Pine Script® v6 Documentation

3 Concepts Part1

TradingView Pine Script Programming Language

Table of Contents

Redirect

Pine Script® v6 Documentation

Alerts

Pine Script® v6 Documentation

User Manual / Concepts / Alerts

Alerts

Introduction

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

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

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

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

- How to use the alert() function to *alert() function calls* in indicators or strategies, which can then be included in *script alerts* created from the charts UI.

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

Keep in mind that:

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

Background

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

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

Which type of alert is best?

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

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

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

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

Script alerts

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

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

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

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

`alert()` function events

The alert() function has the following signature:

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

`message`

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

`freq`

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

- `alert.freq_once_per_bar`: Only the first call per realtime bar triggers the alert (default value).

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

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

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

Using all `alert()` calls

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

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

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

- When RSI crosses the centerline up, the *script alert* will trigger with the "Go long..." message. When RSI crosses the centerline down, the *script alert* will trigger with the "Go short..." message.
- Because no argument is specified for the `freq` parameter in the alert() call, the default value of `alert.freq_once_per_bar` will be used, so the alert will only trigger the first time each of the alert() calls is executed during the realtime bar.

- The message sent with the alert is composed of two parts: a constant string and then the result of the str.tostring() call which will include the value of RSI at the moment where the alert() call is executed by the script. An alert message for a cross up would look like: “Go long (RSI is 53.41)“.
- Because a *script alert* always triggers on any occurrence of a call to alert(), as long as the frequency used in the call allows for it, this particular script does not allow a script user to restrict his *script alert* to longs only, for example.

Note that:

- Contrary to an alertcondition() call which is always placed at column 0 (in the script’s global scope), the alert() call is placed in the local scope of an if branch so it only executes when our triggering condition is met. If an alert() call was placed in the script’s global scope at column 0, it would execute on all bars, which would likely not be the desired behavior.
- An alertcondition() could not accept the same string we use for our alert’s message because of its use of the str.tostring() call. alertcondition() messages must be constant strings.

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

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

Using selective `alert()` calls

When users create a *script alert* on *alert() function calls*, the alert will trigger on any call the script makes to the alert() function, provided its frequency constraints are met. If you want to allow your script’s users to select which alert() function call in your script will trigger a *script alert*, you will need to provide them with the means to indicate their preference in your script’s inputs, and code the appropriate logic in your script. This way, script users will be able to create multiple *script*

alerts from a single script, each behaving differently as per the choices made in the script’s inputs prior to creating the alert in the charts UI.

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

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

Note how:

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

In strategies

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

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

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

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

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

the realtime bar closes because `alert.freq_once_per_bar_close` is the argument used for the `freq` parameter in the alert() calls.

Order fill events

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

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

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

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

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

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

Note that:

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

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

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

alertcondition() events

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

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

The alertcondition() function has the following signature:

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

`condition`

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

`title`

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

`message`

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

will be replaced at runtime by dynamic values that may change bar to bar. See this page's Placeholders section for a list.

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

Using one condition

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

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

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

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

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

Note that:

- The first line uses the `{{plot_0}}` placeholder, where the plot number corresponds to the order of the plot in the script.
- The second line uses the `{{plot("[plot_title]")}}` type of placeholder, which must include the `title` of the plot() call used in our script to plot RSI. Double quotes are used to wrap the plot's title inside the `{{plot("RSI")}}` placeholder. This requires that we use single quotes to wrap the `message` string.

- Using one of these methods, we can include any numeric value that is plotted by our indicator, but as strings cannot be plotted, no string variable can be used.

Using compound conditions

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

This script demonstrates one way to do it:

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

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

Placeholders

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

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

`{{exchange}}`

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

`{{interval}}`

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

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

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

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

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

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

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

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

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

`{{ticker}}`

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

`{{time}}`

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

`{{timenow}}`

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

Avoiding repainting with alerts

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

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

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

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

Next

Backgrounds

Pine Script® v6 Documentation

Backgrounds

The bgcolor() function changes the color of the script's background. If the script is running in `overlay = true` mode, then it will color the chart's background.

The function's signature is:

```
bgcolor(color, offset, editable, show_last, title, force_overlay) →
```

Its `color` parameter allows a "series color" to be used for its argument, so it can be dynamically calculated in an expression.

If the correct transparency is not part of the color to be used, it can be be generated using the color.new() function.

Here is a script that colors the background of trading sessions (try it on 30min EURUSD, for example):

//@version=6 indicator("Session backgrounds", overlay = true) // Default color constants using tranparency of 25. BLUE_COLOR = #0050FF40 PURPLE_COLOR = #0000FF40 PINK_COLOR = #5000FF40 NO_COLOR = color(na) // Allow user to change the colors. preMarketColor = input.color(BLUE_COLOR, "Pre-market") regSessionColor = input.color(PURPLE_COLOR, "Pre-market") postMarketColor = input.color(PINK_COLOR, "Pre-market") // Function returns `true` when the bar's time is timeInRange(tf, session) => time(tf, session) != 0 // Function prints a message at the bottom-right of the chart. f_print(_text) => var table _t = table.new(position.bottom_right, 1, 1) table.cell(_t, 0, 0, _text, bgcolor = color.yellow) var chartIs30MinOrLess = timeframe.isseconds or (timeframe.isintraday and timeframe.multiplier <=30) sessionColor = if

chartIs30MinOrLess switch timeInRange(timeframe.period, "0400-0930") => preMarketColor timeInRange(timeframe.period, "0930-1600") => regSessionColor timeInRange(timeframe.period, "1600-2000") => postMarketColor => NO_COLOR else f_print("No background is displayed.\nChart timeframe must be <= 30min.") NO_COLOR bgcolor(sessionColor)

image

Note that:

- The script only works on chart timeframes of 30min or less. It prints an error message when the chart's timeframe is higher than 30min.
- When the if structure's `else` branch is used because the chart's timeframe is incorrect, the local block returns the `NO_COLOR` color so that no background is displayed in that case.
- We first initialize constants using our base colors, which include the `40` transparency in hex notation at the end. 40 in the hexadecimal notation on the reversed 00-FF scale for transparency corresponds to 75 in Pine Script®'s 0-100 decimal scale for transparency.
- We provide color inputs allowing script users to change the default colors we propose.

In our next example, we generate a gradient for the background of a CCI line:

//@version=6 indicator("CCI Background") bullColor = input.color(color.lime, " ", inline = "1") bearColor = input.color(color.fuchsia, " ", inline = "1") // Calculate CCI. myCCI = ta.cci(hlc3, 20) // Get relative position of CCI in last 100 bars, on a 0-100% scale. myCCIPosition = ta.percentrank(myCCI, 100) // Generate a bull gradient when position is 50-100%, bear gradient when position is 0-50%. backgroundColor = if myCCIPosition >= 50 color.from_gradient(myCCIPosition, 50, 100, color.new(bullColor, 75), bullColor) else color.from_gradient(myCCIPosition, 0, 50, bearColor, color.new(bearColor, 75)) // Wider white line background. plot(myCCI, "CCI", color.white, 3) // Think black line. plot(myCCI, "CCI", color.black,

1) // Zero level. hline(0) // Gradient background.
bgcolor(backgroundColor)

image

Note that:

- We use the ta.cci() built-in function to calculate the indicator value.
- We use the ta.percentrank() built-in function to calculate `myCCIPosition`, i.e., the percentage of past `myCCI` values in the last 100 bars that are below the current value of `myCCI`.
- To calculate the gradient, we use two different calls of the color.from_gradient() built-in: one for the bull gradient when `myCCIPosition` is in the 50-100% range, which means that more past values are below its current value, and another for the bear gradient when `myCCIPosition` is in the 0-49.99% range, which means that more past values are above it.
- We provide inputs so the user can change the bull/bear colors, and we place both color input widgets on the same line using `inline = "1"` in both input.color() calls.
- We plot the CCI signal using two plot() calls to achieve the best contrast over the busy background: the first plot is a 3-pixel wide white background, the second plot() call plots the thin, 1-pixel wide black line.

See the Colors page for more examples of backgrounds.

Previous

Next

Bar coloring

Bar coloring

The barcolor() function colors bars on the main chart, regardless of whether the script is running in the main chart pane or a separate pane.

The function's signature is:

```
barcolor(color, offset, editable, show_last, title, display) → void
```

The coloring can be conditional because the `color` parameter accepts "series color" arguments.

The following script renders *inside* and *outside* bars in different colors:

image

//@version=6 indicator("barcolor example", overlay = true) isUp = close > open isDown = close <= open isOutsideUp = high > high[1] and low < low[1] and isUp isOutsideDown = high > high[1] and low < low[1] and isDown isInside = high < high[1] and low > low[1] barcolor(isInside ? color.yellow : isOutsideUp ? color.aqua : isOutsideDown ? color.purple : na)

Note that:

- The na value leaves bars as is.
- In the barcolor() call, we use embedded ?: ternary operator expressions to select the color.

Next

Bar plotting

Bar plotting

Introduction

The plotcandle() built-in function is used to plot candles. plotbar() is used to plot conventional bars.

Both functions require four arguments that will be used for the OHLC prices (open, high, low, close) of the bars they will be plotting. If one of those is na, no bar is plotted.

