

First steps

- [Introduction](#)
- [Using scripts](#)
 - [Loading scripts from the chart](#)
 - [Browsing Community Scripts](#)
 - [Changing script settings](#)
- [Reading scripts](#)
- [Writing scripts](#)

Introduction

Welcome to the Pine Script® [v5 User Manual](#), which will accompany you in your journey to learn to program your own trading tools in Pine Script®. Welcome also to the very active community of Pine Script® programmers on TradingView.

In this page, we present a step-by-step approach that you can follow to gradually become more familiar with indicators and strategies (also called *scripts*) written in the Pine Script® programming language on [TradingView](#). We will get you started on your journey to:

1. **Use** some of the tens of thousands of existing scripts on the platform.
2. **Read** the Pine Script® code of existing scripts.
3. **Write** Pine Script® scripts.

If you are already familiar with the use of Pine scripts on TradingView and are now ready to learn how to write your own, then jump to the [Writing scripts](#) section of this page.

If you are new to our platform, then please read on!

Using scripts

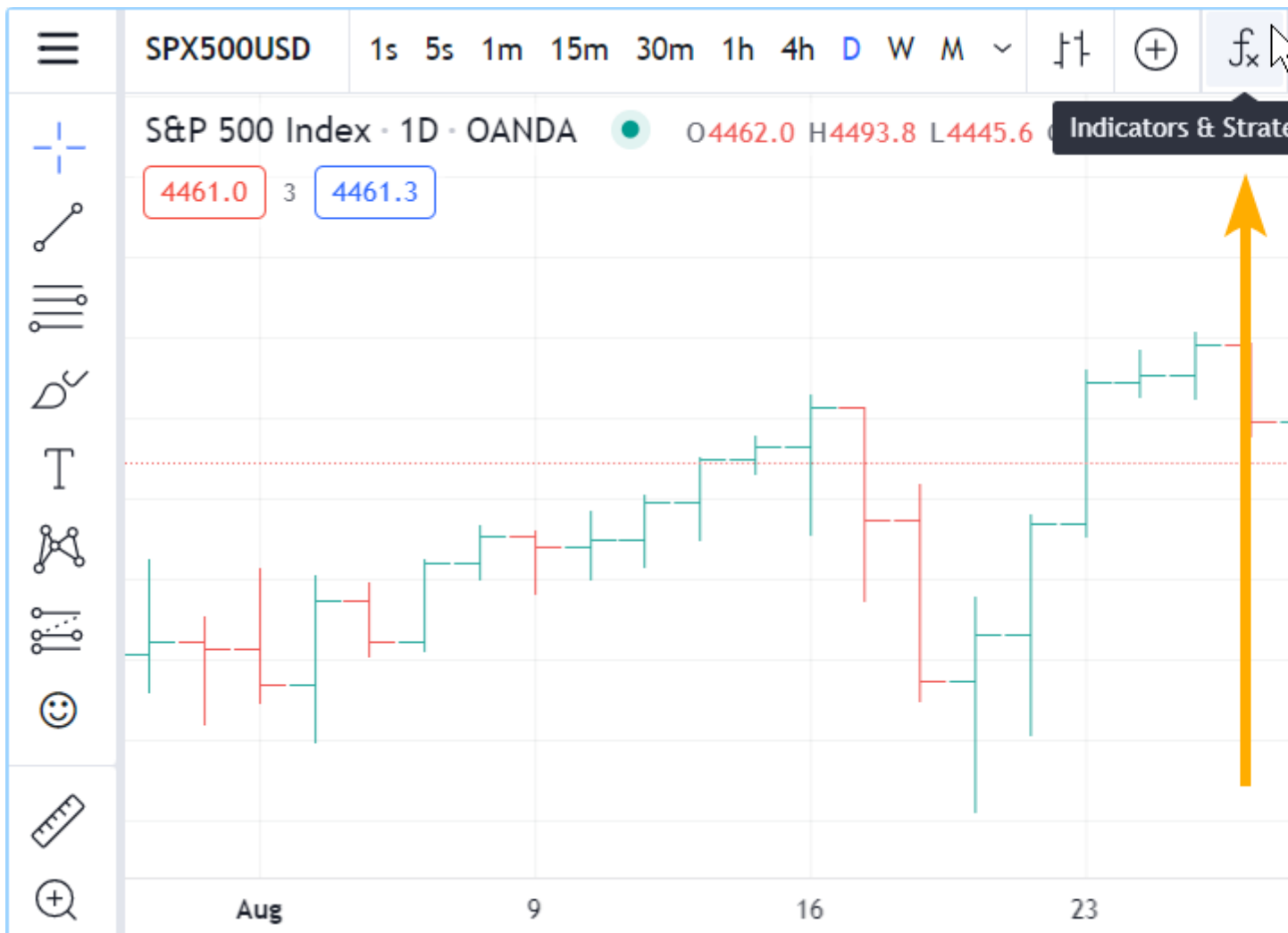
If you are interested in using technical indicators or strategies on TradingView, you can first start exploring the thousands of indicators already available on our platform. You can access existing indicators on the platform in two different ways:

- By using the chart's "Indicators & Strategies" button, or
- By browsing TradingView's [Community Scripts](#), the largest repository of trading scripts in the world, with more than 100,000 scripts, most of which are free and open-source, which means you can see their Pine Script® code.

If you can find the tools you need already written for you, it can be a good way to get started and gradually become proficient as a script user, until you are ready to start your programming journey in Pine Script®.

Loading scripts from the chart

To explore and load scripts from you chart, use the "Indicators & Strategies" button:



The dialog box presents different categories of scripts in its left pane:

- **Favorites** lists the scripts you have “favorited” by clicking on the star that appears to the left of its name when you mouse over it.
- **My scripts** displays the scripts you have written and saved in the Pine Script® Editor. They are saved in TradingView’s cloud.
- **Built-ins** groups all TradingView built-ins organized in four categories: indicators, strategies, candlestick patterns and volume profiles. Most are written in Pine Script® and available for free.
- **Community Scripts** is where you can search from the 100,000+ published scripts written by TradingView users.
- **Invite-only scripts** contains the list of the invite-only scripts you have been granted access to by their authors.

Here, the section containing the TradingView built-ins is selected:

Indicators & Strategies

Search

- Favorites
- My scripts
- Built-ins**
- Community Scripts
- Invite-only scripts

Indicators Strategies Candlestick patterns Volume

SCRIPT NAME

- Accumulation/Distribution
- Advance Decline Line
- Advance Decline Ratio
- Advance/Decline Ratio (Bars)

When you click on one of the indicators or strategies (the ones with the green and red arrows following their name), it loads on your chart.

Browsing Community Scripts

From [TradingView's homepage](#), you can bring up the Community Scripts stream from the "Community" menu. Here, we are pointing to the "Editors' Picks" section, but there are many other categories you can choose from:

SCRIPTS

Oscillators

Centered oscillators

Volatility

Trend analysis

Indicators and Strategies ?

Monthly Returns in PineScript Strategies ?



QuantNomad PREMIUM

Aug 16

I'm not 100% satisfied with the strategy performance output I receive from TradingView. Quite often I want to see something that is not available by default. I usually export raw trades/metrics from TradingView and then do additional analy...

978

30

Picked Education Visible Suggested Note For Author

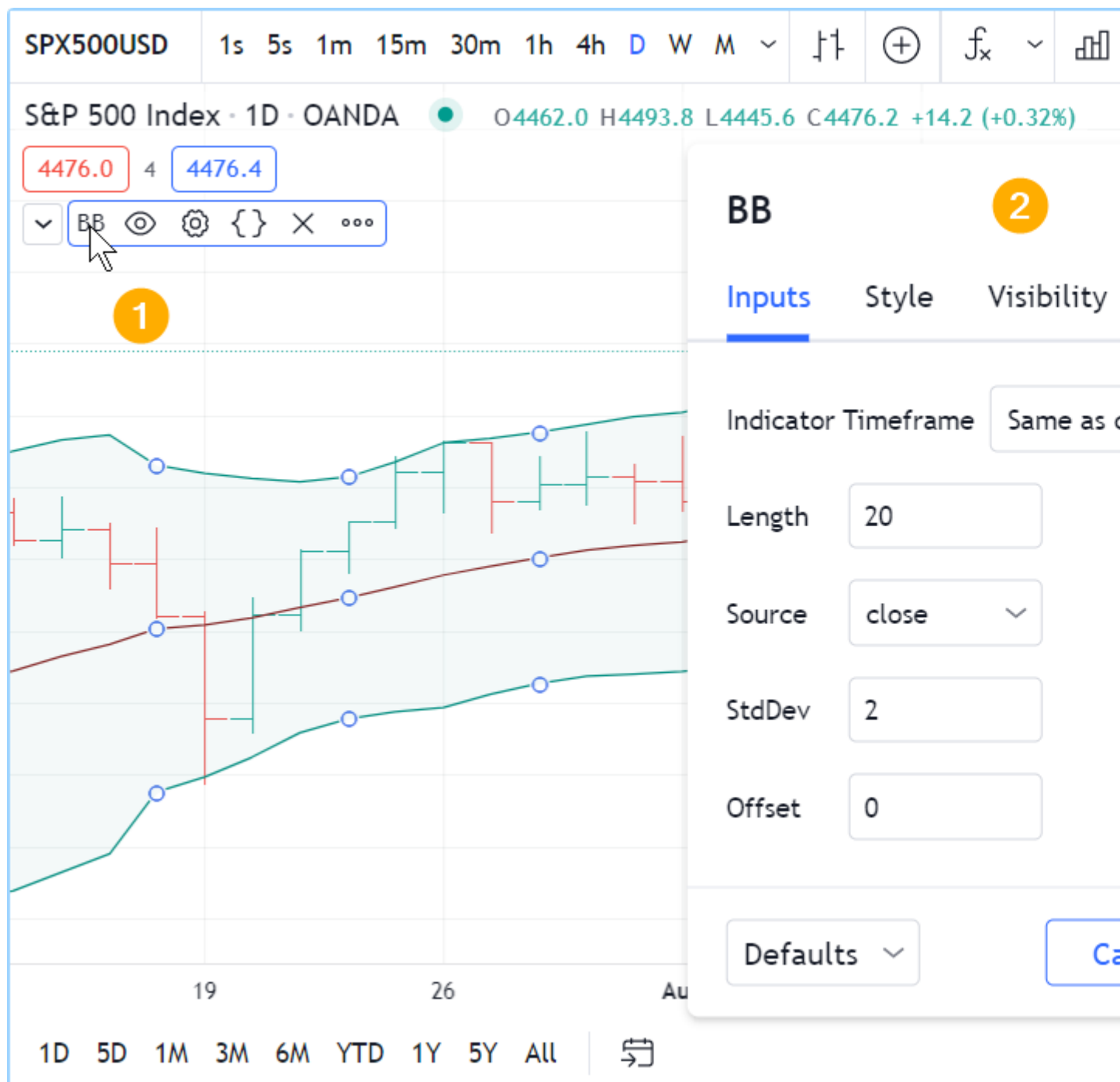
You can also search for scripts using the homepage’s “Search” field, and filter scripts using different criteria. The Help Center has a page explaining the [different types of scripts](#) that are available.

The scripts stream shows script *widgets*, i.e., placeholders showing a miniature view of each publication's chart and description, and its author. By clicking on it you will open the *script's page*, where you can see the script on a chart, read the author's description, like the script, leave comments or read the script's source code if it was published open-source.

Once you find an interesting script in the Community Scripts, follow the instructions in the Help Center to [load it on your chart](#).

Changing script settings

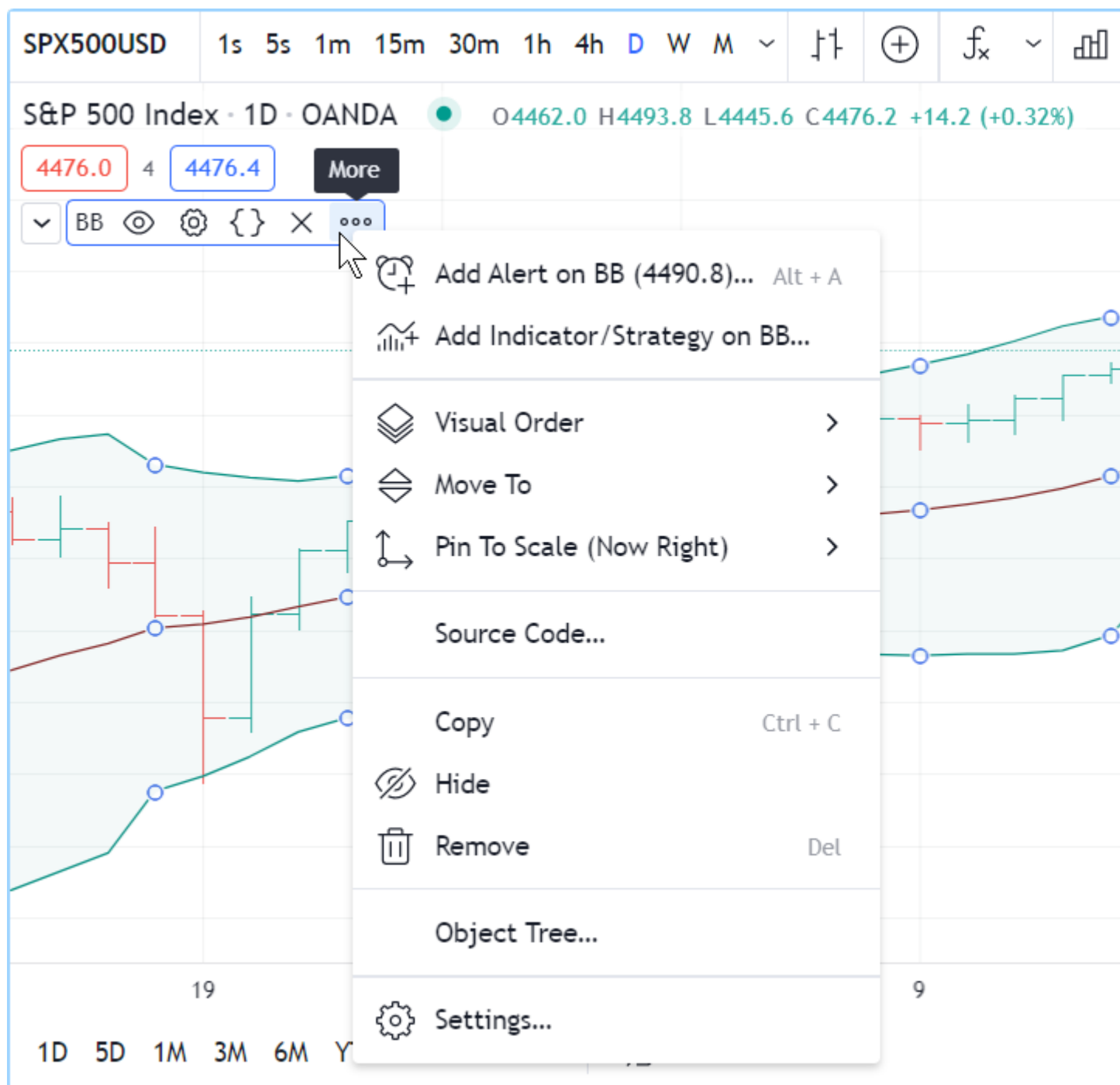
Once a script is loaded on the chart, you can double-click on its name (#1) to bring up its "Settings/Inputs" tab (#2):



The "Inputs" tab allows you to change the settings which the script's author has decided to make editable. You can configure some of the script's visuals using the "Style" tab of the same dialog

box, and which timeframes the script should appear on using the “Visibility” tab.

Other settings are available to all scripts from the buttons that appear to the right of its name when you mouse over it, and from the “More” menu (the three dots):



Reading scripts

Reading code written by **good** programmers is the best way to develop your understanding of the language. This is as true for Pine Script[®] as it is for all other programming languages. Finding good open-source Pine Script[®] code is relatively easy. These are reliable sources of code written by good programmers on TradingView:

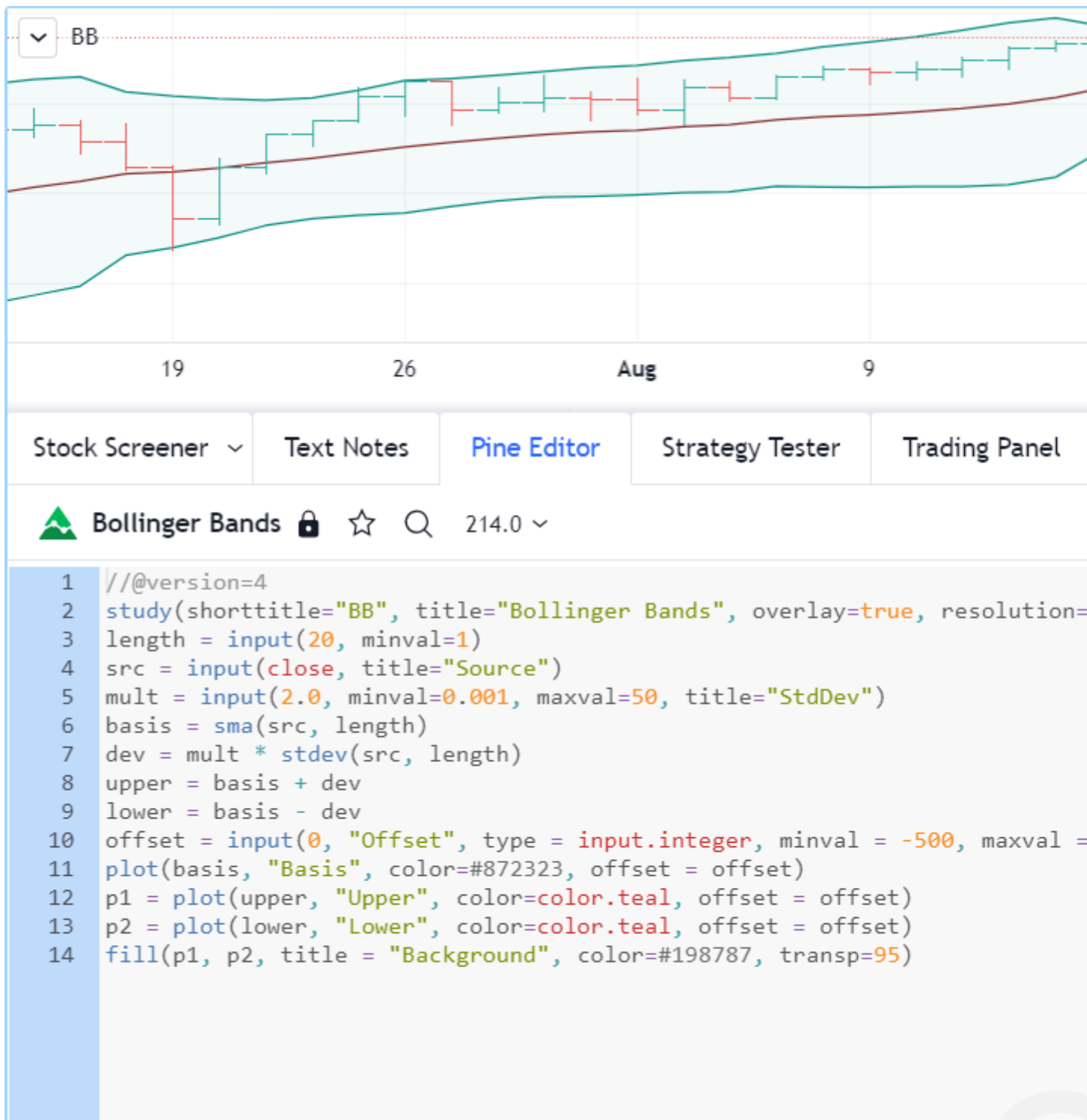
- The TradingView built-in indicators
- Scripts selected as [Editors' Picks](#)

- Scripts by the [authors the PineCoders account follows](#)
- Many scripts by authors with high reputation and open-source publications.

Reading code from [Community Scripts](#) is easy; if you don't see a grey or red "lock" icon in the upper-right corner of the script's widget, this indicates the script is open-source. By opening its script page, you will be able to see its source.

To see the code of TradingView built-ins, load the indicator on your chart, then hover over its name and select the "Source code" curly braces icon (if you don't see it, it's because the indicator's source is unavailable). When you click on the icon, the Pine Script® Editor will open and from there, you can see the script's code. If you want to play with it, you will need to use the Editor's "More" menu button at the top-right of the Editor's pane, and select "Make a copy...". You will then be able to modify and save the code. Because you will have created a different version of the script, you will need to use the Editor's "Add to Chart" button to add that new copy to the chart.

This shows the Pine Script® Editor having just opened after we selected the "View source" button from the indicator on our chart. We are about to make a copy of its source because it is read-only for now (indicated by the "lock" icon near its filename in the Editor):



You can also open TradingView built-in indicators from the Pine Script® Editor (accessible from the “Pine Script® Editor” tab at the bottom of the chart) by using the “Open/New default built-in script...” menu selection.

Writing scripts

We have built Pine Script® to empower both budding and seasoned traders to create their own trading tools. We have designed it so it is relatively easy to learn for first-time programmers — although learning a first programming language, like trading, is rarely **very** easy for anyone — yet powerful enough for knowledgeable programmers to build tools of moderate complexity.

Pine Script® allows you to write three types of scripts:

- **Indicators** like RSI, MACD, etc.
- **Strategies** which include logic to issue trading orders and can be backtested and forward-tested.
- **Libraries** which are used by more advanced programmers to package oft-used functions that can be reused by other scripts.

The next step we recommend is to write your [first indicator](#).

First indicator

- [The Pine Script® Editor](#)
- [First version](#)
- [Second version](#)
- [Next](#)

The Pine Script® Editor

The Pine Script® Editor is where you will be working on your scripts. While you can use any text editor you want to write your Pine scripts, using our Editor has many advantages:

- It highlights your code following Pine Script® syntax.
- It pops up syntax reminders for built-in and library functions when you hover over them.
- It provides quick access to the Pine Script® v5 Reference Manual popup when you `ctrl + click / cmd + click` on Pine Script® keywords.
- It provides an auto-complete feature that you can activate with `ctrl + space / cmd + space`.
- It makes the write/compile/run cycle fast because saving a new version of a script loaded on the chart also executes it immediately.
- While not as feature-rich as the top editors out there, it provides key functionality such as search and replace, multi-cursor and versioning.

To open the Editor, click on the “Pine Script® Editor” tab at the bottom of your TradingView chart. This will open up the Editor’s pane.

First version

We will now create our first working Pine script, an implementation of the [MACD](#) indicator in Pine Script®:

```
2 //@version=5
3 indicator("MACD #1")
4 fast = 12
5 slow = 26
6 fastMA = ta.ema(close, fast)
7 slowMA = ta.ema(close, slow)
8 macd = fastMA - slowMA
9 signal = ta.ema(macd, 9)
10 plot(macd, color = color.blue)
```

```
plot(signal, color = color.orange)
```

- Start by bringing up the “Open” dropdown menu at the top right of the Editor and choose “New blank indicator”.
- Then copy the example script above, taking care not to include the line numbers in your selection.
- Select all the code already in the editor and replace it with the example script.
- Click “Save” and choose a name for your script. Your script is now saved in TradingView’s cloud, but under your account’s name. Nobody but you can use it.
- Click “Add to Chart” in the Editor’s menu bar. The MACD indicator appears in a separate *Pane* under your chart.

Your first Pine script is running on your chart, which should look like this:



Let’s look at our script’s code, line by line:

Line 1: `//@version=5`

This is a [compiler annotation](#) telling the compiler the script will use version 5 of Pine Script®.

Line 2: `indicator("MACD #1")`

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

Line 3: `fast = 12`

Defines a `fast` integer variable which will be the length of the fast EMA.

Line 4: `slow = 26`

Defines a `slow` integer variable which will be the length of the slow EMA.

Line 5: `fastMA = ta.ema(close, fast)`

Defines the variable `fastMA`, containing the result of the EMA calculation (Exponential Moving Average) with a length equal to `fast` (12), on the `close` series, i.e., the closing price of bars.

Line 6: `slowMA = ta.ema(close, slow)`

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

Line 7: `macd = fastMA - slowMA`

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

Line 8: `signal = ta.ema(macd, 9)`

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

Line 9: `plot(macd, color = color.blue)`

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

Line 10: `plot(signal, color = color.orange)`

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

Second version

The first version of our script calculated MACD “manually”, but because Pine Script® is designed to write indicators and strategies, built-in Pine Script® functions exist for many common indicators, including one for... MACD: [ta.macd\(\)](#).

This is the second version of our script:

```

1 //@version=5
2 indicator("MACD #2")
3 fastInput = input(12, "Fast length")
4 slowInput = input(26, "Slow length")
5 [macdLine, signalLine, histLine] = ta.macd(close, fastInput, slowInput, 9)
6 plot(macdLine, color = color.blue)
7 plot(signalLine, color = color.orange)

```

Note that we have:

- Added inputs so we can change the lengths for the MAs
- We now use the [ta.macd\(\)](#) built-in to calculate our MACD, which saves us three lines and makes our code easier to read.

Let's repeat the same process as before to copy that code in a new indicator:

- Start by bringing up the “Open” dropdown menu at the top right of the Editor and choose “New blank indicator”.
- Then copy the example script above, again taking care not to include the line numbers in your selection.
- Select all the code already in the editor and replace it with the second version of our script.
- Click “Save” and choose a name for your script different than the previous one.
- Click “Add to Chart” in the Editor’s menu bar. The “MACD #2” indicator appears in a separate *Pane* under the “MACD #1” indicator.

Your second Pine script is running on your chart. If you double-click on the indicator’s name on your chart, you will bring up the script’s “Settings/Inputs” tab, where you can now change the slow and fast lengths:



Let's look at the lines that have changed in the second version of our script:

Line 2: `indicator("MACD #2")`

We have changed #1 to #2 so the second version of our indicator displays a different name on the chart.

Line 3: `fastInput = input(12, "Fast length")`

Instead of assigning a constant value to a variable, we have used the [input\(\)](#) function so we can change the value in our script’s “Settings/Inputs” tab. 12 will be the default value and the field’s label will be "Fast length". If the value is changed in the “Inputs” tab, the `fastInput` variable’s content will contain the new value and the script will re-execute on the chart with that new value. Note that, as our Pine Script® [Style Guide](#) recommends, we add `Input` to the end of the variable’s name to remind us, later in the script, that its value comes from a user input.

Line 4: `slowInput = input(26, "Slow length")`

We do the same for the slow length, taking care to use a different variable name, default value and text string for the field’s label.

Line 5: `[macdLine, signalLine, histLine] = ta.macd(close, fastInput, slowInput, 9)`

This is where we call the [ta.macd\(\)](#) built-in to perform all the first version’s calculations in one line only. The function requires four parameters (the values after the function name, enclosed in parentheses). It returns three values into the three variables instead of only one, like the functions we used until now, which is why we need to enclose the list of three variables receiving the function’s result in square brackets, to the left of the = sign. Note that

two of the values we pass to the function are the “input” variables containing the fast and slow lengths: `fastInput` and `slowInput`.

Line 6 and 7:

The variable names we are plotting there have changed, but the lines are doing the same thing as in our first version.

Our second version performs the same calculations as our first, but we can change the two lengths used to calculate it. Our code is also simpler and shorter by three lines. We have improved our script.

Next steps

- [“indicators” vs “strategies”](#)
- [How scripts are executed](#)
- [Time series](#)
- [Publishing scripts](#)
- [Getting around the Pine Script® documentation](#)
- [Where to go from here?](#)

After your [first steps](#) and your [first indicator](#), let us explore a bit more of the Pine Script® landscape by sharing some pointers to guide you in your journey to learn Pine Script®.

“indicators” vs “strategies”

Pine Script® [strategies](#) are used to backtest on historical data and forward test on open markets. In addition to indicator calculations, they contain `strategy.*()` calls to send trade orders to Pine Script®’s broker emulator, which can then simulate their execution. Strategies display backtest results in the “Strategy Tester” tab at the bottom of the chart, next to the “Pine Script® Editor” tab.

Pine Script® indicators also contain calculations, but cannot be used in backtesting. Because they do not require the broker emulator, they use less resources and will run faster. It is thus advantageous to use indicators whenever you can.

Both indicators and strategies can run in either overlay mode (over the chart’s bars) or pane mode (in a separate section below or above the chart). Both can also plot information in their respective space, and both can generate [alert events](#).

How scripts are executed

A Pine script is **not** like programs in many programming languages that execute once and then stop. In the Pine Script® *runtime* environment, a script runs in the equivalent of an invisible loop where it is executed once on each bar of whatever chart you are on, from left to right. Chart bars that have already closed when the script executes on them are called *historical bars*. When execution reaches the chart’s last bar and the market is open, it is on the *realtime bar*. The script then executes once every time a price or volume change is detected, and one last time for that realtime bar when it closes. That realtime bar then becomes an *elapsed realtime bar*. Note that when the script executes in realtime, it does not recalculate on all the chart’s historical bars on every price/volume update. It has already calculated once on those bars, so it does not need to recalculate them on every chart tick. See the [Execution model](#) page for more information.

When a script executes on a historical bar, the [close](#) built-in variable holds the value of that bar's close. When a script executes on the realtime bar, [close](#) returns the **current** price of the symbol until the bar closes.

Contrary to indicators, strategies normally execute only once on realtime bars, when they close. They can also be configured to execute on each price/volume update if that is what you need. See the page on [Strategies](#) for more information, and to understand how strategies calculate differently than indicators.

[Time series](#)

The main data structure used in Pine Script[®] is called a [time series](#). Time series contain one value for each bar the script executes on, so they continuously expand as the script executes on more bars. Past values of the time series can be referenced using the history-referencing operator: `[]`.

`close[1]`, for example, refers to the value of [close](#) on the bar preceding the one where the script is executing.

While this indexing mechanism may remind many programmers of arrays, a time series is different and thinking in terms of arrays will be detrimental to understanding this key Pine Script[®] concept. A good comprehension of both the [execution model](#) and [time series](#) is essential in understanding how Pine scripts work. If you have never worked with data organized in time series before, you will need practice to put them to work for you. Once you familiarize yourself with these key concepts, you will discover that by combining the use of time series with our built-in functions specifically designed to handle them efficiently, much can be accomplished in very few lines of code.

[Publishing scripts](#)

TradingView is home to a large community of Pine Script[®] programmers and millions of traders from all around the world. Once you become proficient enough in Pine Script[®], you can choose to share your scripts with other traders. Before doing so, please take the time to learn Pine Script[®] well-enough to supply traders with an original and reliable tool. All publicly published scripts are analyzed by our team of moderators and must comply with our [Script Publishing Rules](#), which require them to be original and well-documented.

If want to use Pine scripts for your own use, simply write them in the Pine Script[®] Editor and add them to your chart from there; you don't have to publish them to use them. If you want to share your scripts with just a few friends, you can publish them privately and send your friends the browser's link to your private publication. See the page on [Publishing](#) for more information.

[Getting around the Pine Script[®] documentation](#)

While reading code from published scripts is no doubt useful, spending time in our documentation will be necessary to attain any degree of proficiency in Pine Script[®]. Our two main sources of documentation on Pine Script[®] are:

- This Pine Script[®] [v5 User Manual](#)
- Our Pine Script[®] [v5 Reference Manual](#)

The Pine Script[®] [v5 User Manual](#) is in HTML format and in English only.

The Pine Script[®] [v5 Reference Manual](#) documents what each variable, function or keyword does. It is an essential tool for all Pine Script[®] programmers; your life will be miserable if you try to write

scripts of any reasonable complexity without consulting it. It exists in two formats: the HTML format we just linked to, and the popup version, which can be accessed from the Pine Script® Editor, by either `ctrl + clicking` on a keyword, or by using the Editor’s “More/Pine Script® reference (pop-up)” menu. The Reference Manual is translated in other languages.

There are five different versions of Pine Script®. Ensure the documentation you use corresponds to the Pine Script® version you are coding with.

[Where to go from here?](#)

This Pine Script® [v5 User Manual](#) contains numerous examples of code used to illustrate the concepts we discuss. By going through it, you will be able to both learn the foundations of Pine Script® and study the example scripts. Reading about key concepts and trying them out right away with real code is a productive way to learn any programming language. As you hopefully have already done in the [First indicator](#) page, copy this documentation’s examples in the Editor and play with them. Explore! You won’t break anything.

This is how the Pine Script® [v5 User Manual](#) you are reading is organized:

- The [Language](#) section explains the main components of the Pine Script® language and how scripts execute.
- The [Concepts](#) section is more task-oriented. It explains how to do things in Pine Script®.
- The [Writing](#) section explores tools and tricks that will help you write and publish scripts.
- The [FAQ](#) section answers common questions from Pine Script® programmers.
- The [Error messages](#) page documents causes and fixes for the most common runtime and compiler errors.
- The [Release Notes](#) page is where you can follow the frequent updates to Pine Script®.
- The [Migration guides](#) section explains how to port between different versions of Pine Script®.
- The [Where can I get more information](#) page lists other useful Pine Script®-related content, including where to ask questions when you are stuck on code.

We wish you a successful journey with Pine Script® ... and trading!

Execution model

- [Calculation based on historical bars](#)
- [Calculation based on realtime bars](#)
- [Events triggering the execution of a script](#)
- [More information](#)
- [Historical values of functions](#)
 - [Why this behavior?](#)
 - [Exceptions](#)

The execution model of the Pine Script® runtime is intimately linked to Pine Script®’s [time series](#) and [type system](#). Understanding all three is key to making the most of the power of Pine Script®.

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

[triggering the execution of a script](#) has occurred.

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

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

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

Calculation based on historical bars

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

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

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

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

```
src = close
```

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

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

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



Calculation based on realtime bars

The behavior of a Pine script on the realtime bar is very different than on historical bars. Recall that the realtime bar is the rightmost bar on the chart when trading is active on the chart's symbol. Also, recall that strategies can behave in two different ways in the realtime bar. By default, they only execute when the realtime bar closes, but the `calc_on_every_tick` parameter of the `strategy` declaration statement can be set to true to modify the strategy's behavior so that it

executes each time the realtime bar updates, as indicators do. The behavior described here for indicators will thus only apply to strategies using `calc_on_every_tick=true`.

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

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

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

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

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

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

To summarize the realtime bar process:

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

Events triggering the execution of a script

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

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

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

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

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

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

More information

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

Historical values of functions

Every function call in Pine leaves a trail of historical values that a script can access on subsequent bars using the `[]` operator. The historical series of functions depend on successive calls to record the output on every bar. When a script does not call functions on each bar, it can produce an inconsistent history that may impact calculations and results, namely when it depends on the continuity of their historical series to operate as expected. The compiler warns users in these cases to make them aware that the values from a function, whether built-in or user-defined, might be misleading.

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



```
//@version=5
indicator("My script")

//@function Calculates the index of the current bar by adding 1 to its own value
from the previous bar.
// The first bar will have an index of 0.
calcBarIndex() =>
    int index = na
```

```

    index := nz(index[1], replacement = -1) + 1

//@variable Returns `true` on every other bar.
condition = bar_index % 2 == 0

int customIndex = na

// Call `calcBarIndex()` when the `condition` is `true`. This prompts the
// compiler to raise a warning.
if condition
    customIndex := calcBarIndex()

plot(bar_index, "Bar index", color = color.green)
plot(customIndex, "Custom index", color = color.red, style = plot.style_cross)

```

Note that:

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

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

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



```

//@version=5
indicator("My script")

//@function Calculates the index of the current bar by adding 1 to its own value
// from the previous bar.
// The first bar will have an index of 0.
calcBarIndex() =>
    int index = na
    index := nz(index[1], replacement = -1) + 1

//@variable Returns `true` on every second bar.
condition = bar_index % 2 == 0

globalScopeBarIndex = calcBarIndex()
int customIndex = na

// Assign `customIndex` to `globalScopeBarIndex` when the `condition` is `true`.
// This won't produce a warning.
if condition
    customIndex := globalScopeBarIndex

plot(bar_index, "Bar index", color = color.green)
plot(customIndex, "Custom index", color = color.red, style = plot.style_cross)

```

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

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



```
//@version=5
indicator("My script")

//@variable Returns `true` on every second bar.
condition = bar_index % 2 == 0

controlSMA = ta.sma(close, 20)
float globalSMA = na
float localSMA = na

// Update `globalSMA` and `localSMA` when `condition` is `true`.
if condition
    globalSMA := controlSMA // No warning.
    localSMA := ta.sma(close, 20) // Raises warning. This function depends on
its history to work as intended.

plot(controlSMA, "Control SMA", color = color.green)
plot(globalSMA, "Global SMA", color = color.blue, style = plot.style_cross)
plot(localSMA, "Local SMA", color = color.red, style = plot.style_cross)
```

Why this behavior?

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

Exceptions

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

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

Time series

Much of the power of Pine Script[®] stems from the fact that it is designed to process *time series* efficiently. Time series are not a form or a type; they are the fundamental structure Pine Script[®] uses to store the successive values of a variable over time, where each value is tethered to a point in time. Since charts are composed of bars, each representing a particular point in time, time series are the ideal data structure to work with values that may change with time.

The notion of time series is intimately linked to Pine Script[®]'s [execution model](#) and [type system](#) concepts. Understanding all three is key to making the most of the power of Pine Script[®].

