chaikin Oscillator* alongside the script for better understanding.

As was stated earlier, the height of the arrow is proportional to the relative value of the first series parameter. The maximum and minimum possible sizes for the arrows (in pixels) can be set using the `minheight` and `maxheight` parameters.

Additionally, the `plotarrow` function allows you to:

- Set the name of a displayed series of data using the `title` parameter.
- Set the color of an *up arrow* using the `colorup` parameter.
- Set the color of a *down arrow* using the `colordown` parameter.
- Shift a series of arrows left/right using the `offset` parameter.
- Set the transparency of arrows with the `transp` parameter.

It's important to note that the `colorup` and `colordown` parameters must receive a constant value of the *color* type. Using expressions for determining color (as is done in `plot`, `plotshape` or `plotchar`) is not allowed.

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Price levels, hline

The hline() annotation function renders a horizontal line at a given level. For example:

//@version=4 study(title="Chaikin Oscillator", shorttitle="Chaikin Osc") short = input(3, minval=1) long = input(10, minval=1) osc = ema(accdist, short) - ema(accdist, long) plot(osc, color=color.red) hline(0, title="Zero", color=color.gray, linestyle=hline.style_dashed)

A *number* must be the first argument of `hline`. Values of *series* type are forbidden. It's possible to create a few horizontal lines with the help of `hline` and fill the background between them with a translucent color using fill().

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Script inputs

The input annotation function makes it possible for script users to modify selected values which the script can then use in its calculation or logic, without the need to modify the script's code.

Specific widgets are supplied in the *Settings/Inputs* dialog box for each type of input. A description of the value as well as minimum/maximum values and a step increment can also be defined. If the type of the input variable cannot be inferred at compile time, it can be explicitly defined using the `type` parameter.

Pine supports the following types of input:

- input.bool,
- input.color,
- input.integer,
- input.float,
- input.string,
- input.symbol,
- input.resolution,
- input.session,
- input.source,
- input.time.

The following examples show how to create each type of input and what its widget looks like.

Boolean input

b = input(title="On/Off", type=input.bool, defval=true) plot(b ? open : na)

Color input

c = input(title="Color", type=input.color, defval=color.red) plot(close, color=c)

Integer input

i = input(title="Offset", type=input.integer, defval=7, minval=-10, maxval=10) plot(offset(close, i))

Float input

f = input(title="Angle", type=input.float, defval=-0.5, minval=-3.14, maxval=3.14, step=0.2) plot(sin(f) > 0 ? close : open)

Symbol and resolution inputs

sym = input(title="Symbol", type=input.symbol, defval="SPY") res = input(title="Resolution", type=input.resolution, defval="60") plot(close, color=color.red) plot(security(sym, res, close), color=color.green)

The symbol input widget has a built-in *symbol search* which activates automatically when the ticker's first characters are typed.

Session input

s = input(title="Session", type=input.session, defval="24x7") plot(time(timeframe.period, s))

Source input

src = input(title="Source", type=input.source, defval=close) ma = sma(src, 9) plot(ma)

Time input

date = input(title="Date", type=input.time, defval=timestamp("20 Feb 2020 00:00 +0300")) plot(date)

Options parameter

The `options` parameter is useful to provide users with a list of constant values they can choose from using a dropdown menu.

choice = input(title="Choice", defval="A", options=["A", "B"]) plot(choice == "A" ? close : choice == "B" ? open : na)

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Study annotation

Every *study* (also known as an *indicator*)[1] script must contain one call of the study annotation function, which has the following signature:

study(title, shorttitle, overlay, format, precision)

The `study` annotation determines the indicator's general properties. Only the `title` parameter is mandatory. It defines the name of the indicator. This name will be used in the *Indicators* dialog box and is independent of the name used to save the script in your Personal Library.

`shorttitle` is the short name of the indicator displayed on the chart, if it must be different than the value of `title`.

`overlay` is a logical argument. If it is true then the study will be added as an overlay on top of the main chart. If it is false then it will be added in a separate pane. False is the default setting. Note that if you change the parameter's value in a script that is already on a chart, you need to use the *Add to Chart* button to apply the change.

`format` defines the type of formatting used for study values appearing on the price axis, in indicator values or in the Data Window. Possible values are: `format.inherit`, `format.price` and `format.volume`. The default is `format.inherit`, which uses the format settings from the chart, unless `precision=` is also used, in which case it will override the effect of `format.inherit`. When `format.price` is used, the default precision will be "2", unless one is specified using `precision=`. When `format.volume` is used, the format is equivalent to `precision=0` used in earlier versions of Pine, where "5183" becomes "5.183K".

`precision` is the number of digits after the floating point used to format study values. It must be a non-negative integer and not greater than 16. If omitted, then formatting from the parent series on the chart will be used. If the format is `format.inherit` and the `precision` parameter is used with a value, then the study will not inherit formatting from the chart's settings and the value specified will be used instead, as if `format=format.price` had been used.

Footnotes

1. Pine also has a strategy annotation function which is used to create a backtesting strategy rather than a study (or indicator). ↩

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

HOWTOs

Get real OHLC price on a Heikin Ashi chart

Suppose, we have a Heikin Ashi chart (or Renko, Kagi, PriceBreak etc) and we've added a pine script on it:

//@version=4 study("Visible OHLC", overlay=true) c = close plot(c)

You may see that variable `c` is a Heikin Ashi *close* price which is not the same as real OHLC price. Because `close` built-in variable is always a value that corresponds to a visible bar (or candle) on the chart.

So, how do we get the real OHLC prices in Pine Script code, if current chart type is non-standard? We should use `security` function in combination with `tickerid` function. Here is an example:

//@version=4 study("Real OHLC", overlay=true) t = tickerid(syminfo.prefix, syminfo.ticker) realC = security(t, timeframe.period, close) plot(realC)

In a similar way we may get other OHLC prices: *open*, *high* and *low*.

Get non-standard OHLC values on a standard chart

Backtesting on non-standard chart types (e.g. Heikin Ashi or Renko) is not recommended because the bars on these kinds of charts do not represent real price movement that you would encounter while trading. If you want your strategy to enter and exit on real prices but still use Heikin Ashi-based signals, you can use the same method to get Heikin Ashi values on a regular candlestick chart:

//@version=4 strategy("BarUpDn Strategy", overlay=true, default_qty_type = strategy.percent_of_equity, default_qty_value = 10) maxIdLossPcnt = input(1, "Max Intraday Loss(%)", type=input.float) strategy.risk.max_intraday_loss(maxIdLossPcnt, strategy.percent_of_equity) needTrade() => close > open and open > close[1] ? 1 : close < open and open < close[1] ? -1 : 0 trade = security(heikinashi(syminfo.tickerid), timeframe.period, needTrade()) if (trade == 1) strategy.entry("BarUp", strategy.long) if (trade == -1) strategy.entry("BarDn", strategy.short)

Plot arrows on the chart

You may use plotshape with style `shape.arrowup` and `shape.arrowdown`:

//@version=4 study('Ex 1', overlay=true) data = close >= open plotshape(data, color=color.lime, style=shape.arrowup, text="Buy") plotshape(not data, color=color.red, style=shape.arrowdown, text="Sell")

You may use `plotchar` function with any unicode character:

//@version=4 study('buy/sell arrows', overlay=true) data = close >= open plotchar(data, char='↓', color=color.lime, text="Buy") plotchar(data, char='↑', location=location.belowbar, color=color.red, text="Sell")

Plot a dynamic horizontal line

There is function `hline` in pine. But it is now limited to only plot constant value. Here is a Pine Script with workaround to plot changing hline:

//@version=4 study("Horizontal line", overlay=true) plot(close[10], trackprice=true, offset=-9999) // trackprice=true plots horizontal line on close[10] // offset=-9999 hides the plot plot(close, color=#FFFFFFFF) // forces to show study

Plot a vertical line on condition

//@version=4 study("Vertical line", overlay=true, scale=scale.none) // scale.none means do not resize the chart to fit this plot // if the bar being evaluated is the last baron the chart (the most recent bar), then cond is true cond = barstate.islast // when cond is true, plot a histogram with a line with height value of 100,000,000,000,000,000,000.00 // (10 to the power of 20) // when cond is false, plot no numeric value (nothing is plotted) // use the style of histogram, a vertical bar plot(cond ? 10e20 : na, style=plot.style_histogram)

Access the previous value

//@version=4 //... s = 0.0 s := nz(s[1]) // Accessing previous values if (condition) s := s + 1

Get a 5-days high

Lookback 5 days from the current bar, find the highest bar, plot a star character at that price level above the current bar

//@version=4 study("Range Analysis", overlay=true) // find which bar is 5 days away from the current time milliseconds_in_5days = 1000 * 60 * 60 * 24 * 5 // millisecs * secs * min * hours * days // plot(milliseconds_in_5days, title="ms in 5d", style=circles) //debug // subtract timestamp of the bar being examined from the current time // if value is less than 5 days ago, set variable "leftborder" as true // this is set true at the bar being examined as the left border of the 5 days lookback window range leftborder = timenow - time < milliseconds_in_5days // true or na when false // plot(leftborder ? 1 : na, title="bar within leftborder") //debug // plot(time, title="bartime") //debug // plot(timenow - time, title="timenow minus bartime") //debug // treat the last bar (most recent bar) as the right edge of the lookback window range rightborder = barstate.islast // initialize variable "max" as na max = float(na) // if bar being examined is not within the lookback window range (i.e., leftborder = false) // change the variable "max" to be na // else, test if value of "max" stored in the previous bar is na // (bcuz first bar being examined in the lookback window will not have a previous value), // if it is na, use the high of the current bar, // else, use the value of "max" stored in the previous bar max := not leftborder ? na : na(max[1]) ? high : max[1] // plot(max ? max : na, title="max b4 compare") // debug // compare high of current bar being examined with previous bar's high // if curr bar high is higher than the max bar high in the lookback window range if high > max // we have a new high max := high // change variable "max" to use current bar's high value max // else keep the previous value of max as the high bar within this lookback window range // plot(max ? max : na, title="max after compare") //debug // if examining the last bar (newest bar, rightborder is true) // set variable "val" to the previous value of series variable "max" // else set to na so nothing is plotted val = rightborder ? max[1] : na // if val is true (a number, not na) // plot character // since no character is specified, a "star" will be plotted // location.absolute uses the value of val as the y axis value // the x axis location will be the last bar (newest bar) plotchar(val, size=size.normal, location=location.absolute) // fill the background of the 5 days lookback window range with aqua color bgcolor(leftborder and not rightborder ? color.aqua : na, transp=70)

Count bars in a dataset

Get a count of all the bars in the loaded dataset. Might be useful for calculating flexible lookback periods based on number of bars.

//@version=4 study("Bar Count", overlay=true, scale=scale.none) plot(bar_index + 1, style=plot.style_histogram)

Enumerate bars in a day

//@version=4 study("My Script", overlay=true, scale=scale.none) is_new_day() => d = dayofweek na(d[1]) or d != d[1] plot(barssince(is_new_day()), style=plot.style_cross)

Find the highest and lowest values for the entire dataset

//@version=4 study("My Script") biggest(series) => max = 0.0 max := nz(max[1], series) if series > max max := series max smallest(series) => min = 0.0 min := nz(min[1], series) if series < min min := series min plot(biggest(close), color=color.green) plot(smallest(close), color=color.red)

Query the last non-na value

You can use the script below to avoid gaps in a series:

//@version=4 study("My Script") series = close >= open ? close : na vw = valuewhen(not na(series), series, 0) plot(series, style=plot.style_linebr, color=color.red) // series has na values plot(vw) // all na values are replaced with the last non-empty value

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Pine compilation and execution errors

Script requesting too many securities

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

//@version=4 study("Securities count") a = security(syminfo.tickerid, '42', close) // (1) first unique security call b = 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 security(syminfo.tickerid, p, close) plot(sym('D')) // (2) one indirect call to security plot(sym('W')) // (3) another one indirect call to security 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 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=4 study("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=4 study("My Script") plot(1)

Loop is too long (> 200 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 200 milliseconds to compute on each of bars, it would have result in more than 16 minutes of loading.

//@version=4 study("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=4 study("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 cannot determine the referencing length of a series. Try using max_bars_back in the study or strategy function

The error appears in cases where Pine 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 to inspect the use of series in branches of conditional statements (`if`, `iff` or `?`), and Pine cannot automatically detect how far back the series is referenced. Here is an example of a script causing this problem:

//@version=4 study("Requires max_bars_back") test = 0.0 if bar_index > 1000 test := roc(close, 20) plot(test)

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

//@version=4 study("Requires max_bars_back", max_bars_back=20) test = 0.0 if bar_index > 1000 test := 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=4 study("My Script") test = 0.0 roc20 = 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 Pine v4 `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=4 study("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=4 study("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.

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Pine version 3 migration guide

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 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 version 3) or `barmerge.lookahead_on` (which is the default for Pine version 2).

Self-referenced variables are removed

Pine version 2 pieces of code, containing a self-referencing variable:

//@version=2 //... s = nz(s[1]) + close

Compiling this piece of code with Pine 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 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 version 3 this will give you an error `Undeclared identifier 'f'`. This example should be rewritten in Pine 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 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, 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 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 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 v4, 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 charts allows for efficient and interactive debugging of Pine code. Once a Pine programmer understands the most appropriate technique to use in each situation, he will be able to debug scripts quickly and thoroughly. This page demonstrates the most useful techniques to debug Pine code.

If you are not yet familiar with Pine's execution model, it is important that you read the execution model page of this User Manual so you understand how your debugging code will behave in the Pine environment.

The lay of the land

Values plotted by Pine scripts can be displayed in four distinct places:

1. Next to the script's name (controlled by the *Indicator Values* checkbox in the *Chart settings/Status Line* tab).
2. In the script's pane, whether your script is a chart overlay or in a separate pane.
3. In the scale (only displays the last bar's value and is controlled by the *Indicator Last Value Label* checkbox in the *Chart settings/Scale* tab).
4. In the Data Window (which you can bring up using the fourth icon down, to the right of your chart).

Note the following in the preceding screenshot:

- The chart's cursor is on the dataset's first bar, where bar_index is zero. That value is reflected next to the indicator's name and in the Data Window. **Moving your cursor on other bars would update those values so they always represent the value of the plot on that bar.** This is a good way to inspect the value of a variable as the script's execution progresses from bar to bar.
- The `title` argument of our plot() call, "Bar Index", is used as the value's legend in the Data Window.
- The precision of the values displayed in the Data Window is dependent on the chart symbol's tick value. You can modify it in two ways:
    - By changing the value of the *Precision* field in the script's *Settings/Style* tab. You can obtain up to eight digits of precision using this method.
    - By using the `precision` parameter in your script's study() or strategy() declaration statement. This method allows specifying up to 16 digits precision.
- The plot() call in our script plots the value of bar_index in the indicator's pane, which shows the increasing value of the variable.
- The scale of the script's pane is automatically sized to accommodate the smallest and largest values plotted by all plot() calls in the script.

Displaying numeric values

When the script's scale is unimportant

The script in the preceding screenshot used the simplest way to inspect numerical values: a plot() call, which plots a line corresponding to the variable's value in the script's display area. Our example script plotted the value of the bar_index built-in variable, which contains the bar's number, a value beginning at zero on the

dataset's first bar and increased by one on each subsequent bar. We used a plot() call to plot the variable to inspect because our script was not plotting anything else; we were not preoccupied with preserving the scale for other plots to continue to plot normally. This is the script we used:

//@version=4 study("Plot `bar_index`") plot(bar_index, "Bar Index")

When the script's scale must be preserved

Plotting values in the script's display area is not always possible. When we already have other plots going on and adding debugging plots of variables whose values fall outside the script's plotting boundaries would make the plots unreadable, another technique must be used to inspect values if we want to preserve the scale of the other plots.

Suppose we want to continue inspecting the value of bar_index, but this time in a script where we are also plotting RSI:

//@version=4 study("Plot RSI and `bar_index`") r = rsi(close, 20) plot(r, "RSI", color.black) plot(bar_index, "Bar Index")

Running the script on a dataset containing a large number of bars yields the following display:

where:

1. The RSI line in black is flat because it varies between zero and 100, but the indicator's pane is scaled to show the maximum value of bar_index, which is `25692.0000`.
2. The value of bar_index on the bar the cursor is on is displayed next to the indicator's name, and its blue plot in the script's pane is flat.
3. The `25692.0000` value of bar_index shown in the scale represents its value on the last bar, so the dataset contains 25693 bars.
4. The value of bar_index on the bar the cursor is on is also displayed in the Data Window, along with that bar's value for RSI just above it.

In order to preserve our plot of RSI while still being able to inspect the value or bar_index, we will plot the variable using plotchar() like this:

//@version=4 study("Plot RSI and `bar_index`") r = rsi(close, 20) plot(r, "RSI", color.black) plotchar(bar_index, "Bar index", "", location.top)

where:

- Because the value of bar_index is no longer being plotted in the script's pane, the pane's boundaries are now those of RSI, which displays normally.
- The value plotted using plotchar() is displayed next to the script's name and in the Data Window.
- We are not plotting a character with our plotchar() call, so the third argument is an empty string (""). We are also specifying location.top as the `location` argument, so that we do not put the symbol's price in play in the calculation of the display area's boundaries.

Displaying strings

Pine labels must be used to display strings. Labels only appear in the script's display area; strings shown in labels do not appear in the Data Window or anywhere else.

Labels on each bar

The following script demonstrates the simplest way to repetitively draw a label showing the symbol's name:

//@version=4 study("Simple label", "", true) label.new(bar_index, high, syminfo.ticker)

By default, only the last 50 labels will be shown on the chart. You can increase this amount up to a maximum of 500 by using the `max_labels_count` parameter in your script's study() or strategy() declaration statement. For example:

study("Simple label", "", true, max_labels_count = 500)

Labels on last bar

As strings manipulated in Pine scripts often do not change bar to bar, the method most frequently used to visualize them is to draw a label on the dataset's last bar. Here, we use a function to create a label that only appears on the chart's last bar. Our `f_print()` function has only one parameter, the text string to be displayed:

//@version=4 study("f_print()", "", true) f_print(_text) => // Create label on the first bar. var _label = label.new(bar_index, na, _text, xloc.bar_index, yloc.price, color(na), label.style_none, color.gray, size.large, text.align_left) // On next bars, update the label's x and y position, and the text it displays. label.set_xy(_label, bar_index, highest(10)[1]) label.set_text(_label, _text) f_print("Multiplier = " + tostring(timeframe.multiplier) + "\nPeriod = " + timeframe.period + "\nHigh = " + tostring(high)) f_print("Hello world!\n\n\n\n")

Note the following in our last code example:

- We use the `f_print()` function to enclose the label-drawing code. While the function is called on each bar, the label is only created on the dataset's first bar because of our use of the var keyword when declaring the `_label` variable inside the function. After creating it, we only update the label's *x* and *y* coordinates and its text on each successive bar. If we did not update those values, the label would remain on the dataset's first bar and would only display the text string's value on that bar. Lastly, note that we use `highest(10)[1]` to position the label vertically, By using the highest high of the **previous** 10 bars, we prevent the label from moving during the realtime bar. You may need to adapt this *y* position in other contexts.
- We call the `f_print()` function twice to show that if you make multiple calls because it makes debugging multiple strings easier, you can superimpose their text by using the correct amount of newlines (`\n`) to separate each one.
- We use the tostring() function to convert numeric values to a string for inclusion in the text to be displayed.

Debugging conditions

Single conditions

Many methods can be used to display occurrences where a condition is met. This code shows six ways to identify bars where RSI is smaller than 30:

//@version=4 study("Single conditions") r = rsi(close, 20) rIsLow = r < 30 hline(30) // Method #1: Change the plot's color. plot(r, "RSI", rIsLow ? color.fuchsia : color.black) // Method #2: Plot a character in the bottom

region of the display. plotchar(rIsLow, "rIsLow char at bottom", "▲", location.bottom, size = size.small) // Method #3: Plot a character on the RSI line. plotchar(rIsLow ? r : na, "rIsLow char on line", "•", location.absolute, color.red, size = size.small) // Method #4: Plot a shape in the top region of the display. plotshape(rIsLow, "rIsLow shape", shape.arrowup, location.top) // Method #5: Plot an arrow. plotarrow(rIsLow ? 1 : na, "rIsLow arrow") // Method #6: Change the background's color. bgcolor(rIsLow ? color.green : na)

Note that:

- We define our condition in the `rIsLow` boolean variable and it is evaluated on each bar. The `r < 30` expression used to assign a value to the variable evaluates to `true` or `false` (or `na` when `r` is `na`, as is the case in the first bars of the dataset).
- **Method #1** uses a change in the color of the RSI plot on the condition. Whenever a plot's color changes, it colors the plot starting from the preceding bar.
- **Method #2** uses plotchar() to plot an up triangle in the bottom part of the indicator's display. Using different combinations of positions and characters allows the simultaneous identification of multiple conditions on a single bar. **This is one of our preferred methods to identify conditions on the chart.**
- **Method #3** also uses a plotchar() call, but this time the character is positioned on the RSI line. In order to achieve this, we use location.absolute and Pine's ternary conditional operator (`?:`) to define a conditional expression where a *y* position is used only when our `rIsLow` condition is true. When it is not true, `na` is used, so no character is displayed.
- **Method #4** uses plotshape() to plot a blue up arrow in the top part of the indicator's display area when our condition is met.
- **Method #5** uses plotarrow() to plot a green up arrow at the bottom of the display when our condition is met.
- **Method #6** uses bgcolor() to change the color of the background when our condition is met. The ternary operator is used once again to evaluate our condition. It will return `color.green` when `rIsLow` is true, and the `na` color (which does not color the background) when `rIsLow` is false or `na`.
- Lastly, note how a boolean variable with a `true` value displays as `1` in the Data Window. `false` values are denoted by a zero value.