Plotting candles with `plotcandle()`

The signature of plotcandle() is:

```
plotcandle(open, high, low, close, title, color, wickcolor, editabl
```

This plots simple candles, all in blue, using the habitual OHLC values, in a separate pane:

//@version=6 indicator("Single-color candles") plotcandle(open, high, low, close)

image

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

//@version=6 indicator("Example 2") paletteColor = close >= open ? color.lime : color.red plotbar(open, high, low, close, color = paletteColor)

image

Note that the `color` parameter accepts “series color” arguments, so constant values such as `color.red`, `color.lime`, `"#FF9090"`, as well as expressions that calculate colors at runtime, as is done with the `paletteColor` variable here, will all work.

You can build bars or candles using values other than the actual OHLC values. For example you could calculate and plot smoothed candles using the following code, which also colors wicks depending on the position of close relative to the smoothed close (c) of our indicator:

//@version=6 indicator("Smoothed candles", overlay = true) lenInput = input.int(9) smooth(source, length) => ta.sma(source, length) o = smooth(open, lenInput) h = smooth(high, lenInput) l = smooth(low, lenInput) c = smooth(close, lenInput) ourWickColor = close > c ? color.green : color.red plotcandle(o, h, l, c, wickcolor = ourWickColor)

image

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

// NOTE: Use this script on an intraday chart. //@version=6 indicator("Daily bars", behind_chart = false, overlay = true) // Use gaps to return data only when the 1D timeframe completes, and to return `na` otherwise. [o, h, l, c] = request.security(syminfo.tickerid, "D", [open, high, low, close], gaps = barmerge.gaps_on) const color UP_COLOR = color.silver const color DN_COLOR = color.blue color wickColor = c >= o ? UP_COLOR : DN_COLOR color bodyColor = c >= o ? color.new(UP_COLOR, 70) : color.new(DN_COLOR, 70) // Plot candles on intraday timeframes, // and when non `na` values are returned by `request.security()` because a HTF bar has completed. plotcandle(timeframe.isintraday ? o : na, h, l, c, color = bodyColor, wickcolor = wickColor)

image

Note that:

- We set the `behind_chart` parameter of the indicator declaration to `false`. This causes our script’s candles to appear on top of the

chart's candles. Selecting "Visual Order/Bring to Front" from the script's "More" menu achieves the same result.

- The script displays candles only when two conditions are met:
  - The chart is using an intraday timeframe (see the check on `timeframe.isintraday` in the plotcandle() call). We do this because it's not useful to show a daily value on timeframes higher or equal to 1D.
  - The request.security() function returns non na values (see `gaps = barmerge.gaps_on` in the function call).
- We use a tuple (`[open, high, low, close]`) with request.security() to fetch four values in one call.
- We create a lighter transparency for the body of our candles in the `bodyColor` variable initialization, so they don't obstruct the chart's candles.

Plotting bars with `plotbar()`

The signature of plotbar() is:

```
plotbar(open, high, low, close, title, color, editable, show_last,
```

Note that plotbar() has no parameter for `bordercolor` or `wickcolor`, as there are no borders or wicks on conventional bars.

This plots conventional bars using the same coloring logic as in the second example of the previous section:

//@version=6 indicator("Dual-color bars") paletteColor = close >= open ? color.lime : color.red plotbar(open, high, low, close, color = paletteColor)

image

Previous

Next

Bar states

Pine Script® v6 Documentation

Bar states

Introduction

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

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

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

Bar state built-in variables

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

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

`barstate.isfirst`

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

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

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

barstate.islast

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

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

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

barstate.ishistory

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

barstate.isrealtime

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

barstate.isnew

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

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

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

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

barstate.isconfirmed

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

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

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

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

barstate.islastconfirmedhistory

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

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

Example

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

//@version=6 indicator("Bar States", overlay = true, max_labels_count = 500) stateText() => string txt = "" txt += barstate.isfirst ? "isfirst\n" : "" txt += barstate.islast ? "islast\n" : "" txt += barstate.ishistory ? "ishistory\n" : "" txt += barstate.isrealtime ? "isrealtime\n" : "" txt += barstate.isnew ? "isnew\n" : "" txt += barstate.isconfirmed ? "isconfirmed\n" : "" txt += barstate.islastconfirmedhistory ? "islastconfirmedhistory\n" : "" labelColor = switch barstate.isfirst => color.fuchsia barstate.islastconfirmedhistory => color.gray barstate.ishistory => color.silver barstate.isconfirmed => color.orange barstate.isnew => color.red => color.yellow label.new(bar_index, na, stateText(), yloc = yloc.abovebar, color = labelColor)

Note that:

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

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

image

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

image

Note that:

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

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

image

Previous

Next

Chart information

Pine Script® v6 Documentation

Chart information

Introduction

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

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

Prices and volume

The built-in variables for OHLCV values are:

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

Other values are available through:

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

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

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

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

Symbol information

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

- syminfo.basecurrency: the base currency, e.g., "BTC" in "BTCUSD", or "EUR" in "EURUSD".
- syminfo.currency: the quote currency, e.g., "USD" in "BTCUSD", or "CAD" in "USDCAD".
- syminfo.description: The long description of the symbol.

- syminfo.main_tickerid: The symbol's *main* ticker identifier. It behaves almost identically to syminfo.tickerid, referencing the symbol's exchange prefix, name, and additional ticker data. However, this variable *always* represents the *current* chart's ticker ID, even within requested contexts.
- syminfo.mincontract: The symbol's smallest tradable amount, which is set by its exchange. For example, the minimum for NASDAQ asset "AAPL" is 1 token, while the minimum for BITSTAMP cryptocurrency "ETHUSD" is 0.0001 tokens.
- syminfo.mintick: The symbol's tick value, or the minimum increment price can move in. Not to be confused with *pips* or *points*. On "ES1!" ("S&P 500 E-Mini") the tick size is 0.25 because that is the minimal increment the price moves in.
- syminfo.pointvalue: The point value is the multiple of the underlying asset determining a contract's value. On "ES1!" ("S&P 500 E-Mini") the point value is 50, so a contract is worth 50 times the price of the instrument.
- syminfo.prefix: The prefix is the exchange or broker's identifier: "NASDAQ" or "BATS" for "AAPL", "CME_MINI_DL" for "ES1!".
- syminfo.root: It is the ticker's prefix for structured tickers like those of futures. It is "ES" for "ES1!", "ZW" for "ZW1!".
- syminfo.session: It reflects the session setting on the chart for that symbol. If the "Chart settings/Symbol/Session" field is set to "Extended", it will only return "extended" if the symbol and the user's feed allow for extended sessions. It is rarely displayed and used mostly as an argument to the `session` parameter in ticker.new().
- syminfo.ticker: It is the symbol's name, without the exchange part (syminfo.prefix): "BTCUSD", "AAPL", "ES1!", "USDCAD".
- syminfo.tickerid: The symbol's ticker identifier, consisting of its exchange prefix and symbol name, e.g., "NASDAQ:MSFT". It can also include ticker information beyond the "prefix:ticker" form, such as extended hours, dividend adjustments, currency conversion, etc. To retrieve the standard "prefix:ticker" form only, pass the variable to ticker.standard(). When used in a `request.*()` call's `expression` argument, this variable references

the *requested* context's ticker ID. Otherwise, it references the current chart's ticker ID.

- syminfo.timezone: The timezone the symbol is traded in. The string is an IANA time zone database name (e.g., "America/New_York").
- syminfo.type: The type of market the symbol belongs to. The values are "stock", "futures", "index", "forex", "crypto", "fund", "dr", "cfd", "bond", "warrant", "structured" and "right".

This script displays these built-in variables and their values for the current symbol in a table on the chart:

image

//@version=6 indicator("`syminfo.*` built-ins demo", overlay = true) //@variable The `syminfo.*` built-ins, displayed in the left column of the table. string txtLeft = "syminfo.basecurrency: " + "\n" + "syminfo.currency: " + "\n" + "syminfo.description: " + "\n" + "syminfo.main_tickerid: " + "\n" + "syminfo.mincontract: " + "\n" + "syminfo.mintick: " + "\n" + "syminfo.pointvalue: " + "\n" + "syminfo.prefix: " + "\n" + "syminfo.root: " + "\n" + "syminfo.session: " + "\n" + "syminfo.ticker: " + "\n" + "syminfo.tickerid: " + "\n" + "syminfo.timezone: " + "\n" + "syminfo.type: " //@variable The values of the `syminfo.*` built-ins, displayed in the right column of the table. string txtRight = syminfo.basecurrency + "\n" + syminfo.currency + "\n" + syminfo.description + "\n" + syminfo.main_tickerid + "\n" + str.tostring(syminfo.mincontract) + "\n" + str.tostring(syminfo.mintick) + "\n" + str.tostring(syminfo.pointvalue) + "\n" + syminfo.prefix + "\n" + syminfo.root + "\n" + syminfo.session + "\n" + syminfo.ticker + "\n" + syminfo.tickerid + "\n" + syminfo.timezone + "\n" + syminfo.type if barstate.islast var table t = table.new(position.middle_right, 2, 1) table.cell(t, 0, 0, txtLeft, bgcolor = color.yellow, text_halign = text.align_right) table.cell(t, 1, 0, txtRight, bgcolor = color.yellow, text_halign = text.align_left)

Chart timeframe

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

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

Additional built-ins return more specific timeframe information:

- timeframe.multiplier returns a “simple int” containing the multiplier of the timeframe unit. A chart timeframe of one hour will return `60` because intraday timeframes are expressed in minutes. A 30sec timeframe will return `30` (seconds), a daily chart will return 1 (day), a quarterly chart will return 3 (months), and a yearly chart will return 12 (months). The value of this variable cannot be used as an argument to `timeframe` parameters in built-in functions, as they expect a string in timeframe specifications format.
- timeframe.period holds a “string” representing the script’s timeframe. It follows Pine’s timeframe string specifications, where the string consists of a quantity (multiplier) and unit, e.g., “1D”, “2W”, “3M”. When used in a `request.*()` call’s `expression` argument, this variable references the *requested* context’s timeframe. Otherwise, it references the script’s main timeframe.
- timeframe.main_period holds a “string” representing the *main* timeframe, which is either the `timeframe` argument specified in the indicator() declaration, or the current chart’s timeframe. It behaves almost identically to timeframe.period. However, this variable *always* represents the script’s *main* timeframe, even within requested contexts.