Take the built-in [open](#) variable, which contains the “open” price of each bar in the dataset, the *dataset* being all the bars on any given chart. If your script is running on a 5min chart, then each value in the [open](#) time series is the “open” price of the consecutive 5min chart bars. When your script refers to [open](#), it is referring to the “open” price of the bar the script is executing on. To refer to past values in a time series, we use the `[]` history-referencing operator. When a script is executing on a given bar, `open[1]` refers to the value of the [open](#) time series on the previous bar.

While time series may remind programmers of arrays, they are totally different. Pine Script[®] does use an array data structure, but it is a completely different concept than a time series.

Time series in Pine Script[®], combined with its special type of runtime engine and built-in functions, are what makes it easy to compute the cumulative total of [close](#) values without using a [for](#) loop, with only `ta.cum(close)`. This is possible because although `ta.cum(close)` appears rather static in a script, it is in fact executed on each bar, so its value becomes increasingly larger as the [close](#) value of each new bar is added to it. When the script reaches the rightmost bar of the chart, `ta.cum(close)` returns the sum of the [close](#) value from all bars on the chart.

Similarly, the mean of the difference between the last 14 [high](#) and [low](#) values can be expressed as `ta.sma(high - low, 14)`, or the distance in bars since the last time the chart made five consecutive higher highs as `barsince(rising(high, 5))`.

Even the result of function calls on successive bars leaves a trace of values in a time series that can be referenced using the `[]` history-referencing operator. This can be useful, for example, when testing the [close](#) of the current bar for a breach of the highest [high](#) in the last 10 bars, but excluding the current bar, which we could write as `breach = close > highest(close, 10)[1]`. The same statement could also be written as `breach = close > highest(close[1], 10)`.

The same looping logic on all bars is applied to function calls such as `plot(open)` which will repeat on each bar, successively plotting on the chart the value of [open](#) for each bar.

Do not confuse “time series” with the “series” form. The *time series* concept explains how consecutive values of variables are stored in Pine Script[®]; the “series” form denotes variables whose values can change bar to bar. Consider, for example, the [timeframe.period](#) built-in variable which is of form “simple” and type “string”, so “simple string”. The “simple” form entails that the variable’s value is known on bar zero (the first bar where the script executes) and will not change during the script’s execution on all the chart’s bars. The variable’s value is the chart’s timeframe in string format, so “D” for a 1D chart, for example. Even though its value cannot change during the script, it would be syntactically correct in Pine Script[®] (though not very useful) to refer to its value 10 bars ago using `timeframe.period[10]`. This is possible because the successive values of [timeframe.period](#) for each bar are stored in a time series, even though all the values in that particular time series are similar. Note, however, that when the `[]` operator is used to access past values of a variable, it yields a result of “series” form, even though the variable without an offset is

of another form, such as “simple” in the case of [timeframe.period](#).

When you grasp how time series can be efficiently handled using Pine Script®’s syntax and its [execution model](#), you can define complex calculations using little code.

Script structure

- [Version](#)
- [Declaration statement](#)
- [Code](#)
- [Comments](#)
- [Line wrapping](#)
- [Compiler annotations](#)

A Pine script follows this general structure:

```
<version>  
<declaration_statement>  
<code>
```

Version

A [compiler annotation](#) in the following form tells the compiler which of the versions of Pine Script® the script is written in:

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

- The version number can be 1 to 5.
- The compiler annotation is not mandatory. When omitted, version 1 is assumed. It is strongly recommended to always use the latest version of the language.
- While it is syntactically correct to place the version compiler annotation anywhere in the script, it is much more useful to readers when it appears at the top of the script.

Notable changes to the current version of Pine Script® are documented in the [Release notes](#).

Declaration statement

All Pine scripts must contain one [declaration statement](#), which is a call to one of these functions:

- [indicator\(\)](#)
- [strategy\(\)](#)
- [library\(\)](#)

The declaration statement:

- Identifies the type of the script, which in turn dictates which content is allowed in it, and how it can be used and executed.
- Sets key properties of the script such as its name, where it will appear when it is added to a chart, the precision and format of the values it displays, and certain values that govern its runtime behavior, such as the maximum number of drawing objects it will display on the chart. With strategies, the properties include parameters that control backtesting, such as initial capital, commission, slippage, etc.

Each type of script has distinct requirements:

- Indicators must contain at least one function call which produces output on the chart (e.g., [plot\(\)](#), [plotshape\(\)](#), [barcolor\(\)](#), [line.new\(\)](#), etc.).
- Strategies must contain at least one `strategy.*()` call, e.g., [strategy.entry\(\)](#).
- Libraries must contain at least one exported [function](#) or [user-defined type](#).

Code

Lines in a script that are not [comments](#) or [compiler annotations](#) are *statements*, which implement the script's algorithm. A statement can be one of these:

- variable declaration
- variable reassignment
- function declaration
- built-in function call, [user-defined function call](#) or [a library function call](#)
- [if](#), [for](#), [while](#), [switch](#) or [type structure](#).

Statements can be arranged in multiple ways:

- Some statements can be expressed in one line, like most variable declarations, lines containing only a function call or single-line function declarations. Lines can also be [wrapped](#) (continued on multiple lines). Multiple one-line statements can be concatenated on a single line by using the comma as a separator.
- Others statements such as structures or multi-line function declarations always require multiple lines because they require a *local block*. A local block must be indented by a tab or four spaces. Each local block defines a distinct *local scope*.
- Statements in the *global scope* of the script (i.e., which are not part of local blocks) cannot begin with white space (a space or a tab). Their first character must also be the line's first character. Lines beginning in a line's first position become by definition part of the script's *global scope*.

A simple valid Pine Script[®] v5 indicator can be generated in the Pine Script[®] Editor by using the “Open” button and choosing “New blank indicator”:

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

This indicator includes three local blocks, one in the `f()` function declaration, and two in the variable declaration using an [if](#) structure:

```
//@version=5

indicator("", "", true) // Declaration statement (global scope)

barIsUp() => // Function declaration (global scope)
    close > open // Local block (local scope)

plotColor = if barIsUp() // Variable declaration (global scope)
    color.green // Local block (local scope)
else
    color.red // Local block (local scope)

bgcolor(color.new(plotColor, 70)) // Call to a built-in function (global scope)
```

You can bring up a simple Pine Script[®] v5 strategy by selecting “New blank strategy” instead:

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

```

strategy("My Strategy", overlay=true, margin_long=100, margin_short=100)

longCondition = ta.crossover(ta.sma(close, 14), ta.sma(close, 28))
if (longCondition)
    strategy.entry("My Long Entry Id", strategy.long)

shortCondition = ta.crossunder(ta.sma(close, 14), ta.sma(close, 28))
if (shortCondition)
    strategy.entry("My Short Entry Id", strategy.short)

```

Comments

Double slashes (//) define comments in Pine Script®. Comments can begin anywhere on the line. They can also follow Pine Script® code on the same line:

```

//@version=5
indicator("")
// This line is a comment
a = close // This is also a comment
plot(a)

```

The Pine Script® Editor has a keyboard shortcut to comment/uncomment lines: `ctrl + /`. You can use it on multiple lines by highlighting them first.

Line wrapping

Long lines can be split on multiple lines, or “wrapped”. Wrapped lines must be indented with any number of spaces, provided it’s not a multiple of four (those boundaries are used to indent local blocks):

```
a = open + high + low + close
```

may be wrapped as:

```

a = open +
    high +
    low +
    close

```

A long `plot()` call may be wrapped as:

```

plot(ta.correlation(src, ovr, length),
    color = color.new(color.purple, 40),
    style = plot.style_area,
    trackprice = true)

```

Statements inside user-defined function declarations can also be wrapped. However, since a local block must syntactically begin with an indentation (4 spaces or 1 tab), when splitting it onto the following line, the continuation of the statement must start with more than one indentation (not equal to a multiple of four spaces). For example:

```

updown(s) =>
    isEqual = s == s[1]
    isGrowing = s > s[1]
    ud = isEqual ?
        0 :
        isGrowing ?
            (nz(ud[1]) <= 0 ?
                1 :

```



```

        nz(ud[1])+1) :
(nz(ud[1]) >= 0 ?
-1 :
nz(ud[1])-1)

```

You can use comments in wrapped lines:

```

//@version=5
indicator("")
c = open > close ? color.red :
  high > high[1] ? color.lime : // A comment
  low < low[1] ? color.blue : color.black
bgcolor(c)

```

Compiler annotations

Compiler annotations are [comments](#) that issue special instructions for a script:

- `//@version=` specifies the PineScript™ version that the compiler will use. The number in this annotation should not be confused with the script's revision number, which updates on every saved change to the code.
- `//@description` sets a custom description for scripts that use the [library\(\)](#) declaration statement.
- `//@function`, `//@param` and `//@returns` add custom descriptions for a user-defined function, its parameters, and its result when placed above the function declaration.
- `//@type` and `//@field` add custom descriptions for a [user-defined type \(UDT\)](#) and its fields when placed above the type declaration.
- `//@variable` adds a custom description for a variable when placed above its declaration.
- `//@strategy_alert_message` provides a default message for strategy scripts to pre-fill the “Message” field in the alert creation dialogue.
- `// #region` and `// #endregion` create collapsible code regions in the Pine Editor. Clicking the dropdown arrow next to `// #region` collapses the lines of code between the two annotations.

This script draws a rectangle using three interactively selected points on the chart. It illustrates how compiler annotations can be used:



```

//@version=5
indicator("Triangle", "", true)

int   TIME_DEFAULT = 0
float PRICE_DEFAULT = 0.0

x1Input = input.time(TIME_DEFAULT, "Point 1", inline = "1", confirm = true)
y1Input = input.price(PRICE_DEFAULT, "", inline = "1", tooltip = "Pick
point 1", confirm = true)
x2Input = input.time(TIME_DEFAULT, "Point 2", inline = "2", confirm = true)
y2Input = input.price(PRICE_DEFAULT, "", inline = "2", tooltip = "Pick
point 2", confirm = true)
x3Input = input.time(TIME_DEFAULT, "Point 3", inline = "3", confirm = true)
y3Input = input.price(PRICE_DEFAULT, "", inline = "3", tooltip = "Pick
point 3", confirm = true)

// @type          Used to represent the coordinates and color to draw a
triangle.

```

```

// @field time1      Time of first point.
// @field time2      Time of second point.
// @field time3      Time of third point.
// @field price1     Price of first point.
// @field price2     Price of second point.
// @field price3     Price of third point.
// @field lineColor  Color to be used to draw the triangle lines.
type Triangle
    int    time1
    int    time2
    int    time3
    float  price1
    float  price2
    float  price3
    color  lineColor

//@function Draws a triangle using the coordinates of the `t` object.
//@param t  (Triangle) Object representing the triangle to be drawn.
//@returns  The ID of the last line drawn.
drawTriangle(Triangle t) =>
    line.new(t.time1, t.price1, t.time2, t.price2, xloc = xloc.bar_time, color =
t.lineColor)
    line.new(t.time2, t.price2, t.time3, t.price3, xloc = xloc.bar_time, color =
t.lineColor)
    line.new(t.time1, t.price1, t.time3, t.price3, xloc = xloc.bar_time, color =
t.lineColor)

// Draw the triangle only once on the last historical bar.
if barstate.islastconfirmedhistory
    //@variable Used to hold the Triangle object to be drawn.
    Triangle triangle = Triangle.new()

    triangle.time1 := x1Input
    triangle.time2 := x2Input
    triangle.time3 := x3Input
    triangle.price1 := y1Input
    triangle.price2 := y2Input
    triangle.price3 := y3Input
    triangle.lineColor := color.purple

    drawTriangle(triangle)

```

Identifiers

Identifiers are names used for user-defined variables and functions:

- They must begin with an uppercase (A-Z) or lowercase (a-z) letter, or an underscore (_).
- The next characters can be letters, underscores or digits (0-9).
- They are case-sensitive.

Here are some examples:

```

myVar
_myVar
my123Var
functionName
MAX_LEN
max_len
maxLen
3barsDown // NOT VALID!

```

The Pine Script® [Style Guide](#) recommends using uppercase SNAKE_CASE for constants, and camelCase for other identifiers:

```
GREEN_COLOR = #4CAF50
MAX_LOOKBACK = 100
int fastLength = 7
// Returns 1 if the argument is `true`, 0 if it is `false` or `na`.
zeroOne(boolValue) => boolValue ? 1 : 0
```

Operators

- [Introduction](#)
- [Arithmetic operators](#)
- [Comparison operators](#)
- [Logical operators](#)
- [`?:` ternary operator](#)
- [`\[\]` history-referencing operator](#)
- [Operator precedence](#)
- [`= ` assignment operator](#)
- [`:=` reassignment operator](#)

Introduction

Some operators are used to build *expressions* returning a result:

- Arithmetic operators
- Comparison operators
- Logical operators
- The [?:](#) ternary operator
- The [\[\]](#) history-referencing operator

Other operators are used to assign values to variables:

- `=` is used to assign a value to a variable, **but only when you declare the variable** (the first time you use it)
- `:=` is used to assign a value to a **previously declared variable**. The following operators can also be used in such a way: `+=`, `-=`, `*=`, `/=`, `%=`

As is explained in the [Type system](#) page, *forms* and *types* play a critical role in determining the type of results that expressions yield. This, in turn, has an impact on how and with what functions you will be allowed to use those results. Expressions always return a form of the strongest one used in the expression, e.g., if you multiply an “input int” with a “series int”, the expression will produce a “series int” result, which you will not be able to use as the argument to `length` in [ta.ema\(\)](#).

This script will produce a compilation error:

```
//@version=5
indicator("")
lenInput = input.int(14, "Length")
factor = year > 2020 ? 3 : 1
adjustedLength = lenInput * factor
ma = ta.ema(close, adjustedLength) // Compilation error!
plot(ma)
```

The compiler will complain: *Cannot call 'ta.ema' with argument 'length'='adjustedLength'. An*

argument of 'series int' type was used but a 'simple int' is expected;. This is happening because lenInput is an "input int" but factor is a "series int" (it can only be determined by looking at the value of [year](#) on each bar). The adjustedLength variable is thus assigned a "series int" value. Our problem is that the Reference Manual entry for [ta.ema\(\)](#) tells us that its length parameter requires values of "simple" form, which is a weaker form than "series", so a "series int" value is not allowed.

The solution to our conundrum requires:

- Using another moving average function that supports a "series int" length, such as [ta.sma\(\)](#), or
- Not using a calculation producing a "series int" value for our length.

Arithmetic operators

There are five arithmetic operators in Pine Script®:

+	Addition and string concatenation
-	Subtraction
*	Multiplication
/	Division
%	Modulo (remainder after division)

The arithmetic operators above are all binary (means they need two *operands* — or values — to work on, like in $1 + 2$). The + and - also serve as unary operators (means they work on one operand, like -1 or $+1$).

If both operands are numbers but at least one of these is of [float](#) type, the result will also be a [float](#). If both operands are of [int](#) type, the result will also be an [int](#). If at least one operand is [na](#), the result is also [na](#).

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

The % operator calculates the modulo by rounding down the quotient to the lowest possible value. Here is an easy example that helps illustrate how the modulo is calculated behind the scenes:

```
//@version=5
indicator("Modulo function")
modulo(series int a, series int b) =>
    a - b * math.floor(nz(a/b))
plot(modulo(-1, 100))
```

Comparison operators

There are six comparison operators in Pine Script®:

<	Less Than
<=	Less Than or

	Equal To
!=	Not Equal
==	Equal
>	Greater Than
>=	Greater Than or Equal To

Comparison operations are binary. If both operands have a numerical value, the result will be of type *bool*, i.e., true, false or [na](#).

Examples

```
1 > 2 // false
1 != 1 // false
close >= open // Depends on values of `close` and `open`
```

Logical operators

There are three logical operators in Pine Script®:

not	Negation
and	Logical Conjunction
or	Logical Disjunction

The operator `not` is unary. When applied to a `true`, operand the result will be `false`, and vice versa.

`and` operator truth table:

a	b	a and b
true	true	true
true	false	false
false	true	false
false	false	false

`or` operator truth table:

a	b	a or b
true	true	true
true	false	true
false	true	true
false	false	false

`?:` ternary operator

The `?:` ternary operator is used to create expressions of the form:

```
condition ? valueWhenConditionIsTrue : valueWhenConditionIsFalse
```

The ternary operator returns a result that depends on the value of `condition`. If it is `true`, then `valueWhenConditionIsTrue` is returned. If `condition` is `false` or `na`, then `valueWhenConditionIsFalse` is returned.

A combination of ternary expressions can be used to achieve the same effect as a [switch](#) structure, e.g.:

```
timeframe.isintraday ? color.red : timeframe.isdaily ? color.green :  
timeframe.ismonthly ? color.blue : na
```

The example is calculated from left to right:

- If [timeframe.isintraday](#) is `true`, then `color.red` is returned. If it is `false`, then [timeframe.isdaily](#) is evaluated.
- If [timeframe.isdaily](#) is `true`, then `color.green` is returned. If it is `false`, then [timeframe.ismonthly](#) is evaluated.
- If [timeframe.ismonthly](#) is `true`, then `color.blue` is returned, otherwise `na` is returned.

Note that the return values on each side of the `:` are expressions — not local blocks, so they will not affect the limit of 500 local blocks per scope.

[\[\]](#) history-referencing operator

It is possible to refer to past values of [time series](#) using the `[]` history-referencing operator. Past values are values a variable had on bars preceding the bar where the script is currently executing — the *current bar*. See the [Execution model](#) page for more information about the way scripts are executed on bars.

The `[]` operator is used after a variable, expression or function call. The value used inside the square brackets of the operator is the offset in the past we want to refer to. To refer to the value of the [volume](#) built-in variable two bars away from the current bar, one would use `volume[2]`.

Because series grow dynamically, as the script moves on successive bars, the offset used with the operator will refer to different bars. Let's see how the value returned by the same offset is dynamic, and why series are very different from arrays. In Pine Script[®], the [close](#) variable, or `close[0]` which is equivalent, holds the value of the current bar's "close". If your code is now executing on the **third** bar of the *dataset* (the set of all bars on your chart), `close` will contain the price at the close of that bar, `close[1]` will contain the price at the close of the preceding bar (the dataset's second bar), and `close[2]`, the first bar. `close[3]` will return `na` because no bar exists in that position, and thus its value is *not available*.

When the same code is executed on the next bar, the **fourth** in the dataset, `close` will now contain the closing price of that bar, and the same `close[1]` used in your code will now refer to the "close" of the third bar in the dataset. The close of the first bar in the dataset will now be `close[3]`, and this time `close[4]` will return `na`.

In the Pine Script[®] runtime environment, as your code is executed once for each historical bar in the dataset, starting from the left of the chart, Pine Script[®] is adding a new element in the series at index 0 and pushing the pre-existing elements in the series one index further away. Arrays, in comparison, can have constant or variable sizes, and their content or indexing structure is not modified by the runtime environment. Pine Script[®] series are thus very different from arrays and only share familiarity with them through their indexing syntax.

When the market for the chart's symbol is open and the script is executing on the chart's last bar, the *realtime bar*, [close](#) returns the value of the current price. It will only contain the actual closing

price of the realtime bar the last time the script is executed on that bar, when it closes.

Pine Script® has a variable that contains the number of the bar the script is executing on: [bar_index](#). On the first bar, [bar_index](#) is equal to 0 and it increases by 1 on each successive bar the script executes on. On the last bar, [bar_index](#) is equal to the number of bars in the dataset minus one.

There is another important consideration to keep in mind when using the `[]` operator in Pine Script®. We have seen cases when a history reference may return the [na](#) value. [na](#) represents a value which is not a number and using it in any expression will produce a result that is also [na](#) (similar to [NaN](#)). Such cases often happen during the script's calculations in the early bars of the dataset, but can also occur in later bars under certain conditions. If your code does not explicitly provide for handling these special cases, they can introduce invalid results in your script's calculations which can ripple through all the way to the realtime bar. The [na](#) and [nz](#) functions are designed to allow for handling such cases.

These are all valid uses of the `[]` operator:

```
high[10]
ta.sma(close, 10)[1]
ta.highest(high, 10)[20]
close > nz(close[1], open)
```

Note that the `[]` operator can only be used once on the same value. This is not allowed:

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

Operator precedence

The order of calculations is determined by the operators' precedence. Operators with greater precedence are calculated first. Below is a list of operators sorted by decreasing precedence:

Precedence	Operator
9	<code>[]</code>
8	unary +, unary -, not
7	<code>*</code> , <code>/</code> , <code>%</code>
6	<code>+</code> , <code>-</code>
5	<code>></code> , <code><</code> , <code>>=</code> , <code><=</code>
4	<code>==</code> , <code>!=</code>
3	and
2	or
1	<code>? :</code>

If in one expression there are several operators with the same precedence, then they are calculated left to right.

If the expression must be calculated in a different order than precedence would dictate, then parts of the expression can be grouped together with parentheses.

`=` assignment operator

The = operator is used to assign a variable when it is initialized — or declared —, i.e., the first time you use it. It says *this is a new variable that I will be using, and I want it to start on each bar with this value.*

These are all valid variable declarations:

```
i = 1
MS_IN_ONE_MINUTE = 1000 * 60
showPlotInput = input.bool(true, "Show plots")
pHi = pivohigh(5, 5)
plotColor = color.green
```

See the [Variable declarations](#) page for more information on how to declare variables.

`:=` reassignment operator

The := is used to *reassign* a value to an existing variable. It says *use this variable that was declared earlier in my script, and give it a new value.*

Variables which have been first declared, then reassigned using :=, are called *mutable* variables. All the following examples are valid variable reassignments. You will find more information on how [var](#) works in the section on the [`var` declaration mode](#):

```
//@version=5
indicator("", "", true)
// Declare `pHi` and initialize it on the first bar only.
var float pHi = na
// Reassign a value to `pHi`
pHi := nz(ta.pivohigh(5, 5), pHi)
plot(pHi)
```

Note that:

- We declare pHi with this code: `var float pHi = na`. The [var](#) keyword tells Pine Script[®] that we only want that variable initialized with [na](#) on the dataset's first bar. The `float` keyword tells the compiler we are declaring a variable of type "float". This is necessary because, contrary to most cases, the compiler cannot automatically determine the type of the value on the right side of the = sign.
- While the variable declaration will only be executed on the first bar because it uses [var](#), the `pHi := nz(ta.pivohigh(5, 5), pHi)` line will be executed on all the chart's bars. On each bar, it evaluates if the [pivohigh\(\)](#) call returns [na](#) because that is what the function does when it hasn't found a new pivot. The [nz\(\)](#) function is the one doing the "checking for [na](#)" part. When its first argument (`ta.pivohigh(5, 5)`) is [na](#), it returns the second argument (`pHi`) instead of the first. When [pivohigh\(\)](#) returns the price point of a newly found pivot, that value is assigned to `pHi`. When it returns [na](#) because no new pivot was found, we assign the previous value of `pHi` to itself, in effect preserving its previous value.

The output of our script looks like this:



Note that:

- The line preserves its previous value until a new pivot is found.
- Pivots are detected five bars after the pivot actually occurs because our `ta.pivohigh(5, 5)` call says that we require five lower highs on both sides of a high point for it to be detected as a pivot.

See the [Variable reassignment](#) section for more information on how to reassign values to variables.

Variable declarations

- [Introduction](#)
 - [Initialization with `na`](#)
 - [Tuple declarations](#)
- [Variable reassignment](#)
- [Declaration modes](#)
 - [On each bar](#)
 - [`var`](#)
 - [`varip`](#)

Introduction

Variables are [identifiers](#) that hold values. They must be *declared* in your code before you use them. The syntax of variable declarations is:

[<declaration_mode>] [<type>] <identifier> = <expression> | <structure>

or

<tuple_declaration> = <function_call> | <structure>

where:

- | means “or”, and parts enclosed in square brackets ([]) can appear zero or one time.
- <declaration_mode> is the variable’s [declaration mode](#). It can be [var](#) or [varip](#), or nothing.
- <type> is optional, as in almost all Pine Script[®] variable declarations (see [types](#)).
- <identifier> is the variable’s [name](#).
- <expression> can be a literal, a variable, an expression or a function call.
- <structure> can be an [if](#), [for](#), [while](#) or [switch structure](#).
- <tuple_declaration> is a comma-separated list of variable names enclosed in square brackets ([]), e.g., [ma, upperBand, lowerBand].

These are all valid variable declarations. The last one requires four lines:

```
BULL_COLOR = color.lime
i = 1
len = input(20, "Length")
float f = 10.5
closeRoundedToTick = math.round_to_mintick(close)
st = ta.supertrend(4, 14)
var barRange = float(na)
var firstBarOpen = open
varip float lastClose = na
[macdLine, signalLine, histLine] = ta.macd(close, 12, 26, 9)
plotColor = if close > open
    color.green
else
    color.red
```

Note

The above statements all contain the = assignment operator because they are **variable declarations**. When you see similar lines using the `:=` reassignment operator, the code is **reassigning** a value to a variable that was **already declared**. Those are **variable reassignments**. Be sure you understand the distinction as this is a common stumbling block for newcomers to Pine Script[®]. See the next [Variable reassignment](#) section for details.

The formal syntax of a variable declaration is:

```
<variable_declaration>
  [<declaration_mode>] [<type>] <identifier> = <expression> | <structure>
  |
  <tuple_declaration> = <function_call> | <structure>

<declaration_mode>
  var | varip

<type>
  int | float | bool | color | string | line | linefill | label | box | table
  | array<type> | matrix<type> | UDF
```

[**Initialization with `na`**](#)

In most cases, an explicit type declaration is redundant because type is automatically inferred from

the value on the right of the = at compile time, so the decision to use them is often a matter of preference. For example:

```
baseLine0 = na           // compile time error!  
float baseLine1 = na    // OK  
baseLine2 = float(na)   // OK
```

In the first line of the example, the compiler cannot determine the type of the `baseLine0` variable because `na` is a generic value of no particular type. The declaration of the `baseLine1` variable is correct because its `float` type is declared explicitly. The declaration of the `baseLine2` variable is also correct because its type can be derived from the expression `float(na)`, which is an explicit cast of the `na` value to the `float` type. The declarations of `baseLine1` and `baseLine2` are equivalent.

Tuple declarations

Function calls or structures are allowed to return multiple values. When we call them and want to store the values they return, a *tuple declaration* must be used, which is a comma-separated set of one or more values enclosed in brackets. This allows us to declare multiple variables simultaneously. As an example, the `ta.bb()` built-in function for Bollinger bands returns three values:

```
[bbMiddle, bbUpper, bbLower] = ta.bb(close, 5, 4)
```

Variable reassignment

A variable reassignment is done using the `:=` reassignment operator. It can only be done after a variable has been first declared and given an initial value. Reassigning a new value to a variable is often necessary in calculations, and it is always necessary when a variable from the global scope must be assigned a new value from within a structure's local block, e.g.:

```
//@version=5  
indicator("", "", true)  
sensitivityInput = input.int(2, "Sensitivity", minval = 1, tooltip = "Higher  
values make color changes less sensitive.")  
ma = ta.sma(close, 20)  
maUp = ta.rising(ma, sensitivityInput)  
maDn = ta.falling(ma, sensitivityInput)  
  
// On first bar only, initialize color to gray  
var maColor = color.gray  
if maUp  
    // MA has risen for two bars in a row; make it lime.  
    maColor := color.lime  
else if maDn  
    // MA has fallen for two bars in a row; make it fuchsia.  
    maColor := color.fuchsia  
  
plot(ma, "MA", maColor, 2)
```

Note that:

- We initialize `maColor` on the first bar only, so it preserves its value across bars.
- On every bar, the `if` statement checks if the MA has been rising or falling for the user-specified number of bars (the default is 2). When that happens, the value of `maColor` must be reassigned a new value from within the `if` local blocks. To do this, we use the `:=` reassignment operator.
- If we did not use the `:=` reassignment operator, the effect would be to initialize a new

maColor local variable which would have the same name as that of the global scope, but actually be a very confusing independent entity that would persist only for the length of the local block, and then disappear without a trace.

All user-defined variables in Pine Script® are *mutable*, which means their value can be changed using the `:=` reassignment operator. Assigning a new value to a variable may change its *form* (see the page on Pine Script®'s [type system](#) for more information). A variable can be assigned a new value as many times as needed during the script's execution on one bar, so a script can contain any number of reassignments of one variable. A variable's [declaration mode](#) determines how new values assigned to a variable will be saved.

Declaration modes

Understanding the impact that declaration modes have on the behavior of variables requires prior knowledge of Pine Script®'s [execution model](#).

When you declare a variable, if a declaration mode is specified, it must come first. Three modes can be used:

- “On each bar”, when none is specified
- [var](#)
- [varip](#)

On each bar

When no explicit declaration mode is specified, i.e. no [var](#) or [varip](#) keyword is used, the variable is declared and initialized on each bar, e.g., the following declarations from our first set of examples in this page's introduction:

```
BULL_COLOR = color.lime
i = 1
len = input(20, "Length")
float f = 10.5
closeRoundedToTick = math.round_to_mintick(close)
st = ta.supertrend(4, 14)
[macdLine, signalLine, histLine] = ta.macd(close, 12, 26, 9)
plotColor = if close > open
    color.green
else
    color.red
```

`var`

When the [var](#) keyword is used, the variable is only initialized once, on the first bar if the declaration is in the global scope, or the first time the local block is executed if the declaration is inside a local block. After that, it will preserve its last value on successive bars, until we reassign a new value to it. This behavior is very useful in many cases where a variable's value must persist through the iterations of a script across successive bars. For example, suppose we'd like to count the number of green bars on the chart:

```
//@version=5
indicator("Green Bars Count")
var count = 0
isGreen = close >= open
if isGreen
    count := count + 1
plot(count)
```



Without the `var` modifier, variable `count` would be reset to zero (thus losing its value) every time a new bar update triggered a script recalculation.

Declaring variables on the first bar only is often useful to manage drawings more efficiently. Suppose we want to extend the last bar's [close](#) line to the right of the right chart. We could write:

```
//@version=5
indicator("Inefficient version", "", true)
closeLine = line.new(bar_index - 1, close, bar_index, close, extend =
extend.right, width = 3)
line.delete(closeLine[1])
```

but this is inefficient because we are creating and deleting the line on each historical bar and on each update in the realtime bar. It is more efficient to use:

```
//@version=5
indicator("Efficient version", "", true)
var closeLine = line.new(bar_index - 1, close, bar_index, close, extend =
extend.right, width = 3)
if barstate.islast
    line.set_xy1(closeLine, bar_index - 1, close)
    line.set_xy2(closeLine, bar_index, close)
```

Note that:

- We initialize `closeLine` on the first bar only, using the [var](#) declaration mode
- We restrict the execution of the rest of our code to the chart's last bar by enclosing our code that updates the line in an [if barstate.islast](#) structure.

There is a very slight penalty performance for using the [var](#) declaration mode. For that reason, when declaring constants, it is preferable not to use [var](#) if performance is a concern, unless the initialization involves calculations that take longer than the maintenance penalty, e.g., functions with complex code or string manipulations.

[`varip`](#)

Understanding the behavior of variables using the [varip](#) declaration mode requires prior knowledge of Pine Script®'s [execution model](#) and [bar states](#).

The [varip](#) keyword can be used to declare variables that escape the *rollback process*, which is explained in the page on Pine Script®'s [execution model](#).

Whereas scripts only execute once at the close of historical bars, when a script is running in realtime, it executes every time the chart's feed detects a price or volume update. At every realtime update, Pine Script®'s runtime normally resets the values of a script's variables to their last committed value, i.e., the value they held when the previous bar closed. This is generally handy, as each realtime script execution starts from a known state, which simplifies script logic.

Sometimes, however, script logic requires code to be able to save variable values **between different executions** in the realtime bar. Declaring variables with [varip](#) makes that possible. The "ip" in [varip](#) stands for *intraday persist*.

Let's look at the following code, which does not use [varip](#):

```
//@version=5
indicator("")
int updateNo = na
```

```

if barstate.isnew
    updateNo := 1
else
    updateNo := updateNo + 1

plot(updateNo, style = plot.style_circles)

```

On historical bars, [barstate.isnew](#) is always true, so the plot shows a value of “1” because the `else` part of the `if` structure is never executed. On realtime bars, [barstate.isnew](#) is only `true` when the script first executes on the bar’s “open”. The plot will then briefly display “1” until subsequent executions occur. On the next executions during the realtime bar, the second branch of the `if` statement is executed because [barstate.isnew](#) is no longer true. Since `updateNo` is initialized to `na` at each execution, the `updateNo + 1` expression yields `na`, so nothing is plotted on further realtime executions of the script.

If we now use [varip](#) to declare the `updateNo` variable, the script behaves very differently:

```

//@version=5
indicator("")
varip int updateNo = na
if barstate.isnew
    updateNo := 1
else
    updateNo := updateNo + 1

plot(updateNo, style = plot.style_circles)

```

The difference now is that `updateNo` tracks the number of realtime updates that occur on each realtime bar. This can happen because the [varip](#) declaration allows the value of `updateNo` to be preserved between realtime updates; it is no longer rolled back at each realtime execution of the script. The test on [barstate.isnew](#) allows us to reset the update count when a new realtime bar comes in.

Because [varip](#) only affects the behavior of your code in the realtime bar, it follows that backtest results on strategies designed using logic based on [varip](#) variables will not be able to reproduce that behavior on historical bars, which will invalidate test results on them. This also entails that plots on historical bars will not be able to reproduce the script’s behavior in realtime.

Conditional structures

- [Introduction](#)
- [`if` structure](#)
 - [`if` used for its side effects](#)
 - [`if` used to return a value](#)
- [`switch` structure](#)
 - [`switch` with an expression](#)
 - [`switch` without an expression](#)
- [Matching local block type requirement](#)

[Introduction](#)

The conditional structures in Pine Script[®] are [if](#) and [switch](#). They can be used:

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

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

Some Pine Script[®] built-in functions cannot be called from within the local blocks of conditional structures. They are: [alertcondition\(\)](#), [barcolor\(\)](#), [fill\(\)](#), [hline\(\)](#), [indicator\(\)](#), [library\(\)](#), [plot\(\)](#), [plotbar\(\)](#), [plotcandle\(\)](#), [plotchar\(\)](#), [plotshape\(\)](#), [strategy\(\)](#). This does not entail their functionality cannot be controlled by conditions evaluated by your script — only that it cannot be done by including them in conditional structures. Note that while `input*.()` function calls are allowed in local blocks, their functionality is the same as if they were in the script's global scope.

The local blocks in conditional structures must be indented by four spaces or a tab.

[if](#) structure

[if](#) used for its side effects

An [if](#) structure used for its side effects has the following syntax:

```
if <expression>
    <local_block>
{else if <expression>
    <local_block>}
[else
    <local_block>]
```

where:

- Parts enclosed in square brackets (`[]`) can appear zero or one time, and those enclosed in curly braces (`{ }`) can appear zero or more times.
- `<expression>` must be of “bool” type or be auto-castable to that type, which is only possible for “int” or “float” values (see the [Type system](#) page).
- `<local_block>` consists of zero or more statements followed by a return value, which can be a tuple of values. It must be indented by four spaces or a tab.
- There can be zero or more `else if` clauses.
- There can be zero or one `else` clause.

When the `<expression>` following the [if](#) evaluates to [true](#), the first local block is executed, the [if](#) structure's execution ends, and the value(s) evaluated at the end of the local block are returned.

When the `<expression>` following the [if](#) evaluates to [false](#), the successive `else if` clauses are evaluated, if there are any. When the `<expression>` of one evaluates to [true](#), its local block is executed, the [if](#) structure's execution ends, and the value(s) evaluated at the end of the local block are returned.

When no `<expression>` has evaluated to [true](#) and an `else` clause exists, its local block is executed, the [if](#) structure's execution ends, and the value(s) evaluated at the end of the local block are returned.

When no `<expression>` has evaluated to [true](#) and no `else` clause exists, [na](#) is returned.

Using [if](#) structures for their side effects can be useful to manage the order flow in strategies, for example. While the same functionality can often be achieved using the `when` parameter in

strategy.*() calls, code using [if](#) structures is easier to read:

```
if (ta.crossover(source, lower))
    strategy.entry("BBandLE", strategy.long, stop=lower,
                 oca_name="BollingerBands",
                 oca_type=strategy.oca.cancel, comment="BBandLE")
else
    strategy.cancel(id="BBandLE")
```

Restricting the execution of your code to specific bars can be done using [if](#) structures, as we do here to restrict updates to our label to the chart's last bar:

```
//@version=5
indicator("", "", true)
var ourLabel = label.new(bar_index, na, na, color = color(na), textcolor =
color.orange)
if barstate.islast
    label.set_xy(ourLabel, bar_index + 2, h12[1])
    label.set_text(ourLabel, str.tostring(bar_index + 1, "# bars in chart"))
```

Note that:

- We initialize the `ourLabel` variable on the script's first bar only, as we use the [var](#) declaration mode. The value used to initialize the variable is provided by the [label.new\(\)](#) function call, which returns a label ID pointing to the label it creates. We use that call to set the label's properties because once set, they will persist until we change them.
- What happens next is that on each successive bar the Pine Script® runtime will skip the initialization of `ourLabel`, and the [if](#) structure's condition ([barstate.islast](#)) is evaluated. It returns `false` on all bars until the last one, so the script does nothing on most historical bars after bar zero.
- On the last bar, [barstate.islast](#) becomes true and the structure's local block executes, modifying on each chart update the properties of our label, which displays the number of bars in the dataset.
- We want to display the label's text without a background, so we make the label's background `na` in the [label.new\(\)](#) function call, and we use `h12[1]` for the label's y position because we don't want it to move all the time. By using the average of the **previous** bar's [high](#) and [low](#) values, the label doesn't move until the moment when the next realtime bar opens.
- We use `bar_index + 2` in our [label.set_xy\(\)](#) call to offset the label to the right by two bars.