Compound conditions

Programmers needing to identify situations where more than one condition is met must build compound conditions by aggregating individual conditions using the and logical operator. Because compound conditions will only perform as expected if their individual conditions trigger correctly, you will save yourself many headaches if you validate the behavior of individual conditions before using a compound condition in your code.

The state of multiple individual conditions can be displayed using a technique like this one, where four individual conditions are used to build our `bull` compound condition:

//@version=4 study("Compound conditions") i_period = input(20) i_bullLevel = input(55) r = rsi(close, i_period) // Condition #1. rsiBull = r > i_bullLevel // Condition #2. hiChannel = highest(r, i_period * 2)[1] aboveHiChannel = r > hiChannel // Condition #3. channelIsOld = hiChannel >= hiChannel[i_period] // Condition #4. historyIsBull = sum(rsiBull ? 1 : -1, i_period * 3) > 0 // Compound condition. bull = rsiBull and aboveHiChannel and channelIsOld and historyIsBull hline(i_bullLevel) plot(r, "RSI", color.black) plot(hiChannel, "High Channel") plotchar(rsiBull ? i_bullLevel : na, "rIsBull", "1", location.absolute,

color.green, size = size.tiny) plotchar(aboveHiChannel ? r : na, "aboveHiChannel", "2", location.absolute, size = size.tiny) plotchar(channelIsOld, "channelIsOld", "3", location.bottom, size = size.tiny) plotchar(historyIsBull, "historyIsBull", "4", location.top, size = size.tiny) bgcolor(bull ? not bull[1] ? color.new(color.green, 50) : color.green : na)

Note that:

- We use a plotchar() call to display each condition's number, taking care to spread them over the indicator's *y* space so they don't overlap.
- The first two plotchar() calls use absolute positioning to place the condition number so that it helps us remember the corresponding condition. The first one which displays "1" when RSI is higher than the user-defined bull level for example, positions the "1" on the bull level.
- We use two different shades of green to color the background: the brighter one indicates the first bar where our compound condition becomes `true`, the lighter green identifies subsequent bars where our compound condition continues to be true.
- While it is not always strictly necessary to assign individual conditions to a variable because they can be used directly in boolean expressions, it makes for more readable code when you assign a condition to a variable name that will remind you and your readers of what it represents. Readability considerations should always prevail in cases like this one, where the hit on performance of assigning conditions to variable names is minimal or null.

Debugging from inside functions

Variables in function are local to the function, so not available for plotting from the script's global scope. In this script we have written the `f_hlca()` function to calculate a weighed average:

//@version=4 study("Debugging from inside functions", "", true) f_hlca() => var float _avg = na _hlca = avg(high, low, close, nz(_avg, close)) _avg := sma(_hlca, 20) hlca = f_hlca() plot(hlca)

We need to inspect the value of `_hlca` in the function's local scope as the function calculates, bar to bar. We cannot access the `_hlca` variable used inside the function from the script's global scope. We thus need another mechanism to pull that variable's value from inside the function's local scope, while still being able to use the function's result. We can use Pine's ability to have functions return a tuple to gain access to the variable:

//@version=4 study("Debugging from inside functions", "", true) f_hlca() => var float _avg = na _instantVal = avg(high, low, close, nz(_avg, close)) _avg := sma(_instantVal, 20) // Return two values instead of one. [_avg, _instantVal] [hlca, instantVal] = f_hlca() plot(hlca, "hlca") plot(instantVal, "instantVal", color.black)

Contrary to global scope variables, array elements of globally defined arrays can be modified from within functions. We can use this feature to write a functionally equivalent script:

//@version=4 study("Debugging from inside functions", "", true) // Create an array containing only one float element. instantVal = array.new_float(1) f_hlca() => var float _avg = na _instantVal = avg(high, low, close, nz(_avg, close)) // Set the array's only element to the current value of `_instantVal`. array.set(instantVal, 0, _instantVal) _avg := sma(_instantVal, 20) hlca = f_hlca() plot(hlca, "hlca") // Retrieve the value of the array's only element which was set from inside the function. plot(array.get(instantVal, 0), "instantVal", color.black)

Debugging from inside 'for' loops

Values inside for loops cannot be plotted using plot() calls in the loop. As in functions, such variables are also local to the loop's scope. Here, we explore three different techniques to inspect variable values originating from for loops, starting from this code example, which calculates the balance of bars in the lookback period which have a higher/lower true range value than the current bar:

```
//@version=4 study("Debugging from inside `for` loops") i_lookBack = input(20, minval = 0) float trBalance = 0 for _i = 1 to i_lookBack trBalance := trBalance + sign(tr - tr[_i]) hline(0) plot(trBalance)
```

Extracting a single value

If we want to inspect the value of a variable at a single point in the loop, we can save it and plot it once the loop is exited. Here, we save the value of tr in the `val` variable at the loop's last iteration:

```
//@version=4 study("Debugging from inside `for` loops", max_lines_count = 500, max_labels_count = 500) i_lookBack = input(20, minval = 0) float val = na float trBalance = 0 for _i = 1 to i_lookBack trBalance := trBalance + sign(tr - tr[_i]) if _i == i_lookBack val := tr[_i] hline(0) plot(trBalance) plot(val, "val", color.black)
```

Using lines and labels

When we want to extract values from more than one loop iteration we can use lines and labels. Here we draw a line corresponding to the value of tr used in each loop iteration. We also use a label to display, for each line, the loop's index and the line's value. This gives us a general idea of the values being used in each loop iteration:

```
//@version=4 study("Debugging from inside `for` loops", max_lines_count = 500, max_labels_count = 500) i_lookBack = input(20, minval = 0) float trBalance = 0 for _i = 1 to i_lookBack trBalance := trBalance + sign(tr - tr[_i]) line.new(bar_index[1], tr[_i], bar_index, tr[_i], color = color.black) label.new(bar_index, tr[_i], tostring(_i) + "•" + tostring(tr[_i]), style = label.style_none, size = size.small) hline(0) plot(trBalance)
```

Note that:

- To show more detail, the scale in the preceding screenshot has been manually expanded by clicking and dragging the scale area.
- We use `max_lines_count = 500, max_labels_count = 500` in our study() declaration statement to display the maximum number of lines and labels.
- Each loop iteration does not necessarily produce a distinct tr value, which is why we may not see 20 distinct lines for each bar.
- If we wanted to show only one level, we could use the same technique while isolating a specific loop iteration as we did in the preceding example.

Extracting multiple values

We can also extract multiple values from loop iterations by building a single string which we will display using a label after the loop executes:

```
//@version=4 study("Debugging from inside `for` loops", max_lines_count = 500, max_labels_count = 500) i_lookBack = input(20, minval = 0) string = "" float trBalance = 0 for _i = 1 to i_lookBack trBalance := trBalance + sign(tr - tr[_i]) string := string + tostring(_i, "00") + "•" + tostring(tr[_i]) + "\n" label.new(bar_index, 0, string, style = label.style_none, size = size.small, textalign = text.align_left) hline(0) plot(trBalance)
```

Note that:

- The scale in the preceding screenshot has been manually expanded by clicking and dragging the scale area so the content of the indicator's display area content could be moved vertically to show only its relevant part.
- We use `tostring(_i, "00")` to force the display of the loop's index to zero-padded two digits so they align neatly.

When loops with numerous iterations make displaying all their values impractical, you can sample a subset of the iterations. This code uses the % (modulo) operator to include values from every second loop iteration:

for _i = 1 to i_lookBack lowerRangeBalance := lowerRangeBalance + sign(tr - tr[_i]) if _i % 2 == 0 string := string + tostring(_i, "00") + "•" + tostring(tr[_i]) + "\n"

Tips

The two techniques we use most frequently to debug our Pine code are:

plotchar(v, "v", "", location.top, size = size.tiny)

to plot variables of type *float*, *int* or *bool* in the indicator's values and the Data Window, and the one-line version of our `f_print()` function to debug strings:

f_print(_text) => var _label = label.new(bar_index, na, _text, xloc.bar_index, yloc.price, color(na), label.style_none, color.gray, size.large, text.align_left), label.set_xy(_label, bar_index, highest(10)[1]), label.set_text(_label, _text) f_print(stringName)

As we use AutoHotkey for Windows to speed repetitive tasks, we include these lines in our AutoHotkey script (this is **not** Pine code):

```
; ————— This is AHK code, not Pine. —————
^+f:: SendInput plotchar(^v, "^v", "", location.top, size = size.tiny){Return}
^+p:: SendInput f_print(_text) => var _label = label.new(bar_index, na, _text, 
```

The second line will type a debugging plotchar() call including an expression or variable name previously copied to the clipboard when we use `CTRL-SHIFT-F`. Copying the `variableName` variable name or the `close > open` conditional expression to the clipboard and hitting `CTRL-SHIFT-F` will, respectively, yield:

plotchar(variableName, "variableName", "", location.top, size = size.tiny) plotchar(close > open, "close > open", "", location.top, size = size.tiny)

The third line triggers on `CTRL-SHIFT-P`. It types our one-line `f_print()` function in a script and on a second line, an empty call to the function with the cursor placed so all that's left to do is type the string we want to display:

f_print(_text) => var _label = label.new(bar_index, na, _text, xloc.bar_index, yloc.price, color(na), label.style_none, color.gray, size.large, text.align_left), label.set_xy(_label, bar_index, highest(10)[1]), label.set_text(_label, _text) f_print()

Note: AutoHotkey works only on Windows systems. Keyboard Maestro or others can be substituted on Apple systems.

Notice!This is documentation for v4, 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 in Pine scripts. 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 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, Pine coders 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 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 studies 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 studies only, *alertcondition() events* which can then be used to create *alertcondition() alerts* from the charts UI.

Keep in mind that:

- No alert-related Pine 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 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 coders can use today to create alert events in their script are the result of successive enhancements deployed throughout Pine's evolution. The alertcondition() function, which works in studies only, was the first feature allowing Pine 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. 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 coders, 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 studies 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 Pine coders to generate alert events on most conditions occurring in their script's execution.

The alertcondition() function remains in Pine 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 a study or a strategy.

A *script alert* created from a **study** will trigger when:

- The study 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*.

> NoticePine studies are often referred to as "indicators" in the charts UI and in the Help Center's user documentation.

'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 the "series" form, 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 studies 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=4 study("All `alert()` calls") r = rsi(close, 20) // Detect crosses. xUp = crossover(r, 50) xDn = crossunder(r, 50) // Trigger an alert on crosses. if xUp alert("Go long (RSI is " + tostring(r, "#.00")) else if xDn alert("Go short (RSI is " + 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 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 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 + 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=4 study("Selective `alert()` calls") i_detectLongs = input(true, "Detect Longs") i_detectShorts = input(true, "Detect Shorts") i_repaint = input(false, "Allow Repainting") r = rsi(close, 20) // Detect crosses. xUp = crossover(r, 50) xDn = crossunder(r, 50) // Only generate entries when the trade's direction is allowed in inputs. enterLong = i_detectLongs and xUp and (i_repaint or barstate.isconfirmed) enterShort = i_detectShorts and xDn and (i_repaint or barstate.isconfirmed) // Trigger the alerts only when the compound condition is met. if enterLong alert("Go long (RSI is " + tostring(r, "#.00")) else if enterShort alert("Go short (RSI is " + 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 the same way as in studies. 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=4 strategy("Strategy with selective `alert()` calls") r = rsi(close, 20) // Detect crosses. xUp = crossover(r, 50) xDn = 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 falling(r, 3) divInShortTrade = strategy.position_size < 0 and 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 a study, 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 coders 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=4 strategy("Strategy using `alert_message`") r = rsi(close, 20) // Detect crosses. xUp = crossover(r, 50) xDn = 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 " +

tostring(high) + ")") else if xDn strategy.entry("Short", strategy.short, stop = low, alert_message = "Stop-sell executed (stop was " + 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 Pine studies. One study 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 Pine **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=4 study("`alertcondition()` on single condition") r = rsi(close, 20) xUp = crossover(r, 50) xDn = 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 `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 study, 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 a study 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=4 study("`alertcondition()` on multiple conditions") i_detectLongs = input(true, "Detect Longs") i_detectShorts = input(true, "Detect Shorts") r = rsi(close, 20) // Detect crosses. xUp = crossover( r, 50) xDn = crossunder(r, 50) // Only generate entries when the trade's direction is allowed in inputs. enterLong = i_detectLongs and xUp enterShort = i_detectShorts 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 > 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=4 study("") r = rsi(close, 14) xUp = crossover(r, 50) plot(r, "RSI") alertcondition(xUp, 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 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 Indicator repainting section may not be preventable by simply triggering alerts on the close of realtime bars.

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Arrays

Introduction

Arrays can be used to store multiple values in one data structure. Think of them as a better way to handle cases where you would otherwise need a set of variables named `price00`, `price01` and `price02`. Arrays are an advanced feature used for scripts requiring intricate data-handling. If you are a beginning Pine programmer, we recommend you become familiar with other, more accessible Pine features before you tackle arrays.

Pine arrays are one-dimensional. All elements of an array are of the same type, which can be *int*, *float*, *bool* or *color*, always of *series* form. Arrays are referenced using an array *id*, similar to label and line id's. Pine does not use an indexing operator to reference individual array elements; instead, functions like array.get() and array.set() are used to read and write values of array elements. Array values can be used in all Pine expressions and functions where a value of *series* form is allowed.

Elements within an array are referred to using an *index*, which starts at 0 and extends to the number or elements in the array, minus one. Arrays in Pine can be sized dynamically, so the number of elements in the array can be modified within one iteration of the script on a bar, and vary across bars. Multiple arrays can be used in the same script. The size of arrays is limited to 100,000.

> NoticeWe 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 *id's*, for the sake of brevity.

Declaring arrays

The following syntax can be used to declare arrays:

```
<type>[] <identifier> = <expression>
var <type>[] <identifier> = <expression>
```

The `[]` modifier is appended to the type name when declaring arrays. However, since type-specific functions are always used to create arrays, the `<type>[]` part of the declaration is redundant, except if you initialize an array variable to `na`, as in the following example where we declare an array variable named `prices`. The variable will be used to designate an array containing *float* values, but no array is created by this declaration yet. For the moment, the array variable contains no valid array id, its value being `na`:

```
float[] prices = na
```

When declaring an array and the `<expression>` is not `na`, one of the `array.new_<type>(size, initial_value)` functions must be used. The arguments of both 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)
```

Similar array creation functions exist for the other types of array elements: array.new_int(), array.new_bool(), array.new_color(), array.new_line(), array.new_label() and array.new_string().

When declaring an array, you can initialize all elements in the array using the `initial_value` parameter. When no argument is supplied for `initial_value`, the array elements are initialized to `na`. The following declaration creates and array id named `prices`. The array is created with two elements, each initialized with the value of the `close` built-in variable on that bar:

```
prices = array.new_float(2, close)
```

There is currently no way to initialize multiple array elements with different values in one statement, whether upon declaration or post-declaration. One is planned in the near future.

Using the 'var' keyword

The var keyword can be used when declaring arrays. It works just as it does for other variables; it causes the declaration to only be executed on the first iteration of the script on the dataset's bar at `bar_index` zero. Because the array is never re-initialized on subsequent bars, its value will persist across bars, as the script iterates on them.

When an array declaration is done using `var` and a new value is pushed at the end of the array on each bar, the array will grow by one on each bar and be of size `bar_index` plus one (bar_index starts at zero) by the time the script executes on the last bar, as this code will do:

```
//@version=4 study("Using `var`") var a = array.new_float(0) array.push(a, close) if barstate.islast label.new(bar_index, 0, "Array size: " + tostring(array.size(a)) + "\nbar_index: " + tostring(bar_index), size = size.large)
```

The same code without the `var` keyword would re-declare the array on each bar. After execution of the `array.push()` call, the array would thus be of size one on all the dataset's bars.

Reading and writing array values

Values can be written to existing individual array elements using array.set(id, index, value), and read using array.get(id, index). As is the case whenever an array index is used in your code, it is imperative that the index never be greater than the array's size, minus one (because array indices start at zero). You can obtain the size of an array by using the array.size(id) function.

The following example uses array.set() to initialize an array of colors to instances of one base color using different transparency levels. It then fetches the proper array element to use it in a bgcolor() call:

```
//@version=4 study("Distance from high", "", true) i_lookBack = input(100) c_fillColor = color.green var c_fills = array.new_color(5) // Initialize the array elements with progressively lighter shades of the fill color. array.set(c_fills, 0, color.new(c_fillColor, 70)) array.set(c_fills, 1, color.new(c_fillColor, 75)) array.set(c_fills, 2, color.new(c_fillColor, 80)) array.set(c_fills, 3, color.new(c_fillColor, 85)) array.set(c_fills, 4, color.new(c_fillColor, 90)) // Find the offset to highest high. Change its sign because the function returns a
```

negative value. lastHiBar = - highestbars(high, i_lookBack) // 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 = min(lastHiBar / (i_lookBack / 5), 4) // Set background to a progressively lighter fill with increasing distance from location of highest high. bgcolor(array.get(c_fills, 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 that can be used to initialize the elements in an array is to declare the array with size zero, and then populate it using array.push() to append **new** elements to the end of the array, increasing the size of the array by one at each call. The following code is functionally identical to the initialization section from the preceding script. Note that we do not use `var` to declare the array in this case. If we did, the set of pushes would add 5 new elements to the array on each bar, since the array would propagate over successive bars:

c_fills = array.new_color(0) // Initialize the array elements with progressively lighter shades of the fill color. array.push(c_fills, color.new(c_fillColor, 70)) array.push(c_fills, color.new(c_fillColor, 75)) array.push(c_fills, color.new(c_fillColor, 80)) array.push(c_fills, color.new(c_fillColor, 85)) array.push(c_fills, color.new(c_fillColor, 90))

The array.fill(id, value, index_from, index_to) function can be used to fill contiguous sets of array elements with a value. Used 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) array.fill(a, close)

are equivalent, but:

a = array.new_float(10) array.fill(a, 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`, needs to be one greater than the last index to be filled. The remaining elements will hold the `na` value, as no intialization value was provided when the array was declared.

Scope

Arrays can be declared in a script's global scope, as well as in the local scope of a function or an `if` branch. One major distinction between Pine arrays and variables declared in the global scope, is that global arrays can be modified from within the local scope of a function. This new capability can be used to implement global variables that can be both read and set from within any function in the script. We use it here to calculate progressively lower or higher levels:

//@version=4 study("Bands", "", true) i_factor = 1 + (input(-2., "Step %") / 100) // Use the lowest average OHLC in last 50 bars from 10 bars back as the our base level. level = array.new_float(1, lowest(ohlc4, 50)[10]) f_nextLevel(_val) => _newLevel = array.get(level, 0) * _val // Write new level to the global array so it can be used as the base in the next call to this function. array.set(level, 0, _newLevel) _newLevel plot(f_nextLevel(1)) plot(f_nextLevel(i_factor)) plot(f_nextLevel(i_factor)) plot(f_nextLevel(i_factor))

History referencing

Past instances of array id's or elements cannot be referenced directly using Pine's [ ] history-referencing operator. One **cannot** write: `array.get(a[1], 0)` to fetch the value of the array's first element on the previous bar.

In Pine, however, each call to a function leaves behind a series trail of function results on previous bars. This series can in turn be used when working with arrays. One can thus write: `ma = sma(array.get(a, 0), 20)` to calculate the simple moving average of the value returned by the `array.get(a, 0)` call on the last 20 bars.

To illustrate this, let's first see how we can fetch the previous bar's `close` value in two, equivalent ways. For `previousClose1` we use the result of the `array.get(a, 0)` function call on the previous bar. Since on the previous bar the array's only element was initialized to that bar's `close` (as it is on every bar), referring to `array.get(a, 0)[1]` returns that bar's `close`, i.e., the value of the `array.get(a, 0)` call on the previous bar.

For `previousClose2` we use the history-referencing operator to fetch the previous bar's `close` in normal Pine fashion:

//@version=4 study("History referencing") // Re-declare the array on each bar. a = array.new_float(1) // Set the value of its only element to `close`. array.set(a, 0, close) previousClose1 = array.get(a, 0)[1] previousClose2 = close[1] plot(previousClose1, "previousClose1", color.gray, 6) plot(previousClose2, "previousClose2", color.white, 2)

In the following example we add two, equivalent calculations of a moving average to our previous code example. For `ma1` we use sma() on the series of values returned by the `array.get(a, 0)` function call on each bar. Since at this point in the script the call returns the current bar's `close`, that is the value used for the average's calculation. We evaluate `ma2` using the usual way we would calculate a simple average in Pine:

//@version=4 study("History referencing") a = array.new_float(1) array.set(a, 0, close) previousClose1 = array.get(a, 0)[1] previousClose2 = close[1] plot(previousClose1, "previousClose1", color.gray, 6) plot(previousClose2, "previousClose2", color.white, 2) ma1 = sma(array.get(a, 0), 20) ma2 = sma(close, 20) plot(ma1, "MA 1", color.aqua, 6) plot(ma2, "MA 2", color.white, 2) // Last set having no impact. array.set(a, 0, 10.0)

Notice the last line of this script. It illustrates how even if we set the value of the array's element to `10.0` at the end of the script, resulting in the final value for the element being committed as `10.0` on the bar's last execution of the script, the earlier call to `array.get(a, 0)` nonetheless returned the `close` value because that was the value of the array element at that point in the script. The series value of the function call will thus be each bar's `close` value.

Inserting and removing array elements

Inserting

Three functions can be used to insert new elements in an array.

array.unshift() inserts a new element at the beginning of an array, at index zero, and shifts any existing elements right by one.

array.insert() can insert a new element at any position in the array. Its `index` parameter is the index where the new element will be added. The element existing at the index used in the function call and any others to its right are shifted one place to the right:

//@version=4 study("`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: " + tostring(a), size = size.large) array.insert(a, 2, 999) label.new(bar_index, 0, "AFTER\na: " + tostring(a), style = label.style_label_up, size = size.large)

array.push() will add a new element at the end of an array.

Removing

Four functions can be used to remove elements from an array. The first three will return the value of the removed element.

array.remove() removes the element at the `index` value used, 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() will remove all elements in the array.

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 arrays can be used as a stack, in which case you will 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 remember successive lows in rallies:

//@version=4 study("Lows from new highs", "", true) var lows = array.new_float(0) flushLows = false // Remove last element from the stack when `_cond` is true. f_array_pop(_id, _cond) => _cond and array.size(_id) > 0 ? array.pop(_id) : float(na) if rising(high, 1) // Rising highs; push a new low on the stack. array.push(lows, low) // Force the return type of this `if` block to be the same as that of the next block. bool(na) else if array.size(lows) >= 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 = f_array_pop(lows, flushLows) plot(lowLevel, "Low 1", low > lowLevel ? color.silver : color.purple, 2, plot.style_linebr) lowLevel := f_array_pop(lows, flushLows) plot(lowLevel, "Low 2", low > lowLevel ? color.silver : color.purple, 3, plot.style_linebr) lowLevel := f_array_pop(lows, flushLows) plot(lowLevel, "Low 3", low > lowLevel ? color.silver : color.purple, 4, plot.style_linebr) lowLevel := f_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. array.clear(lows)

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 `i_pivotCount` elements once again. Note that on the dataset's first bars we will be deleting `na` label id's until the maximum number of labels has been created, but this does not cause runtime errors. Let's look at our code:

//@version=4 MAX_LABELS = 100 study("Show Last n High Pivots", "", true, max_labels_count = MAX_LABELS) i_pivotCount = input(5, "How many pivots to show", minval = 0, maxval = MAX_LABELS) i_pivotLegs = input(3, "Pivot legs", minval = 1, maxval = 5) // Format price to tick precision. f_tickFormat() => _s = tostring(syminfo.mintick) _s := str.replace_all(_s, "25", "00") _s := str.replace_all(_s, "5", "0") _s := str.replace_all(_s, "1", "0") // Create an array containing the user-selected max count of label ids. var labelIds = array.new_label(i_pivotCount) pHi = pivothigh(i_pivotLegs, i_pivotLegs) if not na(pHi) // New pivot found; plot its label `i_pivotLegs` bars back. pLabel = label.new(bar_index[i_pivotLegs], pHi, tostring(pHi, f_tickFormat()), 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's one-dimensional arrays can be viewed as vertical structures residing on each bar. As an array's set of elements is not a series, Pine'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.avg(), array.min(), array.max(), array.median(), array.mode(), array.standardize(), array.stdev(), array.sum(), array.variance(), array.covariance(), array.range().

Note that contrary to the usual mathematical functions in Pine, 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=4 study("`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: " + tostring(a) + "\nb: " + tostring(b), size = size.large) _c = array.concat(a, b) array.push(_c, 4) label.new(bar_index, 0, "AFTER\na: " + tostring(a) + "\nb: " + tostring(b) + "\nc: " + 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=4 study("`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: " + tostring(a) + "\n_b: " + 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=4 study("") 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 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 of form *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=4 study("`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: ") + tostring(a) + "\n\n" + "b " + (_barUp ? "is not sorted: " : "is sorted ▼: ") + tostring(b), size = size.large)
```