See the page on Timeframes for more information.

Session information

Session information is available in different forms:

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

Previous

Next

Colors

Pine Script® v6 Documentation

Colors

Introduction

Script visuals can play a critical role in the usability of the indicators we write in Pine Script®. Well-designed plots and drawings make indicators easier to use and understand. Good visual designs establish a visual hierarchy that allows the more important information to stand out, and the less important one to not get in the way.

Using colors in Pine can be as simple as you want, or as involved as your concept requires. The 4,294,967,296 possible assemblies of color and transparency available in Pine Script can be applied to:

- Any element you can plot or draw in an indicator's visual space, be it lines, fills, text or candles.
- The background of a script's visual space, whether the script is running in its own pane, or in overlay mode on the chart.
- The color of bars or the body of candles appearing on a chart.

A script can only color the elements it places in its own visual space. The only exception to this rule is that a pane indicator can color chart bars or candles.

Pine Script has built-in colors such as color.green, as well as functions like color.rgb() which allow you to dynamically generate any color in the RGBA color space.

Transparency

Each color in Pine Script is defined by four values:

- Its red, green and blue components (0-255), following the RGB color model.
- Its transparency (0-100), often referred to as the Alpha channel outside Pine, as defined in the RGBA color model. Even though transparency is expressed in the 0-100 range, its value can be a "float" when used in functions, which gives you access to the 256 underlying values of the alpha channel.

The transparency of a color defines how opaque it is: zero is fully opaque, 100 makes the color — whichever it is — invisible. Modulating transparency can be crucial in more involved color visuals or when using backgrounds, to control which colors dominate the others, and how they mix together when superimposed.

Z-index

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

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

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

- Background colors
- Fills

- Plots
- Hlines
- LineFills
- Lines
- Boxes
- Labels
- Tables

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

Constant colors

There are 17 built-in colors in Pine Script. This table lists their names, hexadecimal equivalent, and RGB values as arguments to color.rgb():

Name	Hex	RGB values
color.aqua	#00BCD4	color.rgb(0, 188, 212)
color.black	#363A45	color.rgb(54, 58, 69)
color.blue	#2196F3	color.rgb(33, 150, 243)
color.fuchsia	#E040FB	color.rgb(224, 64, 251)
color.gray	#787B86	color.rgb(120, 123, 134)
color.green	#4CAF50	color.rgb(76, 175, 80)
color.lime	#00E676	color.rgb(0, 230, 118)
color.maroon	#880E4F	color.rgb(136, 14, 79)
color.navy	#311B92	color.rgb(49, 27, 146)
color.olive	#808000	color.rgb(128, 128, 0)
color.orange	#FF9800	color.rgb(255, 152, 0)
color.purple	#9C27B0	color.rgb(156, 39, 176)
color.red	#F23645	color.rgb(242, 54, 69)
color.silver	#B2B5BE	color.rgb(178, 181, 190)
color.teal	#089981	color.rgb(8, 153, 129)
color.white	#FFFFFF	color.rgb(255, 255, 255)
color.yellow	#FDD835	color.rgb(253, 216, 53)

The following script shows three different ways to express color.olive with 40% transparency. All three methods are functionally equivalent:

image

//@version=6 indicator("Constant colors demo", overlay = true) // Create a plot using the hex color code equivalent for `color.olive` with `99` as the alpha value (60% opacity). plot(ta.sma(close, 10), "10-bar SMA", #80800099, 3) // Create a plot using `color.new()` to modify `color.olive` with 40% transparency. plot(ta.sma(close, 30), "30-bar SMA", color.new(color.olive, 40), 3) // Create a plot using `color.rgb()` with the `r`, `g`, and `b` components of `color.olive` and 40% transparency. plot(ta.sma(close, 50), "50-bar SMA", color.rgb(128, 128, 0, 40), 3)

Note that:

- An alpha value of 99 in a hexadecimal color code is equivalent to 60% opacity, meaning the resulting color is 40% transparent.
- Transparency does *not* affect plot outputs in the status line, price scale, or Data Window. All these locations show the color with 0% transparency.

The colors in the previous script do not vary as the script executes bar to bar. Sometimes, however, colors need to be created as the script executes on each bar because they depend on conditions that are unknown at compile time, or when the script begins execution on bar zero. For those cases, programmers have two options:

1. Use conditional statements to select colors from a few pre-determined base colors.
2. Build new colors dynamically, by calculating them as the script executes bar to bar, to implement color gradients, for example.

Conditional coloring

Let’s say you want to color a moving average in different colors, depending on some conditions you define. To do so, you can use a conditional statement that will select a different color for each of your

states. Let’s start by coloring a moving average in a bull color when it’s rising, and in a bear color when it’s not:

image

//@version=6 indicator("Conditional colors", "", true) int lengthInput = input.int(20, "Length", minval = 2) color maBullColorInput = input.color(color.green, "Bull") color maBearColorInput = input.color(color.maroon, "Bear") float ma = ta.sma(close, lengthInput) // Define our states. bool maRising = ta.rising(ma, 1) // Build our color. color maColor = maRising ? maBullColorInput : maBearColorInput plot(ma, "MA", maColor, 2)

Note that:

- We provide users of our script a selection of colors for our bull/bear colors.
- We define an `maRising` boolean variable which will hold `true` when the moving average is higher on the current bar than it was on the last.
- We define an `maColor` variable that is assigned one of our two colors, depending on the value of the `maRising` variable. We use the ternary operator to define our conditional expression.

You can also use conditional colors to avoid plotting under certain conditions. Here, we plot high and low pivots using a line, but we do not want to plot anything when a new pivot comes in, to avoid the joints that would otherwise appear in pivot transitions. To do so, we test for pivot changes and use na as the color value when a change is detected, so that no line is plotted on that bar:

image

//@version=6 indicator("Conditional colors", "", true) int legsInput = input.int(5, "Pivot Legs", minval = 1) color pHiColorInput = input.color(color.olive, "High pivots") color pLoColorInput = input.color(color.orange, "Low pivots") // Intialize the pivot level variables. var float pHi = na var float pLo = na // When a new pivot is detected, save its value. pHi := nz(ta.pivothigh(legsInput, legsInput), pHi) pLo := nz(ta.pivotlow(legsInput, legsInput), pLo) // When a new

pivot is detected, do not plot a color. plot(pHi, "High", ta.change(pHi) != 0 ? na : pHiColorInput, 2, plot.style_line) plot(pLo, "Low", ta.change(pLo) != 0 ? na : pLoColorInput, 2, plot.style_line)

To undertand how this code works, one must first know that ta.pivothigh() and ta.pivotlow(), used as they are here without an argument to the `source` parameter, will return a value when they find a high/low pivot, otherwise they return na.

When we test the value returned by the pivot function for na using the nz() function, we allow the value returned to be assigned to the `pHi` or `pLo` variables only when it is not na, otherwise the previous value of the variable is simply reassigned to it, which has no impact on its value. Keep in mind that previous values of `pHi` and `pLo` are preserved bar to bar because we use the var keyword when initializing them, which causes the initialization to only occur on the first bar.

All that's left to do next is, when we plot our lines, to insert a ternary conditional statement that will yield na for the color when the pivot value changes, or the color selected in the script's inputs when the pivot level does not change.

Calculated colors

Using functions like color.new(), color.rgb() and color.from_gradient(), one can build colors on the fly, as the script executes bar to bar.

color.new() is most useful when you need to generate different transparency levels from a base color.

color.rgb() is useful when you need to build colors dynamically from red, green, blue, or tranparency components. While color.rgb() creates a color, its sister functions color.r(), color.g(), color.b() and color.t() can be used to extract the red, green, blue or transparency values from a color, which can in turn be used to generate a variant.

color.from_gradient() is useful to create linear gradients between two base colors. It determines which intermediary color to use by evaluating a source value against minimum and maximum values.

color.new()

Let’s put color.new() to use to create different transparencies for volume columns using one of two bull/bear base colors:

image

//@version=6 indicator("Volume") // We name our color constants to make them more readable. var color GOLD_COLOR = #CCCC00ff var color VIOLET_COLOR = #AA00FFff color bullColorInput = input.color(GOLD_COLOR, "Bull") color bearColorInput = input.color(VIOLET_COLOR, "Bear") int levelsInput = input.int(10, "Gradient levels", minval = 1) // We initialize only once on bar zero with `var`, otherwise the count would reset to zero on each bar. var float riseFallCnt = 0 // Count the rises/falls, clamping the range to: 1 to `i_levels`. riseFallCnt := math.max(1, math.min(levelsInput, riseFallCnt + math.sign(volume - nz(volume[1])))) // Rescale the count on a scale of 80, reverse it and cap transparency to <80 so that colors remains visible. float transparency = 80 - math.abs(80 * riseFallCnt / levelsInput) // Build the correct transparency of either the bull or bear color. color volumeColor = color.new(close > open ? bullColorInput : bearColorInput, transparency) plot(volume, "Volume", volumeColor, 1, plot.style_columns)

Note that:

- In the next to last line of our script, we dynamically calculate the column color by varying both the base color used, depending on whether the bar is up or down, **and** the transparency level, which is calculated from the cumulative rises or falls of volume.
- We offer the script user control over not only the base bull/bear colors used, but also on the number of brightness levels we use. We use this value to determine the maximum number of rises or falls we will track. Giving users the possiblity to manage this value allows them to adapt the indicator’s visuals to the timeframe or market they use.
- We take care to control the maximum level of transparency we use so that it never goes higher than 80. This ensures our colors always retain some visibility.