[if](#) used to return a value

An [if](#) structure used to return one or more values has the following syntax:

```
[<declaration_mode>] [<type>] <identifier> = if <expression>
    <local_block>
{else if <expression>
    <local_block>}
[else
    <local_block>]
```

where:

- Parts enclosed in square brackets (`[]`) can appear zero or one time, and those enclosed in curly braces (`{ }`) can appear zero or more times.
- `<declaration_mode>` is the variable's [declaration mode](#)

- `<type>` is optional, as in almost all Pine Script[®] variable declarations (see [types](#))
- `<identifier>` is the variable's [name](#)
- `<expression>` can be a literal, a variable, an expression or a function call.
- `<local_block>` consists of zero or more statements followed by a return value, which can be a tuple of values. It must be indented by four spaces or a tab.
- The value assigned to the variable is the return value of the `<local_block>`, or [na](#) if no local block is executed.

This is an example:

```
//@version=5
indicator("", "", true)
string barState = if barstate.islastconfirmedhistory
    "islastconfirmedhistory"
else if barstate.isnew
    "isnew"
else if barstate.isrealtime
    "isrealtime"
else
    "other"

f_print(_text) =>
    var table_t = table.new(position.middle_right, 1, 1)
    table.cell(_t, 0, 0, _text, bgcolor = color.yellow)
f_print(barState)
```

It is possible to omit the *else* block. In this case, if the condition is false, an *empty* value (na, false, or "") will be assigned to the `var_declarationX` variable.

This is an example showing how [na](#) is returned when no local block is executed. If `close > open` is false in here, [na](#) is returned:

```
x = if close > open
    close
```

[`switch` structure](#)

The [switch](#) structure exists in two forms. One switches on the different values of a key expression:

```
[[<declaration_mode>] [<type>] <identifier> = ]switch <expression>
    {<expression> => <local_block>}
=> <local_block>
```

The other form does not use an expression as a key; it switches on the evaluation of different expressions:

```
[[<declaration_mode>] [<type>] <identifier> = ]switch
    {<expression> => <local_block>}
=> <local_block>
```

where:

- Parts enclosed in square brackets (`[]`) can appear zero or one time, and those enclosed in curly braces (`{ }`) can appear zero or more times.
- `<declaration_mode>` is the variable's [declaration mode](#)
- `<type>` is optional, as in almost all Pine Script[®] variable declarations (see [types](#))
- `<identifier>` is the variable's [name](#)
- `<expression>` can be a literal, a variable, an expression or a function call.

- `<local_block>` consists of zero or more statements followed by a return value, which can be a tuple of values. It must be indented by four spaces or a tab.
- The value assigned to the variable is the return value of the `<local_block>`, or `na` if no local block is executed.
- The `=>` `<local_block>` at the end allows you to specify a return value which acts as a default to be used when no other case in the structure is executed.

Only one local block of a [switch](#) structure is executed. It is thus a *structured switch* that doesn't *fall through* cases. Consequently, `break` statements are unnecessary.

Both forms are allowed as the value used to initialize a variable.

As with the [if](#) structure, if no local block is executed, `na` is returned.

[switch` with an expression](#)

Let's look at an example of a [switch](#) using an expression:

```
//@version=5
indicator("Switch using an expression", "", true)

string maType = input.string("EMA", "MA type", options = ["EMA", "SMA", "RMA",
"WMA"])
int maLength = input.int(10, "MA length", minval = 2)

float ma = switch maType
    "EMA" => ta.ema(close, maLength)
    "SMA" => ta.sma(close, maLength)
    "RMA" => ta.rma(close, maLength)
    "WMA" => ta.wma(close, maLength)
    =>
        runtime.error("No matching MA type found.")
        float(na)

plot(ma)
```

Note that:

- The expression we are switching on is the variable `maType`, which is of “input int” type (see here for an explanation of what the “[input](#)” form is). Since it cannot change during the execution of the script, this guarantees that whichever MA type the user selects will be executing on each bar, which is a requirement for functions like [ta.ema\(\)](#) which require a “simple int” argument for their `length` parameter.
- If no matching value is found for `maType`, the [switch](#) executes the last local block introduced by `=>`, which acts as a catch-all. We generate a runtime error in that block. We also end it with `float(na)` so the local block returns a value whose type is compatible with that of the other local blocks in the structure, to avoid a compilation error.

[switch` without an expression](#)

This is an example of a [switch](#) structure which does not use an expression:

```
//@version=5
strategy("Switch without an expression", "", true)

bool longCondition = ta.crossover( ta.sma(close, 14), ta.sma(close, 28))
bool shortCondition = ta.crossunder(ta.sma(close, 14), ta.sma(close, 28))

switch
```

```
longCondition => strategy.entry("Long ID", strategy.long)
shortCondition => strategy.entry("Short ID", strategy.short)
```

Note that:

- We are using the [switch](#) to select the appropriate strategy order to emit, depending on whether the `longCondition` or `shortCondition` “bool” variables are true.
- The building conditions of `longCondition` and `shortCondition` are exclusive. While they can both be false simultaneously, they cannot be true at the same time. The fact that only **one** local block of the [switch](#) structure is ever executed is thus not an issue for us.
- We evaluate the calls to [ta.crossover\(\)](#) and [ta.crossunder\(\)](#) **prior** to entry in the [switch](#) structure. Not doing so, as in the following example, would prevent the functions to be executed on each bar, which would result in a compiler warning and erratic behavior:

```
//@version=5
strategy("Switch without an expression", "", true)

switch
  // Compiler warning! Will not calculate correctly!
  ta.crossover( ta.sma(close, 14), ta.sma(close, 28)) =>
strategy.entry("Long ID", strategy.long)
  ta.crossunder(ta.sma(close, 14), ta.sma(close, 28)) =>
strategy.entry("Short ID", strategy.short)
```

Matching local block type requirement

When multiple local blocks are used in structures, the type of the return value of all its local blocks must match. This applies only if the structure is used to assign a value to a variable in a declaration, because a variable can only have one type, and if the statement returns two incompatible types in its branches, the variable type cannot be properly determined. If the structure is not assigned anywhere, its branches can return different values.

This code compiles fine because [close](#) and [open](#) are both of the `float` type:

```
x = if close > open
  close
else
  open
```

This code does not compile because the first local block returns a `float` value, while the second one returns a `string`, and the result of the `if`-statement is assigned to the `x` variable:

```
// Compilation error!
x = if close > open
  close
else
  "open"
```

Loops

- [Introduction](#)
 - [When loops are not needed](#)
 - [When loops are necessary](#)
- [`for`](#)

- [`while`](#)

Introduction

When loops are not needed

Pine Script[®]'s runtime and its built-in functions make loops unnecessary in many situations.

Budding Pine Script[®] programmers not yet familiar with the Pine Script[®] runtime and built-ins who want to calculate the average of the last 10 [close](#) values will often write code such as:

```
//@version=5
indicator("Inefficient MA", "", true)
MA_LENGTH = 10
sumOfCloses = 0.0
for offset = 0 to MA_LENGTH - 1
    sumOfCloses := sumOfCloses + close[offset]
inefficientMA = sumOfCloses / MA_LENGTH
plot(inefficientMA)
```

A [for](#) loop is unnecessary and inefficient to accomplish tasks like this in Pine. This is how it should be done. This code is shorter *and* will run much faster because it does not use a loop and uses the [ta.sma\(\)](#) built-in function to accomplish the task:

```
//@version=5
indicator("Efficient MA", "", true)
thePineMA = ta.sma(close, 10)
plot(thePineMA)
```

Counting the occurrences of a condition in the last bars is also a task which beginning Pine Script[®] programmers often think must be done with a loop. To count the number of up bars in the last 10 bars, they will use:

```
//@version=5
indicator("Inefficient sum")
MA_LENGTH = 10
upBars = 0.0
for offset = 0 to MA_LENGTH - 1
    if close[offset] > open[offset]
        upBars := upBars + 1
plot(upBars)
```

The efficient way to write this in Pine (for the programmer because it saves time, to achieve the fastest-loading charts, and to share our common resources most equitably), is to use the [math.sum\(\)](#) built-in function to accomplish the task:

```
//@version=5
indicator("Efficient sum")
upBars = math.sum(close > open ? 1 : 0, 10)
plot(upBars)
```

What's happening in there is:

- We use the [?:](#) ternary operator to build an expression that yields 1 on up bars and 0 on other bars.
- We use the [math.sum\(\)](#) built-in function to keep a running sum of that value for the last 10 bars.

When loops are necessary

Loops exist for good reason because even in Pine Script[®], they are necessary in some cases. These cases typically include:

- The manipulation of collections ([arrays](#), [matrices](#), and [maps](#)).
- Looking back in history to analyze bars using a reference value that can only be known on the current bar, e.g., to find how many past highs are higher than the [high](#) of the current bar. Since the current bar's [high](#) is only known on the bar the script is running on, a loop is necessary to go back in time and analyze past bars.
- Performing calculations on past bars that cannot be accomplished using built-in functions.

`for`

The [for](#) structure allows the repetitive execution of statements using a counter. Its syntax is:

```
[[<declaration_mode>] [<type>] <identifier> = ]for <identifier> = <expression>
to <expression>[ by <expression>]
    <local_block_loop>
```

where:

- Parts enclosed in square brackets ([]) can appear zero or one time, and those enclosed in curly braces ({ }) can appear zero or more times.
- <declaration_mode> is the variable's [declaration mode](#)
- <type> is optional, as in almost all Pine Script[®] variable declarations (see [types](#))
- <identifier> is a variable's [name](#)
- <expression> can be a literal, a variable, an expression or a function call.
- <local_block_loop> consists of zero or more statements followed by a return value, which can be a tuple of values. It must be indented by four spaces or a tab. It can contain the `break` statement to exit the loop, or the `continue` statement to exit the current iteration and continue on with the next.
- The value assigned to the variable is the return value of the <local_block_loop>, i.e., the last value calculated on the loop's last iteration, or [na](#) if the loop is not executed.
- The identifier in `for <identifier>` is the loop's counter *initial value*.
- The expression in `= <expression>` is the *start value* of the counter.
- The expression in `to <expression>` is the *end value* of the counter. **It is only evaluated upon entry in the loop.**
- The expression in `by <expression>` is optional. It is the step by which the loop counter is increased or decreased on each iteration of the loop. Its default value is 1 when `start value < end value`. It is -1 when `start value > end value`. The step (+1 or -1) used as the default is determined by the start and end values.

This example uses a [for](#) statement to look back a user-defined amount of bars to determine how many bars have a [high](#) that is higher or lower than the [high](#) of the last bar on the chart. A [for](#) loop is necessary here, since the script only has access to the reference value on the chart's last bar. Pine Script[®]'s runtime cannot, here, be used to calculate on the fly, as the script is executing bar to bar:

```
//@version=5
indicator("`for` loop")
lookbackInput = input.int(50, "Lookback in bars", minval = 1, maxval = 4999)
higherBars = 0
lowerBars = 0
if barstate.islast
    var label lbl = label.new(na, na, "", style = label.style_label_left)
```

```

    for i = 1 to lookbackInput
        if high[i] > high
            higherBars += 1
        else if high[i] < high
            lowerBars += 1
        label.set_xy(lbl, bar_index, high)
        label.set_text(lbl, str.tostring(higherBars, "# higher bars\n") +
str.tostring(lowerBars, "# lower bars"))

```

This example uses a loop in its `checkLinesForBreaches()` function to go through an array of pivot lines and delete them when price crosses them. A loop is necessary here because all the lines in each of the `hiPivotLines` and `loPivotLines` arrays must be checked on each bar, and there is no built-in that can do this for us:

```

//@version=5
MAX_LINES_COUNT = 100
indicator("Pivot line breaches", "", true, max_lines_count = MAX_LINES_COUNT)

color hiPivotColorInput = input(color.new(color.lime, 0), "High pivots")
color loPivotColorInput = input(color.new(color.fuchsia, 0), "Low pivots")
int pivotLegsInput = input.int(5, "Pivot legs")
int qtyOfPivotsInput = input.int(50, "Quantity of last pivots to remember",
minval = 0, maxval = MAX_LINES_COUNT / 2)
int maxLineLengthInput = input.int(400, "Maximum line length in bars", minval
= 2)

// —— Queues a new element in an array and de-queues its first element.
qDq(array, qtyOfElements, arrayElement) =>
    array.push(array, arrayElement)
    if array.size(array) > qtyOfElements
        // Only dequeue if array has reached capacity.
        array.shift(array)

// —— Loop through an array of lines, extending those that price has not
crossed and deleting those crossed.
checkLinesForBreaches(arrayOfLines) =>
    int qtyOfLines = array.size(arrayOfLines)
    // Don't loop in case there are no lines to check because "to" value will be
`na` then`.
    for lineNo = 0 to (qtyOfLines > 0 ? qtyOfLines - 1 : na)
        // Need to check that array size still warrants a loop because we may
have deleted array elements in the loop.
        if lineNo < array.size(arrayOfLines)
            line currentLine = array.get(arrayOfLines, lineNo)
            float lineLevel = line.get_price(currentLine, bar_index)
            bool lineWasCrossed = math.sign(close[1] - lineLevel) !=
math.sign(close - lineLevel)
            bool lineIsTooLong = bar_index - line.get_x1(currentLine) >
maxLineLengthInput
            if lineWasCrossed or lineIsTooLong
                // Line stays on the chart but will no longer be extend on
further bars.
                array.remove(arrayOfLines, lineNo)
                // Force type of both local blocks to same type.
                int(na)
            else
                line.set_x2(currentLine, bar_index)
                int(na)

// Arrays of lines containing non-crossed pivot lines.
var line[] hiPivotLines = array.new_line(qtyOfPivotsInput)
var line[] loPivotLines = array.new_line(qtyOfPivotsInput)

```

```

// Detect new pivots.
float hiPivot = ta.pivohigh(pivotLegsInput, pivotLegsInput)
float loPivot = ta.pivotlow(pivotLegsInput, pivotLegsInput)

// Create new lines on new pivots.
if not na(hiPivot)
    line newLine = line.new(bar_index[pivotLegsInput], hiPivot, bar_index,
hiPivot, color = hiPivotColorInput)
    line.delete(qDq(hiPivotLines, qtyOfPivotsInput, newLine))
else if not na(loPivot)
    line newLine = line.new(bar_index[pivotLegsInput], loPivot, bar_index,
loPivot, color = loPivotColorInput)
    line.delete(qDq(loPivotLines, qtyOfPivotsInput, newLine))

// Extend lines if they haven't been crossed by price.
checkLinesForBreaches(hiPivotLines)
checkLinesForBreaches(loPivotLines)

```

`while`

The [while](#) structure allows the repetitive execution of statements until a condition is false. Its syntax is:

```

[[<declaration_mode>] [<type>] <identifier> = ]while <expression>
    <local_block_loop>

```

where:

- Parts enclosed in square brackets ([]) can appear zero or one time.
- <declaration_mode> is the variable's [declaration mode](#)
- <type> is optional, as in almost all Pine Script[®] variable declarations (see [types](#))
- <identifier> is a variable's [name](#)
- <expression> can be a literal, a variable, an expression or a function call. It is evaluated at each iteration of the loop. When it evaluates to `true`, the loop executes. When it evaluates to `false` the loop stops. Note that evaluation of the expression is done before each iteration only. Changes to the expression's value inside the loop will only have an impact on the next iteration.
- <local_block_loop> consists of zero or more statements followed by a return value, which can be a tuple of values. It must be indented by four spaces or a tab. It can contain the `break` statement to exit the loop, or the `continue` statement to exit the current iteration and continue on with the next.
- The value assigned to the <identifier> variable is the return value of the <local_block_loop>, i.e., the last value calculated on the loop's last iteration, or [na](#) if the loop is not executed.

This is the first code example of the [for](#) section written using a [while](#) structure instead of a [for](#) one:

```

//@version=5
indicator("`for` loop")
lookbackInput = input.int(50, "Lookback in bars", minval = 1, maxval = 4999)
higherBars = 0
lowerBars = 0
if barstate.islast
    var label lbl = label.new(na, na, "", style = label.style_label_left)
    // Initialize the loop counter to its start value.
    i = 1
    // Loop until the `i` counter's value is <= the `lookbackInput` value.
    while i <= lookbackInput

```

```

    if high[i] > high
        higherBars += 1
    else if high[i] < high
        lowerBars += 1
    // Counter must be managed "manually".
    i += 1
label.set_xy(lbl, bar_index, high)
label.set_text(lbl, str.toString(higherBars, "# higher bars\n") +
str.toString(lowerBars, "# lower bars"))

```

Note that:

- The `i` counter must be incremented by one explicitly inside the [while](#)'s local block.
- We use the `+=` operator to add one to the counter. `lowerBars += 1` is equivalent to `lowerBars := lowerBars + 1`.

Let's calculate the factorial function using a [while](#) structure:

```

//@version=5
indicator("")
int n = input.int(10, "Factorial of", minval=0)

factorial(int val = na) =>
    int counter = val
    int fact = 1
    result = while counter > 0
        fact := fact * counter
        counter := counter - 1
    fact

// Only evaluate the function on the first bar.
var answer = factorial(n)
plot(answer)

```

Note that:

- We use [input.int\(\)](#) for our input because we need to specify a `minval` value to protect our code. While [input\(\)](#) also supports the input of "int" type values, it does not support the `minval` parameter.
- We have packaged our script's functionality in a `factorial()` function which accepts as an argument the value whose factorial it must calculate. We have used `int val = na` to declare our function's parameter, which says that if the function is called without an argument, as in `factorial()`, then the `val` parameter will initialize to [na](#), which will prevent the execution of the [while](#) loop because its `counter > 0` expression will return [na](#). The [while](#) structure will thus initialize the `result` variable to [na](#). In turn, because the initialization of `result` is the return value of the our function's local block, the function will return [na](#).
- Note the last line of the [while](#)'s local block: `fact`. It is the local block's return value, so the value it had on the [while](#) structure's last iteration.
- Our initialization of `result` is not required; we do it for readability. We could just as well have used:

```

while counter > 0
    fact := fact * counter
    counter := counter - 1
fact

```


Type system

- [Introduction](#)
 - [Forms](#)
 - [Types](#)
- [Using forms and types](#)
 - [Forms](#)
 - [const](#)
 - [input](#)
 - [simple](#)
 - [series](#)
 - [Types](#)
 - [int](#)
 - [float](#)
 - [bool](#)
 - [color](#)
 - [string](#)
 - [plot and hline](#)
 - [line, linefill, label, box and table](#)
 - [Collections](#)
 - [User-defined types](#)
 - [void](#)
- [`na` value](#)
- [Type templates](#)
- [Type casting](#)
- [Tuples](#)

[Introduction](#)

Pine Script[®]'s type system is important because it determines what sort of values can be used when calling Pine Script[®] functions, which is a requirement to do pretty much anything in Pine Script[®]. While it is possible to write very simple scripts without knowing anything about the type system, a reasonable understanding of it is necessary to achieve any degree of proficiency with the language, and in-depth knowledge of its subtleties will allow you to exploit the full potential of Pine Script[®].

The type system uses the *form type* pair to qualify the type of all values, be they literals, a variable, the result of an expression, the value returned by functions or the arguments supplied when calling a function.

The *form* expresses when a value is known.

The *type* denotes the nature of a value.

Note

We will often use “type” to refer to the “form type” pair.

The type system is intimately linked to Pine Script[®]'s [execution model](#) and [time series](#) concepts. Understanding all three is key to making the most of the power of Pine Script[®].

Forms

Pine Script[®] **forms** identify when a variable's value is known. They are:

- “const” for values known at compile time (when adding an indicator to a chart or saving it in the Pine Script[®] Editor)
- “input” for values known at input time (when values are changed in a script's “Settings/Inputs” tab)
- “simple” for values known at bar zero (when the script begins execution on the chart's first historical bar)
- “series” for values known on each bar (any time during the execution of a script on any bar)

Forms are organized in the following hierarchy: **const** < **input** < **simple** < **series**, where “const” is considered a *weaker* form than “input”, for example, and “series” *stronger* than “simple”. The form hierarchy translates into the rule that, whenever a given form is required, a weaker form is also allowed.

An expression's result is always of the strongest form used in the expression's calculation. Furthermore, once a variable acquires a stronger form, that state is irreversible; it can never be converted back to a weaker form. A variable of “series” form can thus never be converted back to a “simple” form, for use with a function that requires arguments of that form.

Note that of all these forms, only the “series” form allows values to change dynamically, bar to bar, during the script's execution over each bar of the chart's history. Such values include [close](#) or [hlc3](#) or any variable calculated using values of “series” form. Variables of “const”, “input” or “simple” forms cannot change values once execution of the script has begun.

Types

Pine Script[®] **types** identify the nature of a value. They are:

- The fundamental types: “int”, “float”, “bool”, “color” and “string”
- The special types: “plot”, “hline”, “line”, “linefill”, “label”, “box”, “table”, “array”, “matrix”, and “map”
- User-defined types (UDTs)
- “void”

Each fundamental type refers to the nature of the value contained in a variable: 1 is of type “int”, 1.0 is of type “float”, "AAPL" is of type “string”, etc. Variables of special types contain an ID referring to an object of the type's name. A variable of type “label” contains an ID (or *pointer*) referring to a label, and so on. The “void” type means no value is returned.

The Pine Script[®] compiler can automatically convert some types into others when a value is not of the required type. The auto-casting rules are: **int** ? **float** ? **bool**. See the [Type casting](#) section of this page for more information on type casting.

Except for parameter definitions appearing in function signatures, Pine Script[®] forms are implicit in code; they are never declared because they are always determined by the compiler. Types, however, can be specified when declaring variables, e.g.:

```
//@version=5
indicator("", "", true)
int periodInput = input.int(100, "Period", minval = 2)
float ma = ta.sma(close, periodInput)
bool xUp = ta.crossover(close, ma)
color maColor = close > ma ? color.lime : color.fuchsia
plot(ma, "MA", maColor)
plotchar(xUp, "Cross Up", "▲", location.top, size = size.tiny)
```

Using forms and types

Forms

const

Values of “const” form must be known at compile time, before your script has access to any information related to the symbol/timeframe information it is running on. Compilation occurs when you save a script in the Pine Script® Editor, which doesn’t even require it to already be running on your chart. “const” variables cannot change during the execution of a script.

Variables of “const” form can be initialized using a *literal* value, or calculated from expressions using only literal values or other variables of “const” form. Our [Style guide](#) recommends using upper case SNAKE_CASE to name variables of “const” form. While it is not a requirement, “const” variables can be declared using the [var](#) keyword so they are only initialized on the first bar of the dataset. See the [section on `var`](#) for more information.

These are examples of literal values:

- *literal int*: 1, -1, 42
- *literal float*: 1., 1.0, 3.14, 6.02E-23, 3e8
- *literal bool*: true, false
- *literal string*: "A text literal", "Embedded single quotes 'text'", 'Embedded double quotes "text"'
- *literal color*: #FF55C6, #FF55C6ff

Note

In Pine Script®, the built-in variables `open`, `high`, `low`, `close`, `volume`, `time`, `hl2`, `hlc3`, `ohlc4`, etc., are of “series” form because their values can change bar to bar.

The “const” form is a requirement for the arguments to the `title` and `shorttitle` parameters in [indicator\(\)](#), for example. All these are valid variables that can be used as arguments for those parameters when calling the function:

```
//@version=5
NAME1 = "My indicator"
var NAME2 = "My Indicator"
var NAME3 = "My" + "Indicator"
var NAME4 = NAME2 + " No. 2"
indicator(NAME4, "", true)
plot(close)
```

This will trigger a compilation error:

```
//@version=5
var NAME = "My indicator for " + syminfo.type
indicator(NAME, "", true)
plot(close)
```

The reason for the error is that the `NAME` variable’s calculation depends on the value of [syminfo.type](#) which is a “simple string” ([syminfo.type](#) returns a string corresponding to the sector the chart’s symbol belongs to, eg., `"crypto"`, `"forex"`, etc.).

Note that using the `:=` operator to assign a new value to a previously declared “const” variable will transform it into a “simple” variable, e.g., here with `name1`, for which we do not use an uppercase name because it is not of “const” form:

```
var name1 = "My Indicator "  
var NAME2 = "No. 2"  
name1 := name1 + NAME2
```

input

Values of “input” form are known when the values initialized through `input.*()` functions are determined. These functions determine the values that can be modified by script users in the script’s “Settings/Inputs” tab. When these values are changed, this always triggers a re-execution of the script from the beginning of the chart’s history (bar zero), so variables of “input” form are always known when the script begins execution, and they do not change during the script’s execution.

Note

The [input.source\(\)](#) function yields a value of “series” type — not “input”. This is because built-in variables such as [open](#), [high](#), [low](#), [close](#), [hl2](#), [hlc3](#), [ohlc4](#), etc., are of “series” form.

The script plots the moving average of a user-defined source and period from a symbol and timeframe also determined through inputs:

```
//@version=5  
indicator("", "", true)  
symbolInput = input.symbol("AAPL", "Symbol")  
timeframeInput = input.timeframe("D", "Timeframe")  
sourceInput = input.source(close, "Source")  
periodInput = input(10, "Period")  
v = request.security(symbolInput, timeframeInput, ta.sma(sourceInput,  
periodInput))  
plot(v)
```

Note that:

- The `symbolInput`, `timeframeInput` and `periodInput` variables are of “input” form.
- The `sourceInput` variable is of “series” form because it is determined from a call to [input.source\(\)](#).
- Our [request.security\(\)](#) call is valid because its `symbol` and `timeframe` parameters require a “simple” argument and the “input” form we use is weaker than “simple”. The function’s `expression` parameter requires a “series” form argument, and that is what form our `sourceInput` variable is. Note that because a “series” form is required there, we could have used “const”, “input” or “simple” forms as well.
- As per our style guide’s recommendations, we use the “Input” suffix with our input variables to help readers of our code remember the origin of these variables.

Wherever an “input” form is required, a “const” form can also be used.

simple

Values of “simple” form are known only when a script begins execution on the first bar of a chart’s history, and they never change during the execution of the script. Built-in variables of the `syminfo.*`, `timeframe.*` and `ticker.*` families, for example, all return results of “simple” form because their value depends on the chart’s symbol, which can only be detected when the script executes on it.

A “simple” form argument is also required for the `length` argument of functions such as [ta.ema\(\)](#) or [ta.rma\(\)](#) which cannot work with dynamic lengths that could change during the script’s execution.

Wherever a “simple” form is required, a “const” or “input” form can also be used.

series

Values of “series” form (also sometimes called *dynamic*) provide the most flexibility because they can change on any bar, or even multiples times during the same bar, in loops for example. Built-in variables such as [open](#), [close](#), [high](#), [time](#) or [volume](#) are of “series” form, as would be the result of expressions calculated using them. Functions such as [barssince\(\)](#) or [crossover\(\)](#) yield a result of “series” form because it varies bar to bar, as does that of the `[]` history-referencing operator used to access past values of a time series. While the “series” form is the most common form used in Pine Script[®], it is not always allowed as arguments to built-in functions.

Suppose you want to display the value of pivots on your chart. This will require converting values into strings, so the string values your code will be using will be of “series string” type. The [label.new\(\)](#) function can be used to place such “series string” text on the chart because its `text` parameter accepts arguments of “series” form:

```
//@version=5
indicator("", "", true)
pivotBarsInput = input(3)
hiP = ta.pivohigh(high, pivotBarsInput, pivotBarsInput)
if not na(hiP)
    label.new(bar_index[pivotBarsInput], hiP, str.tostring(hiP, format.mintick),
        style = label.style_label_down,
        color = na,
        textcolor = color.silver)
plotchar(hiP, "hiP", "•", location.top, size = size.tiny)
```

Note that:

- The `str.tostring(hiP, format.mintick)` call we use to convert the pivot’s value to a string yields a “series string” result, which will work with [label.new\(\)](#).
- While prices appear at the pivot, the pivots actually require `pivotBarsInput` bars to have elapsed before they can be detected. Pivot prices only appear on the pivot because we plot them in the past after the pivot’s detection, using `bar_index[pivotBarsInput]` (the [bar_index](#)’s value, offset `pivotBarsInput` bars back). In real time, these prices would only appear `pivotBarsInput` bars after the actual pivot.
- We print a blue dot using [plotchar\(\)](#) when a pivot is detected in our code.
- Pine Script[®]’s [plotshape\(\)](#) can also be used to position text on the chart, but because its `text` parameter requires a “const string” argument, we could not have used it in place of [label.new\(\)](#) in our script.

Wherever a “series” form is required, a “const”, “input” or “simple” form can also be used.

Types

int

Integer literals must be written in decimal notation, e.g.:

```
1
-1
750
```

Built-in variables such as [bar_index](#), [time](#), [timenow](#), [time_close](#), or [dayofmonth](#) all return values of type “int”.

float

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

```
3.14159      // Rounded value of Pi (π)
- 3.0
6.02e23     // 6.02 * 10^23 (a very large value)
1.6e-19     // 1.6 * 10^-19 (a very small value)
```

The internal precision of floats in Pine Script[®] is 1e-10.

bool

There are only two literals representing *bool* values:

```
true      // true value
false     // false value
```

When an expression of type “bool” returns [na](#) and it is used to test a conditional statement or operator, the “false” branch is executed.

color

Color literals have the following format: `#RRGGBB` or `#RRGGBBAA`. The letter pairs represent 00 to FF hexadecimal values (0 to 255 in decimal) where:

- RR, GG and BB pairs are the values for the color’s red, green and blue components
- AA is an optional value for the color’s transparency (or *alpha* component) where 00 is invisible and FF opaque. When no AA pair is supplied, FF is used.
- The hexadecimal letters can be upper or lower case

Examples:

```
#000000      // black color
#FF0000      // red color
#00FF00      // green color
#0000FF      // blue color
#FFFFFF      // white color
#808080      // gray color
#3ff7a0      // some custom color
#FF000080    // 50% transparent red color
#FF0000ff    // same as #FF0000, fully opaque red color
#FF000000    // completely transparent color
```

Pine Script[®] also has [built-in color constants](#) such as [color.green](#), [color.red](#), [color.orange](#), [color.blue](#) (the default color used in [plot\(\)](#) and other plotting functions), etc.

When using color built-ins, is possible to add transparency information to them with [color.new](#).

Note that when specifying red, green or blue components in `color.*()` functions, a 0-255 decimal value must be used. When specifying transparency in such functions, it is in the form of a 0-100 value (which can be of “float” type to access the underlying 255 potential valoues) where the scale 00-FF scale for color literals is inverted: 100 is thus invisible and 0 is opaque.

Here is an example:

```
//@version=5
indicator("Shading the chart's background", "", true)
BASE_COLOR = color.navy
bgColor = dayofweek == dayofweek.monday      ? color.new(BASE_COLOR, 50) :
```

```

    dayofweek == dayofweek.tuesday    ? color.new(BASE_COLOR, 60) :
    dayofweek == dayofweek.wednesday ? color.new(BASE_COLOR, 70) :
    dayofweek == dayofweek.thursday  ? color.new(BASE_COLOR, 80) :
    dayofweek == dayofweek.friday    ? color.new(BASE_COLOR, 90) :
    color.new(color.blue, 80)
bgcolor(bgColor)

```

See the page on [colors](#) for more information on using colors in Pine Script®.

string

String literals may be enclosed in single or double quotation marks, e.g.:

```

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

```

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

```

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

```

You can escape the string’s delimiter in the string by using a backslash. For example:

```

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

```

You can concatenate strings using the + operator.

plot and hline

Pine Script®’s [fill\(\)](#) function fills the space between two lines with a color. Both lines must have been plotted with either [plot\(\)](#) or [hline\(\)](#) function calls. Each plotted line is referred to in the [fill\(\)](#) function using IDs which are of “plot” or “hline” type, e.g.:

```

//@version=5
indicator("", "", true)
plotID1 = plot(high)
plotID2 = plot(math.max(close, open))
fill(plotID1, plotID2, color.yellow)

```

Note that there is no `plot` or `hline` keyword to explicitly declare the type of [plot\(\)](#) or [hline\(\)](#) IDs.

line, linefill, label, box and table

Drawings appeared in Pine Script® starting with v4. Each drawing has its own type: [line](#), [linefill](#), [label](#), [box](#), [table](#).

Each type is also used as a namespace containing all the built-in functions used to operate on each type of drawing. One of these is a `new()` constructor used to create an object of that type:

[line.new\(\)](#), [linefill.new\(\)](#), [label.new\(\)](#), [box.new\(\)](#) and [table.new\(\)](#).

These functions all return an ID which is a reference that uniquely identifies a drawing object. IDs are always of “series” form, thus their form and type is “series line”, “series label”, etc. Drawing IDs act like a pointer in that they are used to reference a specific instance of a drawing in all the functions of that drawing’s namespace. For example, the line ID returned by a [line.new\(\)](#) call will then be used to refer to it when comes time to delete the line using [line.delete\(\)](#).

Collections

Collections in Pine Script® ([arrays](#), [matrices](#), and [maps](#)) utilize a reference ID, much like other special types (e.g., labels). The type of the ID defines the type of elements the collection will contain. In Pine, we specify array, matrix, and map types by appending a [type template](#) to the [array](#), [matrix](#), or [map](#) keywords:

- `array<int>` defines an array containing “int” elements.
- `array<label>` defines an array containing “label” IDs.
- `array<UDT>` defines an array containing objects of a [user-defined type \(UDT\)](#).
- `matrix<float>` defines a matrix containing “float” elements.
- `matrix<UDT>` defines a matrix containing objects of a [user-defined type \(UDT\)](#).
- `map<string, float>` defines a map containing “string” keys and “float” values.
- `map<int, UDT>` defines a map containing “int” keys and values of a [user-defined type \(UDT\)](#).

For example, one can declare an “int” array with a single element value of 10 in any of the following, equivalent ways:

```
a1 = array.new<int>(1, 10)
array<int> a2 = array.new<int>(1, 10)
a3 = array.from(10)
array<int> a4 = array.from(10)
```

Note that:

- The `int []` syntax can also specify an array of “int” elements, but its use is discouraged. No equivalent exists to specify the types of matrices or maps in that way.
- Type-specific built-ins exist for arrays, such as [array.new_int\(\)](#), but the more generic [array.new<type>](#) form is preferred, which would be `array.new<int>()` to create an array of “int” elements.

User-defined types

The [type](#) keyword allows the creation of *user-defined types* (UDTs) from which [objects](#) can be created. UDTs are composite types; they contain an arbitrary number of *fields* that can be of any type. The syntax to define a *user-defined type* is:

where:

- `export` is used to export the UDT from a library. See the [Libraries](#) page for more information.
- `<UDT_identifier>` is the name of the user-defined type.
- `<field_type>` is the type of the field.
- `<field_name>` is the name of the field.
- `<value>` is an optional default value for the field, which will be assigned to it when new objects of that UDT are created. The field’s default value will be [na](#) if none is specified. The same rules as those governing the default values of parameters in function signatures apply to the default values of fields. For example, the [\[\]](#) history-referencing operator cannot be used with them, and expressions are not allowed.

In this example, we create a UDT containing two fields to hold pivot information, the [time](#) of the pivot’s bar and its price level:

```
type pivotPoint
    int openTime
```



```
float level
```

User-defined types can be embedded, so a field can be of the same type as the UDT it belongs to. Here, we add a field to our previous `pivotPoint` type that will hold the pivot information for another pivot point:

```
type pivotPoint
  int openTime
  float level
  pivotPoint nextPivot
```

Two built-in methods can be used with a UDT: `new()` and `copy()`. Read about them in the [Objects](#) page.

void

There is a “void” type in Pine Script[®]. Functions having only side-effects and returning no usable result return the “void” type. An example of such a function is [alert\(\)](#); it does something (triggers an alert event), but it returns no useful value.

A “void” result cannot be used in an expression or assigned to a variable. No `void` keyword exists in Pine Script[®], as variables cannot be declared using the “void” type.

`na` value

In Pine Script[®], there is a special value called [na](#), which is an acronym for *not available*, meaning the value of an expression or variable is undefined. It is similar to the `null` value in Java, or `None` in Python.

[na](#) values can be automatically cast to almost any type. In some cases, however, the compiler cannot automatically infer a type for an [na](#) value because more than one automatic type-casting rule can be applied. For example:

```
// Compilation error!
myVar = na
```

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

```
float myVar = na
```

or

```
myVar = float(na)
```

To test if some value is [na](#), a special function must be used: [na\(\)](#). For example:

```
myClose = na(myVar) ? 0 : close
```

Do not use the `==` operator to test for [na](#) values, as this method is unreliable.