Reversing

Use array.reverse() to reverse an array:

```
//@version=4 study("`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: " + 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=4 study("`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: " + tostring(a) + "\n_sliceOfA: " + 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: " + tostring(a) + "\n_sliceOfA: " + 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=4 study("Searching in arrays") _value = input(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, _value) _firstIndexFound = array.indexof(a, _value) _lastIndexFound = array.lastindexof(a, _value) label.new(bar_index, 0, "a: " + tostring(a) + "\nFirst " + tostring(_value) + (_firstIndexFound != -1 ? " value was found at index: " + tostring(_firstIndexFound) : " value was not found.") + "\nLast " + tostring(_value) + (_lastIndexFound != -1 ? " value was found at index: " + tostring(_lastIndexFound) : " value was not 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 Reference Manual when in doubt regarding the exact syntax of function calls.

Scripts using arrays can also throw **runtime** errors, which appear in place of the indicator's name on charts. 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 inexistent 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=4 study("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
```

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

//@version=4 study("Protected array size") i_size = input(10, "Array size", minval = 1, maxval = 100000) a = array.new_float(i_size) for _i = 0 to i_size - 1 array.set(a, _i, _i) plot(array.size(a))

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=4 study("Out of bounds index") int[] a = na array.push(a, 111) label.new(bar_index, 0, "a: " + tostring(a))