- We also set the minimum value for the number of levels to 1 in the inputs. When the user selects 1, the volume columns will be either in bull or bear color of maximum brightness — or transparency zero.

color.rgb()

In our next example we use color.rgb() to build colors from RGBA values. We use the result in a holiday season gift for our friends, so they can bring their TradingView charts to parties:

image

//@version=6 indicator("Holiday candles", "", true) float r = math.random(0, 255) float g = math.random(0, 255) float b = math.random(0, 255) float t = math.random(0, 100) color holidayColor = color.rgb(r, g, b, t) plotcandle(open, high, low, close, color = holidayColor, wickcolor = holidayColor, bordercolor = holidayColor)

Note that:

- We generate values in the zero to 255 range for the red, green and blue channels, and in the zero to 100 range for transparency. Also note that because math.random() returns float values, the float 0.0-100.0 range provides access to the full 0-255 transparency values of the underlying alpha channel.
- We use the math.random(min, max, seed) function to generate pseudo-random values. We do not use an argument for the third parameter of the function: seed. Using it is handy when you want to ensure the repeatability of the function's results. Called with the same seed, it will produce the same sequence of values.

color.from_gradient()

Our last examples of color calculations will use color.from_gradient(). Let's first use it in its simplest form, to color a CCI signal in a version of the indicator that otherwise looks like the built-in:

image

```
//@version=6 indicator(title="CCI line gradient", precision=2, timeframe="") var color GOLD_COLOR = #CCCC00 var color VIOLET_COLOR = #AA00FF var color BEIGE_COLOR = #9C6E1B float srcInput = input.source(close, title="Source") int lenInput = input.int(20, "Length", minval = 5) color bullColorInput = input.color(GOLD_COLOR, "Bull") color bearColorInput = input.color(BEIGE_COLOR, "Bear") float signal = ta.cci(srcInput, lenInput) color signalColor = color.from_gradient(signal, -200, 200, bearColorInput, bullColorInput) plot(signal, "CCI", signalColor) bandTopPlotID = hline(100, "Upper Band", color.silver, hline.style_dashed) bandBotPlotID = hline(-100, "Lower Band", color.silver, hline.style_dashed) fill(bandTopPlotID, bandBotPlotID, color.new(BEIGE_COLOR, 90), "Background")
```

Note that:

- To calculate the gradient, color.from_gradient() requires minimum and maximum values against which the argument used for the `value` parameter will be compared. The fact that we want a gradient for an unbounded signal like CCI (i.e., without fixed boundaries such as RSI, which always oscillates between 0-100), does not entail we cannot use color.from_gradient(). Here, we solve our conundrum by providing values of -200 and 200 as arguments. They do not represent the real minimum and maximum values for CCI, but they are at levels from which we do not mind the colors no longer changing, as whenever the series is outside the `bottom_value` and `top_value` limits, the colors used for `bottom_color` and `top_color` will apply.
- The color progression calculated by color.from_gradient() is linear. If the value of the series is halfway between the `bottom_value` and `top_value` arguments, the generated color's RGBA components will also be halfway between those of `bottom_color` and `top_color`.
- Many common indicator calculations are available in Pine Script as built-in functions. Here we use ta.cci() instead of calculating it the long way.

The argument used for `value` in color.from_gradient() does not necessarily have to be the value of the line we are calculating.

Anything we want can be used, as long as arguments for `bottom_value` and `top_value` can be supplied. Here, we enhance our CCI indicator by coloring the band using the number of bars since the signal has been above/below the centerline:

image

//@version=6 indicator(title="CCI line gradient", precision=2, timeframe="") var color GOLD_COLOR = #CCCC00 var color VIOLET_COLOR = #AA00FF var color GREEN_BG_COLOR = color.new(color.green, 70) var color RED_BG_COLOR = color.new(color.maroon, 70) float srcInput = input.source(close, "Source") int lenInput = input.int(20, "Length", minval = 5) int stepsInput = input.int(50, "Gradient levels", minval = 1) color bullColorInput = input.color(GOLD_COLOR, "Line: Bull", inline = "11") color bearColorInput = input.color(VIOLET_COLOR, "Bear", inline = "11") color bullBgColorInput = input.color(GREEN_BG_COLOR, "Background: Bull", inline = "12") color bearBgColorInput = input.color(RED_BG_COLOR, "Bear", inline = "12") // Plot colored signal line. float signal = ta.cci(srcInput, lenInput) color signalColor = color.from_gradient(signal, -200, 200, color.new(bearColorInput, 0), color.new(bullColorInput, 0)) plot(signal, "CCI", signalColor, 2) // Detect crosses of the centerline. bool signalX = ta.cross(signal, 0) // Count no of bars since cross. Capping it to the no of steps from inputs. int gradientStep = math.min(stepsInput, nz(ta.barssince(signalX))) // Choose bull/bear end color for the gradient. color endColor = signal > 0 ? bullBgColorInput : bearBgColorInput // Get color from gradient going from no color to `c_endColor` color bandColor = color.from_gradient(gradientStep, 0, stepsInput, na, endColor) bandTopPlotID = hline(100, "Upper Band", color.silver, hline.style_dashed) bandBotPlotID = hline(-100, "Lower Band", color.silver, hline.style_dashed) fill(bandTopPlotID, bandBotPlotID, bandColor, title = "Band")

Note that:

- The signal plot uses the same base colors and gradient as in our previous example. We have however increased the width of the line from the default 1 to 2. It is the most important component of

our visuals; increasing its width is a way to give it more prominence, and ensure users are not distracted by the band, which has become busier than it was in its original, flat beige color.

- The fill must remain unobtrusive for two reasons. First, it is of secondary importance to the visuals, as it provides complementary information, i.e., the duration for which the signal has been in bull/bear territory. Second, since fills have a greater z-index than plots, the fill will cover the signal plot. For these reasons, we make the fill's base colors fairly transparent, at 70, so they do not mask the plots. The gradient used for the band starts with no color at all (see the na used as the argument to `bottom_color` in the color.from_gradient() call), and goes to the base bull/bear colors from the inputs, which the conditional, `c_endColor` color variable contains.
- We provide users with distinct bull/bear color selections for the line and the band.
- When we calculate the `gradientStep` variable, we use nz() on ta.barssince() because in early bars of the dataset, when the condition tested has not occurred yet, ta.barssince() will return na. Because we use nz(), the value returned is replaced with zero in those cases.

Mixing transparencies

In this example we take our CCI indicator in another direction. We will build dynamically adjusting extremes zone buffers using a Donchian Channel (historical highs/lows) calculated from the CCI. We build the top/bottom bands by making them 1/4 the height of the DC. We will use a dynamically adjusting lookback to calculate the DC. To modulate the lookback, we will calculate a simple measure of volatility by keeping a ratio of a short-period ATR to a long one. When that ratio is higher than 50 of its last 100 values, we consider the volatility high. When the volatility is high/low, we decrease/increase the lookback.

Our aim is to provide users of our indicator with:

- The CCI line colored using a bull/bear gradient, as we illustrated in our most recent examples.
- The top and bottom bands of the Donchian Channel, filled in such a way that their color darkens as a historical high/low becomes older and older.
- A way to appreciate the state of our volatility measure, which we will do by painting the background with one color whose intensity increases when volatility increases.

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

image

And with the dark theme:

image

//@version=6 indicator("CCI DC", precision = 6) color GOLD_COLOR = #CCCC00ff color VIOLET_COLOR = #AA00FFff int lengthInput = input.int(20, "Length", minval = 5) color bullColorInput = input.color(GOLD_COLOR, "Bull") color bearColorInput = input.color(VIOLET_COLOR, "Bear") // ——————— Function clamps `val` between `min` and `max`. clamp(val, min, max) => math.max(min, math.min(max, val)) // ——————— Volatility expressed as 0-100 value. float v = ta.atr(lengthInput / 5) / ta.atr(lengthInput * 5) float vPct = ta.percentrank(v, lengthInput * 5) // ——————— Calculate dynamic lookback for DC. It increases/decreases on low/high volatility. bool highVolatility = vPct > 50 var int lookBackMin = lengthInput * 2 var int lookBackMax = lengthInput * 10 var float lookBack = math.avg(lookBackMin, lookBackMax) lookBack += highVolatility ? -2 : 2 lookBack := clamp(lookBack, lookBackMin, lookBackMax) // ——————— Dynamic lookback length Donchian channel of signal. float signal = ta.cci(close, lengthInput) // `lookBack` is a float; need to cast it to int to be used a length. float hiTop = ta.highest(signal, int(lookBack)) float loBot = ta.lowest(signal, int(lookBack)) // Get margin of 25% of the DC height to build high and low bands. float margin = (hiTop - loBot) / 4 float hiBot = hiTop - margin float loTop = loBot + margin // Center of DC. float center = math.avg(hiTop, loBot) // ——————— Create colors.