Designing your calculations so they are [na](#)-resistant is often useful. In this example, we define a condition that is `true` when the bar’s [close](#) is higher than the previous one. For this calculation to work correctly on the dataset’s first bar where no previous `close` exists and `close[1]` will return [na](#), we use the [nz\(\)](#) function to replace it with the current bar’s [open](#) for that special case:

```
bool risingClose = close > nz(close[1], open)
```

Protecting against [na](#) values can also be useful to prevent an initial [na](#) value from propagating in a calculation's result on all bars. This happens here because the initial value of `ath` is [na](#), and [math.max\(\)](#) returns [na](#) if one of its arguments is [na](#):

```
// Declare `ath` and initialize it with `na` on the first bar.
var float ath = na
// On all bars, calculate the maximum between the `high` and the previous value
of `ath`.
ath := math.max(ath, high)
```

To protect against this, we could instead use:

```
var float ath = na
ath := math.max(nz(ath), high)
```

where we are replacing any [na](#) values of `ath` with zero. Even better would be:

```
var float ath = high
ath := math.max(ath, high)
```

Type templates

Type templates specify the data types that collections ([arrays](#), [matrices](#), and [maps](#)) can contain.

Templates for [arrays](#) and [matrices](#) consist of a single type identifier surrounded by angle brackets, e.g., `<int>`, `<label>`, and `<PivotPoint>` (where `PivotPoint` is a [user-defined type \(UDT\)](#)).

Templates for [maps](#) consist of two type identifiers enclosed in angle brackets, where the first specifies the type of *keys* in each key-value pair, and the second specifies the *value* type. For example, `<string, float>` is a type template for a map that holds `string` keys and `float` values.

Users can construct type templates from:

- Fundamental types: “int”, “float”, “bool”, “color”, and “string”
- The following special types: “line”, “linefill”, “label”, “box”, and “table”
- [User-defined types \(UDTs\)](#)

Note that:

- [Maps](#) can use any of these types as *values*, but they can only accept fundamental types as *keys*.

Scripts use type templates to declare variables that point to collections, and when creating new collection instances. For example:

```
//@version=5
indicator("Type templates demo")

//@variable A variable initially assigned to `na` that accepts arrays of `int`
values.
array<int> intArray = na
//@variable An empty matrix that holds `float` values.
floatMatrix = matrix.new<float>()
//@variable An empty map that holds `string` keys and `color` values.
stringColorMap = map.new<string, color>()
```

Type casting

There is an automatic type-casting mechanism in Pine Script[®] which can *cast* (or convert) certain types to another. The auto-casting rules are: “**int**” ? “**float**” ? “**bool**”, which means that when a “float” is required, an “int” can be used in its place, and when a “bool” value is required, an “int” or “float” value can be used in its place.

See auto-casting in action in this code:

```
//@version=5
indicator("")
plotshape(close)
```

Note that:

- [plotshape\(\)](#) requires a “series bool” argument for its first parameter named `series`. The `true/false` value of that “bool” argument determines if the function plots a shape or not.
- We are here calling [plotshape\(\)](#) with [close](#) as its first argument. This would not be allowed without Pine’s auto-casting rules, which allow a “float” to be cast to a “bool”. When a “float” is cast to a “bool”, any non-zero values are converted to `true`, and zero values are converted to `false`. As a result of this, our code will plot an “X” on all bars, as long as [close](#) is not equal to zero.

It may sometimes be necessary to cast one type into another because auto-casting rules will not suffice. For these cases, explicit type-casting functions exist. They are: [int\(\)](#), [float\(\)](#), [bool\(\)](#), [color\(\)](#), [string\(\)](#), [line\(\)](#), [linefill\(\)](#), [label\(\)](#), [box\(\)](#), and [table\(\)](#).

This is code that will not compile because we fail to convert the type of the argument used for `length` when calling [ta.sma\(\)](#):

```
//@version=5
indicator("")
len = 10.0
s = ta.sma(close, len) // Compilation error!
plot(s)
```

The code fails to compile with the following error: *Cannot call ‘ta.sma’ with argument ‘length’=‘len’. An argument of ‘const float’ type was used but a ‘series int’ is expected.* The compiler is telling us that we supplied a “float” value where an “int” is required. There is no auto-casting rule that can automatically cast a “float” to an “int”, so we will need to do the job ourselves. For this, we will use the [int\(\)](#) function to force the type conversion of the value we supply as a `length` to [ta.sma\(\)](#) from “float” to “int”:

```
//@version=5
indicator("")
len = 10.0
s = ta.sma(close, int(len))
plot(s)
```

Explicit type-casting can also be useful when declaring variables and initializing them to [na](#) which can be done in two ways:

```
// Cast `na` to the "label" type.
lbl = label(na)
// Explicitly declare the type of the new variable.
label lbl = na
```

Tuples

A *tuple* is a comma-separated set of expressions enclosed in brackets that can be used when a function or a local block must return more than one variable as a result. For example:

```
calcSumAndMult(a, b) =>
    sum = a + b
    mult = a * b
    [sum, mult]
```

In this example there is a two-element tuple on the last statement of the function's code block, which is the result returned by the function. Tuple elements can be of any type. There is also a special syntax for calling functions that return tuples, which uses a *tuple declaration* on the left side of the equal sign in what is a multi-variable declaration. The result of a function such as `calcSumAndMult()` that returns a tuple must be assigned to a *tuple declaration*, i.e., a set of comma-separated list of *new* variables that will receive the values returned by the function. Here, the value of the `sum` and `mult` variables calculated by the function will be assigned to the `s` and `m` variables:

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

Note that the type of `s` and `m` cannot be explicitly defined; it is always inferred by the type of the function return results.

Tuples can be useful to request multiple values in one [request.security\(\)](#) call:

```
roundedOHLC() =>
    [math.round_to_mintick(open), math.round_to_mintick(high),
    math.round_to_mintick(low), math.round_to_mintick(close)]
[op, hi, lo, cl] = request.security(syminfo.tickerid, "D", roundedOHLC())
```

or:

```
[op, hi, lo, cl] = request.security(syminfo.tickerid, "D",
[math.round_to_mintick(open), math.round_to_mintick(high),
math.round_to_mintick(low), math.round_to_mintick(close)])
```

or this form if no rounding is required

```
[op, hi, lo, cl] = request.security(syminfo.tickerid, "D", [open, high, low,
close])
```

Tuples can also be used as return results of local blocks, in an [if](#) statement for example:

```
[v1, v2] = if close > open
    [high, close]
else
    [close, low]
```

They cannot be used in ternaries, however, because the return values of a ternary statement are not considered as local blocks. This is not allowed:

```
// Not allowed.
[v1, v2] = close > open ? [high, close] : [close, low]
```

Please note that the items within a tuple returned from a function may be of simple or series form, depending on its contents. If a tuple contains a series value, all other elements within the tuple will also be of the series form. For example:

```
//@version=5
indicator("tuple_typeforms")
```

```

makeTicker(simple string prefix, simple string ticker) =>
    tId = prefix + ":" + ticker // simple string
    source = close // series float
    [tId, source]

// Both variables are series now.
[tId, source] = makeTicker("BATS", "AAPL")

// Error cannot call 'request.security' with 'series string' tId.
r = request.security(tId, "", source)

plot(r)

```

Built-ins

- [Introduction](#)
- [Built-in variables](#)
- [Built-in functions](#)

Introduction

Pine Script[®] has hundreds of *built-in* variables and functions. They provide your scripts with valuable information and make calculations for you, dispensing you from coding them. The better you know the built-ins, the more you will be able to do with your Pine scripts.

In this page we present an overview of some of Pine Script[®]'s built-in variables and functions. They will be covered in more detail in the pages of this manual covering specific themes.

All built-in variables and functions are defined in the Pine Script[®] [v5 Reference Manual](#). It is called a “Reference Manual” because it is the definitive reference on the Pine Script[®] language. It is an essential tool that will accompany you anytime you code in Pine, whether you are a beginner or an expert. If you are learning your first programming language, make the [Reference Manual](#) your friend. Ignoring it will make your programming experience with Pine Script[®] difficult and frustrating — as it would with any other programming language.

Variables and functions in the same family share the same *namespace*, which is a prefix to the function's name. The [ta.sma\(\)](#) function, for example, is in the `ta` namespace, which stands for “technical analysis”. A namespace can contain both variables and functions.

Some variables have function versions as well, e.g.:

- The [ta.tr](#) variable returns the “True Range” of the current bar. The [ta.tr\(true\)](#) function call also returns the “True Range”, but when the previous [close](#) value which is normally needed to calculate it is [na](#), it calculates using `high - low` instead.
- The [time](#) variable gives the time at the [open](#) of the current bar. The [time\(timeframe\)](#) function returns the time of the bar's [open](#) from the `timeframe` specified, even if the chart's timeframe is different. The [time\(timeframe, session\)](#) function returns the time of the bar's [open](#) from the `timeframe` specified, but only if it is within the `session` time. The [time\(timeframe, session, timezone\)](#) function returns the time of the bar's [open](#) from the `timeframe` specified, but only if it is within the `session` time in the specified `timezone`.

Built-in variables

Built-in variables exist for different purposes. These are a few examples:

- Price- and volume-related variables: [open](#), [high](#), [low](#), [close](#), [hl2](#), [hlc3](#), [ohlc4](#), and [volume](#).
- Symbol-related information in the `syminfo` namespace: [syminfo.basecurrency](#), [syminfo.currency](#), [syminfo.description](#), [syminfo.mintick](#), [syminfo.pointvalue](#), [syminfo.prefix](#), [syminfo.root](#), [syminfo.session](#), [syminfo.ticker](#), [syminfo.tickerid](#), [syminfo.timezone](#), and [syminfo.type](#).
- Timeframe (a.k.a. “interval” or “resolution”, e.g., 15sec, 30min, 60min, 1D, 3M) variables in the `timeframe` namespace: [timeframe.isseconds](#), [timeframe.isminutes](#), [timeframe.isintraday](#), [timeframe.isdaily](#), [timeframe.isweekly](#), [timeframe.ismonthly](#), [timeframe.isdwm](#), [timeframe.multiplier](#), and [timeframe.period](#).
- Bar states in the `barstate` namespace (see the [Bar states](#) page): [barstate.isconfirmed](#), [barstate.isfirst](#), [barstate.ishistory](#), [barstate.islast](#), [barstate.islastconfirmedhistory](#), [barstate.isnew](#), and [barstate.isrealtime](#).
- Strategy-related information in the `strategy` namespace: [strategy.equity](#), [strategy.initial_capital](#), [strategy.grossloss](#), [strategy.grossprofit](#), [strategy.wintrades](#), [strategy.losstrades](#), [strategy.position_size](#), [strategy.position_avg_price](#), [strategy.wintrades](#), etc.

Built-in functions

Many functions are used for the result(s) they return. These are a few examples:

- Math-related functions in the `math` namespace: [math.abs\(\)](#), [math.log\(\)](#), [math.max\(\)](#), [math.random\(\)](#), [math.round_to_mintick\(\)](#), etc.
- Technical indicators in the `ta` namespace: [ta.sma\(\)](#), [ta.ema\(\)](#), [ta.macd\(\)](#), [ta.rsi\(\)](#), [ta.supertrend\(\)](#), etc.
- Support functions often used to calculate technical indicators in the `ta` namespace: [ta.barssince\(\)](#), [ta.crossover\(\)](#), [ta.highest\(\)](#), etc.
- Functions to request data from other symbols or timeframes in the `request` namespace: [request.dividends\(\)](#), [request.earnings\(\)](#), [request.financial\(\)](#), [request.quandl\(\)](#), [request.security\(\)](#), [request.splits\(\)](#).
- Functions to manipulate strings in the `str` namespace: [str.format\(\)](#), [str.length\(\)](#), [str.tonumber\(\)](#), [str.tostring\(\)](#), etc.
- Functions used to define the input values that script users can modify in the script’s “Settings/Inputs” tab, in the `input` namespace: [input\(\)](#), [input.color\(\)](#), [input.int\(\)](#), [input.session\(\)](#), [input.symbol\(\)](#), etc.
- Functions used to manipulate colors in the `color` namespace: [color.from_gradient\(\)](#), [color.new\(\)](#), [color.rgb\(\)](#), etc.

Some functions do not return a result but are used for their side effects, which means they do something, even if they don’t return a result:

- Functions used as a declaration statement defining one of three types of Pine scripts, and its properties. Each script must begin with a call to one of these functions: [indicator\(\)](#), [strategy\(\)](#) or [library\(\)](#).
- Plotting or coloring functions: [bgcolor\(\)](#), [plotbar\(\)](#), [plotcandle\(\)](#), [plotchar\(\)](#), [plotshape\(\)](#), [fill\(\)](#).
- Strategy functions placing orders, in the `strategy` namespace: [strategy.cancel\(\)](#), [strategy.close\(\)](#), [strategy.entry\(\)](#), [strategy.exit\(\)](#), [strategy.order\(\)](#), etc.

- Strategy functions returning information on individual past trades, in the `strategy` namespace: [strategy.closedtrades.entry_bar_index\(\)](#), [strategy.closedtrades.entry_price\(\)](#), [strategy.closedtrades.entry_time\(\)](#), [strategy.closedtrades.exit_bar_index\(\)](#), [strategy.closedtrades.max_drawdown\(\)](#), [strategy.closedtrades.max_runup\(\)](#), [strategy.closedtrades.profit\(\)](#), etc.
- Functions to generate alert events: [alert\(\)](#) and [alertcondition\(\)](#).

Other functions return a result, but we don't always use it, e.g.: [hline\(\)](#), [plot\(\)](#), [array.pop\(\)](#), [label.new\(\)](#), etc.

All built-in functions are defined in the Pine Script[®] [v5 Reference Manual](#). You can click on any of the function names listed here to go to its entry in the Reference Manual, which documents the function's signature, i.e., the list of *parameters* it accepts and the form-type of the value(s) it returns (a function can return more than one result). The Reference Manual entry will also list, for each parameter:

- Its name.
- The form-type of the value it requires (we use *argument* to name the values passed to a function when calling it).
- If the parameter is required or not.

All built-in functions have one or more parameters defined in their signature. Not all parameters are required for every function.

Let's look at the [ta.vwma\(\)](#) function, which returns the volume-weighted moving average of a source value. This is its entry in the Reference Manual:



The entry gives us the information we need to use it:

- What the function does.
- Its signature (or definition):
`ta.vwma(source, length) → series float`
- The parameters it includes: `source` and `length`
- The form and type of the result it returns: “series float”.
- An example showing it in use: `plot(ta.vwma(close, 15))`.
- An example showing what it does, but in long form, so you can better understand its calculations. Note that this is meant to explain — not as usable code, because it is more complicated and takes longer to execute. There are only disadvantages to using the long form.
- The “RETURNS” section explains exactly what value the function returns.
- The “ARGUMENTS” section lists each parameter and gives the critical information concerning what form-type is required for arguments used when calling the function.
- The “SEE ALSO” section refers you to related Reference Manual entries.

This is a call to the function in a line of code that declares a `myVwma` variable and assigns the result of `ta.vwma(close, 20)` to it:

```
myVwma = ta.vwma(close, 20)
```

Note that:

- We use the built-in variable [close](#) as the argument for the `source` parameter.
- We use 20 as the argument for the `length` parameter.
- If placed in the global scope (i.e., starting in a line's first position), it will be executed by the Pine Script® runtime on each bar of the chart.

We can also use the parameter names when calling the function. Parameter names are called *keyword arguments* when used in a function call:

```
myVwma = ta.vwma(source = close, length = 20)
```

You can change the position of arguments when using keyword arguments, but only if you use them for all your arguments. When calling functions with many parameters such as [indicator\(\)](#), you can also forego keyword arguments for the first arguments, as long as you don't skip any. If you skip some, you must then use keyword arguments so the Pine Script® compiler can figure out which parameter they correspond to, e.g.:

```
indicator("Example", "Ex", true, max_bars_back = 100)
```

Mixing things up this way is not allowed:

```
indicator(precision = 3, "Example") // Compilation error!
```

When calling built-ins, it is critical to ensure that the arguments you use are of the form and type required, which will vary for each parameter.

To learn how to do this, one needs to understand Pine Script®'s [type system](#). The Reference Manual entry for each built-in function includes an “ARGUMENTS” section which lists the form-type required for the argument supplied to each of the function's parameters.

User-defined functions

- [Introduction](#)
- [Single-line functions](#)
- [Multi-line functions](#)
- [Scopes in the script](#)
- [Functions that return multiple results](#)
- [Limitations](#)

Introduction

User-defined functions are functions that you write, as opposed to the built-in functions in Pine Script®. They are useful to define calculations that you must do repetitively, or that you want to isolate from your script's main section of calculations. Think of user-defined functions as a way to extend the capabilities of Pine Script®, when no built-in function will do what you need.

You can write your functions in two ways:

- In a single line, when they are simple, or
- On multiple lines

Functions can be located in two places:

- If a function is only used in one script, you can include it in the script where it is used. See our [Style guide](#) for recommendations on where to place functions in your script.
- You can create a Pine Script[®] [library](#) to include your functions, which makes them reusable in other scripts without having to copy their code. Distinct requirements exist for library functions. They are explained in the page on [libraries](#).

Whether they use one line or multiple lines, user-defined functions have the following characteristics:

- They cannot be embedded. All functions are defined in the script's global scope.
- They do not support recursion. It is **not allowed** for a function to call itself from within its own code.
- The type of the value returned by a function is determined automatically and depends on the type of arguments used in each particular function call.
- Each function call

Single-line functions

Simple functions can often be written in one line. This is the formal definition of single-line functions:

```
<function_declaration>
  <identifier>(<parameter_list>) => <return_value>

<parameter_list>
  {<parameter_definition>{, <parameter_definition>}}

<parameter_definition>
  [<identifier> = <default_value>]

<return_value>
  <statement> | <expression> | <tuple>
```

Here is an example:

```
f(x, y) => x + y
```

After the function `f()` has been declared, it's possible to call it using different types of arguments:

```
a = f(open, close)
b = f(2, 2)
c = f(open, 2)
```

In the example above, the type of variable `a` is *series* because the arguments are both *series*. The type of variable `b` is *integer* because arguments are both *literal integers*. The type of variable `c` is *series* because the addition of a *series* and *literal integer* produces a *series* result.

Multi-line functions

Pine Script[®] also supports multi-line functions with the following syntax:

```
<identifier>(<parameter_list>) =>
  <local_block>

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

where:

```
<parameter_list>
  {<parameter_definition>{, <parameter_definition>}}

<parameter_definition>
  [<identifier> = <default_value>]
```

The body of a multi-line function consists of several statements. Each statement is placed on a separate line and must be preceded by 1 indentation (4 spaces or 1 tab). The indentation before the statement indicates that it is a part of the body of the function and not part of the script's global scope. After the function's code, the first statement without an indent indicates the body of the function has ended.

Either an expression or a declared variable should be the last statement of the function's body. The result of this expression (or variable) will be the result of the function's call. For example:

```
geom_average(x, y) =>
  a = x*x
  b = y*y
  math.sqrt(a + b)
```

The function `geom_average` has two arguments and creates two variables in the body: `a` and `b`. The last statement calls the function `math.sqrt` (an extraction of the square root). The `geom_average` call will return the value of the last expression: `(math.sqrt(a + b))`.

Scopes in the script

Variables declared outside the body of a function or of other local blocks belong to the *global* scope. User-declared and built-in functions, as well as built-in variables also belong to the global scope.

Each function has its own *local* scope. All the variables declared within the function, as well as the function's arguments, belong to the scope of that function, meaning that it is impossible to reference them from outside — e.g., from the global scope or the local scope of another function.

On the other hand, since it is possible to refer to any variable or function declared in the global scope from the scope of a function (except for self-referencing recursive calls), one can say that the local scope is embedded into the global scope.

In Pine Script[®], nested functions are not allowed, i.e., one cannot declare a function inside another one. All user functions are declared in the global scope. Local scopes cannot intersect with each other.

Functions that return multiple results

In most cases a function returns only one result, but it is possible to return a list of results (a *tuple*-like result):

```
fun(x, y) =>
  a = x+y
  b = x-y
  [a, b]
```

Special syntax is required for calling such functions:

```
[res0, res1] = fun(open, close)
plot(res0)
plot(res1)
```

Limitations

User-defined functions can use any of the Pine Script[®] built-ins, except: [barcolor\(\)](#), [fill\(\)](#), [hline\(\)](#), [indicator\(\)](#), [library\(\)](#), [plot\(\)](#), [plotbar\(\)](#), [plotcandle\(\)](#), [plotchar\(\)](#), [plotshape\(\)](#) and [strategy\(\)](#).

Objects

- [Introduction](#)
- [Creating objects](#)
- [Changing field values](#)
- [Collecting objects](#)
- [Copying objects](#)
- [Shadowing](#)

Note

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

Introduction

Pine Script[®] objects are instances of *user-defined types* (UDTs). They are the equivalent of variables containing parts called *fields*, each able to hold independent values that can be of various types.

Experienced programmers can think of UDTs as methodless classes. They allow users to create custom types that organize different values under one logical entity.

Creating objects

Before an object can be created, its type must be defined. The [User-defined types](#) section of the [Type system](#) page explains how to do so.

Let's define a `pivotPoint` type to hold pivot information:

```
type pivotPoint
    int x
    float y
    string xloc = xloc.bar_time
```

Note that:

- We use the [type](#) keyword to declare the creation of a UDT.
- We name our new UDT `pivotPoint`.
- After the first line, we create a local block containing the type and name of each field.
- The `x` field will hold the x-coordinate of the pivot. It is declared as an “int” because it will hold either a timestamp or a bar index of “int” type.
- `y` is a “float” because it will hold the pivot's price.
- `xloc` is a field that will specify the units of `x`: [xloc.bar_index](#) or [xloc.bar_time](#). We set its default value to [xloc.bar_time](#) by using the `=` operator. When an object is created from that UDT, its `xloc` field will thus be set to that value.

Now that our `pivotPoint` UDT is defined, we can proceed to create objects from it. We create objects using the UDT's `new()` built-in method. To create a new `foundPoint` object from our `pivotPoint` UDT, we use:

```
foundPoint = pivotPoint.new()
```

We can also specify field values for the created object using the following:

```
foundPoint = pivotPoint.new(time, high)
```

Or the equivalent:

```
foundPoint = pivotPoint.new(x = time, y = high)
```

At this point, the `foundPoint` object's `x` field will contain the value of the [time](#) built-in when it is created, `y` will contain the value of [high](#) and the `xloc` field will contain its default value of [xloc.bar_time](#) because no value was defined for it when creating the object.

Object placeholders can also be created by declaring [na](#) object names using the following:

```
pivotPoint foundPoint = na
```

This example displays a label where high pivots are detected. The pivots are detected `legsInput` bars after they occur, so we must plot the label in the past for it to appear on the pivot:

```
//@version=5
indicator("Pivot labels", overlay = true)
int legsInput = input(10)

// Define the `pivotPoint` UDT.
type pivotPoint
  int x
  float y
  string xloc = xloc.bar_time

// Detect high pivots.
pivotHighPrice = ta.pivohigh(legsInput, legsInput)
if not na(pivotHighPrice)
  // A new high pivot was found; display a label where it occurred `legsInput`
  bars back.
  foundPoint = pivotPoint.new(time[legsInput], pivotHighPrice)
  label.new(
    foundPoint.x,
    foundPoint.y,
    str.toString(foundPoint.y, format.mintick),
    foundPoint.xloc,
    textcolor = color.white)
```

Take note of this line from the above example:

```
foundPoint = pivotPoint.new(time[legsInput], pivotHighPrice)
```

This could also be written using the following:

```
pivotPoint foundPoint = na
foundPoint := pivotPoint.new(time[legsInput], pivotHighPrice)
```

When an object is created using [var](#) or [varip](#), those keywords apply to all of the object's fields:

```
//@version=5
indicator("")
type barInfo
  int i = bar_index
```

```

int t = time
float c = close

// Created on bar zero.
var firstBar = barInfo.new()
// Created on every bar.
currentBar = barInfo.new()

plot(firstBar.i)
plot(currentBar.i)

```

Changing field values

The value of an object's fields can be changed using the `:=` reassignment operator.

This line of our previous example:

```
foundPoint = pivotPoint.new(time[legsInput], pivotHighPrice)
```

Could be written using the following:

```
foundPoint = pivotPoint.new()
foundPoint.x := time[legsInput]
foundPoint.y := pivotHighPrice
```

Collecting objects

Pine Script® collections ([arrays](#), [matrices](#), and [maps](#)) can contain objects, allowing users to add virtual dimensions to their data structures. To declare a collection of objects, pass a UDT name into its [type template](#).

This example declares an empty [array](#) that will hold objects of a `pivotPoint` user-defined type:

```
pivotHighArray = array.new<pivotPoint>()
```

To explicitly declare the type of a variable as an [array](#), [matrix](#), or [map](#) of a [user-defined type](#), use the collection's type keyword followed by its [type template](#). For example:

```
var array<pivotPoint> pivotHighArray = na
pivotHighArray := array.new<pivotPoint>()
```

Let's use what we have learned to create a script that detects high pivot points. The script first collects historical pivot information in an [array](#). It then loops through the array on the last historical bar, creating a label for each pivot and connecting the pivots with lines:



```

//@version=5
indicator("Pivot Points High", overlay = true)

int legsInput = input(10)

// Define the `pivotPoint` UDT containing the time and price of pivots.
type pivotPoint
    int openTime
    float level

// Create an empty `pivotPoint` array.
var pivotHighArray = array.new<pivotPoint>()

```

```

// Detect new pivots (`na` is returned when no pivot is found).
pivotHighPrice = ta.pivohigh(legsInput, legsInput)

// Add a new `pivotPoint` object to the end of the array for each detected
pivot.
if not na(pivotHighPrice)
    // A new pivot is found; create a new object of `pivotPoint` type, setting
its `openTime` and `level` fields.
    newPivot = pivotPoint.new(time[legsInput], pivotHighPrice)
    // Add the new pivot object to the array.
    array.push(pivotHighArray, newPivot)

// On the last historical bar, draw pivot labels and connecting lines.
if barstate.islastconfirmedhistory
    var pivotPoint previousPoint = na
    for eachPivot in pivotHighArray
        // Display a label at the pivot point.
        label.new(eachPivot.openTime, eachPivot.level,
str.toString(eachPivot.level, format.mintick), xloc.bar_time, textcolor =
color.white)
        // Create a line between pivots.
        if not na(previousPoint)
            // Only create a line starting at the loop's second iteration
because lines connect two pivots.
            line.new(previousPoint.openTime, previousPoint.level,
eachPivot.openTime, eachPivot.level, xloc = xloc.bar_time)
            // Save the pivot for use in the next iteration.
            previousPoint := eachPivot

```

Copying objects

In Pine, objects are assigned by reference. When an existing object is assigned to a new variable, both point to the same object.

In the example below, we create a `pivot1` object and set its `x` field to 1000. Then, we declare a `pivot2` variable containing the reference to the `pivot1` object, so both point to the same instance. Changing `pivot2.x` will thus also change `pivot1.x`, as both refer to the `x` field of the same object:

```

//@version=5
indicator("")
type pivotPoint
    int x
    float y
pivot1 = pivotPoint.new()
pivot1.x := 1000
pivot2 = pivot1
pivot2.x := 2000
// Both plot the value 2000.
plot(pivot1.x)
plot(pivot2.x)

```

To create a copy of an object that is independent of the original, we can use the built-in `copy()` method in this case.

In this example, we declare the `pivot2` variable referring to a copied instance of the `pivot1` object. Now, changing `pivot2.x` will not change `pivot1.x`, as it refers to the `x` field of a separate object:

```

//@version=5

```

```

indicator("")
type pivotPoint
    int x
    float y
pivot1 = pivotPoint.new()
pivot1.x := 1000
pivot2 = pivotPoint.copy(pivot1)
pivot2.x := 2000
// Plots 1000 and 2000.
plot(pivot1.x)
plot(pivot2.x)

```

It's important to note that the built-in `copy()` method produces a *shallow copy* of an object. If an object has fields with *special types* ([array](#), [matrix](#), [map](#), [line](#), [linefill](#), [label](#), [box](#), or [table](#)), those fields in a shallow copy of the object will point to the same instances as the original.

In the following example, we have defined an `InfoLabel` type with a `label` as one of its fields. The script instantiates a shallow copy of the parent object, then calls a user-defined `set()` [method](#) to update the `info` and `lbl` fields of each object. Since the `lbl` field of both objects points to the same label instance, changes to this field in either object affect the other:

```

//@version=5
indicator("Shallow Copy")

type InfoLabel
    string info
    label lbl

method set(InfoLabel this, int x = na, int y = na, string info = na) =>
    if not na(x)
        this.lbl.set_x(x)
    if not na(y)
        this.lbl.set_y(y)
    if not na(info)
        this.info := info
        this.lbl.set_text(this.info)

var parent = InfoLabel.new("", label.new(0, 0))
var shallow = parent.copy()

parent.set(bar_index, 0, "Parent")
shallow.set(bar_index, 1, "Shallow Copy")

```

To produce a *deep copy* of an object with all of its special type fields pointing to independent instances, we must explicitly copy those fields as well.

In this example, we have defined a `deepCopy()` method that instantiates a new `InfoLabel` object with its `lbl` field pointing to a copy of the original's field. Changes to the deep copy's `lbl` field will not affect the parent object, as it points to a separate instance:

```

//@version=5
indicator("Deep Copy")

type InfoLabel
    string info
    label lbl

method set(InfoLabel this, int x = na, int y = na, string info = na) =>
    if not na(x)
        this.lbl.set_x(x)
    if not na(y)
        this.lbl.set_y(y)

```

```

    if not na(info)
        this.info := info
        this.lbl.set_text(this.info)

method deepCopy(InfoLabel this) =>
    InfoLabel.new(this.info, this.lbl.copy())

var parent = InfoLabel.new("", label.new(0, 0))
var deep   = parent.deepCopy()

parent.set(bar_index, 0, "Parent")
deep.set(bar_index, 1, "Deep Copy")

```

Shadowing

To avoid potential conflicts in the eventuality where namespaces added to Pine Script[®] in the future would collide with UDTs or object names in existing scripts; as a rule, UDTs and object names shadow the language's namespaces. For example, a UDT or object can use the name of built-in types, such as [line](#) or [table](#).

Only the language's five primitive types cannot be used to name UDTs or objects: [int](#), [float](#), [string](#), [bool](#), and [color](#).

Methods

- [Introduction](#)
- [Built-in methods](#)
- [User-defined methods](#)
- [Method overloading](#)
- [Advanced example](#)

Note

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

Introduction

Pine Script[®] methods are specialized functions associated with specific instances of built-in or user-defined [types](#). They are essentially the same as regular functions in most regards but offer a shorter, more convenient syntax. Users can access methods using dot notation on variables directly, just like accessing the fields of a Pine Script[®] [object](#).

Built-in methods

Pine Script[®] includes built-in methods for [array](#), [matrix](#), [map](#), [line](#), [linefill](#), [label](#), [box](#), and [table](#) types. These methods provide users with a more concise way to call specialized routines for these types within their scripts.

When using these special types, the expressions

```
<namespace>.<functionName>([paramName =] <objectName>, ...)
```


and

```
<objectName>.<functionName>(…)
```

are equivalent. For example, rather than using

```
array.get(id, index)
```

to get the value from an array `id` at the specified `index`, we can simply use

```
id.get(index)
```

to achieve the same effect. This notation eliminates the need for users to reference the function's namespace, as `get()` is a method of `id` in this context.

Written below is a practical example to demonstrate the usage of built-in methods in place of functions.

The following script computes Bollinger Bands over a specified number of prices sampled once every `n` bars. It calls `array.push()` and `array.shift()` to queue `sourceInput` values through the `sourceArray`, then `array.avg()` and `array.stdev()` to compute the `sampleMean` and `sampleDev`. The script then uses these values to calculate the `highBand` and `lowBand`, which it plots on the chart along with the `sampleMean`:



```
//@version=5
indicator("Custom Sample BB", overlay = true)

float sourceInput = input.source(close, "Source")
int samplesInput = input.int(20, "Samples")
int n = input.int(10, "Bars")
float multiplier = input.float(2.0, "StdDev")

var array<float> sourceArray = array.new<float>(samplesInput)
var float sampleMean = na
var float sampleDev = na

// Identify if `n` bars have passed.
if bar_index % n == 0
    // Update the queue.
    array.push(sourceArray, sourceInput)
    array.shift(sourceArray)
    // Update the mean and standard deviation values.
    sampleMean := array.avg(sourceArray)
    sampleDev := array.stdev(sourceArray) * multiplier

// Calculate bands.
float highBand = sampleMean + sampleDev
float lowBand = sampleMean - sampleDev

plot(sampleMean, "Basis", color.orange)
plot(highBand, "Upper", color.lime)
plot(lowBand, "Lower", color.red)
```

Let's rewrite this code to utilize methods rather than built-in functions. In this version, we have replaced all built-in `array.*` functions in the script with equivalent methods:

```
//@version=5
indicator("Custom Sample BB", overlay = true)
```

```

float sourceInput = input.source(close, "Source")
int samplesInput = input.int(20, "Samples")
int n = input.int(10, "Bars")
float multiplier = input.float(2.0, "StdDev")

var array<float> sourceArray = array.new<float>(samplesInput)
var float sampleMean = na
var float sampleDev = na

// Identify if `n` bars have passed.
if bar_index % n == 0
    // Update the queue.
    sourceArray.push(sourceInput)
    sourceArray.shift()
    // Update the mean and standard deviation values.
    sampleMean := sourceArray.avg()
    sampleDev := sourceArray.stdev() * multiplier

// Calculate band values.
float highBand = sampleMean + sampleDev
float lowBand = sampleMean - sampleDev

plot(sampleMean, "Basis", color.orange)
plot(highBand, "Upper", color.lime)
plot(lowBand, "Lower", color.red)

```

Note that:

- We call the array methods using `sourceArray.*` rather than referencing the [array](#) namespace.
- We do not include `sourceArray` as a parameter when we call the methods since they already reference the object.

User-defined methods

Pine Script[®] allows users to define custom methods for use with objects of any built-in or user-defined type. Defining a method is essentially the same as defining a function, but with two key differences:

- The [method](#) keyword must be included before the function name.
- The type of the first parameter in the signature must be explicitly declared, as it represents the type of object that the method will be associated with.