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

```
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=4 study("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 v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Bar states. Built-in variables barstate.*

A set of built-in variables of the `barstate` namespace allows users to define the bar states for which scripts actions are performed:

- `barstate.isfirst` - `true` if the current bar is the first in the whole range of bars available, `false` otherwise.
- `barstate.islast` - `true` if the current bar is the last in the whole range of bars available, `false` otherwise. This flag helps to detect *the last historical bar*.
- `barstate.ishistory` - `true` if the current data update is a historical bar update, `false` otherwise (thus it is realtime).
- `barstate.isrealtime` - `true` if the current data update is a real-time bar update, `false` otherwise (thus it is historical). Note that every realtime bar is also the *last* one.
- `barstate.isnew` - `true` if the current data update is the first (opening) update of a new bar, `false` otherwise.
- `barstate.isconfirmed` - `true` if the current data update is the last (closing) update of the current bar, `false` otherwise. The next data update will be an opening update of a new bar[1].
- `barstate.islastconfirmedhistory` - `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.

All historical bars are considered *new* bars. That is because of the fact that the script receives them in a sequential order from the oldest to the newer ones. For bars that update in realtime, a bar is considered new only at the opening tick of this bar.

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

//@version=4 study("Bar States", overlay = true) first = barstate.isfirst last = barstate.islast hist = barstate.ishistory rt = barstate.isrealtime new = barstate.isnew conf = barstate.isconfirmed t = new ? "new" : conf ? "conf" : "intra-bar" t := t + (hist ? "\nhist" : rt ? "\nrt" : "") t := t + (first ? "\nfirst" : last ? "\nlast" : "") label.new(bar_index, na, yloc=yloc.abovebar, text=t, color=hist ? color.green : color.red)

We begin by adding the "Bar States" study to a yearly chart and take a screenshot before any realtime update is received. This shows the *first* and the *last* bars, and the fact that all bars are *new* ones:

When a realtime update is received, the picture changes slightly. The current bar is no longer a historical bar, it has become a realtime bar. Additionally, it is neither *new* nor *confirmed*, which we indicate with the *intra-bar* text in the label.

This is a screenshot of the same symbol at a *1 minute* timeframe, after a few realtime bars have elapsed. The elapsed realtime bars show the *confirmed* state.

Footnotes

1. Variable `barstate.isconfirmed` returns the state of current chart symbol data only. It does not take into account any secondary symbol data requested with the `security` function. ↩

Notice!This is documentation for v4, 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. 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 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 has built-in colors such as color.red, 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 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 in Pine 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 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 lists the groups of visual elements in Pine, 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
- Plots
- Hlines
- Fills
- Boxes
- Labels
- Lines
- Tables

Constant colors

There are 17 built-in colors in Pine. 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.red color with a transparency of 40, but expressed in different ways. All five methods are functionally equivalent:

//@version=4 study("", "", true) // ———— Use `transp` parameter. plot(sma(close, 10), "10", color.olive, transp = 40) plot(sma(close, 30), "30", #808000, transp = 40) // ———— Transparency (#99) is included in the hex value. plot(sma(close, 50), "50", #80800099) // ———— Transparency is included in the color-generating function's arguments. plot(sma(close, 70), "70", color.new(color.olive, 40)) plot(sma(close, 90), "90", color.rgb(128, 128, 0, 40))

> NoticeThe first two plot() calls specify transparency using the `transp` parameter. This use should be avoided as the `transp` is slated for deprecation in Pine 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, Pine 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 gradations, 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=4 study("Conditional colors", "", true) int i_length = input(20, "Length", minval = 2) color i_c_maBull = input(color.green, "Bull") color i_c_maBear = input(color.maroon, "Bear") float ma = sma(close, i_length) // Define our states. bool maRising = rising(ma, 1) // Build our color. color c_ma = maRising ? i_c_maBull : i_c_maBear 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=4 study("Conditional colors", "", true) int i_legs = input(5, "Pivot Legs", minval = 1) color i_c_pHi = input(color.olive, "High pivots") color i_c_pLo = input(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(pivothigh(i_legs, i_legs), pHi) pLo := nz(pivotlow(i_legs, i_legs), pLo) // When a new pivot is detected, do not plot a color. plot(pHi, "High", change(pHi) ? na : i_c_pHi, 2, plot.style_line) plot(pLo, "Low", change(pLo) ? na : i_c_pLo, 2, plot.style_line)

To undertand how this code works, one must first know that pivothigh() and 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=4 study("Volume") // We name our color constants to make them more readable. var color C_GOLD = #CCCC00ff var color C_VIOLET = #AA00FFff color i_c_bull = input(C_GOLD, "Bull") color i_c_bear = input(C_VIOLET, "Bear") int i_levels = input(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 := max(1, min(i_levels, riseFallCnt + 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 - abs(80 * riseFallCnt / i_levels) // Build the correct transparency of either the bull or bear color. color c_volume = color.new(close > open ? i_c_bull : i_c_bear, transparency) plot(volume, "Volume", c_volume, 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 brightnes - 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=4 study("Holiday candles", "", true) float r = random(0, 255) float g = random(0, 255) float b = random(0, 255) float t = random(0, 100) color c_holiday = color.rgb(r, g, b, t) plotcandle(open, high, low, close, color = c_holiday, wickcolor = c_holiday, 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 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 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=4 study(title="CCI line gradient", precision=2, resolution="") var color C_GOLD = #CCCC00 var color C_VIOLET = #AA00FF var color C_BEIGE = #9C6E1B float i_src = input(close, title="Source") int i_len = input(20, "Length", minval = 5) color i_c_bull = input(C_GOLD, "Bull") color i_c_bear = input(C_VIOLET, "Bear") float signal = cci(i_src, i_len) color c_signal = color.from_gradient(signal, -200, 200, C_VIOLET, C_GOLD) plot(signal, "CCI", c_signal) p_bandTop = hline(100, "Upper Band", color.silver, hline.style_dashed) p_bandBot = hline(-100, "Lower Band", color.silver, hline.style_dashed) fill(p_bandTop, p_bandBot, C_BEIGE, title = "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 as built-in functions. Here we use 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=4 study(title="CCI line gradient", precision=2, resolution="") var color C_GOLD = #CCCC00 var color C_VIOLET = #AA00FF var color C_GREEN_BG = color.new(color.green, 70) var color C_RED_BG = color.new(color.maroon, 70) float i_src = input(close, "Source") int i_len = input(20, "Length", minval = 5) int i_steps = input(50, "Gradient levels", minval = 1) color i_c_bull = input(C_GOLD, "Line: Bull", inline = "11") color i_c_bear = input(C_VIOLET, "Bear", inline = "11") color i_c_bullBg = input(C_GREEN_BG, "Background: Bull", inline = "12") color i_c_bearBg = input(C_RED_BG, "Bear", inline = "12") // Plot colored signal line. float signal = cci(i_src, i_len) color c_signal = color.from_gradient(signal, -200, 200, color.new(C_VIOLET, 0), color.new(C_GOLD, 0)) plot(signal, "CCI", c_signal, 2) // Detect crosses of the centerline. bool signalX = cross(signal, 0) // Count no of bars since cross. Capping it to the no of steps from inputs. int gradientStep = min(i_steps, nz(barssince(signalX))) // Choose bull/bear end color for the gradient. color c_endColor = signal > 0 ? i_c_bullBg : i_c_bearBg // Get color from gradient going from no color to `c_endColor` color c_band = color.from_gradient(gradientStep, 0, i_steps, na, c_endColor) p_bandTop = hline(100, "Upper Band", color.silver, hline.style_dashed) p_bandBot = hline(-100, "Lower Band", color.silver, hline.style_dashed) fill(p_bandTop, p_bandBot, c_band, title = "Band")

Note that:

- The signal plot uses the same base colors and gradient as in our previous example. We have however increased the witdth 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 barssince() because in early bars of the dataset, when the condition tested has not occurred yet, 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 our script:

//@version=4 study("CCI DC", precision = 6) var color C_GOLD = #CCCC00ff var color C_VIOLET = #AA00FFff int i_p = input(20, "Length", minval = 5) color i_c_bull = input(C_GOLD, "Bull") color i_c_bear = input(C_VIOLET, "Bear") // ————— Function clamps `_val_` between `_min` and `_max`. f_clamp(_val, _min, _max) => _return = max(_min, min(_max, _val)) // ————— Volatility expressed as 0-100 value. float v = atr(i_p / 5) / atr(i_p * 5) float vPct = percentrank(v, i_p * 5) // ————— Calculate dynamic lookback for DC. It increases/decreases on low/high volatility. bool highVolatility = vPct > 50 var int lookBackMin = i_p * 2 var int lookBackMax = i_p * 10 var float lookBack = avg(lookBackMin, lookBackMax) lookBack += highVolatility ? -2 : 2 lookBack := f_clamp(lookBack, lookBackMin, lookBackMax) // ————— Dynamic lookback length Donchian channel of signal. float signal = cci(close, i_p) // `lookBack` is a float; need to cast it to int to be used a length. float hiTop = highest(signal, int(lookBack)) float loBot = 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 = avg(hiTop, loBot) // ————— Create colors. color c_signal = color.from_gradient(signal, -200, 200, C_VIOLET, C_GOLD) // Bands: Calculate transparencies so the longer since the hi/lo has changed, // the darker the color becomes. Cap highest transparency to 90. float hiTransp = f_clamp(100 - (100 * max(1, nz(barssince(change(hiTop)) + 1)) / 255), 60, 90) float loTransp = f_clamp(100 - (100 * max(1, nz(barssince(change(loBot)) + 1)) / 255), 60, 90) color c_hi = color.new(i_c_bull, hiTransp) color c_lo = color.new(i_c_bear, loTransp) // Background: Rescale the 0-100 range of `vPct` to 0-25 to create 75-100 transparencies. color c_bg = color.new(color.gray, 100 - (vPct / 4)) // ————— Plots // Invisible lines for band fills. p_hiTop = plot(hiTop, color = na) p_hiBot = plot(hiBot, color = na) p_loTop = plot(loTop, color = na) p_loBot = plot(loBot, color = na) // Plot signal and centerline. p_signal = plot(signal, "CCI", c_signal, 2) plot(center, "Centerline", color.silver, 1) // Fill the bands. fill(p_hiTop, p_hiBot, c_hi) fill(p_loTop, p_loBot, c_lo) // ————— Background. bgcolor(c_bg)

This is what our indicator looks like using the light theme:

And with the dark theme:

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 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

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. The Pine Color Magic and Chart Theme Simulator script provides a good selection of base colors to build from.

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=4 study("") 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.

Color selection through script settings

The type of color you use in your scripts has an impact on how users of your script will be able to change the colors of your script's visuals. As long as you don't use colors whose RGBA components have to be calculated at runtime, script users will be able to modify the colors you use by going to your script's "Settings/Style" tab. Our first example script on this page meets that criteria, and the following screenshot shows how we used the script's "Settings/Style" tab to change the color of the first moving average:

If your script uses a calculated color, i.e., a color where at least one of its RGBA components can only be known at runtime, then the "Settings/Style" tab will NOT offer users the usual color widgets they can use to modify your plot colors. Plots of the same script not using calculated colors will also be affected. In this script, for example, our first plot() call uses a calculated color, and the second one doesn't:

//@version=4 study("Calculated colors", "", true) float ma = sma(close, 20) float maHeight = percentrank(ma, 100) float transparency = min(80, 100 - maHeight) // This plot uses a calculated color. plot(ma, "MA1", color.rgb(156, 39, 176, transparency), 2) // This plot does not use a calculated color. plot(close, "Close", color.blue)

The color used in the first plot is a calculated color because its transparency can only be known at runtime. It is calculated using the relative position of the moving average in relation to its past 100 values. The greater percentage of past values are below the current value, the higher the 0-100 value of `maHeight` will be. Since we want the color to be the darkest when `maHeight` is 100, we subtract 100 from it to obtain the zero transparency then. We also cap the calculated `transparency` value to a maximum of 80 so that it always remains visible.

Because that calculated color is used in our script, the "Settings/Style" tab will not show any color widgets:

The solution to enable script users to control the colors used is to supply them with custom inputs, as we do here:

//@version=4 study("Calculated colors", "", true) color i_c_ma = input(color.olive, "MA") color i_c_close = input(color.blue, "Close") float ma = sma(close, 20) float maHeight = percentrank(ma, 100) float transparency = min(80, 100 - maHeight) // This plot uses a calculated color. plot(ma, "MA1", color.new(i_c_ma, transparency), 2) // This plot does not use a calculated color. plot(close, "Close", i_c_close)

Notice how our script's "Settings" now show an "Inputs" tab, where we have created two color inputs. The first one uses color.fuchsia as its default value. Whether the script user changes that color or not, it will then be used in a color.new() call to generate a calculated transparency in the plot() call. The second input uses as its default the built-in color.blue color we previously used in the plot() call, and simply use it as is in the second plot() call.

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Context switching and the security function

The `security` function enables scripts to request data from symbols and/or resolutions other than the ones a script is running on.

Let's assume the following script is running on an IBM chart at *1 minute*. It will display the *close* price of the IBM symbol, but at *15 minutes* resolution.

//@version=4 study("Example security 1", overlay=true) ibm_15 = security("NYSE:IBM", "15", close) plot(ibm_15)

The security function's first argument is the name of the requested symbol. The second argument is the required resolution and the third one is an expression which will be calculated on the requested series *within* the `security` call.

The name of the symbol can be defined using two variants: with a prefix that contains the exchange (or data provider), or without it. For example: `"NYSE:IBM"`, `"BATS:IBM"` or `"IBM"`. When an exchange is not provided, BATS will be used as the default. The current symbol name is stored in the syminfo.ticker and syminfo.tickerid built-in variables. `syminfo.ticker` contains the value of the symbol name without its exchange prefix, for example `"MSFT"`. `syminfo.tickerid` contains the value of the symbol name with its exchange prefix, for example, `"BATS:MSFT"` or `"NASDAQ:MSFT"`. It is recommended to use `syminfo.tickerid` to avoid ambiguity in the values returned by `security`.

The second argument of the `security` function, `resolution` (i.e., the *timeframe*), is also a string. All intraday resolutions are defined using a number of minutes. The lowest resolution is *one minute* and is indicated by the literal `"1"`. It is possible to request any[1] number of minutes: `"5"`, `"10"`, `"21"`, etc. *Hourly* resolution is also set by minutes[2]. For example, the following lines signify one hour, two hours and four hours respectively: `"60"`, `"120"`, `"240"`. A resolution with a value of *1 day* is indicated by `"D"` or `"1D"`. It is possible to request any number of days: `"2D"`, `"3D"`, etc. *Weekly* and *Monthly* resolutions are set in a similar way: `"W"`, `"1W"`, `"2W"`, ..., `"M"`, `"1M"`, `"2M"`. `"M"` and `"1M"` denote the same monthly resolution, and `"W"` and `"1W"` the same weekly resolution. The third parameter of the security function can be any arithmetic expression or a function call, which will be calculated in the context of the chosen series. The resolution of the main chart's symbol is stored in the timeframe.period built-in variable.

With the `security` function, users can view a *1 minute* chart while displaying an SMA (or any other expression) from any other resolution (i.e., daily, weekly, monthly):

//@version=4 study(title="High Time Frame MA", overlay=true) src = close, len = 9 out = sma(src, len) out1 = security(syminfo.tickerid, 'D', out) plot(out1)

One can declare the following variable:

spread = high - low

and calculate it at *1 minute*, *15 minutes* and *60 minutes*:

spread_1 = security(tickerid, '1', spread) spread_15 = security(tickerid, '15', spread) spread_60 = security(tickerid, '60', spread)

The `security` function returns a series which is then adapted to the time scale of the current chart's symbol. This result can be either shown directly on the chart (i.e., with `plot`), or used in further calculations. The "Advance Decline Ratio" script illustrates a more involved use of `security`:

//@version=4 study(title = "Advance Decline Ratio", shorttitle="ADR") ratio(t1, t2, source) => s1 = security(t1, timeframe.period, source) s2 = security(t2, timeframe.period, source) s1 / s2 plot(ratio("USI:ADVN.NY", "USI:DECL.NY", close))

The script requests two additional securities. The results of the requests are then used in an arithmetic formula. As a result, we have a stock market indicator used by investors to measure the number of individual stocks participating in an upward or downward trend.

Barmerge gaps and lookahead

There are two switches that define how data requested with `security` will be mapped to the current timeframe.

The first one, `gaps`, controls gaps in data. With the default value barmerge.gaps_off, data is merged continuously (without `na` gaps). All gaps, if any, are filled with the previous nearest non-`na` value. If barmerge.gaps_on is used, then merged data may contain gaps in the form of `na` values.

The second switch, `lookahead`, was added in Pine Script version 3. The parameter has two possible values: barmerge.lookahead_off and barmerge.lookahead_on to respectively switch between the new, default behavior of security, and the old behavior dating from Pine v1 and v2.

This example shows the difference on a *5 minutes* chart:

//@version=4 study('My Script', overlay=true) a = security(syminfo.tickerid, '60', low, lookahead=barmerge.lookahead_off) plot(a, color=color.red) b = security(syminfo.tickerid, '60', low, lookahead=barmerge.lookahead_on) plot(b, color=color.lime)

The green line on the chart is the *low* price of an hourly bar that is requested with *lookahead on*. It's the old behavior of the security function. The green line based on historical data is displayed at the price level of an hourly *low* right after a new hourly bar is created (dotted blue vertical lines).

The red line is a *low* price of an hourly bar that is requested with *lookahead off*. In this case the requested *low* price of an hourly historical bar will be given only on the last minute bar of the requested hour, when an hourly bar's *low* won't return future data.

The fuchsia dotted line represents the beginning of real-time data. You can see that `barmerge.lookahead_on` and `barmerge.lookahead_off` behave the same way on real-time data, i.e., as `barmerge.lookahead_off` does.

Understanding lookahead

There are many published scripts using the following code:

//@version=2 //... a = security(tickerid, 'D', close[1]) // It's barmerge.lookahead_on, because version is 2

In this case the `close[1]` expression fetches the `close` of the previous day, so the construction **does not use future data**.

In Pine v3 or later, we can rewrite this in two different ways, using `barmerge.lookahead_on` or `barmerge.lookahead_off`. If you use `barmerge.lookahead_on`, then it's quite simple:

//@version=4 //... a = security(syminfo.tickerid, 'D', close[1], lookahead=barmerge.lookahead_on)

Because the original construction doesn't use future data, it is possible to rewrite it using `barmerge.lookahead_off`. If you use `barmerge.lookahead_off`, the script is more complex but shows how the lookahead parameter works:

//@version=4 //... indexHighTF = barstate.isrealtime ? 1 : 0 indexCurrTF = barstate.isrealtime ? 0 : 1 a0 = security(syminfo.tickerid, 'D', close[indexHighTF], lookahead=barmerge.lookahead_off) a = a0[indexCurrTF]

When an indicator is based on historical data (i.e., `barstate.isrealtime` is `false`), we take the current *close* of the daily resolution and shift the result of `security` function call one bar to the right in the current resolution. When an indicator is calculated on real-time data, we take the *close* of the previous day without shifting the `security` data.

Requesting data of a lower timeframe

`security` function was designed to request data of a timeframe *higher* than the current chart timeframe. On a *60 minutes* chart, this would mean requesting 240, D, W, or any higher timeframe.

It is not recommended to request data of a timeframe *lower* that the current chart timeframe, for example *1 minute* data from a *5 minutes* chart. The main problem with such a case is that some part of a 1 minute data will be inevitably lost, as it's impossible to display it on a *5 minutes* chart and not to break the time axis. In such cases the behavior of `security` can be rather unexpected. The next example illustrates this:

// Add this script on a "5" minute chart //@version=4 study("Lookahead On/Off", overlay=true, precision=5) l_on = security(syminfo.tickerid, "1", close, lookahead=barmerge.lookahead_on) l_off = security(syminfo.tickerid, "1", close, lookahead=barmerge.lookahead_off) plot(l_on, color=color.red) plot(l_off, color=color.blue)

This study plots two lines which correspond to different values of the `lookahead` parameter. The red line shows data returned by `security` with `lookahead=barmerge.lookahead_on`. The blue line with `lookahead=barmerge.lookahead_off`. Let's look at the *5 minutes* bar starting at 07:50. The red line at this bar has a value of 1.13151 which corresponds to the value of *the first of the five 1 minute bars* that fall into the time range 07:50 - 07:54. On the other hand, the blue line at the same bar has a value of 1.13121 which corresponds to *the last of the five 1 minute bars* of the same time range.

Footnotes

Footnotes

1. Actually the highest supported minute resolution is "1440" (which is the number of minutes in 24 hours). ↩
2. Requesting data of `"1h"` or `"1H"` resolution would result in an error. Use `"60"` instead. ↩

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Drawings

Starting with Pine v4, indicators and strategies can create *drawing objects* on the chart. Three types of drawings are currently supported: *label*, *line*, and *boxes*. You will find one instance of each on the following chart:

> NoticeOn TradingView charts, a complete set of *Drawing Tools* allows users to create and modify drawings using mouse actions. While they may look similar to drawing objects created with Pine code, they are essentially different entities. Drawing objects created using Pine code cannot be modified with mouse actions.

The line, label, and box drawings in Pine v4 allow you to create indicators with more sophisticated visual components, e.g., pivot points, support/resistance levels, zig zag lines, labels containing dynamic text, etc.

In contrast to indicator plots (plots are created with functions `plot`, `plotshape`, `plotchar`), drawing objects can be created on historical bars as well as in the future, where no bars exist yet.

Creating drawings

Pine drawing objects are created with the label.new , line.new and box.new functions. While each function has many parameters, only the coordinates are mandatory. This is an example of code used to create a label on every bar:

//@version=4 study("My Script", overlay=true) label.new(bar_index, high)

The label is created with the parameters `x=bar_index` (the index of the current bar, bar_index) and `y=high` (high price of the current bar). When a new bar opens, a new label is created on it. Label objects created on previous bars stay on the chart until the indicator deletes them with an explicit call of the label.delete function, or until the automatic garbage collection process removes them.

Here is a modified version of the same script that shows the values of the `x` and `y` coordinates used to create the labels:

//@version=4 study("My Script", overlay=true) label.new(bar_index, high, style=label.style_none, text="x=" + tostring(bar_index) + "\ny=" + tostring(high))

In this example labels are shown without background coloring (because of parameter `style=label.style_none`) but with dynamically created text (`text="x=" + tostring(bar_index) + "\ny=" + tostring(high)`) that prints label coordinates.

This is an example of code that creates line objects on a chart:

//@version=4 study("My Script", overlay=true) line.new(x1=bar_index[1], y1=low[1], x2=bar_index, y2=high)

This is an example of code that creates box objects on a chart:

//@version=4 study("My Script", overlay=true) box.new(left=bar_index[1], top=low[1], right=bar_index, bottom=high)

Calculation of drawings on bar updates

Drawing objects are subject to both *commit* and *rollback* actions, which affect the behavior of a script when it executes in the realtime bar, Execution model.

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

//@version=4 study("My Script", overlay=true) label.new(bar_index, high)

While `label.new` creates a new label on every iteration of the script when price changes in the realtime bar, the most recent label created in the script's previous iteration is also automatically deleted because of rollback before the next iteration. Only the last label created before the realtime bar's close will be committed, and will thus persist.

Coordinates

Drawing objects are positioned on the chart according to *x* and *y* coordinates using a combination of 4 parameters: `x`, `y`, `xloc` and `yloc`. The value of `xloc` determines whether `x` will hold a bar index or time value. When `yloc=yloc.price`, `y` holds a price. `y` is ignored when `yloc` is set to yloc.abovebar or yloc.belowbar.

If a drawing object uses xloc.bar_index, then the x-coordinate is treated as an absolute bar index. The bar index of the current bar can be obtained from the built-in variable `bar_index`. The bar index of previous bars is `bar_index[1]`, `bar_index[2]` and so on. `xloc.bar_index` is the default value for x-location parameters of both label and line drawings.

If a drawing object uses xloc.bar_time, then the x-coordinate is treated as a UNIX time in milliseconds. The start time of the current bar can be obtained from the built-in variable `time`. The bar time of previous bars is `time[1]`, `time[2]` and so on. Time can also be set to an absolute time point with the timestamp function.

The `xloc.bar_time` and `xloc.bar_index` modes makes it possible to place a drawing object in the future, to the right of the current bar. For example:

//@version=4 study("My Script", overlay=true) dt = time - time[1] if barstate.islast label.new(time + 3*dt, close, xloc=xloc.bar_time)

This code places a label object in the future. X-location logic works identically for label, line, and box drawings.

Example for `xloc.bar_index`:

//@version=4 study("My Script", overlay=true) label.new(bar_index+100, high)

In contrast, y-location logic is different for label and line or box drawings. Pine's *line* and *box* drawings always use yloc.price, so their y-coordinate is always treated as an absolute price value.

Label drawings have additional y-location values: yloc.abovebar and yloc.belowbar. When they are used, the value of the `y` parameter is ignored and the drawing object is placed above or below the bar.

Modifying drawings