color signalColor = color.from_gradient(signal, -200, 200, bearColorInput, bullColorInput) // Bands: Calculate transparencies so the longer since the hi/lo has changed, // the darker the color becomes. Cap highest transparency to 90. float hiTransp = clamp(100 - (100 * math.max(1, nz(ta.barssince(ta.change(hiTop) != 0) + 1)) / 255), 60, 90) float loTransp = clamp(100 - (100 * math.max(1, nz(ta.barssince(ta.change(loBot) != 0) + 1)) / 255), 60, 90) color hiColor = color.new(bullColorInput, hiTransp) color loColor = color.new(bearColorInput, loTransp) // Background: Rescale the 0-100 range of `vPct` to 0-25 to create 75-100 transparencies. color bgColor = color.new(color.gray, 100 - (vPct / 4)) // ——————— Plots // Invisible lines for band fills. hiTopPlotID = plot(hiTop, color = na) hiBotPlotID = plot(hiBot, color = na) loTopPlotID = plot(loTop, color = na) loBotPlotID = plot(loBot, color = na) // Plot signal and centerline. p_signal = plot(signal, "CCI", signalColor, 2) plot(center, "Centerline", color.silver, 1) // Fill the bands. fill(hiTopPlotID, hiBotPlotID, hiColor) fill(loTopPlotID, loBotPlotID, loColor) // ——————— Background. bgcolor(bgColor)

Note that:

- We clamp the transparency of the background to a 100-75 range so that it doesn’t overwhelm. We also use a neutral color that will not distract too much. The darker the background is, the higher our measure of volatility.
- We also clamp the transparency values for the band fills between 60 and 90. We use 90 so that when a new high/low is found and the gradient resets, the starting transparency makes the color somewhat visible. We do not use a transparency lower than 60 because we don’t want those bands to hide the signal line.
- We use the very handy ta.percentrank() function to generate a 0-100 value from our ATR ratio measuring volatility. It is useful to convert values whose scale is unknown into known values that can be used to produce transparencies.
- Because we must clamp values three times in our script, we wrote an `f_clamp()` function, instead of explicitly coding the logic three times.

Tips

Maintaining automatic color selectors

Under certain conditions, PineScript can automatically display all of the colors used in script's plots in the "Settings/Style" tab. These plots are graphics created by all `plot*()` functions, barcolor(), and bgcolor(). The user can change the colors using a color picker. This feature allows colors in scripts to be customized without any extra code.

For example, this simple script uses plot() to display a teal and red plot of the close series. The plot's color for each bar depends on the relationship between the bar's close and open values. The script does not specify that the plot's colors should be editable, nor does it create any color-related inputs in the "Settings/Inputs" tab. However, it automatically displays the colors in the "Settings/Style" tab and allows the user to change them, along with the style of the plot:

image

//@version=6 indicator("Color picker showcase") plotColor = close > open ? color.teal : color.red plot(close, color = plotColor)

The colors in the above script can be automatically displayed in this way because they are *not dynamically calculated* and are known as soon as the script has finished compiling. All colors of the "const" type, and all colors of type "input" that are *not modified* via the color.new() or color.rgb() functions can be automatically displayed like this.

However, if *even a single calculated color* is of type "simple color" or "series color", or if an "input color" is passed to color.new() or color.rgb(), *all* colors are calculated in the script's runtime, and no color pickers are available in the "Style" section.

In practice, the creation of "simple" or "series" colors is also most often due to using color.new() and color.rgb() functions. The qualifier of the color that these functions return is the strongest qualifier of the values passed to them. If each call to these functions passes only "const"

values, the resulting colors are also “const”, and the script *does* display them in the “Style” tab.

For example, let’s try to make the plots in the script above semi-transparent by adding a transparency of `50` to its colors via color.new(). The easiest way to do this is to wrap the `plotColor` variable with color.new(), like in the example below:

//@version=6 indicator("Color picker showcase") plotColor = color.new(close > open ? color.teal : color.red, 50) plot(close, color = plotColor)

Unfortunately, with these changes the “Style” tab does not display a color picker any longer. This is because we use the “series bool” condition `close > open` to decide the color, and then pass the result of this expression to a single color.new() call. The qualified type of the calculated color that it returns is “series color”.

image

image

To avoid this, we can ensure that every calculated color created by color.new() is a “const color”. Below, we wrap teal and red separately with color.new() – creating two constant calculated colors in the process – and then decide which one to assign to `plotColor` based on the condition. And while the `plotColor` variable is a “series color”, each color.new() call returns a constant color, so the script displays a color picker n the “Style” tab:

//@version=6 indicator("Color picker showcase") plotColor = close > open ? color.new(color.teal, 50) : color.new(color.red, 50) plot(close, color = plotColor)

image

To calculate the colors at runtime, create custom color inputs for all of the colors that are to be editable. This approach requires more effort, but allows significantly more control over what the user can affect. Learn more about creating color inputs on the Inputs page.

Designing usable colors schemes

If you write scripts intended for other traders, try to avoid colors that will not work well in some environments, whether it be for plots, labels, tables or fills. At a minimum, test your visuals to ensure they perform satisfactorily with both the light and dark TradingView themes; they are the most commonly used. Colors such as black and white, for example, should be avoided.

Build the appropriate inputs to provide script users the flexibility to adapt your script's visuals to their particular environments.

Take care to build a visual hierarchy of the colors you use that matches the relative importance of your script's visual components. Good designers understand how to achieve the optimal balance of color and weight so the eye is naturally drawn to the most important elements of the design. When you make everything stand out, nothing does. Make room for some elements to stand out by toning down the visuals surrounding it.

Providing a selection of color presets in your inputs — rather than a single color that can be changed — can help color-challenged users. Our Technical Ratings demonstrates one way of achieving this.

Plot crisp lines

It is best to use zero transparency to plot the important lines in your visuals, to keep them crisp. This way, they will show through fills more precisely. Keep in mind that fills have a higher z-index than plots, so they are placed on top of them. A slight increase of a line's width can also go a long way in making it stand out.

If you want a special plot to stand out, you can also give it more importance by using multiple plots for the same line. These are examples where we modulate the successive width and transparency of plots to achieve this:

image

```
//@version=6 indicator("") plot(high, "", color.new(color.orange, 80), 8)
plot(high, "", color.new(color.orange, 60), 4) plot(high, "",
color.new(color.orange, 00), 1) plot(hl2, "", color.new(color.orange, 60),
4) plot(hl2, "", color.new(color.orange, 00), 1) plot(low, "",
color.new(color.orange, 0), 1)
```

Customize gradients

When building gradients, adapt them to the visuals they apply to. If you are using a gradient to color candles, for example, it is usually best to limit the number of steps in the gradient to ten or less, as it is more difficult for the eye to perceive intensity variations of discrete objects. As we did in our examples, cap minimum and maximum transparency levels so your visual elements remain visible and do not overwhelm when it’s not necessary.

Previous

Next

Fills

Fills

Introduction

Some of Pine Script's visual outputs, including plots, hlines, lines, boxes, and polylines, allow one to fill the chart space they occupy with colors. Three different mechanisms facilitate filling the space between such outputs:

- The fill() function fills the space between two plots from plot() calls or two horizontal lines (hlines) from hline() calls with a specified color.
- Objects of the linefill type fill the space between line instances created with line.new().
- Other drawing types, namely boxes and polylines, have built-in properties that allow the drawings to fill the visual spaces they occupy.

plot() and hline() fills

The fill() function fills the space between two plots or horizontal lines. It has the following two signatures:

```
fill(plot1, plot2, color, title, editable, show_last, fillgaps) → v
fill(hline1, hline2, color, title, editable, fillgaps) → void
```

The `plot1`, `plot2`, `hline1`, and `hline2` parameters accept plot or hline IDs returned by plot() and hline() function calls. The fill() function is the only built-in that can use these IDs.

This simple example demonstrates how the fill() function works with plot and hline IDs. It calls plot() and hline() three times to display arbitrary values on the chart. Each of these calls returns an ID, which the script assigns to variables for use in the fill() function. The values of p1, p2, and p3 are “plot” IDs, whereas h1, h2, and h3 reference “hline” IDs:

image

//@version=6 indicator("Example 1") // Assign "plot" IDs to the `p1`, `p2`, and `p3` variables. p1 = plot(math.sin(high), "Sine of `high`") p2 = plot(math.cos(low), "Cosine of `low`") p3 = plot(math.sin(close), "Sine of `close`") // Fill the space between `p1` and `p2` with 90% transparent red. fill(p1, p3, color.new(color.red, 90), "`p1`-`p3` fill") // Fill the space between `p2` and `p3` with 90% transparent blue. fill(p2, p3, color.new(color.blue, 90), "`p2`-`p3` fill") // Assign "hline" IDs to the `h1`, `h2`, and `h3` variables. h1 = hline(0, "First level") h2 = hline(1.0, "Second level") h3 = hline(0.5, "Third level") h4 = hline(1.5, "Fourth level") // Fill the space between `h1` and `h2` with 90% transparent yellow. fill(h1, h2, color.new(color.yellow, 90), "`h1`-`h2` fill") // Fill the space between `h3` and `h4` with 90% transparent lime. fill(h3, h4, color.new(color.lime, 90), "`h3`-`h4` fill")

It’s important to note that the fill() function requires *either* two “plot” IDs or two “hline” IDs. One *cannot* mix and match these types in the function call. Consequently, programmers will sometimes need to use plot() where they otherwise might have used hline() if they want to fill the space between a consistent level and a fluctuating series.