```
[export] method <functionName>(<paramType> <paramName> [= <defaultValue>], ...) =>
    <functionBlock>
```

Let's apply user-defined methods to our previous Bollinger Bands example to encapsulate operations from the global scope, which will simplify the code and promote reusability. See this portion from the example:

```

// Identify if `n` bars have passed.
if bar_index % n == 0
    // Update the queue.
    sourceArray.push(sourceInput)
    sourceArray.shift()
    // Update the mean and standard deviation values.
    sampleMean := sourceArray.avg()
    sampleDev := sourceArray.stdev() * multiplier

// Calculate band values.

```

```
float highBand = sampleMean + sampleDev
float lowBand  = sampleMean - sampleDev
```

We will start by defining a simple method to queue values through an array in a single call.

This `maintainQueue()` method invokes the [push\(\)](#) and [shift\(\)](#) methods on a `srcArray` when `takeSample` is true and returns the object:

```
// @function      Maintains a queue of the size of `srcArray`.
//              It appends a `value` to the array and removes its oldest
//              element at position zero.
// @param srcArray (array<float>) The array where the queue is maintained.
// @param value    (float) The new value to be added to the queue.
//              The queue's oldest value is also removed, so its size is
//              constant.
// @param takeSample (bool) A new `value` is only pushed into the queue if this
//              is true.
// @returns       (array<float>) `srcArray` object.
method maintainQueue(array<float> srcArray, float value, bool takeSample = true)
=>
    if takeSample
        srcArray.push(value)
        srcArray.shift()
    srcArray
```

Note that:

- Just as with user-defined functions, we use the `@function` [compiler annotation](#) to document method descriptions.

Now we can replace `sourceArray.push()` and `sourceArray.shift()` with `sourceArray.maintainQueue()` in our example:

```
// Identify if `n` bars have passed.
if bar_index % n == 0
    // Update the queue.
    sourceArray.maintainQueue(sourceInput)
    // Update the mean and standard deviation values.
    sampleMean := sourceArray.avg()
    sampleDev  := sourceArray.stdev() * multiplier

// Calculate band values.
float highBand = sampleMean + sampleDev
float lowBand  = sampleMean - sampleDev
```

From here, we will further simplify our code by defining a method that handles all Bollinger Band calculations within its scope.

This `calcBB()` method invokes the [avg\(\)](#) and [stdev\(\)](#) methods on a `srcArray` to update mean and dev values when `calculate` is true. The method uses these values to return a tuple containing the basis, upper band, and lower band values respectively:

```
// @function      Computes Bollinger Band values from an array of data.
// @param srcArray (array<float>) The array where the queue is maintained.
// @param multiplier (float) Standard deviation multiplier.
// @param calculate (bool) The method will only calculate new values when this
//              is true.
// @returns       A tuple containing the basis, upper band, and lower band
//              respectively.
method calcBB(array<float> srcArray, float mult, bool calculate = true) =>
    var float mean = na
    var float dev  = na
    if calculate
```

```

    // Compute the mean and standard deviation of the array.
    mean := srcArray.avg()
    dev  := srcArray.stdev() * mult
    [mean, mean + dev, mean - dev]

```

With this method, we can now remove Bollinger Band calculations from the global scope and improve code readability:

```

// Identify if `n` bars have passed.
bool newSample = bar_index % n == 0

// Update the queue and compute new BB values on each new sample.
[sampleMean, highBand, lowBand] = sourceArray.maintainQueue(sourceInput,
newSample).calcBB(multiplier, newSample)

```

Note that:

- Rather than using an `if` block in the global scope, we have defined a `newSample` variable that is only true once every `n` bars. The `maintainQueue()` and `calcBB()` methods use this value for their respective `takeSample` and `calculate` parameters.
- Since the `maintainQueue()` method returns the object that it references, we're able to call `calcBB()` from the same line of code, as both methods apply to `array<float>` instances.

Here is how the full script example looks now that we've applied our user-defined methods:

```

//@version=5
indicator("Custom Sample BB", overlay = true)

float sourceInput = input.source(close, "Source")
int samplesInput = input.int(20, "Samples")
int n = input.int(10, "Bars")
float multiplier = input.float(2.0, "StdDev")

var array<float> sourceArray = array.new<float>(samplesInput)

// @function Maintains a queue of the size of `srcArray`.
// It appends a `value` to the array and removes its oldest
// element at position zero.
// @param srcArray (array<float>) The array where the queue is maintained.
// @param value (float) The new value to be added to the queue.
// The queue's oldest value is also removed, so its size is
// constant.
// @param takeSample (bool) A new `value` is only pushed into the queue if this
// is true.
// @returns (array<float>) `srcArray` object.
method maintainQueue(array<float> srcArray, float value, bool takeSample = true)
=>
    if takeSample
        srcArray.push(value)
        srcArray.shift()
    srcArray

// @function Computes Bollinger Band values from an array of data.
// @param srcArray (array<float>) The array where the queue is maintained.
// @param multiplier (float) Standard deviation multiplier.
// @param calculate (bool) The method will only calculate new values when this
// is true.
// @returns A tuple containing the basis, upper band, and lower band
// respectively.
method calcBB(array<float> srcArray, float mult, bool calculate = true) =>

```

```

var float mean = na
var float dev = na
if calculate
    // Compute the mean and standard deviation of the array.
    mean := srcArray.avg()
    dev := srcArray.stdev() * mult
    [mean, mean + dev, mean - dev]

// Identify if `n` bars have passed.
bool newSample = bar_index % n == 0

// Update the queue and compute new BB values on each new sample.
[sampleMean, highBand, lowBand] = sourceArray.maintainQueue(sourceInput,
newSample).calcBB(multiplier, newSample)

plot(sampleMean, "Basis", color.orange)
plot(highBand, "Upper", color.lime)
plot(lowBand, "Lower", color.red)

```

Method overloading

User-defined methods can override and overload existing built-in and user-defined methods with the same identifier. This capability allows users to define multiple routines associated with different parameter signatures under the same method name.

As a simple example, suppose we want to define a method to identify a variable's type. Since we must explicitly specify the type of object associated with a user-defined method, we will need to define overloads for each type that we want it to recognize.

Below, we have defined a `getType()` method that returns a string representation of a variable's type with overloads for the five primitive types:

```

// @function Identifies an object's type.
// @param this Object to inspect.
// @returns (string) A string representation of the type.
method getType(int this) =>
    na(this) ? "int(na)" : "int"

method getType(float this) =>
    na(this) ? "float(na)" : "float"

method getType(bool this) =>
    na(this) ? "bool(na)" : "bool"

method getType(color this) =>
    na(this) ? "color(na)" : "color"

method getType(string this) =>
    na(this) ? "string(na)" : "string"

```

Now we can use these overloads to inspect some variables. This script uses `str.format()` to format the results from calling the `getType()` method on five different variables into a single results string, then displays the string in the `lbl` label using the built-in `set_text()` method:



```

//@version=5
indicator("Type Inspection")

// @function Identifies an object's type.

```

```

// @param this Object to inspect.
// @returns (string) A string representation of the type.
method getType(int this) =>
    na(this) ? "int(na)" : "int"

method getType(float this) =>
    na(this) ? "float(na)" : "float"

method getType(bool this) =>
    na(this) ? "bool(na)" : "bool"

method getType(color this) =>
    na(this) ? "color(na)" : "color"

method getType(string this) =>
    na(this) ? "string(na)" : "string"

a = 1
b = 1.0
c = true
d = color.white
e = "1"

// Inspect variables and format results.
results = str.format(
    "a: {0}\nb: {1}\nc: {2}\nd: {3}\ne: {4}",
    a.getType(), b.getType(), c.getType(), d.getType(), e.getType()
)

var label lbl = label.new(0, 0)
lbl.set_x(bar_index)
lbl.set_text(results)

```

Note that:

- The underlying type of each variable determines which overload of `getType()` the compiler will use.
- The method will append “(na)” to the output string when a variable is `na` to demarcate that it is empty.

Advanced example

Let’s apply what we’ve learned to construct a script that estimates the cumulative distribution of elements in an array, meaning the fraction of elements in the array that are less than or equal to any given value.

There are many ways in which we could choose to tackle this objective. For this example, we will start by defining a method to replace elements of an array, which will help us count the occurrences of elements within a range of values.

Written below is an overload of the built-in [fill\(\)](#) method for `array<float>` instances. This overload replaces elements in a `srcArray` within the range between the `lowerBound` and `upperBound` with an `innerValue`, and replaces all elements outside the range with an `outerValue`:

```

// @function          Replaces elements in a `srcArray` between `lowerBound` and
`upperBound` with an `innerValue`,
//                  and replaces elements outside the range with an
`outerValue`.
// @param srcArray   (array<float>) Array to modify.

```

```

// @param innerValue (float) Value to replace elements within the range with.
// @param outerValue (float) Value to replace elements outside the range with.
// @param lowerBound (float) Lowest value to replace with `innerValue`.
// @param upperBound (float) Highest value to replace with `innerValue`.
// @returns (array<float>) `srcArray` object.
method fill(array<float> srcArray, float innerValue, float outerValue, float
lowerBound, float upperBound) =>
  for [i, element] in srcArray
    if (element >= lowerBound or na(lowerBound)) and (element <= upperBound
or na(upperBound))
      srcArray.set(i, innerValue)
    else
      srcArray.set(i, outerValue)
  srcArray

```

With this method, we can filter an array by value ranges to produce an array of occurrences. For example, the expression

```
srcArray.copy().fill(1.0, 0.0, min, val)
```

copies the `srcArray` object, replaces all elements between `min` and `val` with `1.0`, then replaces all elements above `val` with `0.0`. From here, it's easy to estimate the output of the cumulative distribution function at the `val`, as it's simply the average of the resulting array:

```
srcArray.copy().fill(1.0, 0.0, min, val).avg()
```

Note that:

- The compiler will only use this `fill()` overload instead of the built-in when the user provides `innerValue`, `outerValue`, `lowerBound`, and `upperBound` arguments in the call.
- If either `lowerBound` or `upperBound` is `na`, its value is ignored while filtering the fill range.
- We are able to call `copy()`, `fill()`, and `avg()` successively on the same line of code because the first two methods return an `array<float>` instance.

We can now use this to define a method that will calculate our empirical distribution values. The following `eCDF()` method estimates a number of evenly spaced ascending steps from the cumulative distribution function of a `srcArray` and pushes the results into a `cdfArray`:

```

// @function Estimates the empirical CDF of a `srcArray`.
// @param srcArray (array<float>) Array to calculate on.
// @param steps (int) Number of steps in the estimation.
// @returns (array<float>) Array of estimated CDF ratios.
method eCDF(array<float> srcArray, int steps) =>
  float min = srcArray.min()
  float rng = srcArray.range() / steps
  array<float> cdfArray = array.new<float>()
  // Add averages of `srcArray` filtered by value region to the `cdfArray`.
  float val = min
  for i = 1 to steps
    val += rng
    cdfArray.push(srcArray.copy().fill(1.0, 0.0, min, val).avg())
  cdfArray

```

Lastly, to ensure that our `eCDF()` method functions properly for arrays containing small and large values, we will define a method to normalize our arrays.

This `featureScale()` method uses `array min()` and `array range()` methods to produce a rescaled copy of a `srcArray`. We will use this to normalize our arrays prior to invoking the `eCDF()` method:

```

// @function      Rescales the elements within a `srcArray` to the interval
// [0, 1].
// @param srcArray (array<float>) Array to normalize.
// @returns       (array<float>) Normalized copy of the `srcArray`.
method featureScale(array<float> srcArray) =>
    float min = srcArray.min()
    float rng = srcArray.range()
    array<float> scaledArray = array.new<float>()
    // Push normalized `element` values into the `scaledArray`.
    for element in srcArray
        scaledArray.push((element - min) / rng)
    scaledArray

```

Note that:

- This method does not include special handling for divide by zero conditions. If `rng` is 0, the value of the array element will be `na`.

The full example below queues a `sourceArray` of size `length` with `sourceInput` values using our previous `maintainQueue()` method, normalizes the array's elements using the `featureScale()` method, then calls the `eCDF()` method to get an array of estimates for `n` evenly spaced steps on the distribution. The script then calls a user-defined `makeLabel()` function to display the estimates and prices in a label on the right side of the chart:



```

//@version=5
indicator("Empirical Distribution", overlay = true)

float sourceInput = input.source(close, "Source")
int length       = input.int(20, "Length")
int n            = input.int(20, "Steps")

// @function      Maintains a queue of the size of `srcArray`.
//                It appends a `value` to the array and removes its oldest
//                element at position zero.
// @param srcArray (array<float>) The array where the queue is maintained.
// @param value    (float) The new value to be added to the queue.
//                The queue's oldest value is also removed, so its size is
//                constant.
// @param takeSample (bool) A new `value` is only pushed into the queue if this
//                is true.
// @returns       (array<float>) `srcArray` object.
method maintainQueue(array<float> srcArray, float value, bool takeSample = true)
=>
    if takeSample
        srcArray.push(value)
        srcArray.shift()
    srcArray

// @function      Replaces elements in a `srcArray` between `lowerBound` and
// `upperBound` with an `innerValue`,
//                and replaces elements outside the range with an
// `outerValue`.
// @param srcArray (array<float>) Array to modify.
// @param innerValue (float) Value to replace elements within the range with.
// @param outerValue (float) Value to replace elements outside the range with.
// @param lowerBound (float) Lowest value to replace with `innerValue`.
// @param upperBound (float) Highest value to replace with `innerValue`.
// @returns       (array<float>) `srcArray` object.
method fill(array<float> srcArray, float innerValue, float outerValue, float

```



```

lowerBound, float upperBound) =>
    for [i, element] in srcArray
        if (element >= lowerBound or na(lowerBound)) and (element <= upperBound
or na(upperBound))
            srcArray.set(i, innerValue)
        else
            srcArray.set(i, outerValue)
    srcArray

// @function      Estimates the empirical CDF of a `srcArray`.
// @param srcArray (array<float>) Array to calculate on.
// @param steps    (int) Number of steps in the estimation.
// @returns        (array<float>) Array of estimated CDF ratios.
method eCDF(array<float> srcArray, int steps) =>
    float min = srcArray.min()
    float rng = srcArray.range() / steps
    array<float> cdfArray = array.new<float>()
    // Add averages of `srcArray` filtered by value region to the `cdfArray`.
    float val = min
    for i = 1 to steps
        val += rng
        cdfArray.push(srcArray.copy().fill(1.0, 0.0, min, val).avg())
    cdfArray

// @function      Rescales the elements within a `srcArray` to the interval
// [0, 1].
// @param srcArray (array<float>) Array to normalize.
// @returns        (array<float>) Normalized copy of the `srcArray`.
method featureScale(array<float> srcArray) =>
    float min = srcArray.min()
    float rng = srcArray.range()
    array<float> scaledArray = array.new<float>()
    // Push normalized `element` values into the `scaledArray`.
    for element in srcArray
        scaledArray.push((element - min) / rng)
    scaledArray

// @function      Draws a label containing eCDF estimates in the format
// "{price}: {percent}%"
// @param srcArray (array<float>) Array of source values.
// @param cdfArray (array<float>) Array of CDF estimates.
// @returns        (void)
makeLabel(array<float> srcArray, array<float> cdfArray) =>
    float max      = srcArray.max()
    float rng      = srcArray.range() / cdfArray.size()
    string results = ""
    var label lbl = label.new(0, 0, "", style = label.style_label_left,
text_font_family = font.family_monospace)
    // Add percentage strings to `results` starting from the `max`.
    cdfArray.reverse()
    for [i, element] in cdfArray
        results += str.format("{0}: {1}%\n", max - i * rng, element * 100)
    // Update `lbl` attributes.
    lbl.set_xy(bar_index + 1, srcArray.avg())
    lbl.set_text(results)

var array<float> sourceArray = array.new<float>(length)

// Add background color for the last `length` bars.
bgcolor(bar_index > last_bar_index - length ? color.new(color.orange, 80) : na)

// Queue `sourceArray`, feature scale, then estimate the distribution over `n`
steps.
array<float> distArray =

```

```
sourceArray.maintainQueue(sourceInput).featureScale().eCDF(n)
// Draw label.
makeLabel(sourceArray, distArray)
```

Arrays

- [Introduction](#)
- [Declaring arrays](#)
 - [Using `var` and `varip` keywords](#)
- [Reading and writing array elements](#)
- [Looping through array elements](#)
- [Scope](#)
- [History referencing](#)
- [Inserting and removing array elements](#)
 - [Inserting](#)
 - [Removing](#)
 - [Using an array as a stack](#)
 - [Using an array as a queue](#)
- [Calculations on arrays](#)
- [Manipulating arrays](#)
 - [Concatenation](#)
 - [Copying](#)
 - [Joining](#)
 - [Sorting](#)
 - [Reversing](#)
 - [Slicing](#)
- [Searching arrays](#)
- [Error handling](#)
 - [Index xx is out of bounds. Array size is yy](#)
 - [Cannot call array methods when ID of array is 'na'](#)
 - [Array is too large. Maximum size is 100000](#)
 - [Cannot create an array with a negative size](#)
 - [Cannot use *shift\(\)* if array is empty.](#)
 - [Cannot use *pop\(\)* if array is empty.](#)
 - [Index 'from' should be less than index 'to'](#)
 - [Slice is out of bounds of the parent array](#)

Note

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

[Introduction](#)

Pine Script[®] Arrays are one-dimensional collections that can hold multiple value references. Think of them as a better way to handle cases where one would otherwise need to explicitly declare a set of similar variables (e.g., `price00`, `price01`, `price02`, ...).

All elements within an array must be of the same type, which can be a [built-in](#) or a [user-defined](#)

[type](#), always of “series” form. Scripts reference arrays using an array ID similar to the IDs of lines, labels, and other special types. Pine Script[®] does not use an indexing operator to reference individual array elements. Instead, functions including [array.get\(\)](#) and [array.set\(\)](#) read and write the values of array elements. We can use array values in expressions and functions that allow values of the “series” form.

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

Note

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

Declaring arrays

Pine Script[®] uses the following syntax to declare arrays:

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

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

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

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

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

```
array<float> prices = na
```

We can also write the above example in this form:

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

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

```
prices = array.new<float>(0)
```

Note

The `array.*` namespace also contains type-specific functions for creating arrays, including [array.new_int\(\)](#), [array.new_float\(\)](#), [array.new_bool\(\)](#), [array.new_color\(\)](#), [array.new_string\(\)](#), [array.new_line\(\)](#), [array.new_linefill\(\)](#), [array.new_label\(\)](#), [array.new_box\(\)](#) and [array.new_table\(\)](#).

The [array.new<type>\(\)](#) function can create an array of any type, including [user-defined types](#).

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

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

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

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

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

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

Using ``var`` and ``varip`` keywords

Users can utilize [var](#) and [varip](#) keywords to instruct a script to declare an array variable only once on the first iteration of the script on the first chart bar. Array variables declared using these keywords point to the same array instances until explicitly reassigned, allowing an array and its element references to persist across bars.

When declaring an array variable using these keywords and pushing a new value to the end of the referenced array on each bar, the array will grow by one on each bar and be of size `bar_index + 1` ([bar_index](#) starts at zero) by the time the script executes on the last bar, as this code demonstrates:

```
//@version=5
indicator("Using `var`")
//@variable An array that expands its size by 1 on each bar.
var a = array.new<float>(0)
array.push(a, close)

if barstate.islast
    //@variable A string containing the size of `a` and the current `bar_index`
    value.
    string labelText = "Array size: " + str.toString(a.size()) + "\nbar_index: "
+ str.toString(bar_index)
    // Display the `labelText`.
    label.new(bar_index, 0, labelText, size = size.large)
```

The same code without the [var](#) keyword would re-declare the array on each bar. In this case, after execution of the [array.push\(\)](#) call, the [a.size\(\)](#) call would return a value of 1.

Note

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

Reading and writing array elements

Scripts can write values to existing individual array elements using [array.set\(id, index, value\)](#), and read using [array.get\(id, index\)](#). When using these functions, it is imperative that the `index` in the function call is always less than or equal to the array's size (because array indices start at zero). To get the size of an array, use the [array.size\(id\)](#) function.

The following example uses the [set\(\)](#) method to populate a `fillColors` array with instances of one base color using different transparency levels. It then uses [array.get\(\)](#) to retrieve one of the colors from the array based on the location of the bar with the highest price within the last `lookbackInput` bars:



```
//@version=5
indicator("Distance from high", "", true)
lookbackInput = input.int(100)
FILL_COLOR = color.green
// Declare array and set its values on the first bar only.
var fillColors = array.new<color>(5)
if barstate.isfirst
    // Initialize the array elements with progressively lighter shades of the
    fill color.
    fillColors.set(0, color.new(FILL_COLOR, 70))
    fillColors.set(1, color.new(FILL_COLOR, 75))
    fillColors.set(2, color.new(FILL_COLOR, 80))
    fillColors.set(3, color.new(FILL_COLOR, 85))
    fillColors.set(4, color.new(FILL_COLOR, 90))

// Find the offset to highest high. Change its sign because the function returns
a negative value.
lastHiBar = - ta.highestbars(high, lookbackInput)
// Convert the offset to an array index, capping it to 4 to avoid a runtime
error.
// The index used by `array.get()` will be the equivalent of `floor(fillNo)`.
fillNo = math.min(lastHiBar / (lookbackInput / 5), 4)
// Set background to a progressively lighter fill with increasing distance from
location of highest high.
bgcolor(array.get(fillColors, fillNo))
// Plot key values to the Data Window for debugging.
plotchar(lastHiBar, "lastHiBar", "", location.top, size = size.tiny)
plotchar(fillNo, "fillNo", "", location.top, size = size.tiny)
```

Another technique for initializing the elements in an array is to create an *empty array* (an array with no elements), then use [array.push\(\)](#) to append **new** elements to the end of the array, increasing the size of the array by one on each call. The following code is functionally identical to the initialization section from the preceding script:

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

This code is equivalent to the one above, but it uses [array.unshift\(\)](#) to insert new elements at the

beginning of the fillColors array:

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

We can also use [array.from\(\)](#) to create the same fillColors array with a single function call:

```
//@version=5
indicator("Using `var`")
FILL_COLOR = color.green
var color[] fillColors = array.from(
    color.new(FILL_COLOR, 70),
    color.new(FILL_COLOR, 75),
    color.new(FILL_COLOR, 80),
    color.new(FILL_COLOR, 85),
    color.new(FILL_COLOR, 90)
)
// Cycle background through the array's colors.
bgcolor(array.get(fillColors, bar_index % (fillColors.size())))
```

The [array.fill\(id, value, index_from, index_to\)](#) function points all array elements, or the elements within the index_from to index_to range, to a specified value. Without the last two optional parameters, the function fills the whole array, so:

```
a = array.new<float>(10, close)
```

and:

```
a = array.new<float>(10)
a.fill(close)
```

are equivalent, but:

```
a = array.new<float>(10)
a.fill(close, 1, 3)
```

only fills the second and third elements (at index 1 and 2) of the array with close. Note how [array.fill\(\)](#)'s last parameter, index_to, must be one greater than the last index the function will fill. The remaining elements will hold na values, as the [array.new\(\)](#) function call does not contain an initial_value argument.

Looping through array elements

When looping through an array's element indices and the array's size is unknown, one can use the [array.size\(\)](#) function to get the maximum index value. For example:

```
//@version=5
indicator("Protected `for` loop", overlay = true)
//@variable An array of `close` prices from the 1-minute timeframe.
array<float> a = request.security_lower_tf(syminfo.tickerid, "1", close)

//@variable A string representation of the elements in `a`.
string labelText = ""
```

```

for i = 0 to (array.size(a) == 0 ? na : array.size(a) - 1)
    labelText += str.toString(array.get(a, i)) + "\n"

label.new(bar_index, high, text = labelText)

```

Note that:

- We use the [request.security_lower_tf\(\)](#) function which returns an array of [close](#) prices at the 1 minute timeframe.
- This code example will throw an error if you use it on a chart timeframe smaller than 1 minute.
- [for](#) loops do not execute if the `to` expression is [na](#). Note that the `to` value is only evaluated once upon entry.

An alternative method to loop through an array is to use a [for...in](#) loop. This approach is a variation of the standard `for` loop that can iterate over the value references and indices in an array. Here is an example of how we can write the code example from above using a `for...in` loop:

```

//@version=5
indicator("`for...in` loop", overlay = true)
//@variable An array of `close` prices from the 1-minute timeframe.
array<float> a = request.security_lower_tf(syminfo.tickerid, "1", close)

//@variable A string representation of the elements in `a`.
string labelText = ""
for price in a
    labelText += str.toString(price) + "\n"

label.new(bar_index, high, text = labelText)

```

Note that:

- [for...in](#) loops can return a tuple containing each index and corresponding element. For example, `for [i, price] in a` returns the `i` index and `price` value for each element in `a`.

A [while](#) loop statement can also be used:

```

//@version=5
indicator("`while` loop", overlay = true)
array<float> a = request.security_lower_tf(syminfo.tickerid, "1", close)

string labelText = ""
int i = 0
while i < array.size(a)
    labelText += str.toString(array.get(a, i)) + "\n"
    i += 1

label.new(bar_index, high, text = labelText)

```

Scope

Users can declare arrays within the global scope of a script, as well as the local scopes of [functions](#), [methods](#), and [conditional structures](#). Unlike some of the other built-in types, namely *fundamental* types, scripts can modify globally-assigned arrays from within local scopes, allowing users to implement global variables that any function in the script can directly interact with. We use the functionality here to calculate progressively lower or higher price levels:



```

//@version=5
indicator("Bands", "", true)
//@variable The distance ratio between plotted price levels.
factorInput = 1 + (input.float(-2., "Step %") / 100)
//@variable A single-value array holding the lowest `ohlcv4` value within a 50
bar window from 10 bars back.
level = array.new<float>(1, ta.lowest(ohlcv4, 50)[10])

nextLevel(val) =>
    newLevel = level.get(0) * val
    // Write new level to the global `level` array so we can use it as the
base in the next function call.
    level.set(0, newLevel)
    newLevel

plot(nextLevel(1))
plot(nextLevel(factorInput))
plot(nextLevel(factorInput))
plot(nextLevel(factorInput))

```

History referencing

Pine Script®'s history-referencing operator `[]` can access the history of array variables, allowing scripts to interact with past array instances previously assigned to a variable.

To illustrate this, let's create a simple example to show how one can fetch the previous bar's `close` value in two equivalent ways. This script uses the `[]` operator to get the array instance assigned to `a` on the previous bar, then uses the `get()` method to retrieve the value of the first element (`previousClose1`). For `previousClose2`, we use the history-referencing operator on the `close` variable directly to retrieve the value. As we see from the plots, `previousClose1` and `previousClose2` both return the same value:



```

//@version=5
indicator("History referencing")

//@variable A single-value array declared on each bar.
a = array.new<float>(1)
// Set the value of the only element in `a` to `close`.
array.set(a, 0, close)

//@variable The array instance assigned to `a` on the previous bar.
previous = a[1]

previousClose1 = na(previous) ? na : previous.get(0)
previousClose2 = close[1]

plot(previousClose1, "previousClose1", color.gray, 6)
plot(previousClose2, "previousClose2", color.white, 2)

```

Inserting and removing array elements

Inserting

The following three functions can insert new elements into an array.

[array.unshift\(\)](#) inserts a new element at the beginning of an array (index 0) and increases the index values of any existing elements by one.

[array.insert\(\)](#) inserts a new element at the specified `index` and increases the index of existing elements at or after the `index` by one.



```
//@version=5
indicator("`array.insert()`")
a = array.new<float>(5, 0)
for i = 0 to 4
    array.set(a, i, i + 1)
if barstate.islast
    label.new(bar_index, 0, "BEFORE\na: " + str.tostring(a), size = size.large)
    array.insert(a, 2, 999)
    label.new(bar_index, 0, "AFTER\na: " + str.tostring(a), style =
label.style_label_up, size = size.large)
```

[array.push\(\)](#) adds a new element at the end of an array.

Removing

These four functions remove elements from an array. The first three also return the value of the removed element.

[array.remove\(\)](#) removes the element at the specified `index` and returns that element's value.

[array.shift\(\)](#) removes the first element from an array and returns its value.

[array.pop\(\)](#) removes the last element of an array and returns its value.

[array.clear\(\)](#) removes all elements from an array. Note that clearing an array won't delete any objects its elements referenced. See the example below that illustrates how this works:

```
//@version=5
indicator("`array.clear()` example", overlay = true)

// Create a label array and add a label to the array on each new bar.
var a = array.new<label>()
label lbl = label.new(bar_index, high, "Text", color = color.red)
array.push(a, lbl)

var table t = table.new(position.top_right, 1, 1)
// Clear the array on the last bar. This doesn't remove the labels from the
chart.
if barstate.islast
    array.clear(a)
    table.cell(t, 0, 0, "Array elements count: " + str.tostring(array.size(a)),
bgcolor = color.yellow)
```

Using an array as a stack

Stacks are LIFO (last in, first out) constructions. They behave somewhat like a vertical pile of books to which books can only be added or removed one at a time, always from the top. Pine Script[®] arrays can be used as a stack, in which case we use the [array.push\(\)](#) and [array.pop\(\)](#)

functions to add and remove elements at the end of the array.

`array.push(prices, close)` will add a new element to the end of the `prices` array, increasing the array's size by one.

`array.pop(prices)` will remove the end element from the `prices` array, return its value and decrease the array's size by one.

See how the functions are used here to track successive lows in rallies:



```
//@version=5
indicator("Lows from new highs", "", true)
var lows = array.new<float>(0)
flushLows = false

// Remove last element from the stack when `_cond` is true.
array_pop(id, cond) => cond and array.size(id) > 0 ? array.pop(id) : float(na)

if ta.rising(high, 1)
    // Rising highs; push a new low on the stack.
    lows.push(low)
    // Force the return type of this `if` block to be the same as that of the
    next block.
    bool(na)
else if lows.size() >= 4 or low < array.min(lows)
    // We have at least 4 lows or price has breached the lowest low;
    // sort lows and set flag indicating we will plot and flush the levels.
    array.sort(lows, order.ascending)
    flushLows := true

// If needed, plot and flush lows.
lowLevel = array_pop(lows, flushLows)
plot(lowLevel, "Low 1", low > lowLevel ? color.silver : color.purple, 2,
plot.style_linebr)
lowLevel := array_pop(lows, flushLows)
plot(lowLevel, "Low 2", low > lowLevel ? color.silver : color.purple, 3,
plot.style_linebr)
lowLevel := array_pop(lows, flushLows)
plot(lowLevel, "Low 3", low > lowLevel ? color.silver : color.purple, 4,
plot.style_linebr)
lowLevel := array_pop(lows, flushLows)
plot(lowLevel, "Low 4", low > lowLevel ? color.silver : color.purple, 5,
plot.style_linebr)

if flushLows
    // Clear remaining levels after the last 4 have been plotted.
    lows.clear()
```

Using an array as a queue

Queues are FIFO (first in, first out) constructions. They behave somewhat like cars arriving at a red light. New cars are queued at the end of the line, and the first car to leave will be the first one that arrived to the red light.

In the following code example, we let users decide through the script's inputs how many labels they want to have on their chart. We use that quantity to determine the size of the array of labels we then create, initializing the array's elements to `na`.

When a new pivot is detected, we create a label for it, saving the label's ID in the `pLabel` variable.

We then queue the ID of that label by using [array.push\(\)](#) to append the new label's ID to the end of the array, making our array size one greater than the maximum number of labels to keep on the chart.

Lastly, we de-queue the oldest label by removing the array's first element using [array.shift\(\)](#) and deleting the label referenced by that array element's value. As we have now de-queued an element from our queue, the array contains `pivotCountInput` elements once again. Note that on the dataset's first bars we will be deleting `na` label IDs until the maximum number of labels has been created, but this does not cause runtime errors. Let's look at our code:



```
//@version=5
MAX_LABELS = 100
indicator("Show Last n High Pivots", "", true, max_labels_count = MAX_LABELS)

pivotCountInput = input.int(5, "How many pivots to show", minval = 0, maxval =
MAX_LABELS)
pivotLegsInput  = input.int(3, "Pivot legs", minval = 1, maxval = 5)

// Create an array containing the user-selected max count of label IDs.
var labelIds = array.new<label>(pivotCountInput)

pHi = ta.pivohigh(pivotLegsInput, pivotLegsInput)
if not na(pHi)
    // New pivot found; plot its label `i_pivotLegs` bars back.
    pLabel = label.new(bar_index[pivotLegsInput], pHi, str.tostring(pHi,
format.mintick), textcolor = color.white)
    // Queue the new label's ID by appending it to the end of the array.
    array.push(labelIds, pLabel)
    // De-queue the oldest label ID from the queue and delete the corresponding
label.
    label.delete(array.shift(labelIds))
```

Calculations on arrays

While series variables can be viewed as a horizontal set of values stretching back in time, Pine Script®'s one-dimensional arrays can be viewed as vertical structures residing on each bar. As an array's set of elements is not a time series, Pine Script®'s usual mathematical functions are not allowed on them. Special-purpose functions must be used to operate on all of an array's values. The available functions are: [array.abs\(\)](#), [array.avg\(\)](#), [array.covariance\(\)](#), [array.min\(\)](#), [array.max\(\)](#), [array.median\(\)](#), [array.mode\(\)](#), [array.percentile_linear_interpolation\(\)](#), [array.percentile_nearest_rank\(\)](#), [array.percentrank\(\)](#), [array.range\(\)](#), [array.standardize\(\)](#), [array.stdev\(\)](#), [array.sum\(\)](#), [array.variance\(\)](#).

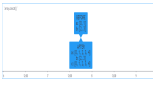
Note that contrary to the usual mathematical functions in Pine Script®, those used on arrays do not return `na` when some of the values they calculate on have `na` values. There are a few exceptions to this rule:

- When all array elements have `na` value or the array contains no elements, `na` is returned. `array.standardize()` however, will return an empty array.
- `array.mode()` will return `na` when no mode is found.

Manipulating arrays

Concatenation

Two arrays can be merged—or concatenated—using [array.concat\(\)](#). When arrays are concatenated, the second array is appended to the end of the first, so the first array is modified while the second one remains intact. The function returns the array ID of the first array:



```
//@version=5
indicator("`array.concat()`")
a = array.new<float>(0)
b = array.new<float>(0)
array.push(a, 0)
array.push(a, 1)
array.push(b, 2)
array.push(b, 3)
if barstate.islast
    label.new(bar_index, 0, "BEFORE\na: " + str.tostring(a) + "\nb: " +
str.tostring(b), size = size.large)
    c = array.concat(a, b)
    array.push(c, 4)
    label.new(bar_index, 0, "AFTER\na: " + str.tostring(a) + "\nb: " +
str.tostring(b) + "\nc: " + str.tostring(c), style = label.style_label_up, size
= size.large)
```

Copying

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



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

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

Joining

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

```
//@version=5
indicator("")
v1 = array.new<string>(10, "test")
v2 = array.new<string>(10, "test")
array.push(v2, "test1")
v3 = array.new_float(5, 5)
v4 = array.new_int(5, 5)
```

```

11 = label.new(bar_index, close, array.join(v1))
12 = label.new(bar_index, close, array.join(v2, ","))
13 = label.new(bar_index, close, array.join(v3, ","))
14 = label.new(bar_index, close, array.join(v4, ","))

```

Sorting

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



```

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

```

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

Reversing

Use [array.reverse\(\)](#) to reverse an array:

```

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

```

Slicing

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

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

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



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

Searching arrays

We can test if a value is part of an array with the [array.includes\(\)](#) function, which returns true if the element is found. We can find the first occurrence of a value in an array by using the [array.indexof\(\)](#) function. The first occurrence is the one with the lowest index. We can also find the last occurrence of a value with [array.lastindexof\(\)](#):

```
//@version=5
indicator("Searching in arrays")
valueInput = input.int(1)
a = array.new<float>(0)
array.push(a, 0)
array.push(a, 1)
array.push(a, 2)
array.push(a, 1)
if barstate.islast
    valueFound = array.includes(a, valueInput)
    firstIndexFound = array.indexof(a, valueInput)
    lastIndexFound = array.lastindexof(a, valueInput)
    label.new(bar_index, 0, "a: " + str.tostring(a) +
"\nFirst " + str.tostring(valueInput) + (firstIndexFound != -1 ? " value
was found at index: " + str.tostring(firstIndexFound) : " value was not found.")
+
"\nLast " + str.tostring(valueInput) + (lastIndexFound != -1 ? " value
was found at index: " + str.tostring(lastIndexFound) : " value was not found."))
```

We can also perform a binary search on an array but note that performing a binary search on an array means that the array will first need to be sorted in ascending order only. The [array.binary_search\(\)](#) function will return the value's index if it was found or -1 if it wasn't. If we want to always return an existing index from the array even if our chosen value wasn't found, then we can use one of the other binary search functions available. The [array.binary_search_leftmost\(\)](#) function, which returns an index if the value was found or the first index to the left where the value would be found. The [array.binary_search_rightmost\(\)](#) function is almost identical and returns an index if the value was found or the first index to the right where the value would be found.

Error handling

Malformed `array.*()` call syntax in Pine scripts will cause the usual **compiler** error messages to appear in Pine Script® Editor's console, at the bottom of the window, when you save a script. Refer to the Pine Script® [v5 Reference Manual](#) when in doubt regarding the exact syntax of function calls.

Scripts using arrays can also throw **runtime** errors, which appear as an exclamation mark next to the indicator's name on the chart. We discuss those runtime errors in this section.

Index xx is out of bounds. Array size is yy

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

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

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

The correct `for` statement is:

```
for i = 0 to 2
```

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

```
//@version=5
indicator("Protected `for` loop")
sizeInput = input.int(0, "Array size", minval = 0, maxval = 100000)
a = array.new<float>(sizeInput)
for i = 0 to (array.size(a) == 0 ? na : array.size(a) - 1)
    array.set(a, i, i)
plot(array.pop(a))
```

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

```
//@version=5
indicator("Protected array size")
sizeInput = input.int(10, "Array size", minval = 1, maxval = 100000)
a = array.new<float>(sizeInput)
for i = 0 to sizeInput - 1
```

```
array.set(a, i, i)
plot(array.size(a))
```

See the [Looping](#) section of this page for more information.

Cannot call array methods when ID of array is 'na'

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

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

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

```
int[] a = array.new_int(0)
```

or:

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

Array is too large. Maximum size is 100000

This error will appear if your code attempts to declare an array with a size greater than 100,000. It will also occur if, while dynamically appending elements to an array, a new element would increase the array's size past the maximum.

Cannot create an array with a negative size

We haven't found any use for arrays of negative size yet, but if you ever do, we may allow them :)

Cannot use *shift()* if array is empty.

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

Cannot use *pop()* if array is empty.

This error will occur if [array.pop\(\)](#) is called to remove the last element of an empty array.

Index 'from' should be less than index 'to'

When two indices are used in functions such as [array.slice\(\)](#), the first index must always be smaller than the second one.

Slice is out of bounds of the parent array

This message occurs whenever the parent array's size is modified in such a way that it makes the shallow copy created by a slice point outside the boundaries of the parent array. This code will reproduce it because after creating a slice from index 3 to 4 (the last two elements of our five-

element parent array), we remove the parent's first element, making its size four and its last index 3. From that moment on, the shallow copy which is still pointing to the "window" at the parent array's indices 3 to 4, is pointing out of the parent array's boundaries:

```
//@version=5
indicator("Slice out of bounds")
a = array.new<float>(5, 0)
b = array.slice(a, 3, 5)
array.remove(a, 0)
c = array.indexOf(b, 2)
plot(c)
```

Matrices

- [Introduction](#)
- [Declaring a matrix](#)
 - [Using `var` and `varip` keywords](#)
- [Reading and writing matrix elements](#)
 - [`matrix.get\(\)` and `matrix.set\(\)`](#)
 - [`matrix.fill\(\)`](#)
- [Rows and columns](#)
 - [Retrieving](#)
 - [Inserting](#)
 - [Removing](#)
 - [Swapping](#)
 - [Replacing](#)
- [Looping through a matrix](#)
 - [`for`](#)
 - [`for...in`](#)
- [Copying a matrix](#)
 - [Shallow copies](#)
 - [Deep copies](#)
 - [Submatrices](#)
- [Scope and history](#)
- [Inspecting a matrix](#)
- [Manipulating a matrix](#)
 - [Reshaping](#)
 - [Reversing](#)
 - [Transposing](#)
 - [Sorting](#)
 - [Concatenating](#)
- [Matrix calculations](#)
 - [Element-wise calculations](#)
 - [Special calculations](#)
 - [`matrix.sum\(\)` and `matrix.diff\(\)`](#)
 - [`matrix.mult\(\)`](#)
 - [`matrix.det\(\)`](#)
 - [`matrix.inv\(\)` and `matrix.pinv\(\)`](#)
 - [`matrix.rank\(\)`](#)
- [Error handling](#)

- [The row/column index \(xx\) is out of bounds, row/column size is \(yy\).](#)
- [The array size does not match the number of rows/columns in the matrix.](#)
- [Cannot call matrix methods when the ID of matrix is 'na'.](#)
- [Matrix is too large. Maximum size of the matrix is 100,000 elements.](#)
- [The row/column index must be 0 <= from_row/column < to_row/column.](#)
- [Matrices 'id1' and 'id2' must have an equal number of rows and columns to be added.](#)
- [The number of columns in the 'id1' matrix must equal the number of rows in the matrix \(or the number of elements in the array\) 'id2'.](#)
- [Operation not available for non-square matrices.](#)

Note

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

Introduction

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

Matrices reference their elements using two indices: one index for their rows and the other for their columns. Each index starts at 0 and extends to the number of rows/columns in the matrix minus one. Matrices in Pine can have dynamic numbers of rows and columns that vary across bars. The [total number of elements](#) within a matrix is the product of the number of [rows](#) and [columns](#) (e.g., a 5x5 matrix has a total of 25). Like [arrays](#), the total number of elements in a matrix cannot exceed 100,000.

Declaring a matrix

Pine Script[®] uses the following syntax for matrix declaration:

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

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

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

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

```
matrix<float> myMatrix = na
```

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

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

```
myMatrix = matrix.new<float>(2, 2, 0.0)
```

Using `var` and `varip` keywords

As with other variables, users can include the [var](#) or [varip](#) keywords to instruct a script to declare a matrix variable only once rather than on every bar. A matrix variable declared with this keyword will point to the same instance throughout the span of the chart unless the script explicitly assigns another matrix to it, allowing a matrix and its element references to persist between script iterations.

This script declares an `m` variable assigned to a matrix that holds a single row of two [int](#) elements using the [var](#) keyword. On every 20th bar, the script adds 1 to the first element on the first row of the `m` matrix. The [plot\(\)](#) call displays this element on the chart. As we see from the plot, the value of [m.get\(0, 0\)](#) persists between bars, never returning to the initial value of 0:



```
//@version=5
indicator("var matrix demo")

//@variable A 1x2 rectangular matrix declared only at `bar_index == 0`, i.e.,
the first bar.
var m = matrix.new<int>(1, 2, 0)

//@variable Is `true` on every 20th bar.
bool update = bar_index % 20 == 0

if update
    int currentValue = m.get(0, 0) // Get the current value of the first row and
column.
    m.set(0, 0, currentValue + 1) // Set the first row and column element value
to `currentValue + 1`.

plot(m.get(0, 0), linewidth = 3) // Plot the value from the first row and
column.
```

Note

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

Reading and writing matrix elements

`matrix.get()` and `matrix.set()`

To retrieve the value from a matrix at a specified `row` and `column` index, use [matrix.get\(\)](#). This function locates the specified matrix element and returns its value. Similarly, to overwrite a specific element's value, use [matrix.set\(\)](#) to assign the element at the specified `row` and `column` to a new value.

The example below defines a square matrix `m` with two rows and columns and an `initial_value` of 0 for all elements on the first bar. The script adds 1 to each element's value on different bars using the [m.get\(\)](#) and [m.set\(\)](#) methods. It updates the first row's first value once every 11 bars, the first row's second value once every seven bars, the second row's first value once

every five bars, and the second row's second value once every three bars. The script plots each element's value on the chart:



```
//@version=5
indicator("Reading and writing elements demo")

//@variable A 2x2 square matrix of `float` values.
var m = matrix.new<float>(2, 2, 0.0)

switch
    bar_index % 11 == 0 => m.set(0, 0, m.get(0, 0) + 1.0) // Adds 1 to the value
    at row 0, column 0 every 11th bar.
    bar_index % 7 == 0 => m.set(0, 1, m.get(0, 1) + 1.0) // Adds 1 to the value
    at row 0, column 1 every 7th bar.
    bar_index % 5 == 0 => m.set(1, 0, m.get(1, 0) + 1.0) // Adds 1 to the value
    at row 1, column 0 every 5th bar.
    bar_index % 3 == 0 => m.set(1, 1, m.get(1, 1) + 1.0) // Adds 1 to the value
    at row 1, column 1 every 3rd bar.

plot(m.get(0, 0), "Row 0, Column 0 Value", color.red, 2)
plot(m.get(0, 1), "Row 0, Column 1 Value", color.orange, 2)
plot(m.get(1, 0), "Row 1, Column 0 Value", color.green, 2)
plot(m.get(1, 1), "Row 1, Column 1 Value", color.blue, 2)
```

[`matrix.fill\(\)`](#)

To overwrite all matrix elements with a specific value, use [matrix.fill\(\)](#). This function points all items in the entire matrix or within the `from_row/column` and `to_row/column` index range to the value specified in the call. For example, this snippet declares a 4x4 square matrix, then fills its elements with a [random](#) value:

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

Note when using [matrix.fill\(\)](#) with matrices containing special types ([line](#), [linefill](#), [label](#), [box](#), or [table](#)) or [UDTs](#), all replaced elements will point to the same object passed in the function call.

This script declares a matrix with four rows and columns of [label](#) references, which it fills with a new [label](#) object on the first bar. On each bar, the script sets the `x` attribute of the label referenced at row 0, column 0 to `bar_index`, and the `text` attribute of the one referenced at row 3, column 3 to the number of labels on the chart. Although the matrix can reference 16 (4x4) labels, each element points to the *same* instance, resulting in only one label on the chart that updates its `x` and `text` attributes on each bar:



```
//@version=5
indicator("Object matrix fill demo")

//@variable A 4x4 label matrix.
var matrix<label> m = matrix.new<label>(4, 4)

// Fill `m` with a new label object on the first bar.
if bar_index == 0
    m.fill(label.new(0, 0, textcolor = color.white, size = size.huge))
```

```
//@variable The number of label objects on the chart.
int numLabels = label.all.size()

// Set the `x` of the label from the first row and column to `bar_index`.
m.get(0, 0).set_x(bar_index)
// Set the `text` of the label at the last row and column to the number of
labels.
m.get(3, 3).set_text(str.format("Total labels on the chart: {0}", numLabels))
```

Rows and columns

Retrieving

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

The script below populates a 3x2 m matrix with the values 1 - 6 on the first chart bar. It calls the [m.row\(\)](#) and [m.col\(\)](#) methods to access the first row and column arrays from the matrix and displays them on the chart in a label along with the array sizes:



```
//@version=5
indicator("Retrieving rows and columns demo")

//@variable A 3x2 rectangular matrix.
var matrix<float> m = matrix.new<float>(3, 2)

if bar_index == 0
    m.set(0, 0, 1.0) // Set row 0, column 0 value to 1.
    m.set(0, 1, 2.0) // Set row 0, column 1 value to 2.
    m.set(1, 0, 3.0) // Set row 1, column 0 value to 3.
    m.set(1, 1, 4.0) // Set row 1, column 1 value to 4.
    m.set(2, 0, 5.0) // Set row 1, column 0 value to 5.
    m.set(2, 1, 6.0) // Set row 1, column 1 value to 6.

//@variable The first row of the matrix.
array<float> row0 = m.row(0)
//@variable The first column of the matrix.
array<float> column0 = m.col(0)

//@variable Displays the first row and column of the matrix and their sizes in a
label.
var label debugLabel = label.new(0, 0, color = color.blue, textcolor =
color.white, size = size.huge)
debugLabel.set_x(bar_index)
debugLabel.set_text(str.format("Row 0: {0}, Size: {1}\nCol 0: {2}, Size: {3}",
row0, m.columns(), column0, m.rows()))
```

Note that:

- To get the sizes of the arrays displayed in the label, we used the [rows\(\)](#) and [columns\(\)](#) methods rather than [array.size\(\)](#) to demonstrate that the size of the `row0` array equals the number of columns and the size of the `column0` array equals the number of rows.

[matrix.row\(\)](#) and [matrix.col\(\)](#) copy the references in a row/column to a new [array](#). Modifications to

the [arrays](#) returned by these functions do not directly affect the elements or the shape of a matrix.

Here, we've modified the previous script to set the first element of `row0` to 10 via the [array.set\(\)](#) method before displaying the label. This script also plots the value from row 0, column 0. As we see, the label shows that the first element of the `row0` array is 10. However, the [plot](#) shows that the corresponding matrix element still has a value of 1:



```
//@version=5
indicator("Retrieving rows and columns demo")

//@variable A 3x2 rectangular matrix.
var matrix<float> m = matrix.new<float>(3, 2)

if bar_index == 0
    m.set(0, 0, 1.0) // Set row 0, column 0 value to 1.
    m.set(0, 1, 2.0) // Set row 0, column 1 value to 2.
    m.set(1, 0, 3.0) // Set row 1, column 0 value to 3.
    m.set(1, 1, 4.0) // Set row 1, column 1 value to 4.
    m.set(2, 0, 5.0) // Set row 1, column 0 value to 5.
    m.set(2, 1, 6.0) // Set row 1, column 1 value to 6.

//@variable The first row of the matrix.
array<float> row0 = m.row(0)
//@variable The first column of the matrix.
array<float> column0 = m.col(0)

// Set the first `row` element to 10.
row0.set(0, 10)

//@variable Displays the first row and column of the matrix and their sizes in a
label.
var label debugLabel = label.new(0, m.get(0, 0), color = color.blue, textcolor =
color.white, size = size.huge)
debugLabel.set_x(bar_index)
debugLabel.set_text(str.format("Row 0: {0}, Size: {1}\nCol 0: {2}, Size: {3}",
row0, m.columns(), column0, m.rows()))

// Plot the first element of `m`.
plot(m.get(0, 0), linewidth = 3)
```

Although changes to an [array](#) returned by [matrix.row\(\)](#) or [matrix.col\(\)](#) do not directly affect a parent matrix, it's important to note the resulting array from a matrix containing [UDTs](#) or special types, including [line](#), [linefill](#), [label](#), [box](#), or [table](#), behaves as a *shallow copy* of a row/column, i.e., the elements within an array returned from these functions point to the same objects as the corresponding matrix elements.

This script contains a custom `myUDT` type containing a `value` field with an initial value of 0. It declares a 1x1 `m` matrix to hold a single `myUDT` instance on the first bar, then calls `m.row(0)` to copy the first row of the matrix as an [array](#). On every chart bar, the script adds 1 to the `value` field of the first row array element. In this case, the `value` field of the matrix element increases on every bar as well since both elements reference the same object:

```
//@version=5
indicator("Row with reference types demo")

//@type A custom type that holds a float value.
type myUDT
    float value = 0.0
```

```

//@variable A 1x1 matrix of `myUDT` type.
var matrix<myUDT> m = matrix.new<myUDT>(1, 1, myUDT.new())
//@variable A shallow copy of the first row of `m`.
array<myUDT> row = m.row(0)
//@variable The first element of the `row`.
myUDT firstElement = row.get(0)

firstElement.value += 1.0 // Add 1 to the `value` field of `firstElement`. Also
affects the element in the matrix.

plot(m.get(0, 0).value, linewidth = 3) // Plot the `value` of the `myUDT` object
from the first row and column of `m`.

```

Inserting

Scripts can add new rows and columns to a matrix via [matrix.add_row\(\)](#) and [matrix.add_col\(\)](#). These functions insert the value references from an [array](#) into a matrix at the specified row/column index. If the id matrix is empty (has no rows or columns), the `array_id` in the call can be of any size. If a row/column exists at the specified index, the matrix increases the index value for the existing row/column and all after it by 1.

The script below declares an empty `m` matrix and inserts rows and columns using the [m.add_row\(\)](#) and [m.add_col\(\)](#) methods. It first inserts an array with three elements at row 0, turning `m` into a 1x3 matrix, then another at row 1, changing the shape to 2x3. After that, the script inserts another array at row 0, which changes the shape of `m` to 3x3 and shifts the index of all rows previously at index 0 and higher. It inserts another array at the last column index, changing the shape to 3x4. Finally, it adds an array with four values at the end row index.

The resulting matrix has four rows and columns and contains values 1-16 in ascending order. The script displays the rows of `m` after each row/column insertion with a user-defined `debugLabel()` function to visualize the process:



```

//@version=5
indicator("Rows and columns demo")

//@function Displays the rows of a matrix in a label with a note.
//@param this The matrix to display.
//@param barIndex The `bar_index` to display the label at.
//@param bgColor The background color of the label.
//@param textColor The color of the label's text.
//@param note The text to display above the rows.
method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""
) =>
    labelText = note + "\n" + str.toString(this)
    if barstate.ishistory
        label.new(
            barIndex, 0, labelText, color = bgColor, style =
label.style_label_center,
            textcolor = textColor, size = size.huge
        )

//Create an empty matrix.
var m = matrix.new<float>()

```

```

if bar_index == last_bar_index - 1
    debugLabel(m, bar_index - 30, note = "Empty matrix")

    // Insert an array at row 0. `m` will now have 1 row and 3 columns.
    m.add_row(0, array.from(5, 6, 7))
    debugLabel(m, bar_index - 20, note = "New row at\nindex 0")

    // Insert an array at row 1. `m` will now have 2 rows and 3 columns.
    m.add_row(1, array.from(9, 10, 11))
    debugLabel(m, bar_index - 10, note = "New row at\nindex 1")

    // Insert another array at row 0. `m` will now have 3 rows and 3 columns.
    // The values previously on row 0 will now be on row 1, and the values from
row 1 will be on row 2.
    m.add_row(0, array.from(1, 2, 3))
    debugLabel(m, bar_index, note = "New row at\nindex 0")

    // Insert an array at column 3. `m` will now have 3 rows and 4 columns.
    m.add_col(3, array.from(4, 8, 12))
    debugLabel(m, bar_index + 10, note = "New column at\nindex 3")

    // Insert an array at row 3. `m` will now have 4 rows and 4 columns.
    m.add_row(3, array.from(13, 14, 15, 16))
    debugLabel(m, bar_index + 20, note = "New row at\nindex 3")

```

Note

Just as the row or column arrays [retrieved](#) from a matrix of [line](#), [linefill](#), [label](#), [box](#), [table](#), or [UDT](#) instances behave as shallow copies, the elements of matrices containing such types reference the same objects as the [arrays](#) inserted into them. Modifications to the element values in either object affect the other in such cases.

Removing

To remove a specific row or column from a matrix, use [matrix.remove_row\(\)](#) and [matrix.remove_col\(\)](#). These functions remove the specified row/column and decrease the index values of all rows/columns after it by 1.

For this example, we've added these lines of code to our "Rows and columns demo" script from the [section above](#):

```

// Removing example

    // Remove the first row and last column from the matrix. `m` will now have 3
rows and 3 columns.
    m.remove_row(0)
    m.remove_col(3)
    debugLabel(m, bar_index + 30, color.red, note = "Removed row 0\nand column
3")

```

This code removes the first row and the last column of the `m` matrix using the [m.remove_row\(\)](#) and [m.remove_col\(\)](#) methods and displays the rows in a label at `bar_index + 30`. As we can see, `m` has a 3x3 shape after executing this block, and the index values for all existing rows are reduced by 1:



Swapping

To swap the rows and columns of a matrix without altering its dimensions, use [matrix.swap_rows\(\)](#) and [matrix.swap_columns\(\)](#). These functions swap the locations of the elements at the row1/column1 and row2/column2 indices.

Let's add the following lines to the [previous example](#), which swap the first and last rows of m and display the changes in a label at bar_index + 40:

```
// Swapping example

    // Swap the first and last row. `m` retains the same dimensions.
    m.swap_rows(0, 2)
    debugLabel(m, bar_index + 40, color.purple, note = "Swapped rows 0\nand 2")
```

In the new label, we see the matrix has the same number of rows as before, and the first and last rows have traded places:



Replacing

It may be desirable in some cases to completely *replace* a row or column in a matrix. To do so, [insert](#) the new array at the desired row/column and [remove](#) the old elements previously at that index.

In the following code, we've defined a `replaceRow()` method that uses the [add_row\(\)](#) method to insert the new values at the row index and uses the [remove_row\(\)](#) method to remove the old row that moved to the row + 1 index. This script uses the `replaceRow()` method to fill the rows of a 3x3 matrix with the numbers 1-9. It draws a label on the chart before and after replacing the rows using the custom `debugLabel()` method:



```
//@version=5
indicator("Replacing rows demo")

//@function Displays the rows of a matrix in a label with a note.
//@param this The matrix to display.
//@param barIndex The `bar_index` to display the label at.
//@param bgColor The background color of the label.
//@param textColor The color of the label's text.
//@param note The text to display above the rows.
method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""
) =>
    labelText = note + "\n" + str.toStringing(this)
    if barstate.ishistory
        label.new(
            barIndex, 0, labelText, color = bgColor, style =
label.style_label_center,
            textcolor = textColor, size = size.huge
        )

//@function Replaces the `row` of `this` matrix with a new array of `values`.
//@param row The row index to replace.
//@param values The array of values to insert.
```

```

method replaceRow(matrix<float> this, int row, array<float> values) =>
    this.add_row(row, values) // Inserts a copy of the `values` array at the
`row`.
    this.remove_row(row + 1) // Removes the old elements previously at the
`row`.

//@variable A 3x3 matrix.
var matrix<float> m = matrix.new<float>(3, 3, 0.0)

if bar_index == last_bar_index - 1
    m.debugLabel(note = "Original")
    // Replace each row of `m`.
    m.replaceRow(0, array.from(1.0, 2.0, 3.0))
    m.replaceRow(1, array.from(4.0, 5.0, 6.0))
    m.replaceRow(2, array.from(7.0, 8.0, 9.0))
    m.debugLabel(bar_index + 10, note = "Replaced rows")

```

Looping through a matrix

`for`

When a script only needs to iterate over the row/column indices in a matrix, the most common method is to use [for](#) loops. For example, this line creates a loop with a `row` value that starts at 0 and increases by one until it reaches one less than the number of rows in the `m` matrix (i.e., the last row index):

```
for row = 0 to m.rows() - 1
```

To iterate over all index values in the `m` matrix, we can create a *nested* loop that iterates over each column index on each row value:

```
for row = 0 to m.rows() - 1
    for column = 0 to m.columns() - 1
```

Let's use this nested structure to create a [method](#) that visualizes matrix elements. In the script below, we've defined a `toTable()` method that displays the elements of a matrix within a [table](#) object. It iterates over each row index and over each column index on every row. Within the loop, it converts each element to a [string](#) to display in the corresponding table cell.

On the first bar, the script creates an empty `m` matrix, populates it with rows, and calls `m.toTable()` to display its elements:



```

//@version=5
indicator("for loop demo", "Matrix to table")

//@function Displays the elements of `this` matrix in a table.
//@param this The matrix to display.
//@param position The position of the table on the chart.
//@param bgColor The background color of the table.
//@param textColor The color of the text in each cell.
//@param note A note string to display on the bottom row of the table.
//@returns A new `table` object with cells corresponding to each element of
`this` matrix.
method toTable(
    matrix<float> this, string position = position.middle_center,
    color bgColor = color.blue, color textColor = color.white,

```

```

    string note = na
) =>
    //@variable The number of rows in `this` matrix.
    int rows = this.rows()
    //@variable The number of columns in `this` matrix.
    int columns = this.columns()
    //@variable A table that displays the elements of `this` matrix with an
optional `note` cell.
    table result = table.new(position, columns, rows + 1, bgColor)

    // Iterate over each row index of `this` matrix.
    for row = 0 to rows - 1
        // Iterate over each column index of `this` matrix on each `row`.
        for col = 0 to columns - 1
            //@variable The element from `this` matrix at the `row` and `col`
index.
            float element = this.get(row, col)
            // Initialize the corresponding `result` cell with the `element`
value.
            result.cell(col, row, str.toString(element), text_color = textColor,
text_size = size.huge)

            // Initialize a merged cell on the bottom row if a `note` is provided.
            if not na(note)
                result.cell(0, rows, note, text_color = textColor, text_size =
size.huge)
                result.merge_cells(0, rows, columns - 1, rows)

        result // Return the `result` table.

//@variable A 3x4 matrix of values.
var m = matrix.new<float>()

if bar_index == 0
    // Add rows to `m`.
    m.add_row(0, array.from(1, 2, 3))
    m.add_row(1, array.from(5, 6, 7))
    m.add_row(2, array.from(9, 10, 11))
    // Add a column to `m`.
    m.add_col(3, array.from(4, 8, 12))
    // Display the elements of `m` in a table.
    m.toTable()

```

[`for...in`](#)

When a script needs to iterate over and retrieve the rows of a matrix, using the [for...in](#) structure is often preferred over the standard `for` loop. This structure directly references the row [arrays](#) in a matrix, making it a more convenient option for such use cases. For example, this line creates a loop that returns a row array for each row in the `m` matrix:

```
for row in m
```

The following indicator calculates the moving average of OHLC data with an input `length` and displays the values on the chart. The custom `rowWiseAvg()` method loops through the rows of a matrix using a `for...in` structure to produce an array containing the [array.avg\(\)](#) of each row.

On the first chart bar, the script creates a new `m` matrix with four rows and `length` columns, which it queues a new column of OHLC data into via the [m.add_col\(\)](#) and [m.remove_col\(\)](#) methods on each subsequent bar. It uses `m.rowWiseAvg()` to calculate the array of row-wise averages, then it plots the element values on the chart:



```
//@version=5
indicator("for...in loop demo", "Average OHLC", overlay = true)

//@variable The number of terms in the average.
int length = input.int(20, "Length", minval = 1)

//@function Calculates the average of each matrix row.
method rowWiseAvg(matrix<float> this) =>
    //@variable An array with elements corresponding to each row's average.
    array<float> result = array.new<float>()
    // Iterate over each `row` of `this` matrix.
    for row in this
        // Push the average of each `row` into the `result`.
        result.push(row.avg())
    result // Return the resulting array.

//@variable A 4x`length` matrix of values.
var matrix<float> m = matrix.new<float>(4, length)

// Add a new column containing OHLC values to the matrix.
m.add_col(m.columns(), array.from(open, high, low, close))
// Remove the first column.
m.remove_col(0)

//@variable An array containing averages of `open`, `high`, `low`, and `close`
over `length` bars.
array<float> averages = m.rowWiseAvg()

plot(averages.get(0), "Average Open", color.blue, 2)
plot(averages.get(1), "Average High", color.green, 2)
plot(averages.get(2), "Average Low", color.red, 2)
plot(averages.get(3), "Average Close", color.orange, 2)
```

Note that:

- `for...in` loops can also reference the index value of each row. For example, `for [i, row] in m` creates a tuple containing the `i` row index and the corresponding row array from the `m` matrix on each loop iteration.

Copying a matrix

Shallow copies

Pine scripts can copy matrices via [matrix.copy\(\)](#). This function returns a *shallow copy* of a matrix that does not affect the shape of the original matrix or its references.

For example, this script assigns a new matrix to the `myMatrix` variable and adds two columns. It creates a new `myCopy` matrix from `myMatrix` using the [myMatrix.copy\(\)](#) method, then adds a new row. It displays the rows of both matrices in labels via the user-defined `debugLabel()` function:



```
//@version=5
indicator("Shallow copy demo")
```

```

//@function Displays the rows of a matrix in a label with a note.
//@param this The matrix to display.
//@param barIndex The `bar_index` to display the label at.
//@param bgColor The background color of the label.
//@param textColor The color of the label's text.
//@param note The text to display above the rows.
method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""
) =>
    labelText = note + "\n" + str.toString(this)
    if barstate.ishistory
        label.new(
            barIndex, 0, labelText, color = bgColor, style =
label.style_label_center,
            textcolor = textColor, size = size.huge
        )

//@variable A 2x2 `float` matrix.
matrix<float> myMatrix = matrix.new<float>()
myMatrix.add_col(0, array.from(1.0, 3.0))
myMatrix.add_col(1, array.from(2.0, 4.0))

//@variable A shallow copy of `myMatrix`.
matrix<float> myCopy = myMatrix.copy()
// Add a row to the last index of `myCopy`.
myCopy.add_row(myCopy.rows(), array.from(5.0, 6.0))

if bar_index == last_bar_index - 1
    // Display the rows of both matrices in separate labels.
    myMatrix.debugLabel(note = "Original")
    myCopy.debugLabel(bar_index + 10, color.green, note = "Shallow Copy")

```

It's important to note that the elements within shallow copies of a matrix point to the same values as the original matrix. When matrices contain special types ([line](#), [linefill](#), [label](#), [box](#), or [table](#)) or [user-defined types](#), the elements of a shallow copy reference the same objects as the original.

This script declares a `myMatrix` variable with a `newLabel` as the initial value. It then copies `myMatrix` to a `myCopy` variable via [myMatrix.copy\(\)](#) and plots the number of labels. As we see below, there's only one [label](#) on the chart, as the element in `myCopy` references the same object as the element in `myMatrix`. Consequently, changes to the element values in `myCopy` affect the values in both matrices:



```

//@version=5
indicator("Shallow copy demo")

//@variable Initial value of the original matrix elements.
var label newLabel = label.new(
    bar_index, 1, "Original", color = color.blue, textcolor = color.white, size
= size.huge
)

//@variable A 1x1 matrix containing a new `label` instance.
var matrix<label> myMatrix = matrix.new<label>(1, 1, newLabel)
//@variable A shallow copy of `myMatrix`.
var matrix<label> myCopy = myMatrix.copy()

//@variable The first label from the `myCopy` matrix.

```

```

label testLabel = myCopy.get(0, 0)

// Change the `text`, `style`, and `x` values of `testLabel`. Also affects the
`newLabel`.
testLabel.set_text("Copy")
testLabel.set_style(label.style_label_up)
testLabel.set_x(bar_index)

// Plot the total number of labels.
plot(label.all.size(), linewidth = 3)

```

Deep copies

One can produce a *deep copy* of a matrix (i.e., a matrix whose elements point to copies of the original values) by explicitly copying each object the matrix references.

Here, we've added a `deepCopy()` user-defined method to our previous script. The method creates a new matrix and uses [nested for loops](#) to assign all elements to copies of the originals. When the script calls this method instead of the built-in [copy\(\)](#), we see that there are now two labels on the chart, and any changes to the label from `myCopy` do not affect the one from `myMatrix`:



```

//@version=5
indicator("Deep copy demo")

//@function Returns a deep copy of a label matrix.
method deepCopy(matrix<label> this) =>
    //@variable A deep copy of `this` matrix.
    matrix<label> that = this.copy()
    for row = 0 to that.rows() - 1
        for column = 0 to that.columns() - 1
            // Assign the element at each `row` and `column` of `that` matrix to
a copy of the retrieved label.
            that.set(row, column, that.get(row, column).copy())
        that

//@variable Initial value of the original matrix.
var label newLabel = label.new(
    bar_index, 2, "Original", color = color.blue, textcolor = color.white, size
= size.huge
)

//@variable A 1x1 matrix containing a new `label` instance.
var matrix<label> myMatrix = matrix.new<label>(1, 1, newLabel)
//@variable A deep copy of `myMatrix`.
var matrix<label> myCopy = myMatrix.deepCopy()

//@variable The first label from the `myCopy` matrix.
label testLabel = myCopy.get(0, 0)

// Change the `text`, `style`, and `x` values of `testLabel`. Does not affect
the `newLabel`.
testLabel.set_text("Copy")
testLabel.set_style(label.style_label_up)
testLabel.set_x(bar_index)

// Change the `x` value of `newLabel`.
newLabel.set_x(bar_index)

// Plot the total number of labels.

```

```
plot(label.all.size(), linewidth = 3)
```

Submatrices

In Pine, a *submatrix* is a [shallow copy](#) of an existing matrix that only includes the rows and columns specified by the `from_row/column` and `to_row/column` parameters. In essence, it is a sliced copy of a matrix.

For example, the script below creates an `mSub` matrix from the `m` matrix via the [m.submatrix\(\)](#) method, then calls our user-defined `debugLabel()` function to display the rows of both matrices in labels:



```
//@version=5
indicator("Submatrix demo")

//@function Displays the rows of a matrix in a label with a note.
//@param this The matrix to display.
//@param barIndex The `bar_index` to display the label at.
//@param bgColor The background color of the label.
//@param textColor The color of the label's text.
//@param note The text to display above the rows.
method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""
) =>
    labelText = note + "\n" + str.toString(this)
    if barstate.ishistory
        label.new(
            barIndex, 0, labelText, color = bgColor, style =
label.style_label_center,
            textcolor = textColor, size = size.huge
        )

//@variable A 3x3 matrix of values.
var m = matrix.new<float>()

if bar_index == last_bar_index - 1
    // Add columns to `m`.
    m.add_col(0, array.from(9, 6, 3))
    m.add_col(1, array.from(8, 5, 2))
    m.add_col(2, array.from(7, 4, 1))
    // Display the rows of `m`.
    m.debugLabel(note = "Original Matrix")

    //@variable A 2x2 submatrix of `m` containing the first two rows and
columns.
    matrix<float> mSub = m.submatrix(from_row = 0, to_row = 2, from_column = 0,
to_column = 2)
    // Display the rows of `mSub`
    debugLabel(mSub, bar_index + 10, bgColor = color.green, note = "Submatrix")
```

Scope and history

Matrix variables leave historical trails on each bar, allowing scripts to use the history-referencing operator `█` to interact with past matrix instances previously assigned to a variable. Additionally, scripts can modify matrices assigned to global variables from within the scopes of [functions](#),

[methods](#), and [conditional structures](#).

This script calculates the average ratios of body and wick distances relative to the bar range over length bars. It displays the data along with values from length bars ago in a table. The user-defined addData() function adds columns of current and historical ratios to the globalMatrix, and the calcAvg() function references previous matrices assigned to globalMatrix using the [] operator to calculate a matrix of averages:



```
//@version=5
indicator("Scope and history demo", "Bar ratio comparison")

int length = input.int(10, "Length", 1)

//@variable A global matrix.
matrix<float> globalMatrix = matrix.new<float>()

//@function Calculates the ratio of body range to candle range.
bodyRatio() =>
    math.abs(close - open) / (high - low)

//@function Calculates the ratio of upper wick range to candle range.
upperWickRatio() =>
    (high - math.max(open, close)) / (high - low)

//@function Calculates the ratio of lower wick range to candle range.
lowerWickRatio() =>
    (math.min(open, close) - low) / (high - low)

//@function Adds data to the `globalMatrix`.
addData() =>
    // Add a new column of data at `column` 0.
    globalMatrix.add_col(0, array.from(bodyRatio(), upperWickRatio(),
lowerWickRatio()))
    //@variable The column of `globalMatrix` from index 0 `length` bars ago.
    array<float> pastValues = globalMatrix.col(0)[length]
    // Add `pastValues` to the `globalMatrix`, or an array of `na` if
`pastValues` is `na`.
    if na(pastValues)
        globalMatrix.add_col(1, array.new<float>(3))
    else
        globalMatrix.add_col(1, pastValues)

//@function Returns the `length`-bar average of matrices assigned to
`globalMatrix` on historical bars.
calcAvg() =>
    //@variable The sum historical `globalMatrix` matrices.
    matrix<float> sums = matrix.new<float>(globalMatrix.rows(),
globalMatrix.columns(), 0.0)
    for i = 0 to length - 1
        //@variable The `globalMatrix` matrix `i` bars before the current bar.
        matrix<float> previous = globalMatrix[i]
        // Break the loop if `previous` is `na`.
        if na(previous)
            sums.fill(na)
            break
        // Assign the sum of `sums` and `previous` to `sums`.
        sums := matrix.sum(sums, previous)
    // Divide the `sums` matrix by the `length`.
    result = sums.mult(1.0 / length)
```



```

// Add data to the `globalMatrix`.
addData()

//@variable The historical average of the `globalMatrix` matrices.
globalAvg = calcAvg()

//@variable A `table` displaying information from the `globalMatrix`.
var table infoTable = table.new(
    position.middle_center, globalMatrix.columns() + 1, globalMatrix.rows() +
1, bgcolor = color.navy
)

// Define value cells.
for [i, row] in globalAvg
    for [j, value] in row
        color textColor = value > 0.333 ? color.orange : color.gray
        infoTable.cell(j + 1, i + 1, str.toString(value), text_color =
textColor, text_size = size.huge)

// Define header cells.
infoTable.cell(0, 1, "Body ratio", text_color = color.white, text_size =
size.huge)
infoTable.cell(0, 2, "Upper wick ratio", text_color = color.white, text_size =
size.huge)
infoTable.cell(0, 3, "Lower wick ratio", text_color = color.white, text_size =
size.huge)
infoTable.cell(1, 0, "Current average", text_color = color.white, text_size =
size.huge)
infoTable.cell(2, 0, str.format("{0} bars ago", length), text_color =
color.white, text_size = size.huge)

```

Note that:

- The `addData()` and `calcAvg()` functions have no parameters, as they directly interact with the `globalMatrix` and `length` variables declared in the outer scope.
- `calcAvg()` calculates the average by adding previous matrices using [matrix.sum\(\)](#) and multiplying all elements by $1 / \text{length}$ using [matrix.mult\(\)](#). We discuss these and other specialized functions in our [Matrix calculations](#) section below.

[Inspecting a matrix](#)

The ability to inspect the shape of a matrix and patterns within its elements is crucial, as it helps reveal important information about a matrix and its compatibility with various calculations and transformations. Pine Script[®] includes several built-ins for matrix inspection, including [matrix.is_square\(\)](#), [matrix.is_identity\(\)](#), [matrix.is_diagonal\(\)](#), [matrix.is_antidiagonal\(\)](#), [matrix.is_symmetric\(\)](#), [matrix.is_antisymmetric\(\)](#), [matrix.is_triangular\(\)](#), [matrix.is_stochastic\(\)](#), [matrix.is_binary\(\)](#), and [matrix.is_zero\(\)](#).

To demonstrate these features, this example contains a custom `inspect()` method that uses conditional blocks with `matrix.is_*()` functions to return information about a matrix. It displays a string representation of an `m` matrix and the description returned from `m.inspect()` in labels on the chart:



```

//@version=5
indicator("Matrix inspection demo")

```

```

//@function Inspects a matrix using `matrix.is_*()` functions and returns a
`string` describing some of its features.
method inspect(matrix<int> this)=>
  //@variable A string describing `this` matrix.
  string result = "This matrix:\n"
  if this.is_square()
    result += "- Has an equal number of rows and columns.\n"
  if this.is_binary()
    result += "- Contains only 1s and 0s.\n"
  if this.is_zero()
    result += "- Is filled with 0s.\n"
  if this.is_triangular()
    result += "- Contains only 0s above and/or below its main diagonal.\n"
  if this.is_diagonal()
    result += "- Only has nonzero values in its main diagonal.\n"
  if this.is_antidiagonal()
    result += "- Only has nonzero values in its main antidiagonal.\n"
  if this.is_symmetric()
    result += "- Equals its transpose.\n"
  if this.is_antisymmetric()
    result += "- Equals the negative of its transpose.\n"
  if this.is_identity()
    result += "- Is the identity matrix.\n"
  result

//@variable A 4x4 identity matrix.
matrix<int> m = matrix.new<int>()

// Add rows to the matrix.
m.add_row(0, array.from(1, 0, 0, 0))
m.add_row(1, array.from(0, 1, 0, 0))
m.add_row(2, array.from(0, 0, 1, 0))
m.add_row(3, array.from(0, 0, 0, 1))

if bar_index == last_bar_index - 1
  // Display the `m` matrix in a blue label.
  label.new(
    bar_index, 0, str.tostring(m), color = color.blue, style =
label.style_label_right,
    textcolor = color.white, size = size.huge
  )
  // Display the result of `m.inspect()` in a purple label.
  label.new(
    bar_index, 0, m.inspect(), color = color.purple, style =
label.style_label_left,
    textcolor = color.white, size = size.huge
  )

```

Manipulating a matrix

Reshaping

The shape of a matrix can determine its compatibility with various matrix operations. In some cases, it is necessary to change the dimensions of a matrix without affecting the number of elements or the values they reference, otherwise known as *reshaping*. To reshape a matrix in Pine, use the [matrix.reshape\(\)](#) function.

This example demonstrates the results of multiple reshaping operations on a matrix. The initial `m` matrix has a 1x8 shape (one row and eight columns). Through successive calls to the [m.reshape\(\)](#) method, the script changes the shape of `m` to 2x4, 4x2, and 8x1. It displays each reshaped matrix in

a label on the chart using the custom `debugLabel()` method:



```
//@version=5
indicator("Reshaping example")