A drawing object can be modified after its creation. The `label.new`, `line.new`, and `box.new` functions return a reference to the created drawing object (of type *series label*, *series line* and *series box* respectively). This reference can then be used as the first argument to the `label.set_*`, `line.set_*`, or `box.set_*` functions used to modify drawings. For example:

//@version=4 study("My Script", overlay=true) l = label.new(bar_index, na) if close >= open label.set_text(l, "green") label.set_color(l, color.green) label.set_yloc(l, yloc.belowbar) label.set_style(l, label.style_label_up) else label.set_text(l, "red") label.set_color(l, color.red) label.set_yloc(l, yloc.abovebar) label.set_style(l, label.style_label_down)

This simple script first creates a label on the current bar and then it writes a reference to it in a variable `l`. Then, depending on whether the current bar is rising or falling (condition `close >= open`), a number of label drawing properties are modified: text, color, *y* coordinate location (`yloc`) and label style.

One may notice that `na` is passed as the `y` argument to the `label.new` function call. The reason for this is that the example's label uses either `yloc.belowbar` or `yloc.abovebar` y-locations, which don't require a y value. A finite value for `y` is needed only if a label uses `yloc.price`.

The available *setter* functions for label drawings are:

- label.set_color — changes color of label
- label.set_size — changes size of label
- label.set_style — changes style of label
- label.set_text — changes text of label
- label.set_textcolor — changes color of text
- label.set_x — changes x-coordinate of label
- label.set_y — changes y-coordinate of label
- label.set_xy — changes both x and y coordinates of label
- label.set_xloc — changes x-location of label
- label.set_yloc — changes y-location of label
- label.set_tooltip — changes tooltip of label

The available *setter* functions for line drawings are:

- line.set_color — changes color of line
- line.set_extend — changes attribute that makes:
  - `extend.none` - a line segment
  - `extend.left`/`extend.right` - a ray
  - `extend.both` - an endless line
- line.set_style — changes [style of line](drawings_label_styles.interpreted-text role“ref”)
- line.set_width — changes width of line
- line.set_xloc — changes x-location of line (both x1 and x2)
- line.set_x1 — changes x1-coordinate of line
- line.set_y1 — changes y1-coordinate of line
- line.set_xy1 — changes both x1 and y1 coordinates of line
- line.set_x2 — changes x2-coordinate of line
- line.set_y2 — changes y2-coordinate of line
- line.set_xy2 — changes both x2 and y2 coordinates of line at once

The available *setter* functions for box drawings are:

- box.set_border_color — changes border color of the box
- box.set_bgcolor — changes background color of the box
- box.set_extend — changes attribute that makes:
  - `extend.none` - the horizontal borders start at the left border and end at the right border
  - `extend.left`/`extend.right` - the horizontal borders are extended indefinitely to the left/right of the box
  - `extend.both` - the horizontal borders are extended on both sides
- box.set_border_style — changes border style of the box
- box.set_border_width — changes border width of the box
- box.set_bottom — changes bottom coordinate of the box
- box.set_right — changes right coordinate of the box
- box.set_rightbottom — changes both right and bottom coordinates of the box at once
- box.set_top — changes top coordinate of the box
- box.set_left — changes left coordinate of the box
- box.set_lefttop — changes both left and top coordinates of the box at once

Label styles

Various styles can be applied to labels with either the label.new or label.set_style function:

Label style name	Label	Label with text
`label.style_none`		
`label.style_xcross`		
`label.style_cross`		
`label.style_triangleup`		
`label.style_triangledown`		
`label.style_flag`		
`label.style_circle`		
`label.style_arrowup`		
`label.style_arrowdown`		
`label.style_label_up`		
`label.style_label_down`		
`label.style_square`		
`label.style_diamond`		

Line and box styles

Various styles can be applied to lines with either the line.new, box.new, line.set_style or box.set_border_style function:

Line style name	Line	Box
`line.style_solid`		
`line.style_dotted`		

Line style name	Line	Box
`line.style_dashed`		
`line.style_arrow_left`		Not supported
`line.style_arrow_right`		Not supported
`line.style_arrow_both`		Not supported

Deleting drawings

The label.delete, line.delete and box.delete functions delete *label*, *line*, or *box* drawing objects from the chart.

Here is Pine code that keeps just one label drawing object on the current bar, *deleting the old ones*:

//@version=4 study("Last Bar Close 1", overlay=true) c = close >= open ? color.lime : color.red l = label.new(bar_index, na, text=tostring(close), color=c, style=label.style_label_down, yloc=yloc.abovebar) label.delete(l[1])

On every new bar update of the "Last Bar Close 1" study, a new label object is created and written to variable `l`. Variable `l` is of type *series label*, so the `[]` operator is used to get the previous bar's label object. That previous label is then passed to the `label.delete` function to delete it.

Functions `label.delete` and `line.delete` do nothing if the `na` value is used as an id, which makes code like the following unnecessary:

if not na(l[1]) label.delete(l[1])

The previous script's behavior can be reproduced using another approach:

//@version=4 study("Last Bar Close 2", overlay=true) var label l = na label.delete(l) c = close >= open ? color.lime : color.red l := label.new(bar_index, na, text=tostring(close), color=c, style=label.style_label_down, yloc=yloc.abovebar)

When the study "Last Bar Close 2" gets a new bar update, variable `l` is still referencing the old label object created on the previous bar. This label is deleted with the `label.delete(l)` call. A new label is then created and its id saved to `l`. Using this approach there is no need to use the `[]` operator.

Note the use of the new Pine v4 var keyword. It creates variable `l` and initializes it with the `na` value only once. `label.delete(l)` would have no object to delete if it weren't for the fact that `l` is initialized only once.

There is yet another way to achieve the same objective as in the two previous scripts, this time by modifying the label rather than deleting it:

//@version=4 study("Last Bar Close 3", overlay=true) var label l = label.new(bar_index, na, style=label.style_label_down, yloc=yloc.abovebar) c = close >= open ? color.lime : color.red label.set_color(l, c) label.set_text(l, tostring(close)) label.set_x(l, bar_index)

Once again, the use of new var keyword is essential. It is what allows the `label.new` call to be executed only once, on the very first historical bar.

Examples of classic indicators

Pivot Points Standard

//@version=4 study("Pivot Points Standard", overlay=true) higherTF = input("D", type=input.resolution) prevCloseHTF = security(syminfo.tickerid, higherTF, close[1], lookahead=true) prevOpenHTF = security(syminfo.tickerid, higherTF, open[1], lookahead=true) prevHighHTF = security(syminfo.tickerid, higherTF, high[1], lookahead=true) prevLowHTF = security(syminfo.tickerid, higherTF, low[1], lookahead=true) pLevel = (prevHighHTF + prevLowHTF + prevCloseHTF) / 3 r1Level = pLevel * 2 - prevLowHTF s1Level = pLevel * 2 - prevHighHTF var line r1Line = na var line pLine = na var line s1Line = na if pLevel[1] != pLevel line.set_x2(r1Line, bar_index) line.set_x2(pLine, bar_index) line.set_x2(s1Line, bar_index) line.set_extend(r1Line, extend.none) line.set_extend(pLine, extend.none) line.set_extend(s1Line, extend.none) r1Line := line.new(bar_index, r1Level, bar_index, r1Level, extend=extend.right) pLine := line.new(bar_index, pLevel, bar_index, pLevel, width=3, extend=extend.right) s1Line := line.new(bar_index, s1Level, bar_index, s1Level, extend=extend.right) label.new(bar_index, r1Level, "R1", style=label.style_none) label.new(bar_index, pLevel, "P", style=label.style_none) label.new(bar_index, s1Level, "S1", style=label.style_none) if not na(pLine) and line.get_x2(pLine) != bar_index line.set_x2(r1Line, bar_index) line.set_x2(pLine, bar_index) line.set_x2(s1Line, bar_index)

Pivot Points High/Low

//@version=4 study("Pivot Points High Low", shorttitle="Pivots HL", overlay=true) lenH = input(title="Length High", type=input.integer, defval=10, minval=1) lenL = input(title="Length Low", type=input.integer, defval=10, minval=1) fun(src, len, isHigh, _style, _yloc, _color) => p = nz(src[len]) isFound = true for i = 0 to len - 1 if isHigh and src[i] > p isFound := false if not isHigh and src[i] < p isFound := false for i = len + 1 to 2 * len if isHigh and src[i] >= p isFound := false if not isHigh and src[i] <= p isFound := false if isFound label.new(bar_index[len], p, tostring(p), style=_style, yloc=_yloc, color=_color) fun(high, lenH, true, label.style_label_down, yloc.abovebar, color.lime) fun(low, lenL, false, label.style_label_up, yloc.belowbar, color.red)

Linear Regression

//@version=4 study("Linear Regression", shorttitle="LinReg", overlay=true) upperMult = input(title="Upper Deviation", defval=2) lowerMult = input(title="Lower Deviation", defval=-2) useUpperDev = input(title="Use Upper Deviation", defval=true) useLowerDev = input(title="Use Lower Deviation", defval=true) showPearson = input(title="Show Pearson's R", defval=true) extendLines = input(title="Extend Lines", defval=false) len = input(title="Count", defval=100) src = input(title="Source", defval=close) extend = extendLines ? extend.right : extend.none calcSlope(src, len) => if not barstate.islast or len <= 1 [float(na), float(na), float(na)] else sumX = 0.0 sumY = 0.0 sumXSqr = 0.0 sumXY = 0.0 for i = 0 to len - 1 val = src[i] per = i + 1.0 sumX := sumX + per sumY := sumY + val sumXSqr := sumXSqr + per * per sumXY := sumXY + val * per slope = (len * sumXY - sumX * sumY) / (len * sumXSqr - sumX * sumX) average = sumY / len intercept = average - slope * sumX / len + slope [slope, average, intercept] [s, a, i] = calcSlope(src, len) startPrice = i + s * (len - 1) endPrice = i var line baseLine = na if na(baseLine) and not na(startPrice) baseLine := line.new(bar_index - len + 1, startPrice, bar_index, endPrice, width=1, extend=extend, color=color.red) else line.set_xy1(baseLine, bar_index - len + 1, startPrice) line.set_xy2(baseLine, bar_index, endPrice) na calcDev(src, len, slope, average, intercept) => upDev = 0.0 dnDev = 0.0 stdDevAcc = 0.0 dsxx = 0.0 dsyy = 0.0 dsxy = 0.0 periods = len - 1 daY = intercept + (slope * periods) / 2 val = intercept for i = 0 to periods price = high[i] - val if (price > upDev) upDev := price price := val - low[i] if (price > dnDev) dnDev := price price :=

src[i] dxt = price - average dyt = val - daY price := price - val stdDevAcc := stdDevAcc + price * price dsxx := dsxx + dxt * dxt dsyy := dsyy + dyt * dyt dsxy := dsxy + dxt * dyt val := val + slope stdDev = sqrt(stdDevAcc / (periods == 0 ? 1 : periods)) pearsonR = dsxx == 0 or dsyy == 0 ? 0 : dsxy / sqrt(dsxx * dsyy) [stdDev, pearsonR, upDev, dnDev] [stdDev, pearsonR, upDev, dnDev] = calcDev(src, len, s, a, i) upperStartPrice = startPrice + (useUpperDev ? upperMult * stdDev : upDev) upperEndPrice = endPrice + (useUpperDev ? upperMult * stdDev : upDev) var line upper = na lowerStartPrice = startPrice + (useLowerDev ? lowerMult * stdDev : -dnDev) lowerEndPrice = endPrice + (useLowerDev ? lowerMult * stdDev : -dnDev) var line lower = na if na(upper) and not na(upperStartPrice) upper := line.new(bar_index - len + 1, upperStartPrice, bar_index, upperEndPrice, width=1, extend=extend, color=#0000ff) else line.set_xy1(upper, bar_index - len + 1, upperStartPrice) line.set_xy2(upper, bar_index, upperEndPrice) na if na(lower) and not na(lowerStartPrice) lower := line.new(bar_index - len + 1, lowerStartPrice, bar_index, lowerEndPrice, width=1, extend=extend, color=#0000ff) else line.set_xy1(lower, bar_index - len + 1, lowerStartPrice) line.set_xy2(lower, bar_index, lowerEndPrice) na // Pearson's R var label r = na transparent = color.new(color.white, 100) label.delete(r[1]) if showPearson and not na(pearsonR) r := label.new(bar_index - len + 1, lowerStartPrice, tostring(pearsonR, "#.################"), color=transparent, textcolor=#0000ff, size=size.normal, style=label.style_label_up)

Zig Zag

//@version=4 study("Zig Zag", overlay=true) dev_threshold = input(title="Deviation (%)", type=input.float, defval=5, minval=1, maxval=100) depth = input(title="Depth", type=input.integer, defval=10, minval=1) pivots(src, length, isHigh) => p = nz(src[length]) if length == 0 [bar_index, p] else isFound = true for i = 0 to length - 1 if isHigh and src[i] > p isFound := false if not isHigh and src[i] < p isFound := false for i = length + 1 to 2 * length if isHigh and src[i] >= p isFound := false if not isHigh and src[i] <= p isFound := false if isFound and length * 2 <= bar_index [bar_index[length], p] else [int(na), float(na)] [iH, pH] = pivots(high, floor(depth / 2), true) [iL, pL] = pivots(low, floor(depth / 2), false) calc_dev(base_price, price) => 100 * (price - base_price) / base_price var line lineLast = na var int iLast = 0 var float pLast = 0 var bool isHighLast = true // otherwise the last pivot is a low pivot var int linesCount = 0 pivotFound(dev, isHigh, index, price) => if isHighLast == isHigh and not na(lineLast) // same direction if isHighLast ? price > pLast : price < pLast if linesCount <= 1 line.set_xy1(lineLast, index, price) line.set_xy2(lineLast, index, price) [lineLast, isHighLast, false] else [line(na), bool(na), false] else // reverse the direction (or create the very first line) if na(lineLast) id = line.new(index, price, index, price, color=color.red, width=2) [id, isHigh, true] else // price move is significant if abs(dev) >= dev_threshold id = line.new(iLast, pLast, index, price, color=color.red, width=2) [id, isHigh, true] else [line(na), bool(na), false] if not na(iH) and not na(iL) and iH == iL dev1 = calc_dev(pLast, pH) [id2, isHigh2, isNew2] = pivotFound(dev1, true, iH, pH) if isNew2 linesCount := linesCount + 1 if not na(id2) lineLast := id2 isHighLast := isHigh2 iLast := iH pLast := pH dev2 = calc_dev(pLast, pL) [id1, isHigh1, isNew1] = pivotFound(dev2, false, iL, pL) if isNew1 linesCount := linesCount + 1 if not na(id1) lineLast := id1 isHighLast := isHigh1 iLast := iL pLast := pL else if not na(iH) dev1 = calc_dev(pLast, pH) [id, isHigh, isNew] = pivotFound(dev1, true, iH, pH) if isNew linesCount := linesCount + 1 if not na(id) lineLast := id isHighLast := isHigh iLast := iH pLast := pH else if not na(iL) dev2 = calc_dev(pLast, pL) [id, isHigh, isNew] = pivotFound(dev2, false, iL, pL) if isNew linesCount := linesCount + 1 if not na(id) lineLast := id isHighLast := isHigh iLast := iL pLast := pL

Limits

Total number of drawings

Drawing objects consume server resources, which is why there is a limit to the total number of drawings per study or strategy. When too many drawings are created, old ones are automatically deleted by the Pine runtime, in a process referred to as *garbage collection*.

This code creates a drawing on every bar:

//@version=4 study("My Script", overlay=true) label.new(bar_index, high)

Scrolling the chart left, one will see there are no drawings after approximately 50 bars:

You can change the drawing limit to a value in range from 1 to 500 using the max_lines_count, max_labels_count, or max_boxes_count parameters for the study and strategy functions:

//@version=4 study("My Script", overlay=true, max_labels_count=100) label.new(bar_index, high)

Bars count in future with xloc.bar_index

Objects positioned using xloc.bar_index cannot be drawn further than 500 bars into the future.

Additional securities

Pine code sometimes uses additional symbols and/or timeframes with the security function. Drawing functions can only be used in the main symbol's context.

max_bars_back of time

Use of `barstate.isrealtime` in combination with drawings may sometimes produce unexpected results. This code's intention, for example, is to ignore all historical bars and create a label drawing on the *realtime* bar:

//@version=4 study("My Script", overlay=true) if barstate.isrealtime label.new(bar_index[10], na, text="Label", yloc=yloc.abovebar)

It will, however, fail at runtime. The reason for the error is that Pine cannot determine the buffer size for historical values of the `time` plot, even though the `time` built-in variable isn't mentioned in the code. This is due to the fact that the built-in variable `bar_index` uses the `time` series in its inner workings. Accessing the value of the bar index 10 bars back requires that the history buffer size of the `time` series be of size 10 or more.

In Pine, there is a mechanism that automaticaly detects the required historical buffer size for most cases. Autodetection works by letting Pine code access historical values any number of bars back for a limited duration. In this script's case, the `if barstate.isrealtime` condition prevents any such accesses to occur, so the required historical buffer size cannot be inferred and the code fails.

The solution to this conundrum is to use the max_bars_back function to explicitly set the historical buffer size for the `time` series:

//@version=4 study("My Script", overlay=true) max_bars_back(time, 10) if barstate.isrealtime label.new(bar_index[10], na, text="Label", yloc=yloc.abovebar)

Such occurrences are confusing, but rare. In time, the Pine team hopes to eliminate them.

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Extended and regular sessions

On TradingView you can access extended hours sessions by *right-clicking* on a chart and choosing *Settings Symbol Extended Hours (Intraday only)*. There are two types of sessions: *regular* (excluding pre- and post-market data) and *extended* (including pre- and post-market data). Pine scripts may request additional session data using the security function.

The `security` function can be called with a symbol name (*"EXCHANGE_PREFIX:TICKER"*, e.g., "BATS:AAPL") as its first argument. Used this way, the `security` function will return data for the regular session. For example:

//@version=4 study("Example 1: Regular Session Data") cc = security("BATS:AAPL", timeframe.period, close, true) plot(cc, style=plot.style_linebr)

If you want the `security` call to return extended session data, you must first use the tickerid function to build `security` call's first argument:

//@version=4 study(“Example 2: Extended Session Data”) t = tickerid(“BATS”, “AAPL”, session.extended) cc = security(t, timeframe.period, close, true) plot(cc, style=plot.style_linebr)

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

The first argument of the `tickerid` function is an exchange prefix ("BATS") and the second argument is a ticker ("AAPL"). The third argument specifies the type of the session (`session.extended` or `session.regular`). So *Example 1* could be rewritten as:

//@version=4 study("Example 3: Regular Session Data (using tickerid)") t = tickerid("BATS", "AAPL", session.regular) cc = security("BATS:AAPL", timeframe.period, close, true) plot(cc, style=plot.style_linebr)

If you want to request the same session specification used for the chart's main symbol, omit the third argument; it is optional. Or, if you want your code to explicitly declare your intention, use the `syminfo.session` built-in variable as the third argument to `tickerid` function, as it holds the session type of the chart's main symbol:

//@version=4 study("Example 4: Same as Main Symbol Session Type Data") t = tickerid("BATS", "AAPL", syminfo.session) cc = security(t, timeframe.period, close, true) plot(cc, style=plot.style_linebr)

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Indicator repainting

Historical data does not include records of intra-bar movements of price; only open, high, low and close (OHLC). This leads to a script sometimes working differently on historical data and in real-time, where only the open price is known and where price will typically move many times before the real-time bar's final high, low and close values are set after the real-time bar closes.

If we add a script on a chart, wait until it calculates on a number of real-time bars and then reload the page, we will sometimes see a script's plots change slightly. This behavior is one of a few different types of behaviors commonly referred to as *indicator repainting*. It is the type of repainting we are concerned with here and which we will refer to when using *repainting*. It is due to the fact that when certain features are used in scripts, they will calculate differently on historical and real-time bars.

Other types of behavior rightly or wrongly referred to as *repainting* include plotting with a negative offset on past bars and using otherwise unavailable future information received through misunderstood calls to the `security` function, which can introduce data not available in real-time into script calculations.

Not all indicators are subject to the type of repainting we discuss here. In most cases it depends on whether or not certain functions or language constructs are used in the code. Please note that this repainting effect is **not** a bug; it is the result of the inherent differences between historic bars and real-time bar information on TradingView.

We can see repainting in the following cases:

1. Strategies using `calc_on_every_tick=true`. A strategy with parameter `calc_on_every_tick = false` may also be prone to repainting, but to a lesser degree.

2. Using `security` for requesting data from a resolution *higher* than the resolution of the chart's main symbol:

// Add this study on 1 minute chart //@version=4 study("My Script") c = security(syminfo.tickerid, "5", close) plot(close) plot(c, color=color.red)

This study will calculate differently on real-time and historical data, regardless of `lookahead` parameter's value (see `understanding_lookahead`).

3. Using `security` to request data from a resolution *lower* than the resolution of chart's main symbol (more information `here <requesting_data_of_a_lower_timeframe>`). If `lookahead=false`, repainting will occur. When `lookahead=true`, repainting is less probable. It may still happen when 1 and 5 minute updates outrun each other.

4. All scripts which calculations depending on a *starting point*. Intraday data gets aligned to the beginning of the week, month or year, depending on the resolution. Due to this, the results produced by such scripts can differ from time to time. These are cases where scripts will be relying on a starting point:

    - when they use valuewhen(), barssince() or ema() functions (due to peculiarities in their algorithm).
    - any backtesting strategy (regardless of how the `calc_on_every_tick` parameter is defined).

    There is a dependency between the resolution and the alignment of a starting point:

- 1--14 minutes - aligns to the beginning of a week.
- 15--29 minutes - aligns to the beginning of a month.
- from 30 minutes and higher - aligns to the beginning of a year.

The following limitations of history lengths are taken into account when processing the data:

- 10000 historical bars for all Pro plans.
- 5000 historical bars for other plans.

5. Changes in historical data, for example, due to a *split*.

6. Presence of the following variables in the script usually leads to repainting:

   - barstate.isconfirmed, barstate.isfirst, barstate.ishistory, barstate.islast, barstate.isnew, barstate.isrealtime;
   - timenow;
   - bar_index.

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Non-standard chart types data

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 `heikinashi`, `renko`, `linebreak`, `kagi` and `pointfigure`. All of them work in the same manner; they create a special ticker identifier to be used as the first argument in a `security` function call.

heikinashi function

Heikin-Ashi means *average bar* in Japanese. The open, high, low and close prices of Heikin-Ashi candlesticks are synthetic; they are not actual prices. Each value is calculated using combinations of normal OHLC values from the current and previous bar. The calculations used make Heikin-Ashi bars less noisy than normal candlesticks.

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

This script requests low prices of Heikin-Ashi bars and plots them on top of the usual candlesticks:

//@version=4 study("Example 5", overlay=true) ha_t = heikinashi(syminfo.tickerid) ha_low = security(ha_t, timeframe.period, low) plot(ha_low)

Note that the low prices of Heikin-Ashi bars are different from the low prices of the normal candlesticks.

If you wanted to switch off extended hours data in *Example 5*, you would need to use the `tickerid` function first, instead of using the `syminfo.tickerid` variable directly:

//@version=4 study("Example 6", overlay=true) t = tickerid(syminfo.prefix, syminfo.ticker, session.regular) ha_t = heikinashi(t) ha_low = security(ha_t, timeframe.period, low, gaps=barmerge.gaps_on) plot(ha_low, style=plot.style_linebr)

Note that we use an additional fourth parameter with `security`: `gaps=barmerge.gaps_on`, which instructs the function not to use previous values to fill slots where data is absent. This means we will get empty areas during extended hours. To be able to see this on the chart, we also need to use a special plot style (`style=plot.style_linebr`), the *Line With Breaks* style.

You may plot Heikin-Ashi bars from a script so they look exactly like a chart's Heikin-Ashi bars:

//@version=4 study("Example 6.1") ha_t = heikinashi(syminfo.tickerid) ha_open = security(ha_t, timeframe.period, open) ha_high = security(ha_t, timeframe.period, high) ha_low = security(ha_t, timeframe.period, low) ha_close = security(ha_t, timeframe.period, close) palette = ha_close >= ha_open ? color.green : color.red plotcandle(ha_open, ha_high, ha_low, ha_close, color=palette)

You will find more information on the plotcandle() and plotbar() functions in the Custom OHLC bars and candles section.

Renko function

Renko bars only plot price movements, without taking time or volume into consideration. They are constructed from ticks and look like bricks stacked in adjacent columns[1]. A new brick is drawn after the price

passes the top or bottom by a predetermined amount.

//@version=4 study("Example 7", overlay=true) renko_t = renko(syminfo.tickerid, "ATR", 10) renko_low = security(renko_t, timeframe.period, low) plot(renko_low)

Please note that you cannot plot Renko bricks from Pine script exactly as they look. You can only get a series of numbers similar to OHLC values for Renko bars and use them in your algorithms.

For detailed information, see renko().

Linebreak function

The *Line Break* chart type displays a series of vertical boxes that are based on price changes[^2].

//@version=4 study("Example 8", overlay=true) lb_t = linebreak(syminfo.tickerid, 3) lb_close = security(lb_t, timeframe.period, close) plot(lb_close)

Please note that you cannot plot Line Break boxes from Pine script exactly as they look. You can only get a series of numbers similar to OHLC values for Line Break charts and use them in your algorithms.

For detailed information, see linebreak().

Kagi function

Kagi charts are made of a continuous line that changes directions. The direction changes when the price changes[^3] beyond a predetermined amount.

//@version=4 study("Example 9", overlay=true) kagi_t = kagi(syminfo.tickerid, 1) kagi_close = security(kagi_t, timeframe.period, close) plot(kagi_close)

Please note that you cannot plot Kagi lines from Pine script exactly as they look. You can only get a series of numbers similar to OHLC values for Kagi charts and use them in your algorithms.

For detailed information, see kagi().

Pointfigure function

Point and Figure (PnF) charts only plot price movements[^4], without taking time into consideration. A column of X's is plotted as the price rises, and O's are plotted when price drops.

Please note that you cannot plot PnF X's and O's from Pine script exactly as they look. You can only get a series of numbers that are similar to OHLC values for PnF charts and use them in your algorithms. Every column of X's or O's is represented with four numbers. You may think of them as synthetic OHLC PnF values.

//@version=4 study("Example 10", overlay=true) pnf_t = pointfigure(syminfo.tickerid, "hl", "ATR", 14, 3) pnf_open = security(pnf_t, timeframe.period, open, true) pnf_close = security(pnf_t, timeframe.period, close, true) plot(pnf_open, color=color.green, style=plot.style_linebr, linewidth=4) plot(pnf_close, color=color.red, style=plot.style_linebr, linewidth=4)

For detailed information, see pointfigure().

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. ↩

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Session and time information

The "time" function and variable

Pine provides means to work with trade session, time and date information. On this IBM chart at *30 minutes*, two scripts are running: "Bar date/time" and "Session bars".

This is the "Bar date/time" script:

```
//@version=4 study("Bar date/time") plot(time)
```

The time variable returns the date/time (timestamp) of each bar's opening time in UNIX format[1] and in the exchange's timezone. As can be seen from the screenshot, the `time` value on the last bar is equal to 1397593800000. This value is the number of *milliseconds* that have passed since 00:00:00 UTC, 1 January, 1970 and corresponds to Tuesday, 15th of April, 2014 at 20:30:00 UTC. The chart's time gauge in the screenshot shows the time of the last bar as 2014-04-15 16:30 because it has a 4-hour difference between the exchange's timezone, which is the default time returned by the `time` function.

The second script is "Session bars":

```
//@version=4 study("Session bars") t = time(timeframe.period, "0930-1600") plot(na(t) ? 0 : 1)
```

This shows how the user can distinguish between regular session and extended hours bars by using the built-in time function rather than the `time` variable. Note that the background behind these bars is colored because of the chart's settings; not because of the script. The `time` function returns the time of the bar's start in milliseconds UNIX time, or `na` if the bar is located outside the given trade session (09:30--16:00 in our example). The `time` function accepts two arguments: the `resolution` used to determine the timestamp of bars and `session`, the session specification in the form of a string containing the beginning and end of the trade session in the exchange's timezone. The string "0930-1600" corresponds to the trade session of the IBM symbol. These are examples of trade session specifications:

0000-0000

A monday to friday 24-hour session beginning at midnight.

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 Monday through Friday.

2000-1630:1234567

An overnight session that begins at 20:00 and ends at 16:30 the next day.

0930-1700:146

A session that begins at 9:30 and ends at 17:00 on Sundays (1), Wednesdays (4) and Fridays (6) (other days of the week are days off).

24x7

A complete 24-hour session beginning at 00:00 every day.

0000-0000:1234567

Same as "24x7".

0000-0000:23456

Same as previous example, but only Monday to Friday.

1700-1700:23456

An *overnight session*. Monday session starts Sunday at 17:00 and ends Monday at 17:00. Applies to Monday through Friday. Equivalent to 1700-1700

1000-1001:26

A weird session that lasts only one minute on Mondays (2) and one minute on Fridays (6).

Session specification used for the `time` function's second argument does not need to correspond to the symbol's real trade session. Hypothetical session specifications can be used to highlight other bars of a data series.

Pine provides an overloaded version of the `time` function which does not require custom session specification. This version of the function uses the regular session of a symbol. For example, it is possible to highlight the beginning of each half-hour bar on a minute chart in the following way:

//@version=4 study("new 30 min bar") is_newbar(res) => t = time(res) not na(t) and (na(t[1]) or t > t[1]) plot(is_newbar("30") ? 1 : 0)

The previous example's `is_newbar` custom function can be used in many situations. Here, we use it to display the market's opening high and low on an intraday chart:

//@version=4 study("Opening high/low", overlay=true) highTimeFrame = input("D", type=input.resolution) sessSpec = input("0930-1600", type=input.session) is_newbar(res, sess) => t = time(res, sess) na(t[1]) and not na(t) or t[1] < t newbar = is_newbar("1440", sessSpec) var float s1 = na var float s2 = na if newbar s1 := low s2 := high plot(s1, style=plot.style_circles, linewidth=3, color=color.red) plot(s2, style=plot.style_circles, linewidth=3, color=color.lime)

Pay attention to the variables `highTimeFrame` and `sessSpec`. They are defined using the input function and its `type` parameter to make their type explicit.

Built-in variables for working with time

Pine's standard library has an assortment of built-in variables and functions which make it possible to use time in the script's logic.

The most basic variables:

- time - UNIX time of the *current bar start* in milliseconds, UTC timezone.
- timenow - Current UNIX time in milliseconds, UTC timezone.
- syminfo.timezone - Exchange timezone of the chart main symbol series.

Variables that give information about the current bar start time:

- year - Current bar year.
- month - Current bar month.
- weekofyear - Week number of current bar.
- dayofmonth - Date of current bar.
- dayofweek - Day of week for current bar. You can use `sunday`, `monday`, `tuesday`, `wednesday`, `thursday`, `friday` and `saturday` variables for comparisons.
- hour - Hour of the current bar start time (in exchange timezone).
- minute - Minute of the current bar start time (in exchange timezone).
- second - Second of the current bar start time (in exchange timezone).

Functions for UNIX time "construction":

- year(t) - Returns year for provided UTC time `t`.
- month(t) - Returns month for provided UTC time `t`.
- weekofyear(t) - Returns week of year for provided UTC time `t`.
- dayofmonth(t) - Returns day of month for provided UTC time `t`.
- dayofweek(t) - Returns day of week for provided UTC time `t`.
- hour(t) - Returns hour for provided UTC time `t`.
- minute(t) - Returns minute for provided UTC time `t`.
- second(t) - Returns second for provided UTC time `t`.
- timestamp(year, month, day, hour, minute) -Returns UNIX time of specified date and time. Note, there is also an overloaded version with an additional `timezone` parameter.

All these variables and functions return time in the **exchange time zone**, except for the `time` and `timenow` variables which return time in **UTC timezone**.

Footnotes

1. UNIX time is measured in seconds. Pine Script uses UNIX time multiplied by 1000, so it's in millisecods. ↩

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Strategies

A *strategy* is a Pine script that can send, modify and cancel *buy/sell orders*. Strategies allow you to perform *backtesting* (emulation of a strategy trading on historical data) and *forwardtesting* (emulation of a strategy trading on real-time data) according to your algorithms.

A strategy written in Pine has many of the same capabilities as a Pine *study*, a.k.a. *indicator*. When you write a strategy, it must start with the strategy() annotation call (instead of `study`). Strategies may plot data, but they can also place, modify and cancel orders. They also have access to essential strategy performance information through specific keywords. The same information is available externally in the *Strategy Tester* tab. Once a strategy is calculated on historical data, you can see hypothetical order fills.

A simple strategy example

//@version=4 strategy("test") if bar_index < 100 strategy.entry("buy", strategy.long, 10, when=strategy.position_size <= 0) strategy.entry("sell", strategy.short, 10, when=strategy.position_size > 0) plot(strategy.equity)

As soon as the script is compiled and applied to a chart, you can see filled order marks on it and how your balance was changing during backtesting (*equity* curve). This is a very basic strategy that buys and sells on every bar.

The `strategy("test")` line states that the script is a strategy named "test". `strategy.entry()` is a command that can be used to send both "buy" and "sell" orders. `plot(strategy.equity)` plots the equity curve.

Applying a strategy to a chart

To test your strategy, apply it to the chart. Use the symbol and time intervals that you want to test. You can use a built-in strategy from the *Indicators & Strategies* dialog box, or write your own.

> NoticeWhen applying strategies to non-standard types of charts (Heikin Ashi, Renko, etc.), it is very important to realize that results will not reflect real market conditions. Orders on these types of charts will be executed at the synthetic price levels used on these charts, which often **do not reflect real market prices** and thus lead to unrealistic backtesting results. We therefore highly recommend using only standard chart types for backtesting strategies.

Backtesting and forwardtesting

On TradingView, strategies are calculated on all the chart's available historical data (*backtesting*), and then automatically continue calculations when real-time data comes in (*forwardtesting*).

By default, during both historical and real-time calculation, code is calculated on the bar's close.

When forwardtesting, you have the option of configuring script calculation to occur on every real-time tick. To enable this, check the *Recalculate On Every Tick* option in the strategy's *Settings/Properties*, or specify it in the script's code using: `strategy(..., calc_on_every_tick=true)`.

You can set the strategy to perform one additional calculation after an order is filled. For this you need to check *Recalculate After Order filled* in the strategy's *Settings/Properties*, or do it in the script's code using: `strategy(..., calc_on_order_fills=true)`.

Broker emulator

TradingView uses a *broker emulator* when running strategies. Unlike in real trading, the emulator only fills orders at chart prices, which is why an order can only be filled on the next tick in forwardtesting and on the next bar or later in backtesting, i.e., after the strategy calculates.

The following logic is used to emulate order fills:

1. If the bar's high is closer to bar's open than the bar's low, the broker emulator assumes that intrabar price was moving this way: open → high → low → close.
2. If the bar's low is closer to bar's open than the bar's high, the broker emulator assumes that intrabar price was moving this way: open → low → high → close.
3. The broker emulator assumes that there are no gaps inside bars, meaning the full range of intrabar prices is available for order execution.
4. Even if the *Recalculate On Every Tick* option is enabled in strategy properties (or the script's `strategy` call uses `calc_on_every_tick=true`), the broker emulator's behavior still uses the above logic.

Here is a strategy demonstrating how orders are filled by the broker emulator:

//@version=4 strategy("History SAW demo", overlay=true, pyramiding=100, calc_on_order_fills=true) strategy.entry("LE", strategy.long)

This code is calculated once per bar on the close, but an additional calculation occurs as soon as an order is filled. That is why you can see 4 filled orders on every bar: 2 orders on open, 1 order on high and 1 order on low. This is for backtesting. In real-time, orders would be executed on every new tick.

It is also possible to emulate an *order queue*. The setting is called *Verify Price For Limit Orders* and can be found in strategy properties, or set in the script's code with `strategy(..., backtest_fill_limits_assumption=X)`. The specified value is a minimum price movements in number of points/pips (default value is 0). A limit order is filled if the current price is better (higher for sell orders, lower for buy orders) by the specified number of points/pips. The execution price still matches the limit order price. Example:

- `backtest_fill_limits_assumption = 1`. Minimum price movement is `0.25`.
- A buy limit order is placed at price `12.50`.
- Current price is `12.50`.
- The order cannot be filled at the current price because `backtest_fill_limits_assumption = 1`. To fill the order the price must be `0.25*1` lower. The order is put in the queue.
- Assume that the next tick comes at price `12.00`. This price is 2 points lower, meaning the condition `backtest_fill_limits_assumption = 1` is satisfied, so the order should be filled. The order is filled at `12.50` (original order price), even if the price is not available anymore.

Order placement commands

All keywords related to strategies start with a `strategy.` prefix. The following commands are used for placing orders: `strategy.entry`, `strategy.order` and `strategy.exit`.

strategy.entry

This command only places entry orders. It is affected by the `pyramiding` setting in the strategy's properties and by the `strategy.risk.allow_entry_in` function. If there is an open market position when an opposite direction order is generated, the number of contracts/shares/lots/units will be increased by the number of currently open contracts (script equivalent: `strategy.position_size + quantity`). As a result, the size of the opened market position will be equal to the order size specified in the `strategy.entry` command.

strategy.order

This command places both entry and exit orders. It is not affected by pyramiding settings or by the `strategy.risk.allow_entry_in` function. It allows you to create complex entry and exit order constructions when the functionality of `strategy.entry` and `strategy.exit` will not do.

strategy.exit

This command allows you to exit a market position or form multiple exit order strategies using a stop loss, profit target or trailing stop. All such orders are part of the same `strategy.oca.reduce` group. An exit order cannot be placed if there is no open market position or there is no active entry order (an exit order is bound to the ID of an entry order). It is not possible to exit a position with a market order using the command `strategy.exit`. For this, the strategy.close or strategy.close_all commands should be used. If the number of contracts/shares/lots/units specified for the `strategy.exit` is less than the size of current open positions, the exit will be partial. It is possible to exit from the same entry order more than once using the same exit order ID, which allows you to create exit strategies with multiple levels. In cases where a market position is formed by multiple entry orders (pyramiding enabled), each exit order must be linked to a matching entry order.

Example 1:

//@version=4 strategy("revers demo") if bar_index < 100 strategy.entry("buy", strategy.long, 4, when=strategy.position_size <= 0) strategy.entry("sell", strategy.short, 6, when=strategy.position_size > 0) plot(strategy.equity)

The above strategy constantly reverses market position from +4 to -6, back and forth, which the plot shows.

Example 2:

//@version=4 strategy("exit once demo") strategy.entry("buy", strategy.long, 4, when=strategy.position_size <= 0) strategy.exit("bracket", "buy", 2, profit=10, stop=10)

This strategy demonstrates a case where a market position is never closed because it uses a partial exit order to close the market position and it cannot be executed more than once. If you double the line for exiting, the strategy will close the market position completely.

Example 3:

//@version=4 strategy("Partial exit demo") if bar_index < 100 strategy.entry("buy", strategy.long, 4, when=strategy.position_size <= 0) strategy.exit("bracket1", "buy", 2, profit=10, stop=10) strategy.exit("bracket2", "buy", profit=20, stop=20)

This code generates 2 levels of brackets (2 take profit orders and 2 stop loss orders). Both levels are activated at the same time: first level to exit 2 contracts and the second one to exit all the rest.

The first take profit and stop loss orders (level 1) are in an OCA group. The other orders (level 2) are in another OCA group. This means that as the order from level 1 is filled, the orders from level 2 are not cancelled; they stay active.

Every command placing an order has an ID (string value) which is a unique order identifier. If an order with the same ID is already placed but not yet filled, the last command modifies the existing order. If modification is not possible (conversion from buy to sell), the old order is cancelled and the new order is placed. `strategy.entry` and `strategy.order` work with the same IDs (they can modify the same entry order). `strategy.exit` works with other order IDs (it is possible to have an entry order and an exit order with the same ID).

To cancel a specific order using its ID, the strategy.cancel(string ID) command should be used. To cancel all pending orders the strategy.cancel_all() command should be used. Strategy orders are placed as soon as their conditions are satisfied and command is called in code. The broker emulator doesn't execute orders before the next tick comes after the code was calculated, while in real trading an order can get filled sooner. When a market order is generated at the close of the current bar, the broker emulator only executes it at the open price of the next.

Example:

//@version=4 strategy("next bar open execution demo") if bar_index < 100 strategy.order("buy", strategy.long, when=strategy.position_size == 0) strategy.order("sell", strategy.short, when=strategy.position_size != 0)

If this code is applied to a chart, all orders are filled at the open of every bar.

Conditions for order placement (`when`, `pyramiding`, `strategy.risk`) are checked when the script is calculated. If all conditions are satisfied, the order is placed. If any condition is not satisfied, the order is not placed. It is important to cancel price orders (limit, stop and stop-limit orders).

Example (for MSFT, 1D):

//@version=4 strategy("Priced Entry demo") var c = 0 if year > 2020 c := c + 1 if c == 1 strategy.entry("LE1", strategy.long, 2, stop = high + 35 * syminfo.mintick) strategy.entry("LE2", strategy.long, 2, stop = high + 2 * syminfo.mintick)

Even though pyramiding is disabled, both these orders are filled in backtesting because when they are generated there is no opened long market position. Both orders are placed and when price satisfies order execution conditions, they both get executed. It is recommended to put the orders in an OCA group using `strategy.oca.cancel`. This way only one order is filled and the other one is cancelled. Here is the modified code:

//@version=4 strategy("Priced Entry demo") var c = 0 if year > 2020 c := c + 1 if c == 1 strategy.entry("LE1", strategy.long, 2, stop = high + 35 * syminfo.mintick, oca_type = strategy.oca.cancel, oca_name = "LE") strategy.entry("LE2", strategy.long, 2, stop = high + 2 * syminfo.mintick, oca_type = strategy.oca.cancel, oca_name = "LE")

If, for some reason, order placing conditions are not met when executing the command, the entry order will not be placed. For example, if pyramiding settings are set to 2, the existing position already contains two entries and the strategy tries to place a third one, it will not be placed. Entry conditions are evaluated at the order generation stage and not at the execution stage. Therefore, if you submit two price type entries with

pyramiding disabled, once one of them is executed the other will not be cancelled automatically. To avoid issues we recommend using `strategy.oca.cancel` groups for entries so when one entry order is filled the others are cancelled.

The same is true for price type exits. Orders will be placed once their conditions are met, i.e., an entry order with a matching ID is filled.

Closing market position

Despite the fact that it is possible to exit from a specific entry in code, when orders are shown in the *List of Trades* in the *Strategy Tester* tab, they all are linked according to FIFO (first in, first out) rules. If an entry order ID is not specified for an exit order in code, the exit order closes the first entry order that opened market position. Let's study the following example:

//@version=4 strategy("exit Demo", pyramiding=2, overlay=true) strategy.entry("Buy1", strategy.long, 5, when = strategy.position_size == 0 and year > 2014) strategy.entry("Buy2", strategy.long, 10, stop = strategy.position_avg_price + strategy.position_avg_price*0.1, when = strategy.position_size == 5) strategy.exit("bracket", loss=10, profit=10, when=strategy.position_size == 15)

The code given above places 2 orders sequentially: "Buy1" at market price and "Buy2" at a 10% higher price (stop order). The exit order is placed only after entry orders have been filled. If you apply the code to a chart, you will see that each entry order is closed by an exit order, though we did not specify entry order ID to close in this line: `strategy.exit("bracket", loss=10, profit=10, when=strategy.position_size == 15)`

Another example:

//@version=4 strategy("exit Demo", pyramiding=2, overlay=true) strategy.entry("Buy1", strategy.long, 5, when = strategy.position_size == 0) strategy.entry("Buy2", strategy.long, 10, stop = strategy.position_avg_price + strategy.position_avg_price*0.1, when = strategy.position_size == 5) strategy.close("Buy2",when=strategy.position_size == 15) strategy.exit("bracket", "Buy1", loss=10, profit=10, when=strategy.position_size == 15) plot(strategy.position_avg_price)

- It opens a 5-contract long position with the order "Buy1".
- It extends the long position by purchasing 10 more contracts at 10% higher price with the order "Buy2".
- The exit order (strategy.close) to sell 10 contracts (exit from "Buy2") is filled.

If you take a look at the plot, you can see that average entry price = "Buy2" execution price and our strategy closed exactly this entry order, while on the *Trade List* tab we can see that it closed the first "Buy1" order and half of the second "Buy2". It means that no matter which entry order you specify for your strategy to close, the broker emulator will still close the first one, according to FIFO rules. It works the same way as when trading with a real broker.

OCA groups

It is possible to put orders in 2 different One-Cancells-All (OCA) groups in Pine Script.

strategy.oca.cancel

As soon as an order from the group is filled (even partially) or cancelled, the other orders from the same group get cancelled. One should keep in mind that if order prices are the same or they are close, more than 1 order of the same group may be filled. This OCA group type is available only for entry orders because all exit orders are placed in `strategy.oca.reduce`.

Example:

//@version=4 strategy("oca_cancel demo") if year > 2014 and year < 2016 strategy.entry("LE", strategy.long) strategy.entry("SE", strategy.short)

You may think that this is a reverse strategy since pyramiding is not allowed, but in fact both orders will get filled because they are market orders, which means they are to be executed immediately at the current price. The second order doesn't get cancelled because both are filled almost at the same moment and the system doesn't have time to process the first order fill and cancel the second one before it gets executed. The same would happen if these were price orders with same or similar prices. The strategy places all orders allowed according to market position, etc.

The strategy places all orders that do not contradict the rules (in our case market position is flat, therefore any entry order can be filled). At each tick calculation, firstly all orders with the satisfied conditions are executed and only then the orders from the group where an order was executed are cancelled.

To turn the above strategy into a reverse strategy you need to place orders in the OCA group:

//@version=4 strategy("oca_cancel demo") if year > 2014 and year < 2016 strategy.entry("LE", strategy.long, oca_type = strategy.oca.cancel, oca_name="Entry") strategy.entry("SE", strategy.short, oca_type = strategy.oca.cancel, oca_name="Entry")

strategy.oca.reduce

This group type allows multiple orders within the group to be filled. As one of the orders within the group starts to be filled, the size of other orders is reduced by the filled contracts amount. It is very useful for the exit strategies. Once the price touches your take-profit order and it is being filled, the stop-loss is not cancelled but its amount is reduced by the filled contracts amount, thus protecting the rest of the open position.

strategy.oca.none

The order is placed outside of the group (default value for the `strategy.order` and `strategy.entry` functions).

Every group has its own unique id, like orders. If two groups have the same id, but different type, they will be considered a different groups. Example:

//@version=4 strategy("My Script") if year > 2014 and year < 2016 strategy.entry("Buy", strategy.long, oca_name="My oca", oca_type=strategy.oca.reduce) strategy.exit("FromBy", "Buy", profit=100, loss=200, oca_name="My oca") strategy.entry("Sell", strategy.short, oca_name="My oca", oca_type=strategy.oca.cancel) strategy.order("Order", strategy.short, oca_name="My oca", oca_type=strategy.oca.none)

"Buy" and "Sell" will be placed in different groups as their type is different. "Order" will be outside of any group as its type is set to `strategy.oca.none`. Moreover, "Buy" will be placed in the exit group as exits are always placed in the `strategy.oca.reduce_size` type group.

Risk management

It is not easy to create a universally profitable strategy. Usually, strategies are created for certain market patterns and can produce uncontrollable losses when applied to other data. Therefore, stopping auto trading when too many losses occur is important. A special group of strategy commands help you manage risk. They all start with the `strategy.risk.` prefix.

In any given strategy you can combine any number of risk management criteria in any combination. Every risk category command is calculated at every tick as well as at every order execution event, regardless of the `calc_on_order_fills` strategy setting. There is no way to disable any risk rule at runtime from a script. Regardless of where in the script the risk rule is located it will always be applied unless the line with the rule is deleted and the script is recompiled.

When a risk management rule is triggered, no orders will be generated starting from the next calculation of the script. Therefore, if a strategy has several rules of the same type with different parameters, it will stop calculating when the rule with the most strict parameters is triggered. When a strategy is stopped, all unexecuted orders are cancelled and then a market order is sent to close the position if it is not flat.

Furthermore, it is worth remembering that when using resolutions higher than 1 day, the whole bar is considered to be 1 day for the rules starting with prefix `strategy.risk.max_intraday_`.

Example (MSFT, 1):

//@version=4 strategy("multi risk demo", overlay=true, pyramiding=10, calc_on_order_fills = true) if year > 2014 strategy.entry("LE", strategy.long) strategy.risk.max_intraday_filled_orders(5) strategy.risk.max_intraday_filled_orders(2)

The position will be closed and trading will be stopped until the end of every trading session after two orders are executed within this session, as the second rule is triggered earlier and is valid until the end of the trading session.

One should remember that the `strategy.risk.allow_entry_in` rule is applied to entries only so it will be possible to enter in a trade using the `strategy.order` command, as this command is not an entry command per se. Moreover, when the `strategy.risk.allow_entry_in` rule is active, entries in a "prohibited trade" become exits instead of reverse trades.

Example (MSFT, 1D):

//@version=4 strategy("allow_entry_in demo", overlay=true) if year > 2014 strategy.entry("LE", strategy.long, when=strategy.position_size <= 0) strategy.entry("SE", strategy.short, when=strategy.position_size > 0) strategy.risk.allow_entry_in(strategy.direction.long)

As short entries are prohibited by the risk rules, long exit trades will be made instead of reverse trades.

Currency

TradingView strategies can operate in a currency that is different from the instrument's currency. *Net Profit* and *Open Profit* are recalculated in the account currency. Account currency is set in the strategy properties' *Base Currency* drop-down list or in the script via the `strategy(..., currency=currency.*)` parameter. Performance report values are calculated in the selected currency.

Trade profit (open or closed) is calculated based on the profit in the instrument currency multiplied by the cross-rate on the *close* of the trading day previous to the bar where the strategy is calculated.

Example: we trade EURUSD, D and have selected `currency.EUR` as the strategy currency. Our strategy buys and exits the position using 1 point profit target or stop loss.

//@version=4 strategy("Currency test", currency=currency.EUR) if year > 2020 strategy.entry("LE", true, 1000) strategy.exit("LX", "LE", profit=1, loss=1) profit = strategy.netprofit plot(abs((profit - profit[1])*100), "1 point profit", color=color.blue, linewidth=2) plot(1 / close[1], "prev usdeur", color=color.red)

After adding this strategy to the chart we can see that the plot lines are matching. This demonstrates that the rate to calculate the profit for every trade was based on the *close* of the previous day.

When trading on intra-day resolutions, the cross-rate on the close of the trading day previous to the bar where the strategy is calculated will be used and it will not change during the trading session.

When trading on resolutions higher than 1 day, the cross-rate on the close of the trading day previous to the close of the bar where the strategy is calculated will be used. Let's say we trade on a weekly chart, then the cross rate on Thursday's session close will always be used to calculate the profits.

In real-time, yesterday's session close rate is used.

Leverage

When you open any position, the margin required to maintain the position will be calculated. If there are not enough funds, then a margin call occurs - the forced closure of part or all of the positions by a market order so that there is again enough funds to maintain the positions.

It is calculated by the formula:

margin = MVS * margin_percent, where margin is cash collateral for opening a position, in %, and MVS is the current market value of the security.

MVS is calculated according to the following formula: MVS = last_price * point_value * contracts.

margin_percent is a parameter that can be set in the strategy settings.

If margin == strategy.equity, then we can no longer open new positions. If margin > strategy.equity, then the margin call occurs, forcibly closing part or all of the positions until the margin < strategy.equity condition is met.

Notice!This is documentation for v4, 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, 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=4 study("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 `atr()` outside the `if` block so it executes on each bar. myAtr = atr(14) if barstate.islast // We only populate the table on the last bar. table.cell(atrDisplay, 0, 0, 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 `atr(14)` prior to entry in our if block so that it evaluates on each bar. Had we used `tostring(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=4 study("ATR", "", true) i_atrP = input(14, "ATR period", minval = 1, tooltip = "Using a period of 1 yields True Range.") // ———— Produces a string format usable with `tostring()` to restrict precision to ticks. f_tickFormat() => _s = tostring(syminfo.mintick) _s := str.replace_all(_s, "25", "00") _s := str.replace_all(_s, "5", "0") _s := str.replace_all(_s, "1", "0") var table atrDisplay = table.new(position.top_right, 1, 1, bgcolor = color.gray, frame_width = 2, frame_color = color.black) myAtr = atr(i_atrP) if barstate.islast table.cell(atrDisplay, 0, 0, tostring(myAtr, f_tickFormat()), 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 used the `f_tickFormat()` 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=4 study("Chart background", "", true) i_c_bull = input(color.new(color.green, 95), "Bull", inline = "1") i_c_bear = input(color.new(color.red, 95), "Bear", inline = "1") // ————— Function colors chart bg on RSI bull/bear state. f_colorChartBg(_c_bull, _c_bear) => var table _chartBg = table.new(position.middle_center, 1, 1) float _r = rsi(close, 20) color _c_bg = _r > 50 ? _c_bull : _r < 50 ? _c_bear : na if barstate.islast table.cell(_chartBg, 0, 0, width = 100, height = 100, bgcolor = _c_bg) f_colorChartBg(i_c_bull, i_c_bear)

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 `f_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=4 study("Price vs MA", "", true) var string GP1 = "Moving averages" int i_masQty = input(20, "Quantity", minval = 1, maxval = 40, group = GP1, tooltip = "1-40") int i_masStart = input(20, "Periods begin at", minval = 2, maxval = 200, group = GP1, tooltip = "2-200") int i_masStep = input(20, "Periods increase by", minval = 1, maxval = 100, group = GP1, tooltip = "1-100") var string GP2 = "Display" string i_tableYpos = input("top", "Panel position", inline = "11", options = ["top", "middle", "bottom"], group = GP2) string i_tableXpos = input("right", "", inline = "11", options = ["left", "center", "right"], group = GP2) color i_c_bull = input(color.new(color.green, 30), "Bull", inline = "12", group = GP2) color i_c_bear = input(color.new(color.red, 30), "Bear", inline = "12", group = GP2) color i_c_neutral = input(color.new(color.gray, 30), "Neutral", inline = "12", group = GP2) // ————— Produces a string format usable with `tostring()` to restrict precision to ticks. f_tickFormat() => string _s = tostring(syminfo.mintick) _s := str.replace_all(_s, "25", "00") _s := str.replace_all(_s, "5", "0") _s := str.replace_all(_s, "1", "0") var table panel = table.new(i_tableYpos + "_" + i_tableXpos, 2, i_masQty + 1) if barstate.islast // Table header. table.cell(panel, 0, 0, "MA", bgcolor = i_c_neutral) table.cell(panel, 1, 0, "Value", bgcolor = i_c_neutral) int period = i_masStart for _i = 1 to i_masQty // ————— Call MAs on each bar. float _ma = sma(close,

period) // ————— Only execute table code on last bar. if barstate.islast // Period in left column. table.cell(panel, 0, _i, tostring(period), bgcolor = i_c_neutral) // If MA is between the open and close, use neutral color. If close is lower/higher than MA, use bull/bear color. _c_bg = close > _ma ? open < _ma ? i_c_neutral : i_c_bull : open > _ma ? i_c_neutral : i_c_bear // MA value in right column. table.cell(panel, 1, _i, tostring(_ma, f_tickFormat()), text_color = color.black, bgcolor = _c_bg) period += i_masStep

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 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=4 study("Price vs Past", "", true) var int MAX_LOOKBACK = 300 int i_lookBack = input(150, minval = 1, maxval = MAX_LOOKBACK, step = 10) color i_c_Bull = input(#00FF00ff, "Bull", inline = "11") color i_c_Bear = input(#FF0080ff, "Bear", inline = "11") // ————— Function returning `_color` with `_transp` transparency. f_colorNew(_color, _transp) => // color _color : base color to derive a new transparency from. Its current transparency is ignored. // float _transp: 0-100 transparency of `_color` to return. _r = color.r(_color) _g = color.g(_color) _b = color.b(_color) color _return = color.rgb(_r, _g, _b, _transp) // ————— Function draws a heatmap showing the position of the current `_src` relative to its past `_lookBack` values. f_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 i_lookBack float _transp = 100. * _i / _lookBack if _src > _src[_i] table.cell(_heatmap, _lookBack - _i, 0, bgcolor = f_colorNew(i_c_Bull, _transp)) else table.cell(_heatmap, _lookBack - _i, 0, bgcolor = f_colorNew(i_c_Bear, _transp)) f_drawHeatmap(high, i_lookBack)

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 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 v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Comments

Pine Script supports single-line comments. Any text from the symbol `//` until the end of the line is considered as comment. An example:

```
//@version=4 study("Test") // This line is a comment a = close // This is also a comment plot(a)
```

The *Pine Editor* has a hotkey for commenting/uncommenting: `Ctrl + /`. Highlight a code fragment and press `Ctrl + /` to comment/uncomment whole blocks of code at a time.

Comments cannot be placed in the middle of a statement continued on multiple lines. Read more on this here: `Line_wrapping`.

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Declaring functions

Pine has an extensive library of built-in functions which can be used to create scripts. Users can also create their own functions.

Single-line functions

Simple functions can often be written on one line. This is the syntax of single-line functions:

```
<identifier>(<list of arguments>) => <expression>
```

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)

The type of the value returned by function `f` is determined automatically and depends on the type of the arguments used in each particular function call. 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.

Pine Scipt functions do not support recursion. It is **not allowed** for a function to call itself from within its own code.

Multi-line functions

Pine also supports multi-line functions with the following syntax:

```
<identifier>(<list of arguments>) =>
    <variable declaration>
    ...
    <variable declaration or expression>
```

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 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 `sqrt` (an extraction of the square root). The `geom_average` call will return the value of the last expression: `(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 buit-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, 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)

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Execution model

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 *studies* will execute once every time an *update* occurs, i.e., price or volume changes. Pine *strategies* will by default only execute when the rightmost bar closes, but they can also be configured to execute on every update, like studies do.

All symbol/resolution 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/resolution pair or approximately 10000 bars have been loaded[1]. 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).

All bars in a dataset are *historical bars*, except the rightmost one if a trading session is active. When trading is active in 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 a historical 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=4 study("My Script", overlay=true) src = close a = sma(src, 5) b = sma(src, 50) c = 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 study'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 studies do. The behavior described here for studies 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. The Pine 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 resolution is loaded on a chart.
- A script is saved or added to the chart from the Pine Editor.
- 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 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 bar 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 is due to 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 also referred to as *repainting*.

Additional resources

A number of `barstate.*` built-in variables provide information about the current type of bar update (historical, realtime, closing, etc.). The page where they are documented also contains a script that allows you to visualize the distinction described above between elapsed realtime bars and historical bars: `/essential/Bar_states_Built-in_variables_barstate`.

This page explains the details of strategy calculations: `/essential/Strategies`.

Footnotes

1. The upper limit for the total number of historical bars is about 10000 for *Pro/Premium* users. *Free* users are able to see about 5000 bars. ↩

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Expressions, declarations and statements

Expressions

An expression is a sequence where operators or function calls are applied to operands (variables or values) to define the calculations and actions required by the script. Expressions in Pine almost always produce a result (exceptions are the functions `study`, `fill`, `strategy.entry`, etc., which produce side effects and will be covered later).

Here are some examples of simple expressions:

(high + low + close)/3 sma(high - low, 10) + sma(close, 20)

Variable declaration

Variables in Pine are declared with the special symbol `=` and an optional `var` keyword in one of the following ways:

```
<identifier> = <expression>
<type> <identifier> = <expression>
var <identifier> = <expression>
var <type> <identifier> = <expression>
```

`<identifier>` is the name of the declared variable, see `Identifiers`.

`<type>` can be one of the predefined keywords: `float`, `int`, `bool`, `color`, `string`, `line` or `label`. However, in most cases, an explicit type declaration is redundant because type is automatically inferred from the `<expression>` 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 `na` value to `float` type. The declarations of `baseLine1` and `baseLine2` are equivalent.

The `var` keyword is a special modifier that instructs the compiler to *create and initialize the variable only once*. This behavior is very useful in 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=4 study("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 it's value) every time a new bar update triggered a script recalculation.

In Pine v3 the study "Green Bars Count" could be written without using the `var` keyword:

//@version=3 study("Green Bars Count") count = 0 // These two lines could be replaced in v4 count := nz(count[1], count) // with 'var count = 0' isGreen = close >= open if isGreen count := count + 1 plot(count)

The v4 code is more readable and can be more efficient if, for example, the `count` variable is initialized with an expensive function call instead of `0`.

Examples of simple variable declarations:

src = close len = 10 ma = sma(src, len) + high

Examples with type modifiers and var keyword:

float f = 10 // NOTE: while the expression is of type int, the variable is float i = int(close) // NOTE: explicit cast of float expression close to type int r = round(close) // NOTE: round() and int() are different... int() simply throws fractional part away var hl = high - low

Example, illustrating the effect of `var` keyword:

// Creates a new label object on every bar: label lb = label.new(bar_index, close, text="Hello, World!") // Creates a label object only on the first bar in history: var label lb = label.new(bar_index, close, text="Hello, World!")

Variable assignment

A mutable variable is a variable which can be given a new value. The operator `:=` must be used to give a new value to a variable. A variable must be declared before you can assign a value to it (see declaration of variables above).

The type of a variable is identified at declaration time. From then on, a variable can be given a value of expression only if both the expression and the variable belong to the same type, otherwise a compilation error will occur.

Variable assignment example:

//@version=4 study("My Script") price = close if hl2 > price price := hl2 plot(price)

if statement

An `if` statement defines a block of statements to be executed when the `if`'s conditional expression evaluates to `true`, and optionally, an alternative block to be executed when the expression is `false`.

General code form:

```
<var_declarationX> = if <condition>
    <var_decl_then0>
    <var_decl_then1>
    ...
    <var_decl_thenN>
else if [optional block]
```

```
    <var_decl_else0>
    <var_decl_else1>
    ...
    <var_decl_elseN>
else
    <var_decl_else0>
    <var_decl_else1>
    ...
    <var_decl_elseN>
    <return_expression_else>
```

where:

- `var_declarationX` - this variable is assigned the value of the `if` statement as a whole.
- `condition` - if the `condition` expression is true, the logic from the *then* block immediately following the `if` first line (`var_decl_then0`, `var_decl_then1`, etc.) is used, if the `condition` is false, the logic from the *else* block (`var_decl_else0`, `var_decl_else1`, etc.) is used.
- `return_expression_then`, `return_expression_else` - the last expression from the *then* block or from the *else* block will determine the final value of the whole `if` statement.

The type of the returning value of the `if` statement is determined by the type of `return_expression_then` and `return_expression_else`. Their types must match. It is not possible to return an integer value from the *then* block if the *else* block returns a string value.

Example:

// This code compiles x = if close > open close else open // This code doesn't compile x = if close > open close else "open"

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.

Example:

x = if close > open close // If current close > current open, then x = close. // Otherwise the x = na.

It is possible to use either multiple *else if* blocks or none at all.

Example:

x = if open > close 5 else if high > low close else open

The *then*, *else if* and *else* blocks are shifted by four spaces[1]. `if` statements can be nested by adding four more spaces:

x = if close > open b = if close > close[1] close else close[1] b else open

It is possible and quite frequent to ignore the resulting value of an `if` statement (`var_declarationX =` can be omited). This form is used when you need the side effect of the expression, for example in `strategy trading:

if (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")

for statement

The `for` statement allows to execute a number of instructions repeatedly:

```
<var_declarationX> = for <i> = <from> to <to> by <step>
    <var_decl0>
    <var_decl1>
    ...
    continue
    ...
    break
    ...
    <var_declN>
    <return_expression>
```

where:

- `i` - a loop counter variable.
- `from` - start value of the counter.
- `to` - end value of the counter. When the counter becomes greater than `to` (or less than `to` in the case where `from > to`) the loop is stopped.
- `step` - loop step. Optional. Default is 1. If `from` is greater than `to`, the loop step will automatically change direction; no need to use a negative step.
- `var_decl0`, ...`var_declN`, `return_expression` - body of the loop. It must be indented by 4 spaces.
- `return_expression` - returning value. When a loop is finished or broken, the returning value is assigned to `var_declarationX`.
- `continue` - a keyword. Can only be used in loops. It jumps to the loop's next iteration.
- `break` - a keyword. Can be used only in loops. It exits the loop.

`for` loop example:

//@version=4 study("For loop") my_sma(price, length) => sum = price for i = 1 to length-1 sum := sum + price[i] sum / length plot(my_sma(close,14))

Variable `sum` is a mutable variable so a new value can be given to it by the operator `:=` in the loop's body. Note that we recommend using the built-in sma() function for simple moving averages, as it calculates faster.

Note that some built-in functions may behave unexpectedly in for loop. Let's look at the following example:

//@version=4 study("RMA in for loop") sum = 0.0 for i = 1 to 2 sum := sum + rma(close, i) plot(sum)

While you may expect that `sum` will contain `rma(close, 1) + rma(close, 2)`, this is not so. It will contain `rma(close, 1) + rma(close, 1)` because once `rma` is initialized with length 1, this length is stored until the

script is removed from chart. To avoid this you may use your own, stateless function implementation. This is the list of built-in functions which have the same behavior:

- `rma(source, length)`: `length` is stateful.
- `ema(source, length)`: `length` is stateful.
- `valuewhen(condition, source, occurrence)`: `occurrence` is stateful.
- `rsi(x, y)`: when `y` is of type integer and behaves like a length, `y` is stateful.

Footnotes

1. TradingView's *Pine Editor* automatically replaces **Tab** with 4 spaces. ↩

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Functions and annotations

Pine Script distinguishes between *functions* and *annotation functions* (or just *annotations*). Syntactically they are similar, but they serve different purposes.

While functions are generally used to calculate values and for the result they return, annotations are mostly used for their side effects and only occasionally for the result some of them return.

Functions may be built-in, such as sma(), ema(), rsi(), or user-defined. All annotations are built-in.

The side effects annotations are used for include:

- assigning a name or other global properties to a script using study() or strategy()
- determining the inputs of a script using input()
- determining the outputs of a script using plot()

In addition to having side effects, a few annotations such as `plot` and `hline` also return a result which may be used or not. This result, however, can only be used in other annotations and can't take part in the script's calculations (see fill() annotation).

A detailed overview of Pine annotations can be found here.

Syntactically, user-defined functions, built-in functions and annotation functions are similar, i.e., we call them by name with a list of arguments in parentheses. Differences between them are mostly semantic, except for the fact that annotations and built-in functions accept keyword arguments while user-defined functions do not.

Example of an annotation call with positional arguments:

study('Example', 'Ex', true)

The same call with keyword arguments:

study(title='Example', shorttitle='Ex', overlay=true)

It's possible to mix positional and keyword arguments. Positional arguments must go first and keyword arguments should follow them. So the following call is not valid:

study(precision=3, 'Example') // Compilation error!

Execution of Pine functions and historical context inside function blocks

The history of series variables used inside Pine functions is created through each successive call to the function. If the function is not called on each bar the script runs on, this will result in disparities between the historic values of series inside vs outside the function's local block. Hence, series referenced inside and outside the function using the same index value will not refer to the same point in history if the function is not called on each bar.

Let's look at this example script where the `f` and `f2` functions are called every second bar:

//@version=4 study("My Script", overlay=true) // Returns the value of "a" the last time the function was called 2 bars ago. f(a) => a[1] // Returns the value of last bar's "close", as expected. f2() => close[1] oneBarInTwo

= bar_index % 2 == 0 plot(oneBarInTwo ? f(close) : na, color=color.maroon, linewidth=6, style=plot.style_cross) plot(oneBarInTwo ? f2() : na, color=color.lime, linewidth=6, style=plot.style_circles) plot(close[2], color=color.maroon) plot(close[1], color=color.lime)

As can be seen with the resulting plots, `a[1]` returns the previous value of a in the function's context, so the last time `f` was called two bars ago - not the close of the previous bar, as `close[1]` does in `f2`. This results in `a[1]` in the function block referring to a different past value than `close[1]` even though they use the same index of 1.

Why this behavior?

This behavior is required because forcing execution of functions on each bar would lead to unexpected results, as would be the case for a `label.new` function call inside an if branch, which must not execute unless the if condition requires it.

On the other hand, this behavior leads to unexpected results with certain built-in functions which require being executed each bar to correctly calculate their results. Such functions will not return expected results if they are placed in contexts where they are not executed every bar, such as if branches.

The solution in these cases is to take those function calls outside their context so they can be executed on every bar.

In this script, `barssince` is not called on every bar because it is inside a ternary operator's conditional branch:

//@version=4 study("Barssince",overlay=false) res = close>close[1] ? barssince(close<close[1]) : -1 plot(res, style=plot.style_histogram, color=res >= 0 ? color.red : color.blue)

This leads to incorrect results because `barssince` is not executed on every bar:

The solution is to take the barssince call outside the conditional branch to force its execution on every bar:

//@version=4 study("Barssince",overlay=false) b = barssince(close<close[1]) res = close>close[1] ? b : -1 plot(res, style=plot.style_histogram, color=res >= 0 ? color.red : color.blue)

Using this technique we get the expected output:

Exceptions

Not all built-in functions need to be executed every bar. These are the functions which do not require it, and so do not need special treatment:

abs, acos, asin, atan, ceil, cos, dayofmonth, dayofweek, exp, floor, heikinashi, hour, kagi, linebreak, log, log10, max, min, minute, month, na, nz, pow, renko, round, second, sign, sin, sqrt, tan, tickerid, time, timestamp, tostring, weekofyear, year

> NoticeFunctions called from within a `for` loop use the same context in each of the loop's iterations. In the example below, each `lowest` call on the same bar uses the value that was passed to it (i.e., `bar_index`), so function calls used in loops do not require special treatment.

//@version=4 study("My Script") va = 0.0 for i = 1 to 2 by 1 if (i + bar_index) % 2 == 0 va := lowest(bar_index, 10) // same context on each call plot(va)

Notice!This is documentation for v4, 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 can be composed of:

- upper (`A–Z`) or lower (`a–z`) case letters,
- underscore (_),
- digits (`0–9`).

An identifier cannot begin with a digit. Identifiers are case-sensitive. Here are some examples of valid identifiers:

myVar _myVar my123Var MAX_LEN max_len

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Line wrapping

Any statement that is too long in Pine Script can be placed on more than one line. Syntactically, a statement **must** begin at the beginning of the line. If it wraps to the next line then the continuation of the statement **must** begin with one or several (different from multiple of 4) spaces. For example:

a = open + high + low + close

may be wrapped as:

a = open + high + low + close

A long `plot` call may be wrapped as:

plot(correlation(src, ovr, length), color=color.purple, style=plot.style_area, transp=40)

Statements inside user functions can also be wrapped. However, since a local statement 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 multiple of 4 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)

Do not use comments with line wrapping. The following code does NOT compile:

//@version=4 study("My Script") c = open > close ? color.red : high > high[1] ? color.lime : // a comment low < low[1] ? color.blue : color.black bgcolor(c)

The compiler fails with the error: `Add to Chart operation failed, reason: line 3: syntax error at input 'end of line without line continuation'`. To make this code compile, simply remove the `// a comment` comment.

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Operators

Arithmetic operators

There are five arithmetic operators in Pine Script:

Symbol	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo (remainder after division)

The arithmetic operators above are all binary, whith + and - also serving as unary operators.

When using arithmetic operators, the type of the result depends on the type of the operands. If at least one of the operands is a *series*, then the result will also have a *series* type. If both operands are numeric, but at least one of these has the type *float*, then the result will also have the type *float*. If both operands are of type *integer*, then the result will also have the type *integer*. If at least one operand is `na` then the result is also `na`.

The + operator also serves as the concatenation operator for strings. `"EUR"+"USD"` constructs the "EURUSD" string.

Comparison operators

There are six comparison operators in Pine Script:

Symbol	Operation
<	Less Than
<=	Less Than or Equal To
!=	Not Equal
==	Equal
>	Greater Than
>=	Greater Than or Equal To

Comparison operations are binary. The result is determined by the type of the operands. If at least one of these operands has a *series* type, then the type of the result will also be *series* (a series of logical values). If both operands have a numerical type, then the result will be of the logical type *bool*.

Logical operators

There are three logical operators in Pine Script:

Symbol	Operation
`not`	Negation

Symbol	Operation
`and`	Logical Conjunction
`or`	Logical Disjunction

All logical operators can operate with *bool* operands, numerical operands, or *series* type operands. As is the case with arithmetic and comparison operators, if at least one of the operands is of *series* type, then the result will also be of *series* type. In all other cases the type of the result will be the logical type *bool*.

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

?: conditional operator and the iff function

The `?:` conditional ternary operator calculates the first expression (condition) and returns the value of either the second operand (if the condition is `true`) or of the third operand (if the condition is `false`). Syntax is:

condition ? result1 : result2

If `condition` is `true` then the ternary operator will return `result1`, otherwise it will return `result2`.

A combination of conditional operators can build constructs similar to *switch* statements in other languages. For example:

isintraday ? red : isdaily ? green : ismonthly ? blue : na

The example is calculated from left to right. First, the `isintraday` condition is calculated; if it is `true` then `red` will be the result. If it is `false` then `isdaily` is calculated, if this is `true`, then `green` will be the result. If it is `false`, then `ismonthly` is calculated. If it is `true`, then `blue` will be the result, otherwise `na` will be the result.

For those who find using the `?:` operator syntax inconvenient, there is an alternative: the built-in `iff` function. The function has the following signature:

iff(condition, result1, result2)

The function acts identically to the `?:` operator, i.e., if the condition is `true` then it returns `result1`, otherwise `result2`. This is the equivalent of the previous example using `iff`:

iff(isintraday, red, iff(isdaily, green, iff(ismonthly, blue, na)))

History reference operator []

It is possible to refer to the historical values of any variable of the *series* type with the `[]` operator. *Historical* values are variable values for the previous bars.

Most data in Pine is stored in series (somewhat like arrays, but with a dynamic index). Let's see how the index is dynamic, and why series are also very different from arrays. In Pine, the `close` variable, or `close[0]` which is equivalent, holds the price at the close of the current bar. If your code is now executing on the **third** bar of the dataset, `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 second), and `close[2]`, the first. `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. 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 runtime environment, as your code is executed once for each historical bar in the dataset, starting from the left of the chart, Pine 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, are usually static in size and their content or indexing structure is not modified by the runtime environment. Pine series are thus very different from arrays and share familiarity with them mostly through their indexing syntax.

At the realtime, `close` variable represents the current price and will only contain the actual closing price of the realtime bar the last time the script is executed on that bar, and from then on, when it is referred to using the history-referencing operator.

Pine has a variable that keeps track of the bar count: `bar_index`. On the first bar, `bar_index` is equal to 0 and it increases by 1 at each new bar, so at 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. 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 math 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 Pine 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.

Note 1. Almost all built-in functions in Pine's standard library return a *series* result. It is therefore possible to apply the `[]` operator directly to function calls, as is done here:

sma(close, 10)[1]

Note 2. Despite the fact that the `[]` operator returns a result of *series* type, it is prohibited to apply this operator to the same operand over and over again. Here is an example of incorrect use which will generate a compilation error:

close[1][2] // Error: incorrect use of [] operator

In some situations, the user may want to shift the series to the left. Negative arguments for the operator `[]` are prohibited. This can be accomplished using the `offset` parameter in the `plot` annotation, which supports both positive and negative values. Note though that it is a visual shift., i.e., it will be applied after all calculations. Further details on `plot` and its parameters can be found here.

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.

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Structure of the script

A script in Pine usually consists of:

- `//@version=4` A compiler directive in a comment that specifies which version of Pine the script will use. If the `@version` directive is missing, version 1 will be used. It is strongly recommended to always use the latest version available.
- A study() or a strategy() annotation call. Their parameters define the script's title and other properties.
- A series of statements which implement the script's algorithm. Each statement is usually placed on a separate line. It is possible to place more than one statement on a line by using the comma (`,`) as a separator. Lines containing statements that are not part of local blocks cannot begin with white space. Their first character must also be the line's first character. Statements may be one of three kinds:
  - variable declarations
  - function declarations
  - functions and annotations calls
- A *study* must contain at least one function/annotation call which produces some output on the chart (e.g., `plot`, `plotshape`, `barcolor`, `line.new`, etc.). A strategy must contain at least one `strategy.*` call, e.g., `strategy.entry`.

  The simplest valid Pine v4 study can be generated using *Pine Editor New Blank indicator script*:

  //@version=4 study("My Script") plot(close)

  A simple valid Pine v4 strategy can be generated using *Pine Editor New Blank strategy script*:

  //@version=4 strategy("My Strategy", overlay=true) longCondition = crossover(sma(close, 14), sma(close, 28)) if (longCondition) strategy.entry("My Long Entry Id", strategy.long) shortCondition = crossunder(sma(close, 14), sma(close, 28)) if (shortCondition) strategy.entry("My Short Entry Id", strategy.short)

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Type system

Pine has 9 fundamental data **types**. They are: *int*, *float*, *bool*, *color*, *string*, *line*, *label*, *plot*, *hline*. All of these types exist in several **forms**. There are 5 forms of types: *literal*, *const*, *input*, *simple* and a *series*. We will often refer to a pair *form type* as a *type*. The Pine compiler distinguishes between a *literal bool* type, an *input bool* type, a *series bool* type and so on.

There is also an *array* type, a *void* type, a *na* (not available) value and a compound *tuple* type.

Type forms

Literal

A *literal* is a special notation for representing a fixed value in Pine. This fixed value itself is an expression and such literal expressions are always of one of the 5 following types:

- *literal float* (`3.14`, `6.02E-23`, `3e8`)
- *literal int* (`42`)
- *literal bool* (`true`, `false`)
- *literal string* (`"A text literal"`)
- *literal color* (`#FF55C6`)

> NoticeIn Pine, the built-in names `open`, `high`, `low`, `close`, `volume`, `time`, `hl2`, `hlc3`, `ohlc4` are not literals. They are of the *series* form.

Const

Values of the form *const* are ones that:

- do not change during script execution
- are known or can be calculated at compile time

For example:

c1 = 0 c2 = c1 + 1 c3 = c1 + 1 if open > close c3 := 0

The type of `c1` is *const int* because it is initialized with a *literal int* expression. The type of `c2` is also *const int* because it is initialized with an arithmetic expression of *const int* type. The type of `c3` is *series int* because it changes at runtime.

Input

Values of the form *input* are ones that:

- do not change during script execution
- are unknown at compile time
- originate from an input() function

For example:

```
p = input(10, title="Period")
```

The type of `p` variable is *input integer*.

Simple

Values of the form *simple* are ones that:

- do not change during script execution
- are unknown at compile time

They are values that come from the main chart's symbol information. For example, the syminfo.mintick built-in variable is a *simple float*. The word *simple* is usually omitted when referring to this form, so we use *float* rather than *simple float*.

Series

Values of the form *series* are ones that:

- change during the script execution
- store a sequence of historical values associated with bars of the main chart's symbol
- can be accessed using the `[]` operator. Note that only the last value in the series, i.e., the one associated with the current bar, is available for both reading and writing

The *series* form is the most common form in Pine. Examples of built-in *series* variables are: `open`, `high`, `low`, `close`, `volume` and `time`. The size of these series is equal to the quantity of available bars for the current ticker and timeframe (resolution). Series may contain numbers or a special value: `na`, meaning that a value is *not available*. Further information about the `na` value can be found here. Any expression that contains a series variable will be treated as a series itself. For example:

a = open + close // Addition of two series b = high / 2 // Division of a series variable by // an integer literal constant c = close[1] // Referring to the previous ``close`` value

> NoticeThe `[]` operator also returns a value of the *series* form.

Fundamental types

int

Integer literals must be written in decimal notation. For example:

1 750 94572 100

There are 5 forms of int type in Pine:

- *literal int*
- *const int*
- *input int*
- *int*

- *series int*

float

Floating-point literals contain a delimiter (the symbol `.`) and may also contain the symbol `e` (which means "multiply by 10 to the power of X", where X is the number after the symbol `e`). Examples:

3.14159 // 3.14159 6.02e23 // 6.02 * 10^23 1.6e-19 // 1.6 * 10^-19 3.0 // 3.0

The first number is the rounded number Pi (π), the second number is very large, while the third is very small. The fourth number is simply the number `3` as a floating point number.

> NoticeIt's possible to use uppercase `E` instead of lowercase `e`.

There are 5 forms of float type in Pine:

- *literal float*
- *const float*
- *input float*
- *float*
- *series float*

The internal precision of floats in Pine is 1e-10.

bool

There are only two literals representing *bool* values:

true // true value false // false value

There are 5 forms of bool type in Pine:

- *literal bool*
- *const bool*
- *input bool*
- *bool*
- *series bool*

color

Color literals have the following format: `#` followed by 6 or 8 hexadecimal digits matching RGB or RGBA value. The first two digits determine the value for the **red** color component, the next two, **green**, and the third, **blue**. Each component value must be a hexadecimal number from `00` to `FF` (0 to 255 in decimal).

The fourth pair of digits is optional. When used, it specifies the **alpha** (opacity) component, a value also from `00` (fully transparent) to `FF` (fully opaque). Examples:

#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 color

> NoticeHexadecimal notation is not case-sensitive.

There are 5 forms of color type in Pine:

- *literal color*
- *const color*
- *input color*
- *color*
- *series color*

One might ask how a value can be of type *input color* if it is impossible to use input() to input a color in Pine. The answer is: through an arithmetic expression with other input types and color literals/constants. For example:

b = input(true, "Use red color") c = b ? color.red : #000000 // c has color input type

This is an arithmetic expression using Pine's ternary operator `?:` where three different types of values are used: `b` of type *input bool*, `color.red` of type *const color* and `#000000` of type *literal color*. In determining the result's type, the Pine compiler takes into account its automatic type-casting rules (see the end of this section) and the available overloads of the `?:` operator. The resulting type is the narrowest type fitting these criteria: *input color*.

The following built-in *color* variables can be used to avoid hexadecimal color literals: `color.black`, `color.silver`, `color.gray`, `color.white`, `color.maroon`, `color.red`, `color.purple`, `color.fuchsia`, `color.green`, `color.lime`, `color.olive`, `color.yellow`, `color.navy`, `color.blue`, `color.teal`, `color.aqua`, `color.orange`.

It is possible to change the transparency of the color using a built-in function color.new.

Here is an example:

//@version=4 study(title="Shading the chart's background", overlay=true) c = color.navy bgColor = (dayofweek == dayofweek.monday) ? color.new(c, 50) : (dayofweek == dayofweek.tuesday) ? color.new(c, 60) : (dayofweek == dayofweek.wednesday) ? color.new(c, 70) : (dayofweek == dayofweek.thursday) ? color.new(c, 80) : (dayofweek == dayofweek.friday) ? color.new(c, 90) : color.new(color.blue, 80) bgcolor(color=bgColor)

string

String literals may be enclosed in single or double quotation marks. Examples:

```
"This is a double quoted string literal"
'This is a single quoted string literal'
```

Single and double quotation marks are functionally equivalent. A string enclosed within double quotation marks may contain any number of single quotation marks, and vice versa:

```
"It's an example"
```

```
'The "Star" indicator'
```

If you need to enclose the string's delimiter in the string, it must be preceded by a backslash. For example:

```
'It\'s an example'
"The \"Star\" indicator"
```

There are 5 forms of string type in Pine:

- *literal string*
- *const string*
- *input string*
- *string*
- *series string*

line and label

New drawing objects were introduced in Pine v4. These objects are created with the line.new() and label.new() functions. Their type is *series line* and *series label*, respectively. There is only one form of the *line* and *label* types in Pine: *series*.

plot and hline

A few function annotations (in particular `plot` and `hline`) return values which represent objects created on the chart. The function `plot` returns an object of the type *plot*, represented as a line or diagram on the chart. The function `hline` returns an object of the type *hline*, represented as a horizontal line. These objects can be passed to the fill() function to color the area in between them.

array

Arrays in Pine are identified by an *array id*. There is no single type representing an array id, but rather an overloaded version of a subset of fundamental Pine types which reflects the type of an array's elements. These type names are constructed by appending the `[]` suffix to one of the four fundamental types allowed in arrays:

- `int[]`
- `float[]`
- `bool[]`
- `color[]`

void

There is a *void* type in Pine Script. Most functions and annotation functions which produce a *side effect* return a void result. E.g., strategy.entry(), plotshape() etc.

A void result cannot be used in any arithmetic expression or be assigned to a variable.

na value

In Pine there is a special built-in variable `na`, which is an acronym for *not available*, meaning an expression or variable has no value. This is very similar to the `null` value in Java or `None` in Python.

There are a few things to know about `na` values. First, the `na` value can be automatically cast to almost any type.

Secondly, in some cases the Pine compiler cannot automatically infer a type for a `na` value because more that one automatic type-casting rule can be applied. For example:

myVar = na // Compilation error!

Here, the compiler cannot determine if `myVar` will be used to plot something, as in `plot(myVar)` where its type would be *series float*, or to set some text as in `label.set_text(lb, text=myVar)` where its type would be *series string*, or for some other purpose. Such cases must be explicitly resolved in one of two ways:

float myVar = na

or:

myVar = float(na)

Thirdly, to test if some value is *not available*, a special function must be used: na. For example:

myClose = na(myVar) ? 0 : close

Do not use the operator == to test for `na` values, as this is not guaranteed to work.

Tuples

In Pine Script there is limited support for a tuple type. A tuple is an immutable sequence of values used when a function must return more than one variable as a result. For example:

calcSumAndMul(a, b) => sum = a + b mul = a * b [sum, mul]

In this example there is a 2-tuple on the last statement of the function `calcSumAndMul`. Tuple elements can be of any type. There is also a special syntax for calling functions that return tuples. For example, `calcSumAndMul` must be called as follows:

[s, m] = calcSumAndMul(high, low)

where the value of local variables `sum` and `mul` will be written to the `s` and `m` variables of the outer scope.

Type casting

There is an automatic type-casting mechanism in Pine Script. In the following picture, an arrow denotes the Pine compiler's ability to automatically cast one type to another:

For example:

//@version=4 study("My Script") plotshape(series=close)

The type of the `series` parameter of the `plotshape` function is *series bool*. But the function is called with the `close` argument of type *series float*. The types do not match, but an automatic type-casting rule *series float*

series bool (see the diagram) does the proper conversion.

Sometimes there is no automatic *X Y* type-casting rule. For these cases, explicit type-casting functions were introduced in Pine v4. They are:

- int
- float
- string
- bool
- color
- line
- label

Here is an example:

//@version=4 study("My Script") len = 10.0 s = sma(close, len) // Compilation error! plot(s)

This code fails to compile with an error: `Add to Chart operation failed, reason: line 4: Cannot call sma with arguments (series[float], const float); available overloads: sma(series[float], integer) => series[float];` The compiler says that while the type of the `len` variable is *const float*, the `sma` function expected an `integer`. There is no automatic type casting from *const float* to *integer*, but an explicit type-casting function can be used:

//@version=4 study("My Script") len = 10.0 s = sma(close, int(len)) plot(s)

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Versions

There are currently four versions of the Pine Script Language. A compiler directive must be used in the first line of a script to specify the version or Pine used by the script: `//@version=N` where `N` is the version number (1-4). Note that different Pine Script Language versions are incompatible with each other.

Notable changes to the language are documented in Release notes.

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Public Library

Description

On TradingView you may choose from our built-in scripts, from the user-published scripts in the *Public Library*, or write your own custom scripts using Pine Script. You can search the library for scripts before writing one. You may also ask users who publish scripts for help by sending them a private message (PM) on TradingView.

The Public Library contains all user-published scripts. All published scripts can be viewed either from the Scripts section of tradingview.com's page, or from the *Indicators Public Library* tab:

Copying a script from the Public to your Personal Library

To copy a script, use one of the two following methods:

1. Click the star next to the needed script in the Indicators window:
2. Open the needed script from the Scripts section and click the *Add to Favorite Scripts* button:

The script will be added to your Favorites section in the Indicators window, where you can easily find it when needed:

Scripts in the *Indicators* window are sorted by popularity. Scripts with the most likes appear first:

Editing a script from the Public Library

To edit a script from the Public Library:

1. Open the script (the script will be unavailable for editing and will be colored lighter than normal):
2. Click on the *Make a Copy...* button, enter the desired name and save a copy in your personal library:

A saved copy of the script will appear in the list of your scripts in the *Indicators My Scripts* tab:

Now you can open and edit the script.

If the same script is published more than once, a notification on older versions will appear, indicating a new version is available.

Searching for a script

To search for a script, use the *Search* field in the *Indicators* dialog:

This will search both your Personal and Public Library at the same time.

You may also search for scripts from the tradingview.com home page by first selecting *Scripts* from the *Ticker* dropdown menu and entering your search criteria.

Notice!This is documentation for v4, 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.

> NoticeIf 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 study() 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 study() or strategy() declaration statement.
- 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 `Public_Library` 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 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 Pro, Pro+ or Premium accounts 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 v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Quickstart guide

A script written in Pine is composed of functions and variables. Functions contain instructions that describe the required calculations. Variables save the values used or created during those calculations.

A script must contain a `study` or `strategy` annotation which defines the script's name and other properties. The script's body contains the functions and variables necessary to calculate results which will be rendered on a chart with a `plot` function, or some other function that plots the script's output.

Example of a Pine script

Let's look at the implementation of the MACD indicator in Pine:

//@version=4 study("MACD") fast = 12, slow = 26 fastMA = ema(close, fast) slowMA = ema(close, slow) macd = fastMA - slowMA signal = sma(macd, 9) plot(macd, color=color.blue) plot(signal, color=color.orange)

Line 1: `//@version=4`

This is a comment containing a compiler directive that tells the compiler the script will use version 4 of Pine.

Line 2: `study("MACD")`

Defines the name of the script that will appear on the chart as "MACD".

Line 3: `fast = 12, slow = 26`

Defines two integer variables: `fast` and `slow`.

Line 4: `fastMA = 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 5: `slowMA = ema(close, slow)`

Defines the variable `slowMA`, containing the result of the EMA calculation with a length equal to `slow` (26), from `close`.

Line 6: `macd = fastMA - slowMA`

Defines the variable `macd` as the difference between the two EMAs.

Line 7: `signal = sma(macd, 9)`

Defines the variable `signal` as a smoothed value of `macd` using the SMA algorithm (Simple Moving Average) with a length of 9.

Line 8: `plot(macd, color=color.blue)`

Calls the `plot` function to output the variable `macd` using a blue line.

Line 9: `plot(signal, color=color.orange)`

Calls the `plot` function to output the variable `signal` using an orange line.

After adding the "MACD" script to the chart you will see the following:

Pine contains a variety of built-in functions for the most popular algorithms (SMA, EMA, WMA, etc.). You can also define your custom functions. You will find a description of all available built-in functions here.

"study" vs "strategy"

Pine strategies are used to run backtests. In addition to normal script calculations, they also contain `strategy.*()` calls to send buy and sell orders to the broker emulator, which can then simulate their execution. See Strategies.

Pine studies, as the one in the previous example, also contain calculations, but cannot be used in backtesting. Because they do not make use of the broker emulator, they use less resources and will run faster.

Both strategies and studies can run in either overlay or pane mode, and plot information in that space. Both can also generate alert events. See Alerts.

Execution model of Pine scripts

A Pine script is **not** like many normal programs that execute once and then stop. In the Pine runtime environment, a script runs in the equivalent of an invisible loop where it is executed once on each historical bar. When execution reaches the last, real-time bar, the script executes once every time a price or volume change is detected, then one final time when the real-time bar closes and becomes a historical bar.

By default, Pine *strategies* only execute once at the close of real-time bars, but they can also be instructed to execute on each price change, as *studies* do. See Execution model.

Series

The main data type used in Pine scripts is called a *series*. It is a continuous list of values that stretches back in time from the current bar and where one value exists for each bar. While this structure may remind many of an array, a Pine series is totally different and thinking in terms of arrays will be detrimental to understanding this key Pine concept. You can read about series here and get more information on how to use them here.

Understanding scripts

If you intend to write Pine scripts of any reasonable complexity, a good comprehension of both the Pine execution model and series is essential in understanding how Pine scripts work. If you have never worked with data organized in series before, you will need practice in putting them to work for you. When you familiarize yourself with Pine's fundamental concepts, you will discover that by combining the use of series with our built-in functions designed to efficiently process series information, much can be accomplished in very few lines of Pine code.

Pine Editor

The Pine Editor is where you will be working on your scripts. To open it, click on the *Pine Editor* tab at the bottom of your TradingView chart. This will open up the editor's window. We will create our first working Pine script. Start by bringing up the “New” dropdown menu at the top right of the editor and choose *Blank indicator script*. Then copy the previous example script, select all code already in the editor and replace it with the example script. Click *Save*, choose a name and then click *Add to Chart*. The MACD indicator will appear in a separate *Pane* under the chart.

From here, you can change the script’s code. For example, change the last line’s `color.orange` for `color.fuchsia`. When you save the script, the change will be reflected in the indicator’s pane. Your first Pine script is running!

Where to go from here?

This documentation contains numerous examples of code used to illustrate how functions, variables and operators are used in Pine. By going through it, you will be able to both learn the foundations of Pine and study the example scripts.

The fastest way to learn a programming language is to read about key concepts and try them out with real code. As we’ve just done, copy this documentation’s examples in the editor and play with them. Explore! You won’t break anything.

You will also find examples of Pine scripts in the editor’s *New* dropdown menu and in TradingView's extensive Public Library of scripts which contains thousands of open source Pine scripts. Enjoy and welcome to Pine!

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Release notes

This page contains release notes of notable changes in Pine Script.

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 our User Manual page on tables.

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, making it possible to draw rectangles on charts using the Pine syntax. For more details see the Pine Script reference and the User Manual page on drawings .

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 User Manual page on colors.

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.

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 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)

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

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 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="🐻 ")

- 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 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 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 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

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.

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.

December 2016

- Added invite-only scripts. The invite-only indicators are visible in the Public Library, 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 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-based indicators.

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. 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.

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 coders 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.

December 2013

- The first version of Pine is introduced to all TradingView users, initially as an open beta, on December 13th.

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Introduction

TradingView has designed its own scripting language called **Pine Script**. It allows users to create custom indicators and run them on our servers. Pine was designed as a lightweight language focused on the specific task of developing indicators and strategies. Most of TradingView's built-in indicators have been coded in Pine. It is our explicit goal to keep Pine accessible and easy to understand for the broadest possible audience.

Pine is cloud-based and therefore different in nature to client-side programming languages. While we will not develop Pine into a full-fledged language with high-end coding capabilities for building very complex tools, constantly improving Pine is one of our highest priorities and we are happy to consider any 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 impose as few limits as possible while enforcing as many as needed. We must ensure the platform keeps running smoothly so nobody is negatively affected by scripts that consume a disproportionate amount of resources. The imposed limits apply to elements such as the amount of data from additional symbols, execution time, memory usage and script size. Additionally, we keep Pine syntax and semantics simple so it can handle common tasks efficiently.

We will continue to improve Pine's documentation and support to ensure that anyone who wishes to learn and use Pine may do so, as this helps grow the Pine development community and provides TradingView users with more robust and helpful tools.

Notice!This is documentation for v4, which is no longer actively maintained.
For up-to-date documentation, see the latest version.

Where can I get more information?

- Additional descriptions of all built-in operators, variables, functions and annotation functions can be found in the Pine Script Reference.
- 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 *New* button. You'll find the TradingView built-in scripts in the dropdown list.
- Information about major releases and modifications to Pine Script (as well as other features) is regularly published on TradingView's blog.
- TradingView's *Public Library* contains all user-published scripts. All published scripts are also listed in the Scripts section of the tradingview.com home page and in the *Indicators Public Library* dialog box.

External resources

- PineCoders publishes useful information for Pine coders.
- Kodify has TradingView tutorials on various topics for beginners and more experienced coders alike. Topics include plotting, alerts, strategy orders, and complete example indicators and strategies.
- There is a Pine Script community growing on StackOverflow where you can ask questions. See questions tagged with *pine-script*.