For example, this script calculates an `oscillator` based on the percentage distance between the chart’s close price and the 10-bar moving average from a ta.sma() call, then plots it on the chart pane. In this case, we wanted to fill the area between the `oscillator` and zero. Although we can display the zero level with hline() since its value does not change, we cannot pass a “plot” and “hline” ID to the fill() function. Therefore, we must use a plot() call for the level to allow the script to fill the space:

image

//@version=6 indicator("Example 2") //@variable The 10-bar moving average of `close` prices. float ma = ta.sma(close, 10) //@variable The distance from the `ma` to the `close` price, as a percentage of the `ma`. float oscillator = 100 * (ma - close) / ma //@variable The ID of the `oscillator` plot for use in the `fill()` function. oscPlotID = plot(oscillator, "Oscillator") //@variable The ID of the zero level plot for use in the `fill()` function. // Requires a "plot" ID since the `fill()` function can't use "plot" and "hline" IDs at the same time. zeroPlotID = plot(0, "Zero level", color.silver, 1, plot.style_circles) // Filll the space between the `oscPlotID` and `zeroPlotID` with 90% transparent blue. fill(oscPlotID, zeroPlotID, color.new(color.blue, 90), "Oscillator fill")

The `color` parameter of the fill() function accepts a “series color” argument, meaning the fill’s color can change across chart bars. For example, this code fills the space between two moving average plots with 90% transparent green or red colors based on whether `ma1` is above `ma2`:

image

//@version=6 indicator("Example 3", overlay = true) //@variable The 5-bar moving average of `close` prices. float ma1 = ta.sma(close, 5) //@variable The 20-bar moving average of `close` prices. float ma2 = ta.sma(close, 20) //@variable The 90% transparent color of the space between MA plots. Green if `ma1 > ma2`, red otherwise. color fillColor = ma1 > ma2 ? color.new(color.green, 90) : color.new(color.red, 90) //@variable The ID of the `ma1` plot for use in the `fill()` function. ma1PlotID = plot(ma1, "5-bar SMA") //@variable The ID of the `ma2` plot for use in the `fill()` function. ma2PlotID = plot(ma2, "20-bar SMA") // Fill the space between the `ma1PlotID` and `ma2PlotID` using the `fillColor`. fill(ma1PlotID, ma2PlotID, fillColor, "SMA plot fill")

Line fills

While the fill() function allows a script to fill the space between two plots or hlines, it does not work with line objects. When a script needs to fill the space between lines, it requires a linefill object created by the linefill.new() function. The function has the following signature:

```
linefill.new(line1, line2, color) → series linefill
```

The `line1` and `line2` parameters accept line IDs. These IDs determine the chart region that the linefill object will fill with its specified `color`. A script can update the `color` property of a linefill ID returned by this function by calling linefill.set_color() with the ID as its `id` argument.

The behavior of linefills depends on the lines they reference. Scripts cannot move linefills directly, as the lines that a linefill uses determine the space it will fill. To retrieve the IDs of the lines referenced by a linefill object, use the linefill.get_line1() and linefill.get_line2() functions.

Any pair of line instances can only have *one* linefill between them. Successive calls to linefill.new() using the same `line1` and `line2` arguments will create a new linefill ID that *replaces* the previous one associated with them.

The example below demonstrates a simple use case for linefills. The script calculates a `pivotHigh` and `pivotLow` series using the built-in ta.pivothigh() and ta.pivotlow() functions with constant `leftbars` and `rightbars` arguments. On the last confirmed historical bar, the script draws two extended lines. The first line connects the two most recent non-na `pivotHigh` values, and the second connects the most recent non-na `pivotLow` values.

To emphasize the “channel” formed by these lines, the script fills the space between them using linefill.new():

image

//@version=6 indicator("Linefill demo", "Channel", true) //@variable The number bars to the left of a detected pivot. int LEFT_BARS = 15 // @variable The number bars to the right for pivot confirmation. int RIGHT_BARS = 5 //@variable The price of the pivot high point. float pivotHigh = ta.pivothigh(LEFT_BARS, RIGHT_BARS) //@variable The price of the pivot low point. float pivotLow = ta.pivotlow(LEFT_BARS, RIGHT_BARS) // Initialize the chart points the lines will use. var firstHighPoint = chart.point.new(na, na, na) var secondHighPoint = chart.point.new(na, na, na) var firstLowPoint = chart.point.new(na, na,

na) var secondLowPoint = chart.point.new(na, na, na) // Update the `firstHighPoint` and `secondHighPoint` when `pivotHigh` is not `na`. if not na(pivotHigh) firstHighPoint := secondHighPoint secondHighPoint := chart.point.from_index(bar_index - RIGHT_BARS, pivotHigh) // Update the `firstLowPoint` and `secondLowPoint` when `pivotlow` is not `na`. if not na(pivotLow) firstLowPoint := secondLowPoint secondLowPoint := chart.point.from_index(bar_index - RIGHT_BARS, pivotLow) if barstate.islastconfirmedhistory //@variable An extended line that passes through the `firstHighPoint` and `secondHighPoint`. line pivotHighLine = line.new(firstHighPoint, secondHighPoint, extend = extend.right) //@variable An extended line that passes through the `firstLowPoint` and `secondLowPoint`. line pivotLowLine = line.new(firstLowPoint, secondLowPoint, extend = extend.right) //@variable The color of the space between the lines. color fillColor = switch secondHighPoint.price > firstHighPoint.price and secondLowPoint.price > firstLowPoint.price => color.lime secondHighPoint.price < firstHighPoint.price and secondLowPoint.price < firstLowPoint.price => color.red => color.silver //@variable A linefill that colors the space between the `pivotHighLine` and `pivotLowLine`. linefill channelFill = linefill.new(pivotHighLine, pivotLowLine, color.new(fillColor, 90))

Box and polyline fills

The box and polyline types allow scripts to draw geometric shapes and other formations on the chart. Scripts create boxes and polylines with the box.new() and polyline.new() functions, which include parameters that allow the drawings to fill their visual spaces.

To fill the space inside the borders of a box with a specified color, include a `bgcolor` argument in the box.new() function. To fill a polyline's visual space, pass a `fill_color` argument to the polyline.new() function.

For example, this script draws an octagon with a polyline and an inscribed rectangle with a box on the last confirmed historical bar. It determines the size of the drawings using the value from the `radius` variable, which corresponds to approximately one-fourth of the number

of bars visible on the chart. We included `fill_color = color.new(color.blue, 60)` in the polyline.new() call to fill the octagon with a translucent blue color, and we used `bgcolor = color.purple` in the box.new() call to fill the inscribed rectangle with opaque purple:

image

//@version=6 indicator("Box and polyline fills demo") //@variable The number of visible chart bars, excluding the leftmost and rightmost bars. var int barCount = 0 if time > chart.left_visible_bar_time and time < chart.right_visible_bar_time barCount += 1 //@variable The approximate radius used to calculate the octagon and rectangle coordinates. int radius = math.ceil(barCount / 4) if barstate.islastconfirmedhistory //@variable An array of chart points. The polyline uses all points in this array, but the box only needs two. array<chart.point> points = array.new<chart.point>() //@variable The counterclockwise angle of each point, in radians. Updates on each loop iteration. float angle = 0.0 //@variable The radians to add to the `angle` on each loop iteration. float increment = 0.25 * math.pi // Loop 8 times to calculate octagonal points. for i = 0 to 7 //@variable The point's x-coordinate (bar offset). int x = int(math.round(math.cos(angle) * radius)) //@variable The point's y-coordinate. float y = math.round(math.sin(angle) * radius) // Push a new chart point into the `points` array and increase the `angle`. points.push(chart.point.from_index(bar_index - radius + x, y)) angle += increment // Create a closed polyline to draw the octagon and fill it with translucent blue. polyline.new(points, closed = true, fill_color = color.new(color.blue, 60)) // Create a box for the rectangle using index 3 and 7 for the top-left and bottom-right corners, // and fill it with opaque purple. box.new(points.get(3), points.get(7), bgcolor = color.purple)

See this manual's Lines and boxes page to learn more about working with these types.

Previous

Next

Inputs

Pine Script® v6 Documentation

Inputs

Introduction

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

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

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

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

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

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

- Double-click on the name of an on-chart indicator
- Right-click on the script's name and choose the "Settings" item from the dropdown menu

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

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

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

image

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

Input functions

Pine Script® features the following input functions:

- input()
- input.int()
- input.float()
- input.bool()
- input.color()
- input.string()
- input.text_area()
- input.timeframe()
- input.symbol()
- input.source()
- input.session()
- input.time()
- input.price()
- input.enum()

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

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

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

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

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

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

Input function parameters

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

All these parameters expect “const” arguments. The only exceptions are for the `defval` and `options` parameters of the source and enum

inputs, as input.source() uses “series float” value, and input.enum() uses *members* of an enum type.