//@function Displays the rows of a matrix in a label with a note.
//@param this The matrix to display.
//@param barIndex The `bar_index` to display the label at.
//@param bgColor The background color of the label.
//@param textColor The color of the label's text.
//@param note The text to display above the rows.
method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""
) =>
    labelText = note + "\n" + str.toString(this)
    if barstate.ishistory
        label.new(
            barIndex, 0, labelText, color = bgColor, style =
label.style_label_center,
            textcolor = textColor, size = size.huge
        )

//@variable A matrix containing the values 1-8.
matrix<int> m = matrix.new<int>()

if bar_index == last_bar_index - 1
    // Add the initial vector of values.
    m.add_row(0, array.from(1, 2, 3, 4, 5, 6, 7, 8))
    m.debugLabel(note = "Initial 1x8 matrix")

    // Reshape. `m` now has 2 rows and 4 columns.
    m.reshape(2, 4)
    m.debugLabel(bar_index + 10, note = "Reshaped to 2x4")

    // Reshape. `m` now has 4 rows and 2 columns.
    m.reshape(4, 2)
    m.debugLabel(bar_index + 20, note = "Reshaped to 4x2")

    // Reshape. `m` now has 8 rows and 1 column.
    m.reshape(8, 1)
    m.debugLabel(bar_index + 30, note = "Reshaped to 8x1")
```

Note that:

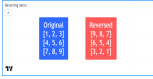
- The order of elements in `m` does not change with each `m.reshape()` call.
- When reshaping a matrix, the product of the `rows` and `columns` arguments must equal the `matrix.elements_count()` value, as `matrix.reshape()` cannot change the number of elements in a matrix.

Reversing

One can reverse the order of all elements in a matrix using `matrix.reverse()`. This function moves the references of an `m`-by-`n` matrix `id` at the `i`-th row and `j`-th column to the `m - 1 - i` row and `n - 1 - j` column.

For example, this script creates a 3x3 matrix containing the values 1-9 in ascending order, then uses the `reverse()` method to reverse its contents. It displays the original and modified versions of the

matrix in labels on the chart via `m.debugLabel()`:



```
//@version=5
indicator("Reversing demo")

//@function Displays the rows of a matrix in a label with a note.
//@param this The matrix to display.
//@param barIndex The `bar_index` to display the label at.
//@param bgColor The background color of the label.
//@param textColor The color of the label's text.
//@param note The text to display above the rows.
method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""
) =>
    labelText = note + "\n" + str.toString(this)
    if barstate.ishistory
        label.new(
            barIndex, 0, labelText, color = bgColor, style =
label.style_label_center,
            textcolor = textColor, size = size.huge
        )

//@variable A 3x3 matrix.
matrix<float> m = matrix.new<float>()

// Add rows to `m`.
m.add_row(0, array.from(1, 2, 3))
m.add_row(1, array.from(4, 5, 6))
m.add_row(2, array.from(7, 8, 9))

if bar_index == last_bar_index - 1
    // Display the contents of `m`.
    m.debugLabel(note = "Original")
    // Reverse `m`, then display its contents.
    m.reverse()
    m.debugLabel(bar_index + 10, color.red, note = "Reversed")
```

Transposing

Transposing a matrix is a fundamental operation that flips all rows and columns in a matrix about its *main diagonal* (the diagonal vector of all values in which the row index equals the column index). This process produces a new matrix with reversed row and column dimensions, known as the *transpose*. Scripts can calculate the transpose of a matrix using [matrix.transpose\(\)](#).

For any m-row, n-column matrix, the matrix returned from [matrix.transpose\(\)](#) will have n rows and m columns. All elements in a matrix at the i-th row and j-th column correspond to the elements in its transpose at the j-th row and i-th column.

This example declares a 2x4 m matrix, calculates its transpose using the [m.transpose\(\)](#) method, and displays both matrices on the chart using our custom `debugLabel()` method. As we can see below, the transposed matrix has a 4x2 shape, and the rows of the transpose match the columns of the original:



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

```

indicator("Transpose example")

//@function Displays the rows of a matrix in a label with a note.
//@param this The matrix to display.
//@param barIndex The `bar_index` to display the label at.
//@param bgColor The background color of the label.
//@param textColor The color of the label's text.
//@param note The text to display above the rows.
method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""
) =>
    labelText = note + "\n" + str.toStringing(this)
    if barstate.ishistory
        label.new(
            barIndex, 0, labelText, color = bgColor, style =
label.style_label_center,
            textcolor = textColor, size = size.huge
        )

//@variable A 2x4 matrix.
matrix<int> m = matrix.new<int>()

// Add columns to `m`.
m.add_col(0, array.from(1, 5))
m.add_col(1, array.from(2, 6))
m.add_col(2, array.from(3, 7))
m.add_col(3, array.from(4, 8))

//@variable The transpose of `m`. Has a 4x2 shape.
matrix<int> mt = m.transpose()

if bar_index == last_bar_index - 1
    m.debugLabel(note = "Original")
    mt.debugLabel(bar_index + 10, note = "Transpose")

```

Sorting

Scripts can sort the contents of a matrix via [matrix.sort\(\)](#). Unlike [array.sort\(\)](#), which sorts *elements*, this function organizes all *rows* in a matrix in a specified order ([order.ascending](#) by default) based on the values in a specified `column`.

This script declares a 3x3 `m` matrix, sorts the rows of the `m1` copy in ascending order based on the first column, then sorts the rows of the `m2` copy in descending order based on the second column. It displays the original matrix and sorted copies in labels using our `debugLabel()` method:



```

//@version=5
indicator("Sorting rows example")

//@function Displays the rows of a matrix in a label with a note.
//@param this The matrix to display.
//@param barIndex The `bar_index` to display the label at.
//@param bgColor The background color of the label.
//@param textColor The color of the label's text.
//@param note The text to display above the rows.
method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""

```

```

) =>
    labelText = note + "\n" + str.toString(this)
    if barstate.ishistory
        label.new(
            barIndex, 0, labelText, color = bgColor, style =
label.style_label_center,
            textcolor = textColor, size = size.huge
        )

//@variable A 3x3 matrix.
matrix<int> m = matrix.new<int>()

if bar_index == last_bar_index - 1
    // Add rows to `m`.
    m.add_row(0, array.from(3, 2, 4))
    m.add_row(1, array.from(1, 9, 6))
    m.add_row(2, array.from(7, 8, 9))
    m.debugLabel(note = "Original")

    // Copy `m` and sort rows in ascending order based on the first column
(default).
    matrix<int> m1 = m.copy()
    m1.sort()
    m1.debugLabel(bar_index + 10, color.green, note = "Sorted using col
0\n(Ascending)")

    // Copy `m` and sort rows in descending order based on the second column.
    matrix<int> m2 = m.copy()
    m2.sort(1, order.descending)
    m2.debugLabel(bar_index + 20, color.red, note = "Sorted using col
1\n(Descending)")

```

It's important to note that [matrix.sort\(\)](#) does not sort the columns of a matrix. However, one *can* use this function to sort matrix columns with the help of [matrix.transpose\(\)](#).

As an example, this script contains a `sortColumns()` method that uses the [sort\(\)](#) method to sort the [transpose](#) of a matrix using the column corresponding to the row of the original matrix. The script uses this method to sort the `m` matrix based on the contents of its first row:



```

//@version=5
indicator("Sorting columns example")

//@function Displays the rows of a matrix in a label with a note.
//@param this The matrix to display.
//@param barIndex The `bar_index` to display the label at.
//@param bgColor The background color of the label.
//@param textColor The color of the label's text.
//@param note The text to display above the rows.
method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""
) =>
    labelText = note + "\n" + str.toString(this)
    if barstate.ishistory
        label.new(
            barIndex, 0, labelText, color = bgColor, style =
label.style_label_center,
            textcolor = textColor, size = size.huge
        )

```

```

//@function Sorts the columns of `this` matrix based on the values in the
specified `row`.
method sortColumns(matrix<int> this, int row = 0, bool ascending = true) =>
    //@variable The transpose of `this` matrix.
    matrix<int> thisT = this.transpose()
    //@variable Is `order.ascending` when `ascending` is `true`,
`order.descending` otherwise.
    order = ascending ? order.ascending : order.descending
    // Sort the rows of `thisT` using the `row` column.
    thisT.sort(row, order)
    //@variable A copy of `this` matrix with sorted columns.
    result = thisT.transpose()

//@variable A 3x3 matrix.
matrix<int> m = matrix.new<int>()

if bar_index == last_bar_index - 1
    // Add rows to `m`.
    m.add_row(0, array.from(3, 2, 4))
    m.add_row(1, array.from(1, 9, 6))
    m.add_row(2, array.from(7, 8, 9))
    m.debugLabel(note = "Original")

    // Sort the columns of `m` based on the first row and display the result.
    m.sortColumns(0).debugLabel(bar_index + 10, note = "Sorted using row
0\n(Ascending)")

```

Concatenating

Scripts can *concatenate* two matrices using [matrix.concat\(\)](#). This function appends the rows of an `id2` matrix to the end of an `id1` matrix with the same number of columns.

To create a matrix with elements representing the *columns* of a matrix appended to another, [transpose](#) both matrices, use [matrix.concat\(\)](#) on the transposed matrices, then [transpose\(\)](#) the result.

For example, this script appends the rows of the `m2` matrix to the `m1` matrix and appends their columns using *transposed copies* of the matrices. It displays the `m1` and `m2` matrices and the results after concatenating their rows and columns in labels using the custom `debugLabel()` method:



```

//@version=5
indicator("Concatenation demo")

//@function Displays the rows of a matrix in a label with a note.
//@param this The matrix to display.
//@param barIndex The `bar_index` to display the label at.
//@param bgColor The background color of the label.
//@param textColor The color of the label's text.
//@param note The text to display above the rows.
method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""
) =>
    labelText = note + "\n" + str.toString(this)
    if barstate.ishistory
        label.new(
            barIndex, 0, labelText, color = bgColor, style =
label.style_label_center,
            textcolor = textColor, size = size.huge
        )

```

```

//@variable A 2x3 matrix filled with 1s.
matrix<int> m1 = matrix.new<int>(2, 3, 1)
//@variable A 2x3 matrix filled with 2s.
matrix<int> m2 = matrix.new<int>(2, 3, 2)

//@variable The transpose of `m1`.
t1 = m1.transpose()
//@variable The transpose of `m2`.
t2 = m2.transpose()

if bar_index == last_bar_index - 1
    // Display the original matrices.
    m1.debugLabel(note = "Matrix 1")
    m2.debugLabel(bar_index + 10, note = "Matrix 2")
    // Append the rows of `m2` to the end of `m1` and display `m1`.
    m1.concat(m2)
    m1.debugLabel(bar_index + 20, color.blue, note = "Appended rows")
    // Append the rows of `t2` to the end of `t1`, then display the transpose of
    `t1.
    t1.concat(t2)
    t1.transpose().debugLabel(bar_index + 30, color.purple, note = "Appended
columns")

```

Matrix calculations

Element-wise calculations

Pine scripts can calculate the *average*, *minimum*, *maximum*, and *mode* of all elements within a matrix via [matrix.avg\(\)](#), [matrix.min\(\)](#), [matrix.max\(\)](#), and [matrix.mode\(\)](#). These functions operate the same as their `array.*` equivalents, allowing users to run element-wise calculations on a matrix, its [submatrices](#), and its [rows and columns](#) using the same syntax. For example, the built-in `*.avg()` functions called on a 3x3 matrix with values 1-9 and an [array](#) with the same nine elements will both return a value of 5.

The script below uses `*.avg()`, `*.max()`, and `*.min()` methods to calculate developing averages and extremes of OHLC data in a period. It adds a new column of [open](#), [high](#), [low](#), and [close](#) values to the end of the `ohlcData` matrix whenever `queueColumn` is true. When false, the script uses the [get\(\)](#) and [set\(\)](#) matrix methods to adjust the elements in the last column for developing HLC values in the current period. It uses the `ohlcData` matrix, a [submatrix\(\)](#), and [row\(\)](#) and [col\(\)](#) arrays to calculate the developing OHLC4 and HL2 averages over `length` periods, the maximum high and minimum low over `length` periods, and the current period's developing OHLC4 price:



```

//@version=5
indicator("Element-wise calculations example", "Developing values", overlay =
true)

//@variable The number of data points in the averages.
int length = input.int(3, "Length", 1)
//@variable The timeframe of each reset period.
string timeframe = input.timeframe("D", "Reset Timeframe")

//@variable A 4x`length` matrix of OHLC values.
var matrix<float> ohlcData = matrix.new<float>(4, length)

```



```

//@variable Is `true` at the start of a new bar at the `timeframe`.
bool queueColumn = timeframe.change(timeframe)

if queueColumn
    // Add new values to the end column of `ohlData`.
    ohlcData.add_col(length, array.from(open, high, low, close))
    // Remove the oldest column from `ohlData`.
    ohlcData.remove_col(0)
else
    // Adjust the last element of column 1 for new highs.
    if high > ohlcData.get(1, length - 1)
        ohlcData.set(1, length - 1, high)
    // Adjust the last element of column 2 for new lows.
    if low < ohlcData.get(2, length - 1)
        ohlcData.set(2, length - 1, low)
    // Adjust the last element of column 3 for the new closing price.
    ohlcData.set(3, length - 1, close)

//@variable The `matrix.avg()` of all elements in `ohlData`.
avgOHL4 = ohlcData.avg()
//@variable The `matrix.avg()` of all elements in rows 1 and 2, i.e., the
average of all `high` and `low` values.
avgHL2 = ohlcData.submatrix(from_row = 1, to_row = 3).avg()
//@variable The `matrix.max()` of all values in `ohlData`. Equivalent to
`ohlData.row(1).max()`.
maxHigh = ohlcData.max()
//@variable The `array.min()` of all `low` values in `ohlData`. Equivalent to
`ohlData.min()`.
minLow = ohlcData.row(2).min()
//@variable The `array.avg()` of the last column in `ohlData`, i.e., the
current OHL4.
ohl4Value = ohlcData.col(length - 1).avg()

plot(avgOHL4, "Average OHL4", color.purple, 2)
plot(avgHL2, "Average HL2", color.navy, 2)
plot(maxHigh, "Max High", color.green)
plot(minLow, "Min Low", color.red)
plot(ohl4Value, "Current OHL4", color.blue)

```

Note that:

- In this example, we used [array.*\(\)](#) and [matrix.*\(\)](#) methods interchangeably to demonstrate their similarities in syntax and behavior.
- Users can calculate the matrix equivalent of [array.sum\(\)](#) by multiplying the [matrix.avg\(\)](#) by the [matrix.elements_count\(\)](#).

Special calculations

Pine Script® features several built-in functions for performing essential matrix arithmetic and linear algebra operations, including [matrix.sum\(\)](#), [matrix.diff\(\)](#), [matrix.mult\(\)](#), [matrix.pow\(\)](#), [matrix.det\(\)](#), [matrix.inv\(\)](#), [matrix.pinv\(\)](#), [matrix.rank\(\)](#), [matrix.trace\(\)](#), [matrix.eigenvalues\(\)](#), [matrix.eigenvectors\(\)](#), and [matrix.kron\(\)](#). These functions are advanced features that facilitate a variety of matrix calculations and transformations.

Below, we explain a few fundamental functions with some basic examples.

[matrix.sum\(\)](#) and [matrix.diff\(\)](#)

Scripts can perform addition and subtraction of two matrices with the same shape or a matrix and a

scalar value using the [matrix.sum\(\)](#) and [matrix.diff\(\)](#) functions. These functions use the values from the `id2` matrix or scalar to add to or subtract from the elements in `id1`.

This script demonstrates a simple example of matrix addition and subtraction in Pine. It creates a 3x3 matrix, calculates its [transpose](#), then calculates the [matrix.sum\(\)](#) and [matrix.diff\(\)](#) of the two matrices. This example displays the original matrix, its [transpose](#), and the resulting sum and difference matrices in labels on the chart:



```
//@version=5
indicator("Matrix sum and diff example")

//@function Displays the rows of a matrix in a label with a note.
//@param this The matrix to display.
//@param barIndex The `bar_index` to display the label at.
//@param bgColor The background color of the label.
//@param textColor The color of the label's text.
//@param note The text to display above the rows.
method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""
) =>
    labelText = note + "\n" + str.toString(this)
    if barstate.ishistory
        label.new(
            barIndex, 0, labelText, color = bgColor, style =
label.style_label_center,
            textcolor = textColor, size = size.huge
        )

//@variable A 3x3 matrix.
m = matrix.new<float>()

// Add rows to `m`.
m.add_row(0, array.from(0.5, 1.0, 1.5))
m.add_row(1, array.from(2.0, 2.5, 3.0))
m.add_row(2, array.from(3.5, 4.0, 4.5))

if bar_index == last_bar_index - 1
    // Display `m`.
    m.debugLabel(note = "A")
    // Get and display the transpose of `m`.
    matrix<float> t = m.transpose()
    t.debugLabel(bar_index + 10, note = "AT")
    // Calculate the sum of the two matrices. The resulting matrix is symmetric.
    matrix.sum(m, t).debugLabel(bar_index + 20, color.green, note = "A + AT")
    // Calculate the difference between the two matrices. The resulting matrix
    is antisymmetric.
    matrix.diff(m, t).debugLabel(bar_index + 30, color.red, note = "A - AT")
```

Note that:

- In this example, we've labeled the original matrix as "A" and the transpose as "A^T".
- Adding "A" and "A^T" produces a [symmetric](#) matrix, and subtracting them produces an [antisymmetric](#) matrix.

[matrix.mult\(\)](#)

Scripts can multiply two matrices via the [matrix.mult\(\)](#) function. This function also facilitates the

multiplication of a matrix by an [array](#) or a scalar value.

In the case of multiplying two matrices, unlike addition and subtraction, matrix multiplication does not require two matrices to share the same shape. However, the number of columns in the first matrix must equal the number of rows in the second one. The resulting matrix returned by [matrix.mult\(\)](#) will contain the same number of rows as `id1` and the same number of columns as `id2`. For instance, a 2x3 matrix multiplied by a 3x4 matrix will produce a matrix with two rows and four columns, as shown below. Each value within the resulting matrix is the [dot product](#) of the corresponding row in `id1` and column in `id2`:



```
//@version=5
indicator("Matrix mult example")

//@function Displays the rows of a matrix in a label with a note.
//@param this The matrix to display.
//@param barIndex The `bar_index` to display the label at.
//@param bgColor The background color of the label.
//@param textColor The color of the label's text.
//@param note The text to display above the rows.
method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""
) =>
    labelText = note + "\n" + str.toString(this)
    if barstate.ishistory
        label.new(
            barIndex, 0, labelText, color = bgColor, style =
label.style_label_center,
            textcolor = textColor, size = size.huge
        )

//@variable A 2x3 matrix.
a = matrix.new<float>()
//@variable A 3x4 matrix.
b = matrix.new<float>()

// Add rows to `a`.
a.add_row(0, array.from(1, 2, 3))
a.add_row(1, array.from(4, 5, 6))

// Add rows to `b`.
b.add_row(0, array.from(0.5, 1.0, 1.5, 2.0))
b.add_row(1, array.from(2.5, 3.0, 3.5, 4.0))
b.add_row(0, array.from(4.5, 5.0, 5.5, 6.0))

if bar_index == last_bar_index - 1
    //@variable The result of `a` * `b`.
    matrix<float> ab = a.mult(b)
    // Display `a`, `b`, and `ab` matrices.
    debugLabel(a, note = "A")
    debugLabel(b, bar_index + 10, note = "B")
    debugLabel(ab, bar_index + 20, color.green, note = "A * B")
```

Note that:

- In contrast to the multiplication of scalars, matrix multiplication is *non-commutative*, i.e., `matrix.mult(a, b)` does not necessarily produce the same result as `matrix.mult(b, a)`. In the context of our example, the latter will raise a runtime error because the number of columns in `b` doesn't equal the number of rows in `a`.

When multiplying a matrix and an [array](#), this function treats the operation the same as multiplying `id1` by a single-column matrix, but it returns an [array](#) with the same number of elements as the number of rows in `id1`. When [matrix.mult\(\)](#) passes a scalar as its `id2` value, the function returns a new matrix whose elements are the elements in `id1` multiplied by the `id2` value.

[matrix.det\(\)](#)

A *determinant* is a scalar value associated with a [square](#) matrix that describes some of its characteristics, namely its invertibility. If a matrix has an [inverse](#), its determinant is nonzero. Otherwise, the matrix is *singular* (non-invertible). Scripts can calculate the determinant of a matrix via [matrix.det\(\)](#).

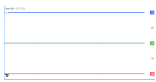
Programmers can use determinants to detect similarities between matrices, identify *full-rank* and *rank-deficient* matrices, and solve systems of linear equations, among other applications.

For example, this script utilizes determinants to solve a system of linear equations with a matching number of unknown values using [Cramer's rule](#). The user-defined `solve()` function returns an [array](#) containing solutions for each unknown value in the system, where the *n*-th element of the [array](#) is the determinant of the coefficient matrix with the *n*-th column replaced by the column of constants divided by the determinant of the original coefficients.

In this script, we've defined the matrix `m` that holds coefficients and constants for these three equations:

$$\begin{aligned} 3 * x_0 + 4 * x_1 - 1 * x_2 &= 8 \\ 5 * x_0 - 2 * x_1 + 1 * x_2 &= 4 \\ 2 * x_0 - 2 * x_1 + 1 * x_2 &= 1 \end{aligned}$$

The solution to this system is ($x_0 = 1$, $x_1 = 2$, $x_2 = 3$). The script calculates these values from `m` via `m.solve()` and plots them on the chart:



```
//@version=5
indicator("Determinants example", "Cramer's Rule")

//@function Solves a system of linear equations with a matching number of
unknowns using Cramer's rule.
//@param this An augmented matrix containing the coefficients for each
unknown and the results of
// the equations. For example, a row containing the values 2, -1, and 3
represents the equation
// `2 * x0 + (-1) * x1 = 3`, where `x0` and `x1` are the unknown values
in the system.
//@returns An array containing solutions for each variable in the system.
solve(matrix<float> this) =>
    //@variable The coefficient matrix for the system of equations.
    matrix<float> coefficients = this.submatrix(from_column = 0, to_column =
this.columns() - 1)
    //@variable The array of resulting constants for each equation.
    array<float> constants = this.col(this.columns() - 1)
    //@variable An array containing solutions for each unknown in the system.
    array<float> result = array.new<float>()

    //@variable The determinant value of the coefficient matrix.
    float baseDet = coefficients.det()
    matrix<float> modified = na
    for col = 0 to coefficients.columns() - 1
        modified := coefficients.copy()
```

```

        modified.add_col(col, constants)
        modified.remove_col(col + 1)

        // Calculate the solution for the column's unknown by dividing the
determinant of `modified` by the `baseDet`.
        result.push(modified.det() / baseDet)

    result

//@variable A 3x4 matrix containing coefficients and results for a system of
three equations.
m = matrix.new<float>()

// Add rows for the following equations:
// Equation 1: 3 * x0 + 4 * x1 - 1 * x2 = 8
// Equation 2: 5 * x0 - 2 * x1 + 1 * x2 = 4
// Equation 3: 2 * x0 - 2 * x1 + 1 * x2 = 1
m.add_row(0, array.from(3.0, 4.0, -1.0, 8.0))
m.add_row(1, array.from(5.0, -2.0, 1.0, 4.0))
m.add_row(2, array.from(2.0, -2.0, 1.0, 1.0))

//@variable An array of solutions to the unknowns in the system of equations
represented by `m`.
solutions = solve(m)

plot(solutions.get(0), "x0", color.red, 3) // Plots 1.
plot(solutions.get(1), "x1", color.green, 3) // Plots 2.
plot(solutions.get(2), "x2", color.blue, 3) // Plots 3.

```

Note that:

- Solving systems of equations is particularly useful for *regression analysis*, e.g., linear and polynomial regression.
- Cramer's rule works fine for small systems of equations. However, it's computationally inefficient on larger systems. Other methods, such as [Gaussian elimination](#), are often preferred for such use cases.

[`matrix.inv\(\)` and `matrix.pinv\(\)`](#)

For any non-singular [square](#) matrix, there is an inverse matrix that yields the [identity](#) matrix when [multiplied](#) by the original. Inverses have utility in various matrix transformations and solving systems of equations. Scripts can calculate the inverse of a matrix **when one exists** via the [matrix.inv\(\)](#) function.

For singular (non-invertible) matrices, one can calculate a generalized inverse ([pseudoinverse](#)), regardless of whether the matrix is square or has a nonzero determinant, via the [matrix.pinv\(\)](#) function. Keep in mind that unlike a true inverse, the product of a pseudoinverse and the original matrix does not necessarily equal the identity matrix unless the original matrix *is invertible*.

The following example forms a 2x2 m matrix from user inputs, then uses the [m.inv\(\)](#) and [m.pinv\(\)](#) methods to calculate the inverse or pseudoinverse of m. The script displays the original matrix, its inverse or pseudoinverse, and their product in labels on the chart:



```

//@version=5
indicator("Inverse example")

// Element inputs for the 2x2 matrix.
float r0c0 = input.float(4.0, "Row 0, Col 0")

```

```

float r0c1 = input.float(3.0, "Row 0, Col 1")
float r1c0 = input.float(2.0, "Row 1, Col 0")
float r1c1 = input.float(1.0, "Row 1, Col 1")

//@function Displays the rows of a matrix in a label with a note.
//@param this The matrix to display.
//@param barIndex The `bar_index` to display the label at.
//@param bgColor The background color of the label.
//@param textColor The color of the label's text.
//@param note The text to display above the rows.
method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""
) =>
    labelText = note + "\n" + str.toString(this)
    if barstate.ishistory
        label.new(
            barIndex, 0, labelText, color = bgColor, style =
label.style_label_center,
            textcolor = textColor, size = size.huge
        )

//@variable A 2x2 matrix of input values.
m = matrix.new<float>()

// Add input values to `m`.
m.add_row(0, array.from(r0c0, r0c1))
m.add_row(1, array.from(r1c0, r1c1))

//@variable Is `true` if `m` is square with a nonzero determinant, indicating
invertibility.
bool isInvertible = m.is_square() and m.det()

//@variable The inverse or pseudoinverse of `m`.
mInverse = isInvertible ? m.inv() : m.pinv()

//@variable The product of `m` and `mInverse`. Returns the identity matrix when
`isInvertible` is `true`.
matrix<float> product = m.mult(mInverse)

if bar_index == last_bar_index - 1
    // Display `m`, `mInverse`, and their `product`.
    m.debugLabel(note = "Original")
    mInverse.debugLabel(bar_index + 10, color.purple, note = isInvertible ?
"Inverse" : "Pseudoinverse")
    product.debugLabel(bar_index + 20, color.green, note = "Product")

```

Note that:

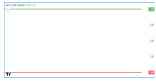
- This script will only call [m.inv\(\)](#) when `isInvertible` is `true`, i.e., when `m` is [square](#) and has a nonzero [determinant](#). Otherwise, it uses [m.pinv\(\)](#) to calculate the generalized inverse.

[`matrix.rank\(\)`](#)

The *rank* of a matrix represents the number of linearly independent vectors (rows or columns) it contains. In essence, matrix rank measures the number of vectors one cannot express as a linear combination of others, or in other words, the number of vectors that contain **unique** information. Scripts can calculate the rank of a matrix via [matrix.rank\(\)](#).

This script identifies the number of linearly independent vectors in two 3x3 matrices (`m1` and `m2`)

and plots the values in a separate pane. As we see on the chart, the `m1.rank()` value is 3 because each vector is unique. The `m2.rank()` value, on the other hand, is 1 because it has just one unique vector:



```
//@version=5
indicator("Matrix rank example")

//@variable A 3x3 full-rank matrix.
m1 = matrix.new<float>()
//@variable A 3x3 rank-deficient matrix.
m2 = matrix.new<float>()

// Add linearly independent vectors to `m1`.
m1.add_row(0, array.from(3, 2, 3))
m1.add_row(1, array.from(4, 6, 6))
m1.add_row(2, array.from(7, 4, 9))

// Add linearly dependent vectors to `m2`.
m2.add_row(0, array.from(1, 2, 3))
m2.add_row(1, array.from(2, 4, 6))
m2.add_row(2, array.from(3, 6, 9))

// Plot `matrix.rank()` values.
plot(m1.rank(), color = color.green, linewidth = 3)
plot(m2.rank(), color = color.red, linewidth = 3)
```

Note that:

- The highest rank value a matrix can have is the minimum of its number of rows and columns. A matrix with the maximum possible rank is known as a *full-rank* matrix, and any matrix without full rank is known as a *rank-deficient* matrix.
- The [determinants](#) of full-rank square matrices are nonzero, and such matrices have [inverses](#). Conversely, the [determinant](#) of a rank-deficient matrix is always 0.
- For any matrix that contains nothing but the same value in each of its elements (e.g., a matrix filled with 0), the rank is always 0 since none of the vectors hold unique information. For any other matrix with distinct values, the minimum possible rank is 1.

Error handling

In addition to usual **compiler** errors, which occur during a script's compilation due to improper syntax, scripts using matrices can raise specific **runtime** errors during their execution. When a script raises a runtime error, it displays a red exclamation point next to the script title. Users can view the error message by clicking this icon.

In this section, we discuss runtime errors that users may encounter while utilizing matrices in their scripts.

The row/column index (xx) is out of bounds, row/column size is (yy).

This runtime error occurs when trying to access indices outside the matrix dimensions with functions including [matrix.get\(\)](#), [matrix.set\(\)](#), [matrix.fill\(\)](#), and [matrix.submatrix\(\)](#), as well as some of the functions relating to the [rows and columns](#) of a matrix.

For example, this code contains two lines that will produce this runtime error. The [m.set\(\)](#) method references a row index that doesn't exist (2). The [m.submatrix\(\)](#) method references all column

indices up to `to_column - 1`. A `to_column` value of 4 results in a runtime error because the last column index referenced (3) does not exist in `m`:

```
//@version=5
indicator("Out of bounds demo")

//@variable A 2x3 matrix with a max row index of 1 and max column index of 2.
matrix<float> m = matrix.new<float>(2, 3, 0.0)

m.set(row = 2, column = 0, value = 1.0) // The `row` index is out of bounds
on this line. The max value is 1.
m.submatrix(from_column = 1, to_column = 4) // The `to_column` index is invalid
on this line. The max value is 3.

if bar_index == last_bar_index - 1
  label.new(bar_index, 0, str.tostring(m), color = color.navy, textcolor =
color.white, size = size.huge)
```

Users can avoid this error in their scripts by ensuring their function calls do not reference indices greater than or equal to the number of rows/columns.

The array size does not match the number of rows/columns in the matrix.

When using `matrix.add_row()` and `matrix.add_col()` functions to [insert](#) rows and columns into a non-empty matrix, the size of the inserted array must align with the matrix dimensions. The size of an inserted row must match the number of columns, and the size of an inserted column must match the number of rows. Otherwise, the script will raise this runtime error. For example:

```
//@version=5
indicator("Invalid array size demo")

// Declare an empty matrix.
m = matrix.new<float>()

m.add_col(0, array.from(1, 2)) // Add a column. Changes the shape of `m` to
2x1.
m.add_col(1, array.from(1, 2, 3)) // Raises a runtime error because `m` has 2
rows, not 3.

plot(m.col(0).get(1))
```

Note that:

- When `m` is empty, one can insert a row or column array of *any* size, as shown in the first `m.add_col()` line.

Cannot call matrix methods when the ID of matrix is 'na'.

When a matrix variable is assigned to `na`, it means that the variable doesn't reference an existing object. Consequently, one cannot use built-in `matrix.*()` functions and methods with it. For example:

```
//@version=5
indicator("na matrix methods demo")

//@variable A `matrix` variable assigned to `na`.
matrix<float> m = na

mCopy = m.copy() // Raises a runtime error. You can't copy a matrix that doesn't
exist.
```



```

if bar_index == last_bar_index - 1
    label.new(bar_index, 0, str.toString(mCopy), color = color.navy, textcolor =
color.white, size = size.huge)

```

To resolve this error, assign `m` to a valid matrix instance before using `matrix.*()` functions.

Matrix is too large. Maximum size of the matrix is 100,000 elements.

The total number of elements in a matrix (`matrix.elements_count()`) cannot exceed **100,000**, regardless of its shape. For example, this script will raise an error because it inserts 1000 rows with 101 elements into the `m` matrix:

```

//@version=5
indicator("Matrix too large demo")

var matrix<float> m = matrix.new<float>()

if bar_index == 0
    for i = 1 to 1000
        // This raises an error because the script adds 101 elements on each
iteration.
        // 1000 rows * 101 elements per row = 101000 total elements. This is too
large.
        m.add_row(m.rows(), array.new<float>(101, i))

plot(m.get(0, 0))

```

The row/column index must be 0 <= from_row/column < to_row/column.

When using `matrix.*()` functions with `from_row/column` and `to_row/column` indices, the `from_*` values must be less than the corresponding `to_*` values, with the minimum possible value being 0. Otherwise, the script will raise a runtime error.

For example, this script shows an attempt to declare a submatrix from a 4x4 `m` matrix with a `from_row` value of 2 and a `to_row` value of 2, which will result in an error:

```

//@version=5
indicator("Invalid from_row, to_row demo")

//@variable A 4x4 matrix filled with a random value.
matrix<float> m = matrix.new<float>(4, 4, math.random())

matrix<float> mSub = m.submatrix(from_row = 2, to_row = 2) // Raises an error.
`from_row` can't equal `to_row`.

plot(mSub.get(0, 0))

```

Matrices 'id1' and 'id2' must have an equal number of rows and columns to be added.

When using `matrix.sum()` and `matrix.diff()` functions, the `id1` and `id2` matrices must have the same number of rows and the same number of columns. Attempting to add or subtract two matrices with mismatched dimensions will raise an error, as demonstrated by this code:

```

//@version=5
indicator("Invalid sum dimensions demo")

//@variable A 2x3 matrix.

```

```

matrix<float> m1 = matrix.new<float>(2, 3, 1)
//@variable A 3x4 matrix.
matrix<float> m2 = matrix.new<float>(3, 4, 2)

mSum = matrix.sum(m1, m2) // Raises an error. `m1` and `m2` don't have matching
dimensions.

plot(mSum.get(0, 0))

```

The number of columns in the ‘id1’ matrix must equal the number of rows in the matrix (or the number of elements in the array) ‘id2’.

When using [matrix.mult\(\)](#) to multiply an id1 matrix by an id2 matrix or array, the [matrix.rows\(\)](#) or [array.size\(\)](#) of id2 must equal the [matrix.columns\(\)](#) in id1. If they don't align, the script will raise this error.

For example, this script tries to multiply two 2x3 matrices. While *adding* these matrices is possible, *multiplying* them is not:

```

//@version=5
indicator("Invalid mult dimensions demo")

//@variable A 2x3 matrix.
matrix<float> m1 = matrix.new<float>(2, 3, 1)
//@variable A 2x3 matrix.
matrix<float> m2 = matrix.new<float>(2, 3, 2)

mSum = matrix.mult(m1, m2) // Raises an error. The number of columns in `m1` and
rows in `m2` aren't equal.

plot(mSum.get(0, 0))

```

Operation not available for non-square matrices.

Some matrix operations, including [matrix.inv\(\)](#), [matrix.det\(\)](#), [matrix.eigenvalues\(\)](#), and [matrix.eigenvectors\(\)](#) only work with **square** matrices, i.e., matrices with the same number of rows and columns. When attempting to execute such functions on non-square matrices, the script will raise an error stating the operation isn't available or that it cannot calculate the result for the matrix id. For example:

```

//@version=5
indicator("Non-square demo")

//@variable A 3x5 matrix.
matrix<float> m = matrix.new<float>(3, 5, 1)

plot(m.det()) // Raises a runtime error. You can't calculate the determinant of
a 3x5 matrix.

```

Maps

- [Introduction](#)
- [Declaring a map](#)
 - [Using `var` and `varip` keywords](#)
- [Reading and writing](#)
 - [Putting and getting key-value pairs](#)

- [Inspecting keys and values](#)
 - [`map.keys\(\)` and `map.values\(\)`](#)
 - [`map.contains\(\)`](#)
- [Removing key-value pairs](#)
- [Combining maps](#)
- [Looping through a map](#)
- [Copying a map](#)
 - [Shallow copies](#)
 - [Deep copies](#)
- [Scope and history](#)
- [Maps of other collections](#)

Note

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

[Introduction](#)

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

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

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

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

[Declaring a map](#)

Pine Script[®] uses the following syntax to declare maps:

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

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

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

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

```
map<string, float> myMap = na
```

When the `<expression>` is not `na`, the compiler does not require explicit type declaration, as it

will infer the type information from the assigned map object.

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

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

Using ``var`` and ``varip`` keywords

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

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



```
//@version=5
indicator("var map demo")

//@variable A map associating color values with string keys.
var colorMap = map.new<string, color>()

// Put `<string, color>` pairs into `colorMap` on the first bar.
if bar_index == 0
    colorMap.put("Bull", color.green)
    colorMap.put("Bear", color.red)
    colorMap.put("Neutral", color.gray)

//@variable The 14-bar RSI of `close`.
float oscillator = ta.rsi(close, 14)