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

Let’s go over each parameter:

- `defval` is the first parameter of all input functions. It is the default value that appears in the input widget. It requires an argument of the type of input value that the function applies to.
- `title` requires a “const string” argument. It is the field’s label.
- `tooltip` requires a “const string” argument. When the parameter is used, a question mark icon will appear to the right of the field. When users hover over it, the tooltip’s text will appear. Note that if multiple input fields are grouped on one line using `inline`, the tooltip will always appear to the right of the rightmost field, and display the text of the last `tooltip` argument used in the line. Newlines (`\n`) are supported in the argument string.
- `inline` requires a “const string” argument. Using the same argument for the parameter in multiple `input.*()` calls will group their input widgets on the same line. There is a limit to the width the “Inputs” tab will expand, so a limited quantity of input fields can be fitted on one line. Using one `input.*()` call with a unique argument for `inline` has the effect of bringing the input field left, immediately after the label, foregoing the default left-alignment of all input fields used when no `inline` argument is used.
- `group` requires a “const string” argument. Use it to group any number of inputs in an organized section. The string used as the `group` argument becomes the section’s heading. All `input.*()` calls to be grouped together must use the same string for their `group` argument.
- `options` requires a comma-separated list of elements enclosed in square brackets (e.g., `["ON", "OFF"]`, `[1, 2, 3]`, `[myEnum.On, myEnum.Off]`). The input uses the specified elements as menu selections in its resulting dropdown widget. Users can only select one menu item at a time. When supplying an `options` list, the

`defval` value must be one of the list's elements. Inputs that allow `minval`, `maxval`, or `step` parameters cannot use those parameters and the `options` parameter simultaneously.

- `minval` requires a "const int/float" argument, depending on the type of the `defval` value. It is the minimum valid value for the input field.
- `maxval` requires a "const int/float" argument, depending on the type of the `defval` value. It is the maximum valid value for the input field.
- `step` is the increment by which the field's value will move when the widget's up/down arrows are used.
- `confirm` requires a "const bool" (`true` or `false`) argument. This parameter affects the behavior of the script when it is added to a chart. `input.*()` calls using `confirm = true` will cause the "Settings/Inputs" tab to popup when the script is added to the chart. `confirm` is useful to ensure that users configure a particular field.

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

Input types

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

Generic input

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

Its signature is:

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

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

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

image

Integer input

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

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

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

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

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

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

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

image

Float input

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

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

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

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

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

image

Boolean input

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

image

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

Note that:

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

Color input

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

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

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

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

image

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

Note that:

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

String input

The input.string() function creates a string input with either a single-line *text field* or a *dropdown menu* of predefined text options. Other

`input.*()` functions also return “string” values. However, most of them are specialized for specific tasks, such as defining timeframes, symbols, and sessions.

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

Like the input.text_area() function, the input.string() text can contain up to 4096 characters, including horizontal whitespaces. However, because the input’s field in the “Settings/Inputs” tab is *narrow*, input.string() is best suited for defining small strings or for providing a quick set of input options for customizing calculations.

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

image

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

at the bar's `high` to show the `displayText`. label.new(bar_index, high, displayText)

Note that:

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

Text area input

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

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

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

image

//@version=6 indicator("Text area input demo", overlay = true) //@variable A comma-separated list of symbol names with optional exchange prefixes. string symbolListInput = input.text_area("AAPL,GOOG,NVDA,MSFT", "Symbol list") //@variable An array of symbol substrings formed by splitting the `symbolListInput` by its commas. var array<string> symbols = str.split(symbolListInput, ",") if barstate.islast //@variable A table displaying requested volume

data for each symbol in the `symbols` array. var table display = table.new(position.bottom_right, 2, symbols.size()) for [i, symbol] in symbols display.cell(0, i, symbol, text_color = chart.fg_color, text_size = 20) float vol = request.security(symbol, "", volume) display.cell(1, i, str.tostring(vol, format.volume), text_color = chart.fg_color, text_size = 20)

Note that:

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

Timeframe input

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

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

image

//@version=6 indicator("Timeframe input demo", "MA", true) // @variable The timeframe of the requested data. string tfInput = input.timeframe("1D", "Timeframe") // Get the typical number of seconds in the chart's timeframe and the `tfInput` timeframe. int chartSeconds = timeframe.in_seconds() int tfSeconds =

timeframe.in_seconds(tfInput) // Raise an error if the `tfInput` is a lower timeframe. if tfSeconds < chartSeconds runtime.error("The 'Timeframe' input must represent a timeframe higher than or equal to the chart's.") //@variable The offset of the requested expression. 1 when `tfInput` is a higher timeframe, 0 otherwise. int offset = chartSeconds == tfSeconds ? 0 : 1 //@variable The 20-bar SMA of `close` prices for the current symbol from the `tfInput` timeframe. float maHTF = request.security(syminfo.tickerid, tfInput, ta.sma(close, 20)[offset], lookahead = barmerge.lookahead_on) // Plot the `maHTF` value. plot(maHTF, "MA", color.aqua)

Note that:

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

Symbol input

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

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

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

Note that:

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

Session input

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

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

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

//@version=6 indicator("Session input", "", true) string sessionInput = input.session("0600-1700", "Session") string daysInput = input.string("1234567", tooltip = "1 = Sunday, 7 = Saturday") sessionString = sessionInput + ":" + daysInput inSession = not na(time(timeframe.period, sessionString)) bgcolor(inSession ? color.silver : na)

Note that:

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

image

Source input

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

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

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

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

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

image

Time input

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

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

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

image

//@version=6 indicator("Time input demo", overlay = true) //@variable A millisecond UNIX timestamp calculated from a specified date and time. // The input date and time are values in the chart's time zone, but the resulting UNIX timestamp // is time zone-agnostic. int dateAndTimeInput = input.time(timestamp("27 Dec 2024 09:30 -0500"), "Date and time") //@variable Is `true` if the bar's opening time is beyond the input date and time; `false` otherwise. bool barIsLater = time > dateAndTimeInput // Highlight the background when `barIsLater` is `true`. bgcolor(barIsLater ? color.new(color.blue, 70) : na, title = "Bar opened later highlight")

Note that:

- The vertical line to the left of the background highlight is visible when selecting the script's status line or opening the "Settings"

menu. Moving this line *changes* the input timestamp. Users can also change the time by choosing “Reset points” from the script’s “More” menu and selecting a new point directly on the chart.
- Changing the time zone in the chart’s settings can change the values shown in the input fields. However, the underlying UNIX timestamp does **not** change because it is unaffected by time zones.
- Users can *pair* time inputs with price inputs to create interactive chart points. See the next section to learn more.

Price input

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

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

image

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

Programmers can also *pair* price inputs and time inputs to add *interactive points* for custom calculations or drawings. When a script creates pairs of time and price inputs that belong to the same group, and each pair has a unique, matching `inline` argument, it adds *point*

markers on the chart instead of separate horizontal and vertical markers. Users can move these point markers to adjust input price and time values simultaneously.

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

image

//@version=6 indicator("Price and time input demo", overlay = true) // Create price and time inputs with the same `inline` arguments to set them together on the chart. // Price and time for the first point. float price1Input = input.price(0, "Price 1", inline = "1", confirm = true) int time1Input = input.time(0, "Time 1", inline = "1", confirm = true) // Price and time for the second point. float price2Input = input.price(0, "Price 2", inline = "2", confirm = true) int time2Input = input.time(0, "Time 2", inline = "2", confirm = true) // Price and time for the third point. float price3Input = input.price(0, "Price 3", inline = "3", confirm = true) int time3Input = input.time(0, "Time 3", inline = "3", confirm = true) // Price and time for the fourth point. float price4Input = input.price(0, "Price 4", inline = "4", confirm = true) int time4Input = input.time(0, "Time 4", inline = "4", confirm = true) //@variable An array of chart points created from the time and price inputs. var array<chart.point> points = array.from(chart.point.from_time(time1Input, price1Input), chart.point.from_time(time2Input, price2Input), chart.point.from_time(time3Input, price3Input), chart.point.from_time(time4Input, price4Input)) // Draw a closed, curved polyline connecting the points from the `points` array on the last bar. if barstate.islast var polyline shape = polyline.new(points, true, true, xloc.bar_time, color.purple, color.new(color.blue, 60))

Note that:

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

Enum input

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

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

image

//@version=6 indicator("Enum input demo", overlay = true) //@enum An enumeration of named values representing signal display modes. // @field long Named value to specify that only long signals are allowed. //@field short Named value to specify that only short signals are allowed. //@field both Named value to specify that either signal type is allowed. //@field none Named value to specify that no signals are allowed. enum SignalType long = "Only long signals" short = "Only short signals" both = "Long and short signals" none //@variable An

enumerator (member) of the `SignalType` enum. Controls the script's signals. SignalType sigInput = input.enum(SignalType.long, "Signal type") // Calculate moving averages. float ma1 = ta.sma(ohlc4, 10) float ma2 = ta.sma(ohlc4, 200) // Calculate cross signals. bool longCross = ta.crossover(close, math.max(ma1, ma2)) bool shortCross = ta.crossunder(close, math.min(ma1, ma2)) // Calculate long and short signals based on the selected `sigInput` value. bool longSignal = (sigInput == SignalType.long or sigInput == SignalType.both) and longCross bool shortSignal = (sigInput == SignalType.short or sigInput == SignalType.both) and shortCross // Plot shapes for the `longSignal` and `shortSignal`. plotshape(longSignal, "Long signal", shape.triangleup, location.belowbar, color.teal, size = size.normal) plotshape(shortSignal, "Short signal", shape.triangledown, location.abovebar, color.maroon, size = size.normal) // Plot the moving averages. plot(ma1, "Fast MA") plot(ma2, "Slow MA")

Note that:

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

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

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

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

image

Other features affecting Inputs

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

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

image

Tips

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

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

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

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

```
//@version=6 indicator("Aligned inputs", "", true) var GRP1 = "Not aligned" ma1SourceInput = input(close, "MA source", inline = "11", group = GRP1) ma1LengthInput = input(close, "Length", inline = "11", group = GRP1) long1SourceInput = input(close, "Signal source", inline = "12", group = GRP1) long1LengthInput = input(close, "Length", inline = "12", group = GRP1) var GRP2 = "Aligned" // The three spaces after "MA source" are Unicode EN spaces (U+2002). ma2SourceInput = input(close, "MA source   ", inline = "21", group = GRP2) ma2LengthInput = input(close, "Length", inline = "21", group = GRP2) long2SourceInput = input(close, "Signal source", inline = "22", group = GRP2) long2LengthInput = input(close, "Length", inline = "22", group = GRP2) plot(ta.vwma(close, 10))
```

image

Note that:

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

Previous

Next

Levels

Pine Script® v6 Documentation

User Manual / Concepts / Levels

Levels

`hline()` levels

Levels are lines plotted using the hline() function. It is designed to plot **horizontal** levels using a **single color**, i.e., it does not change on different bars. See the Levels section of the page on plot() for alternative ways to plot levels when hline() won't do what you need.

The function has the following signature:

```
hline(price, title, color, linestyle, linewidth, editable) → hline
```

hline() has a few constraints when compared to plot():

- Since the function's objective is to plot horizontal lines, its `price` parameter requires an "input int/float" argument, which means that "series float" values such as close or dynamically-calculated values cannot be used.
- Its `color` parameter requires an "input int" argument, which precludes the use of dynamic colors, i.e., colors calculated on each bar — or "series color" values.
- Three different line styles are supported through the `linestyle` parameter: `hline.style_solid`, `hline.style_dotted` and `hline.style_dashed`.

Let's see hline() in action in the "True Strength Index" indicator:

//@version=6 indicator("TSI") myTSI = 100 * ta.tsi(close, 25, 13) hline(50, "+50", color.lime) hline(25, "+25", color.green) hline(0, "Zero", color.gray, linestyle = hline.style_dotted) hline(-25, "-25", color.maroon) hline(-50, "-50", color.red) plot(myTSI)

image

image

Note that:

- We display 5 levels, each of a different color.
- We use a different line style for the zero centerline.
- We choose colors that will work well on both light and dark themes.
- The usual range for the indicator’s values is +100 to -100. Since the ta.tsi() built-in returns values in the +1 to -1 range, we make the adjustment in our code.

Fills between levels

The space between two levels plotted with hline() can be colored using fill(). Keep in mind that **both** plots must have been plotted with hline().

Let’s put some background colors in our TSI indicator:

//@version=6 indicator("TSI") myTSI = 100 * ta.tsi(close, 25, 13) plus50Hline = hline(50, "+50", color.lime) plus25Hline = hline(25, "+25", color.green) zeroHline = hline(0, "Zero", color.gray, linestyle = hline.style_dotted) minus25Hline = hline(-25, "-25", color.maroon) minus50Hline = hline(-50, "-50", color.red) // ———— Function returns a color in a light shade for use as a background. fillColor(color col) => color.new(col, 90) fill(plus50Hline, plus25Hline, fillColor(color.lime)) fill(plus25Hline, zeroHline, fillColor(color.teal)) fill(zeroHline, minus25Hline, fillColor(color.maroon)) fill(minus25Hline, minus50Hline, fillColor(color.red)) plot(myTSI)

image

image

Note that:

- We have now used the return value of our hline() function calls, which is of the hline special type. We use the `plus50Hline`, `plus25Hline`, `zeroHline`, `minus25Hline` and `minus50Hline`

variables to store those “hline” IDs because we will need them in our fill() calls later.

- To generate lighter color shades for the background colors, we declare a `fillColor()` function that accepts a color and returns its 90 transparency. We use calls to that function for the `color` arguments in our fill() calls.
- We make our fill() calls for each of the four different fills we want, between four different pairs of levels.
- We use `color.teal` in our second fill because it produces a green that fits the color scheme better than the `color.green` used for the 25 level.

Previous

Next

Libraries

Pine Script® v6 Documentation

Libraries

Introduction

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

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

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

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

Creating a library

A library is a special kind of script that begins with the library() declaration statement, rather than indicator() or strategy(). A library contains exportable function, method, UDT, and enum definitions, which constitute the only visible part of the library when imported by another script. Like other script types, libraries can also include Pine

Script code in their global scopes. Programmers typically use a library's global code to demonstrate how other scripts can use its exported structures.

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

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

Note that:

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

This is an example library:

//@version=6 // @description Provides functions calculating the all-time high/low of values. library("AllTimeHighLow", true) // @function Calculates the all-time high of a series. // @param val Series to use (`high` is used if no argument is supplied). // @returns The all-time

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

Library functions

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

In exported library function signatures (their first line):

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

Exported library functions have the following constraints:

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

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

Qualified type control

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

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

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

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

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

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

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

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

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

User-defined types and objects

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

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

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

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

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

Note that:

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

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

When a library only uses a UDT internally, it does not need to export the type. The following library uses the `point` type internally, but it

only exports the `drawPivots()` function, which does not have a parameter of the `point` type or return a `point` object:

//@version=6 library("PivotLabels", true) // We use this `point` UDT in the library, but it does NOT require exporting because // 1. The exported function's parameters do not use the UDT. // 2. The exported function does not return a UDT result. type point int x float y bool isHi bool wasBreached = false fillPivotsArray(qtyLabels, leftLegs, rightLegs) => // Create an array of the specified qty of pivots to maintain. var pivotsArray = array.new<point>(math.max(qtyLabels, 0)) // Detect pivots. float pivotHi = ta.pivothigh(leftLegs, rightLegs) float pivotLo = ta.pivotlow(leftLegs, rightLegs) // Create a new `point` object when a pivot is found. point foundPoint = switch not na(pivotHi) => point.new(time[rightLegs], pivotHi, true) not na(pivotLo) => point.new(time[rightLegs], pivotLo, false) => na // Add new pivot info to the array and remove the oldest pivot. if not na(foundPoint) array.push(pivotsArray, foundPoint) array.shift(pivotsArray) array<point> result = pivotsArray detectBreaches(pivotsArray) => // Detect breaches. for [i, eachPoint] in pivotsArray if not na(eachPoint) if not eachPoint.wasBreached bool hiWasBreached = eachPoint.isHi and high[1] <= eachPoint.y and high > eachPoint.y bool loWasBreached = not eachPoint.isHi and low[1] >= eachPoint.y and low < eachPoint.y if hiWasBreached or loWasBreached // This pivot was breached; change its `wasBreached` field. point p = array.get(pivotsArray, i) p.wasBreached := true array.set(pivotsArray, i, p) drawLabels(pivotsArray) => for eachPoint in pivotsArray if not na(eachPoint) label.new(eachPoint.x, eachPoint.y, str.tostring(eachPoint.y, format.mintick), xloc.bar_time, color = eachPoint.wasBreached ? color.gray : eachPoint.isHi ? color.teal : color.red, style = eachPoint.isHi ? label.style_label_down: label.style_label_up, textcolor = eachPoint.wasBreached ? color.silver : color.white) // @function Displays a label for each of the last `qtyLabels` pivots. // Colors high pivots in green, low pivots in red, and breached pivots in gray. // @param qtyLabels (simple int) Quantity of last labels to display. // @param leftLegs (simple int) Left pivot legs. // @param rightLegs (simple int) Right pivot legs. // @returns Nothing. export drawPivots(int qtyLabels, int leftLegs, int rightLegs) => // Gather pivots as they occur. pointsArray = fillPivotsArray(qtyLabels, leftLegs,

rightLegs) // Mark breached pivots. detectBreaches(pointsArray) // Draw labels once. if barstate.islastconfirmedhistory drawLabels(pointsArray) // Example use of the function. drawPivots(20, 10, 5)

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

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

Enum types

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

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

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

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

//@version=6 indicator("") import userName/Signal/1 as Signal // Calculate the median and quarter range values. float medianValue = ta.median(close, 100) float rangeValue = ta.range(close, 100) * 0.25 // Calculate upper and lower channel values. float upper = medianValue + rangeValue float lower = medianValue - rangeValue //@variable Returns `Signal.State.long`, `Signal.State.short`, or

`Signal.State.neutral` based on the price action. Signal.State mySignal = switch close > upper => Signal.State.long close < lower => Signal.State.short => Signal.State.neutral plot(close, color = mySignal == Signal.State.long ? color.green : mySignal == Signal.State.short ? color.red : na)

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

Publishing a library

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

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

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

image

Note that:

- We leave the library's title as is (the `title` argument in our library() declaration statement is used as the default). While you can change the publication's title, it is preferable to keep its default value because the `title` argument is used to reference imported libraries in the import statement. It makes life easier for library users when your publication's title matches the actual name of the library.
- A default description is built from the compiler annotations we used in our library. We will publish the library wihout retouching it.

- We chose to publish our library publicly, so it will be visible to all TradingViewers.
- We do not have the possibility of selecting a visibility type other than “Open” because libraries are always open-source.
- The list of categories for libraries is different than for indicators and strategies. We have selected the “Statistics and Metrics” category.
- We have added some custom tags: “all-time”, “high” and “low”.

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

House Rules

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

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

Using a library

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

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

where:

- The <username>/<libraryName>/<libraryVersion> path will uniquely identify the library.

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

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

import PineCoders/AllTimeHighLow/1 as allTime

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

image

This is an indicator that reuses our library:

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

Note that:

- We have chosen to use the "allTime" alias for the library's instance in our script. When typing that alias in the Editor, a popup will appear to help you select the particular function you want to use from the library.
- We use the library's `hi()` and `lo()` functions without an argument, so the default high and low built-in variables will be used for their series, respectively.

- We use a second call to `allTime.hi()`, but this time using close as its argument, to plot the highest close in the chart's history.

Previous

Next