//@variable The color of the `oscillator`.
color oscColor = switch
    oscillator > 50 => colorMap.get("Bull")
    oscillator < 50 => colorMap.get("Bear")
    => colorMap.get("Neutral")

// Plot the `oscillator` using the `oscColor` from our `colorMap`.
plot(oscillator, "Histogram", oscColor, 2, plot.style_histogram, histbase = 50)
plot(oscillator, "Line", oscColor, 3)
```

Note

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

Reading and writing

Putting and getting key-value pairs

The [map.put\(\)](#) function is one that map users will utilize quite often, as it's the primary method to

put a new key-value pair into a map. It associates the `key` argument with the `value` argument in the call and adds the pair to the map `id`.

If the `key` argument in the `map.put()` call already exists in the map's `keys`, the new pair passed into the function will **replace** the existing one.

To retrieve the value from a map `id` associated with a given `key`, use `map.get()`. This function returns the value if the `id` map **contains** the `key`. Otherwise, it returns `na`.

The following example calculates the difference between the `bar_index` values from when `close` was last **rising** and **falling** over a given `length` with the help of `map.put()` and `map.get()` methods. The script puts a ("Rising", `bar_index`) pair into the data map when the price is rising and puts a ("Falling", `bar_index`) pair into the map when the price is falling. It then puts a pair containing the "Difference" between the "Rising" and "Falling" values into the map and plots its value on the chart:



```
//@version=5
indicator("Putting and getting demo")

//@variable The length of the `ta.rising()` and `ta.falling()` calculation.
int length = input.int(2, "Length")

//@variable A map associating `string` keys with `int` values.
var data = map.new<string, int>()

// Put a new ("Rising", `bar_index`) pair into the `data` map when `close` is
rising.
if ta.rising(close, length)
    data.put("Rising", bar_index)
// Put a new ("Falling", `bar_index`) pair into the `data` map when `close` is
falling.
if ta.falling(close, length)
    data.put("Falling", bar_index)

// Put the "Difference" between current "Rising" and "Falling" values into the
`data` map.
data.put("Difference", data.get("Rising") - data.get("Falling"))

//@variable The difference between the last "Rising" and "Falling" `bar_index`.
int index = data.get("Difference")

//@variable Returns `color.green` when `index` is positive, `color.red` when
negative, and `color.gray` otherwise.
color indexColor = index > 0 ? color.green : index < 0 ? color.red : color.gray

plot(index, color = indexColor, style = plot.style_columns)
```

Note that:

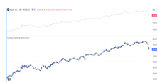
- This script replaces the values associated with the "Rising", "Falling", and "Difference" keys on successive `data.put()` calls, as each of these keys is unique and can only appear once in the data map.
- Replacing the pairs in a map does not change the internal *insertion order* of its keys. We discuss this further in the [next section](#).

Similar to working with other collections, when putting a value of a *special type* ([line](#), [linefill](#), [label](#), [box](#), or [table](#)) or a *user-defined type* into a map, it's important to note the inserted pair's value

points to that same object without copying it. Modifying the value referenced by a key-value pair will also affect the original object.

For example, this script contains a custom `ChartData` type with `o`, `h`, `l`, and `c` fields. On the first chart bar, the script declares a `myMap` variable and adds the pair `("A", myData)`, where `myData` is a `ChartData` instance with initial field values of `na`. It adds the pair `("B", myData)` to `myMap` and updates the object from this pair on every bar via the user-defined `update()` method.

Each change to the object with the “B” key affects the one referenced by the “A” key, as shown by the candle plot of the “A” object’s fields:



```
//@version=5
indicator("Putting and getting objects demo")

//@type A custom type to hold OHLC data.
type ChartData
    float o
    float h
    float l
    float c

//@function Updates the fields of a `ChartData` object.
method update(ChartData this) =>
    this.o := open
    this.h := high
    this.l := low
    this.c := close

//@variable A new `ChartData` instance declared on the first bar.
var myData = ChartData.new()
//@variable A map associating `string` keys with `ChartData` instances.
var myMap = map.new<string, ChartData>()

// Put a new pair with the "A" key into `myMap` only on the first bar.
if bar_index == 0
    myMap.put("A", myData)

// Put a pair with the "B" key into `myMap` on every bar.
myMap.put("B", myData)

//@variable The `ChartData` value associated with the "A" key in `myMap`.
ChartData oldest = myMap.get("A")
//@variable The `ChartData` value associated with the "B" key in `myMap`.
ChartData newest = myMap.get("B")

// Update `newest`. Also affects `oldest` and `myData` since they all reference
the same `ChartData` object.
newest.update()

// Plot the fields of `oldest` as candles.
plotcandle(oldest.o, oldest.h, oldest.l, oldest.c)
```

Note that:

- This script would behave differently if it passed a copy of `myData` into each `myMap.put()` call. For more information, see [this](#) section of our User Manual’s page on [objects](#).

Inspecting keys and values

`map.keys()` and `map.values()`

To retrieve all keys and values put into a map, use [map.keys\(\)](#) and [map.values\(\)](#). These functions copy all key/value references within a map `id` to a new [array](#) object. Modifying the array returned from either of these functions does not affect the `id` map.

Although maps are *unordered* collections, Pine Script® internally maintains the *insertion order* of a map's key-value pairs. As a result, the [map.keys\(\)](#) and [map.values\(\)](#) functions always return [arrays](#) with their elements ordered based on the `id` map's insertion order.

The script below demonstrates this by displaying the key and value arrays from an `m` map in a [label](#) once every 50 bars. As we see on the chart, the order of elements in each array returned by `m.keys()` and `m.values()` aligns with the insertion order of the key-value pairs in `m`:



```
//@version=5
indicator("Keys and values demo")

if bar_index % 50 == 0
    //@variable A map containing pairs of `string` keys and `float` values.
    m = map.new<string, float>()

    // Put pairs into `m`. The map will maintain this insertion order.
    m.put("First", math.round(math.random(0, 100)))
    m.put("Second", m.get("First") + 1)
    m.put("Third", m.get("Second") + 1)

    //@variable An array containing the keys of `m` in their insertion order.
    array<string> keys = m.keys()
    //@variable An array containing the values of `m` in their insertion order.
    array<float> values = m.values()

    //@variable A label displaying the `size` of `m` and the `keys` and `values`
arrays.
    label debugLabel = label.new(
        bar_index, 0,
        str.format("Pairs: {0}\nKeys: {1}\nValues: {2}", m.size(), keys,
values),
        color = color.navy, style = label.style_label_center,
        textcolor = color.white, size = size.huge
    )
```

Note that:

- The value with the “First” key is a [random](#) whole number between 0 and 100. The “Second” value is one greater than the “First”, and the “Third” value is one greater than the “Second”.

It's important to note a map's internal insertion order **does not** change when replacing its key-value pairs. The locations of the new elements in the [keys\(\)](#) and [values\(\)](#) arrays will be the same as the old elements in such cases. The only exception is if the script completely [removes](#) the key beforehand.

Below, we've added a line of code to [put](#) a new value with the “Second” key into the `m` map, overwriting the previous value associated with that key. Although the script puts this new pair into the map *after* the one with the “Third” key, the pair's key and value are still second in the `keys` and `values` arrays since the key was already present in `m` *before* the change:



```
//@version=5
indicator("Keys and values demo")

if bar_index % 50 == 0
    //@variable A map containing pairs of `string` keys and `float` values.
    m = map.new<string, float>()

    // Put pairs into `m`. The map will maintain this insertion order.
    m.put("First", math.round(math.random(0, 100)))
    m.put("Second", m.get("First") + 1)
    m.put("Third", m.get("Second") + 1)

    // Overwrite the "Second" pair in `m`. This will NOT affect the insertion
    order.
    // The key and value will still appear second in the `keys` and `values`
    arrays.
    m.put("Second", -2)

    //@variable An array containing the keys of `m` in their insertion order.
    array<string> keys = m.keys()
    //@variable An array containing the values of `m` in their insertion order.
    array<float> values = m.values()

    //@variable A label displaying the `size` of `m` and the `keys` and `values`
    arrays.
    label debugLabel = label.new(
        bar_index, 0,
        str.format("Pairs: {0}\nKeys: {1}\nValues: {2}", m.size(), keys,
values),
        color = color.navy, style = label.style_label_center,
        textcolor = color.white, size = size.huge
    )
```

Note

The elements in a [map.values\(\)](#) array point to the same values as the map id. Consequently, when the map's values are of *reference types*, including [line](#), [linefill](#), [label](#), [box](#), [table](#), or [UDTs](#), modifying the instances referenced by the [map.values\(\)](#) array will also affect those referenced by the map id since the contents of both collections point to identical objects.

[map.contains\(\)](#)

To check if a specific key exists within a map id, use [map.contains\(\)](#). This function is a convenient alternative to calling [array.includes\(\)](#) on the [array](#) returned from [map.keys\(\)](#).

For example, this script checks if various keys exist within an m map, then displays the results in a [label](#):

Inspecting keys demo

Tested keys: [A, B, C, D, E, F]
Keys found: [A, C, E]



```
//@version=5
indicator("Inspecting keys demo")

//@variable A map containing `string` keys and `string` values.
m = map.new<string, string>()

// Put key-value pairs into the map.
m.put("A", "B")
m.put("C", "D")
m.put("E", "F")

//@variable An array of keys to check for in `m`.
array<string> testKeys = array.from("A", "B", "C", "D", "E", "F")

//@variable An array containing all elements from `testKeys` found in the keys
of `m`.
array<string> mappedKeys = array.new<string>()

for key in testKeys
    // Add the `key` to `mappedKeys` if `m` contains it.
    if m.contains(key)
        mappedKeys.push(key)

//@variable A string representing the `testKeys` array and the elements found
within the keys of `m`.
string testText = str.format("Tested keys: {0}\nKeys found: {1}", testKeys,
mappedKeys)

if bar_index == last_bar_index - 1
    //@variable Displays the `testText` in a label at the `bar_index` before the
last.
    label debugLabel = label.new(
        bar_index, 0, testText, style = label.style_label_center,
        textcolor = color.white, size = size.huge
    )
```

Removing key-value pairs

To remove a specific key-value pair from a map `id`, use [map.remove\(\)](#). This function removes the key and its associated value from the map while preserving the insertion order of other key-value pairs. It returns the removed value if the map [contained](#) the key. Otherwise, it returns [na](#).

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

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



```
//@version=5
indicator("Removing key-value pairs demo")

//@function Returns a label to display the keys and values from a map.
method debugLabel(
    map<string, int> this, int barIndex = bar_index,
    color bgColor = color.blue, string note = ""
) =>
    //@variable A string representing the size, keys, and values in `this` map.
    string repr = str.format(
        "{0}\nSize: {1}\nKeys: {2}\nValues: {3}",
        note, this.size(), str.toString(this.keys()),
        str.toString(this.values())
    )
    label.new(
        barIndex, 0, repr, color = bgColor, style = label.style_label_center,
        textcolor = color.white, size = size.huge
    )

if bar_index == last_bar_index - 1
    //@variable A map containing `string` keys and `int` values.
    m = map.new<string, int>()

    // Put key-value pairs into `m`.
    for [i, key] in array.from("A", "B", "C", "D", "E")
        m.put(key, i)
    m.debugLabel(bar_index, color.green, "Added pairs")

    //@variable An array of keys to remove from `m`.
    array<string> removeKeys = array.from("B", "B", "D", "F", "a")

    // Remove each `key` in `removeKeys` from `m`.
    for key in removeKeys
        m.remove(key)
    m.debugLabel(bar_index + 10, color.red, "Removed pairs")

    // Remove all remaining keys from `m`.
    m.clear()
    m.debugLabel(bar_index + 20, color.purple, "Cleared the map")
```

Note that:

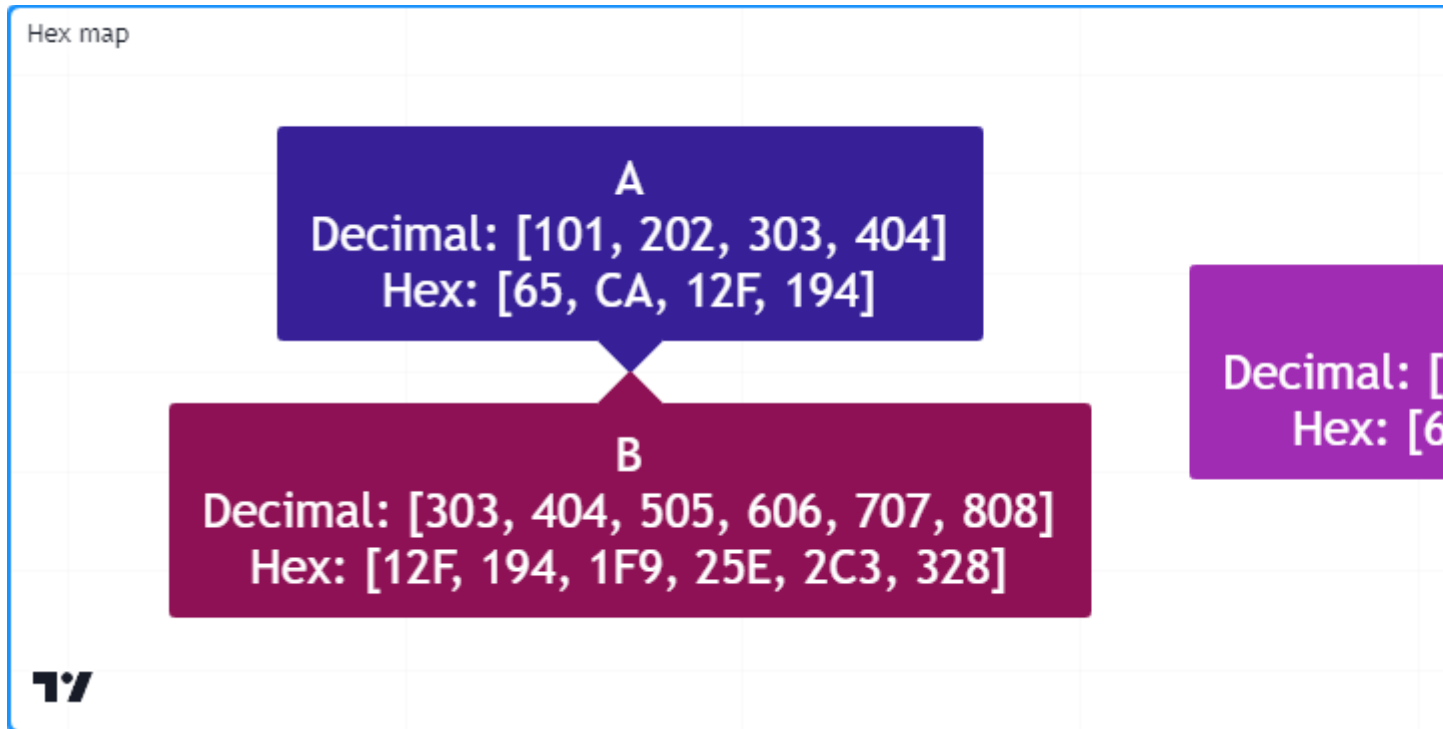
- Not all strings in the `removeKeys` array were present in the keys of `m`. Attempting to remove non-existent keys (“F”, “a”, and the second “B” in this example) has no effect on a map’s contents.

Combining maps

Scripts can combine two maps via [map.put_all\(\)](#). This function puts *all* key-value pairs from the `id2` map, in their insertion order, into the `id1` map. As with [map.put\(\)](#), if any keys in `id2` are also present in `id1`, this function **replaces** the key-value pairs that contain those keys without affecting their initial insertion order.

This example contains a user-defined `hexMap()` function that maps decimal [int](#) keys to [string](#) representations of their [hexadecimal](#) forms. The script uses this function to create two maps, `mapA` and `mapB`, then uses `mapA.put_all(mapB)` to put all key-value pairs from `mapB` into `mapA`.

The script uses a custom `debugLabel()` function to display labels showing the [keys](#) and [values](#) of `mapA` and `mapB`, then another label displaying the contents of `mapA` after putting all key-value pairs from `mapB` into it:



```
//@version=5
indicator("Combining maps demo", "Hex map")

//@variable An array of string hex digits.
var array<string> hexDigits = str.split("0123456789ABCDEF", "")

//@function Returns a hexadecimal string for the specified `value`.
hex(int value) =>
    //@variable A string representing the hex form of the `value`.
    string result = ""
    //@variable A temporary value for digit calculation.
    int tempValue = value
    while tempValue > 0
        //@variable The next integer digit.
        int digit = tempValue % 16
        // Add the hex form of the `digit` to the `result`.
        result := hexDigits.get(digit) + result
        // Divide the `tempValue` by the base.
        tempValue := int(tempValue / 16)
    result

//@function Returns a map holding the `numbers` as keys and their `hex` strings
as values.
hexMap(array<int> numbers) =>
    //@variable A map associating `int` keys with `string` values.
    result = map.new<int, string>()
    for number in numbers
        // Put a pair containing the `number` and its `hex()` representation
into the `result`.
        result.put(number, hex(number))
```

```

result

//@function Returns a label to display the keys and values of a hex map.
debugLabel(
    map<int, string> this, int barIndex = bar_index, color bgColor =
color.blue,
    string style = label.style_label_center, string note = ""
) =>
    string repr = str.format(
        "{0}\nDecimal: {1}\nHex: {2}",
        note, str.toString(this.keys()), str.toString(this.values())
    )
    label.new(
        barIndex, 0, repr, color = bgColor, style = style,
        textcolor = color.white, size = size.huge
    )

if bar_index == last_bar_index - 1
    //@variable A map with decimal `int` keys and hexadecimal `string` values.
    map<int, string> mapA = hexMap(array.from(101, 202, 303, 404))
    debugLabel(mapA, bar_index, color.navy, label.style_label_down, "A")

    //@variable A map containing key-value pairs to add to `mapA`.
    map<int, string> mapB = hexMap(array.from(303, 404, 505, 606, 707, 808))
    debugLabel(mapB, bar_index, color.maroon, label.style_label_up, "B")

    // Put all pairs from `mapB` into `mapA`.
    mapA.put_all(mapB)
    debugLabel(mapA, bar_index + 10, color.purple, note = "Merge B into A")

```

Looping through a map

There are several ways scripts can iteratively access the keys and values in a map. For example, one could loop through a map's [keys\(\)](#) array and [get\(\)](#) the value for each key, like so:

```

for key in thisMap.keys()
    value = thisMap.get(key)

```

However, we recommend using a `for . . . in` loop directly on a map, as it iterates over the map's key-value pairs in their insertion order, returning a tuple containing the next pair's key and value on each iteration.

For example, this line of code loops through each key and value in `thisMap`, starting from the first key-value pair put into it:

```

for [key, value] in thisMap

```

Let's use this structure to write a script that displays a map's key-value pairs in a [table](#). In the example below, we've defined a custom `toTable()` method that creates a [table](#), then uses a `for . . . in` loop to iterate over the map's key-value pairs and populate the table's cells. The script uses this method to visualize a map containing length-bar averages of price and volume data:



```

//@version=5
indicator("Looping through a map demo", "Table of averages")

//@variable The length of the moving average.
int length = input.int(20, "Length")

```

```

//@variable The size of the table text.
string txtSize = input.string(
    size.huge, "Text size",
    options = [size.auto, size.tiny, size.small, size.normal, size.large,
size.huge]
)

//@function Displays the pairs of `this` map within a table.
//@param    this A map with `string` keys and `float` values.
//@param    position The position of the table on the chart.
//@param    header The string to display on the top row of the table.
//@param    textSize The size of the text in the table.
//@returns  A new `table` object with cells displaying each pair in `this`.
method toTable(
    map<string, float> this, string position = position.middle_center, string
header = na,
    string textSize = size.huge
) =>
    // Color variables
    borderColor = #000000
    headerColor = color.rgb(1, 88, 80)
    pairColor    = color.maroon
    textColor    = color.white

    //@variable A table that displays the key-value pairs of `this` map.
    table result = table.new(
        position, this.size() + 1, 3, border_width = 2, border_color =
borderColor
    )
    // Initialize top and side header cells.
    result.cell(1, 0, header, bgcolor = headerColor, text_color = textColor,
text_size = textSize)
    result.merge_cells(1, 0, this.size(), 0)
    result.cell(0, 1, "Key", bgcolor = headerColor, text_color = textColor,
text_size = textSize)
    result.cell(0, 2, "Value", bgcolor = headerColor, text_color = textColor,
text_size = textSize)

    //@variable The column index of the table. Updates on each loop iteration.
    int col = 1

    // Loop over each `key` and `value` from `this` map in the insertion order.
    for [key, value] in this
        // Initialize a `key` cell in the `result` table on row 1.
        result.cell(
            col, 1, str.toString(key), bgcolor = color.maroon,
            text_color = color.white, text_size = textSize
        )
        // Initialize a `value` cell in the `result` table on row 2.
        result.cell(
            col, 2, str.toString(value), bgcolor = color.maroon,
            text_color = color.white, text_size = textSize
        )
        // Move to the next column index.
        col += 1
    result // Return the `result` table.

//@variable A map with `string` keys and `float` values to hold `length`-bar
averages.
averages = map.new<string, float>()

// Put key-value pairs into the `averages` map.
averages.put("Open", ta.sma(open, length))
averages.put("High", ta.sma(high, length))

```

```

averages.put("Low", ta.sma(low, length))
averages.put("Close", ta.sma(close, length))
averages.put("Volume", ta.sma(volume, length))

//@variable The text to display at the top of the table.
string headerText = str.format("{0} {1}-bar averages", "" + syminfo.tickerid +
"", length)
// Display the `averages` map in a `table` with the `headerText`.
averages.toTable(header = headerText, textSize = txtSize)

```

Copying a map

Shallow copies

Scripts can make a *shallow copy* of a map `id` using the [map.copy\(\)](#) function. Modifications to a shallow copy do not affect the original `id` map or its internal insertion order.

For example, this script constructs an `m` map with the keys “A”, “B”, “C”, and “D” assigned to four [random](#) values between 0 and 10. It then creates an `mCopy` map as a shallow copy of `m` and updates the values associated with its keys. The script displays the key-value pairs in `m` and `mCopy` on the chart using our custom `debugLabel()` method:



```

//@version=5
indicator("Shallow copy demo")

//@function Displays the key-value pairs of `this` map in a label.
method debugLabel(
    map<string, float> this, int barIndex = bar_index, color bgColor =
color.blue,
    color textColor = color.white, string note = ""
) =>
    //@variable The text to display in the label.
    labelText = note + "\n{"
    for [key, value] in this
        labelText += str.format("{0}: {1}, ", key, value)
    labelText := str.replace(labelText, ", ", "}", this.size() - 1)

    if barstate.ishistory
        label result = label.new(
            barIndex, 0, labelText, color = bgColor, style =
label.style_label_center,
            textcolor = textColor, size = size.huge
        )

if bar_index == last_bar_index - 1
    //@variable A map of `string` keys and random `float` values.
    m = map.new<string, float>()

    // Assign random values to an array of keys in `m`.
    for key in array.from("A", "B", "C", "D")
        m.put(key, math.random(0, 10))

    //@variable A shallow copy of `m`.
    mCopy = m.copy()

    // Assign the insertion order value `i` to each `key` in `mCopy`.
    for [i, key] in mCopy.keys()

```

```

mCopy.put(key, i)

// Display the labels.
m.debugLabel(bar_index, note = "Original")
mCopy.debugLabel(bar_index + 10, color.purple, note = "Copied and changed")

```

Deep copies

While a [shallow copy](#) will suffice when copying maps that have values of a *fundamental type*, it's important to remember that shallow copies of a map holding values of a *reference type* ([line](#), [linefill](#), [label](#), [box](#), [table](#), or a [UDT](#)) point to the same objects as the original. Modifying the objects referenced by a shallow copy will affect the instances referenced by the original map and vice versa.

To ensure changes to objects referenced by a copied map do not affect instances referenced in other locations, one can make a *deep copy* by creating a new map with key-value pairs containing copies of each value in the original map.

This example creates an original map of [string](#) keys and [label](#) values and [puts](#) a key-value pair into it. The script copies the map to a shallow variable via the built-in [copy\(\)](#) method, then to a deep variable using a custom `deepCopy()` method.

As we see from the chart, changes to the label retrieved from the shallow copy also affect the instance referenced by the original map, but changes to the one from the deep copy do not:



```

//@version=5
indicator("Deep copy demo")

//@function Returns a deep copy of `this` map.
method deepCopy(map<string, label> this) =>
    //@variable A deep copy of `this` map.
    result = map.new<string, label>()
    // Add key-value pairs with copies of each `value` to the `result`.
    for [key, value] in this
        result.put(key, value.copy())
    result //Return the `result`.

//@variable A map containing `string` keys and `label` values.
var original = map.new<string, label>()

if bar_index == last_bar_index - 1
    // Put a new key-value pair into the `original` map.
    map.put(
        original, "Test",
        label.new(bar_index, 0, "Original", textcolor = color.white, size =
size.huge)
    )

    //@variable A shallow copy of the `original` map.
    map<string, label> shallow = original.copy()
    //@variable A deep copy of the `original` map.
    map<string, label> deep = original.deepCopy()

    //@variable The "Test" label from the `shallow` copy.
    label shallowLabel = shallow.get("Test")
    //@variable The "Test" label from the `deep` copy.
    label deepLabel = deep.get("Test")

```

```

// Modify the "Test" label's `y` attribute in the `original` map.
// This also affects the `shallowLabel`.
original.get("Test").set_y(label.all.size())

// Modify the `shallowLabel`. Also modifies the "Test" label in the
`original` map.
shallowLabel.set_text("Shallow copy")
shallowLabel.set_color(color.red)
shallowLabel.set_style(label.style_label_up)

// Modify the `deepLabel`. Does not modify any other label instance.
deepLabel.set_text("Deep copy")
deepLabel.set_color(color.navy)
deepLabel.set_style(label.style_label_left)
deepLabel.set_x(bar_index + 5)

```

Note that:

- The `deepCopy()` method loops through the original map, copying each value and [putting](#) key-value pairs containing the copies into a [new](#) map instance.

Scope and history

As with other collections in Pine, map variables leave historical trails on each bar, allowing a script to access past map instances assigned to a variable using the history-referencing operator `[]`. Scripts can also assign maps to global variables and interact with them from the scopes of [functions](#), [methods](#), and [conditional structures](#).

As an example, this script uses a global map and its history to calculate an aggregate set of [EMAs](#). It declares a `globalData` map of [int](#) keys and [float](#) values, where each key in the map corresponds to the length of each EMA calculation. The user-defined `update()` function calculates each key-length EMA by mixing the values from the previous map assigned to `globalData` with the current source value.

The script plots the [maximum](#) and [minimum](#) values in the global map's `values()` array and the value from `globalData.get(50)` (i.e., the 50-bar EMA):



```

//@version=5
indicator("Scope and history demo", overlay = true)

//@variable The source value for EMA calculation.
float source = input.source(close, "Source")

//@variable A map containing global key-value pairs.
globalData = map.new<int, float>()

//@function Calculates a set of EMAs and updates the key-value pairs in
`globalData`.
update() =>
    //@variable The previous map instance assigned to `globalData`.
    map<int, float> previous = globalData[1]

    // Put key-value pairs with keys 10-200 into `globalData` if `previous` is
`na`.
    if na(previous)
        for i = 10 to 200
            globalData.put(i, source)

```



```

else
    // Iterate each `key` and `value` in the `previous` map.
    for [key, value] in previous
        //@variable The smoothing parameter for the `key`-length EMA.
        float alpha = 2.0 / (key + 1.0)
        //@variable The `key`-length EMA value.
        float ema = (1.0 - alpha) * value + alpha * source
        // Put the `key`-length `ema` into the `globalData` map.
        globalData.put(key, ema)

// Update the `globalData` map.
update()

//@variable The array of values from `globalData` in their insertion order.
array<float> values = globalData.values()

// Plot the max EMA, min EMA, and 50-bar EMA values.
plot(values.max(), "Max EMA", color.green, 2)
plot(values.min(), "Min EMA", color.red, 2)
plot(globalData.get(50), "50-bar EMA", color.orange, 3)

```

Maps of other collections

Maps cannot directly use other maps, [arrays](#), or [matrices](#) as values, but they can hold values of a [user-defined type](#) that contains collections within its fields.

For example, suppose we want to create a “2D” map that uses [string](#) keys to access *nested maps* that hold pairs of [string](#) keys and [float](#) values. Since maps cannot use other collections as values, we will first create a *wrapper type* with a field to hold a `map<string, float>` instance, like so:

```

//@type A wrapper type for maps with `string` keys and `float` values.
type Wrapper
    map<string, float> data

```

With our `Wrapper` type defined, we can create maps of [string](#) keys and `Wrapper` values, where the `data` field of each value in the map points to a `map<string, float>` instance:

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

The script below uses this concept to construct a map containing maps that hold OHLCV data requested from multiple tickers. The user-defined `requestData()` function requests price and volume data from a ticker, creates a `<string, float>` map, [puts](#) the data into it, then returns a `Wrapper` instance containing the new map.

The script [puts](#) the results from each call to `requestData()` into the `mapOfMaps`, then creates a [string](#) representation of the nested maps with a user-defined `toString()` method, which it displays on the chart in a [label](#):



```

//@version=5
indicator("Nested map demo")

//@variable The timeframe of the requested data.
string tf = input.timeframe("D", "Timeframe")
// Symbol inputs.
string symbol1 = input.symbol("EURUSD", "Symbol 1")
string symbol2 = input.symbol("GBPUSD", "Symbol 2")
string symbol3 = input.symbol("EURGBP", "Symbol 3")

```

```

//@type A wrapper type for maps with `string` keys and `float` values.
type Wrapper
    map<string, float> data

//@function Returns a wrapped map containing OHLCV data from the `tickerID` at
the `timeframe`.
requestData(string tickerID, string timeframe) =>
    // Request a tuple of OHLCV values from the specified ticker and timeframe.
    [o, h, l, c, v] = request.security(
        tickerID, timeframe,
        [open, high, low, close, volume]
    )
    //@variable A map containing requested OHLCV data.
    result = map.new<string, float>()
    // Put key-value pairs into the `result`.
    result.put("Open", o)
    result.put("High", h)
    result.put("Low", l)
    result.put("Close", c)
    result.put("Volume", v)
    //Return the wrapped `result`.
    Wrapper.new(result)

//@function Returns a string representing `this` map of `string` keys and
`Wrapper` values.
method toString(map<string, Wrapper> this) =>
    //@variable A string representation of `this` map.
    string result = "{"

    // Iterate over each `key1` and associated `wrapper` in `this`.
    for [key1, wrapper] in this
        // Add `key1` to the `result`.
        result += key1

        //@variable A string representation of the `wrapper.data` map.
        string innerStr = ": {"
        // Iterate over each `key2` and associated `value` in the wrapped map.
        for [key2, value] in wrapper.data
            // Add the key-value pair's representation to `innerStr`.
            innerStr += str.format("{0}: {1}, ", key2, str.toString(value))

        // Replace the end of `innerStr` with `}"` and add to `result`.
        result += str.replace(innerStr, ", ", "},\n", wrapper.data.size() - 1)

    // Replace the blank line at the end of `result` with `}"`.
    result := str.replace(result, ",\n", "}", this.size() - 1)
    result

//@variable A map of wrapped maps containing OHLCV data from multiple tickers.
var mapOfMaps = map.new<string, Wrapper>()

//@variable A label showing the contents of the `mapOfMaps`.
var debugLabel = label.new(
    bar_index, 0, color = color.navy, textcolor = color.white, size =
size.huge,
    style = label.style_label_center, text_font_family = font.family_monospace
)

// Put wrapped maps into `mapOfMaps`.
mapOfMaps.put(symbol1, requestData(symbol1, tf))
mapOfMaps.put(symbol2, requestData(symbol2, tf))
mapOfMaps.put(symbol3, requestData(symbol3, tf))

```

```
// Update the label.
debugLabel.set_text(mapOfMaps.toString())
debugLabel.set_x(bar_index)
```

Alerts

- [Introduction](#)
 - [Background](#)
 - [Which type of alert is best?](#)
- [Script alerts](#)
 - [`alert\(\)` function events](#)
 - [Using all `alert\(\)` calls](#)
 - [Using selective `alert\(\)` calls](#)
 - [In strategies](#)
 - [Order fill events](#)
- [`alertcondition\(\)` events](#)
 - [Using one condition](#)
 - [Using compound conditions](#)
 - [Placeholders](#)
- [Avoiding repainting with 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 the "series" form, it can be generated at runtime and differ bar to bar, making it dynamic.

freq

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

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

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

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

Using all `alert()` calls

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

```
//@version=5
indicator("All `alert()` calls")
r = ta.rsi(close, 20)

// Detect crosses.
xUp = ta.crossover( r, 50)
xDn = ta.crossunder(r, 50)
// Trigger an alert on crosses.
if xUp
    alert("Go long (RSI is " + str.tostring(r, "#.00"))
else if xDn
```

```

alert("Go short (RSI is " + str.tostring(r, "#.00"))

plotchar(xUp, "Go Long", "▲", location.bottom, color.lime, size = size.tiny)
plotchar(xDn, "Go Short", "▼", location.top, color.red, size = size.tiny)
hline(50)
plot(r)

```

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

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

Note that:

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

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

```

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

```

Using selective `alert()` calls

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

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

```

//@version=5
indicator("Selective `alert()` calls")
detectLongsInput = input.bool(true, "Detect Longs")
detectShortsInput = input.bool(true, "Detect Shorts")
repaintInput = input.bool(false, "Allow Repainting")

```

```

r = ta.rsi(close, 20)
// Detect crosses.
xUp = ta.crossover( r, 50)
xDn = ta.crossunder(r, 50)
// Only generate entries when the trade's direction is allowed in inputs.
enterLong = detectLongsInput and xUp and (repaintInput or
barstate.isconfirmed)
enterShort = detectShortsInput and xDn and (repaintInput or
barstate.isconfirmed)
// Trigger the alerts only when the compound condition is met.
if enterLong
    alert("Go long (RSI is " + str.tostring(r, "%.00"))
else if enterShort
    alert("Go short (RSI is " + str.tostring(r, "%.00"))

plotchar(enterLong, "Go Long", "▲", location.bottom, color.lime, size =
size.tiny)
plotchar(enterShort, "Go Short", "▼", location.top, color.red, size =
size.tiny)
hline(50)
plot(r)

```

Note how:

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

In strategies

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

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

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

```

//@version=5
strategy("Strategy with selective `alert()` calls")
r = ta.rsi(close, 20)

// Detect crosses.
xUp = ta.crossover( r, 50)
xDn = ta.crossunder(r, 50)
// Place orders on crosses.
if xUp
    strategy.entry("Long", strategy.long)
else if xDn

```

```

strategy.entry("Short", strategy.short)

// Trigger an alert when RSI diverges from our trade's direction.
divInLongTrade = strategy.position_size > 0 and ta.falling(r, 3)
divInShortTrade = strategy.position_size < 0 and ta.rising(r, 3)
if divInLongTrade
    alert("WARNING: Falling RSI", alert.freq_once_per_bar_close)
if divInShortTrade
    alert("WARNING: Rising RSI", alert.freq_once_per_bar_close)

plotchar(xUp, "Go Long", "▲", location.bottom, color.lime, size = size.tiny)
plotchar(xDn, "Go Short", "▼", location.top, color.red, size = size.tiny)
plotchar(divInLongTrade, "WARNING: Falling RSI", "•", location.top,
color.red, size = size.tiny)
plotchar(divInShortTrade, "WARNING: Rising RSI", "•", location.bottom,
color.lime, size = size.tiny)
hline(50)
plot(r)

```

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

Order fill events

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

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

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

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

```

//@version=5
strategy("Strategy using `alert_message`")
r = ta.rsi(close, 20)

// Detect crosses.

```



```

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

plotchar(xUp, "Go Long", "▲", location.bottom, color.lime, size = size.tiny)
plotchar(xDn, "Go Short", "▼", location.top, color.red, size = size.tiny)
hline(50)
plot(r)

```

Note that:

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

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

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

[alertcondition\(\)](#) events

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

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

The [alertcondition\(\)](#) function has the following signature:

```

alertcondition(condition, title, message)

```

condition

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

title

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

message

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

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

Using one condition

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

```
//@version=5
indicator("`alertcondition()` on single condition")
r = ta.rsi(close, 20)

xUp = ta.crossover( r, 50)
xDn = ta.crossunder(r, 50)

plot(r, "RSI")
hline(50)
plotchar(xUp, "Long", "▲", location.bottom, color.lime, size = size.tiny)
plotchar(xDn, "Short", "▼", location.top, color.red, size = size.tiny)

alertcondition(xUp, "Long Alert", "Go long")
alertcondition(xDn, "Short Alert", "Go short ")
```

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

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

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

Note that:

- The first line uses the `{{plot_0}}` placeholder, where the plot number corresponds to the order of the plot in the script.
- The second line uses the `{{plot("[plot_title]")}}` type of placeholder, which must include the `title` of the [plot\(\)](#) call used in our script to plot RSI. Double quotes are

used to wrap the plot's title inside the `{{plot("RSI")}}` placeholder. This requires that we use single quotes to wrap the message string.

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

Using compound conditions

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

This script demonstrates one way to do it:

```
//@version=5
indicator("`alertcondition()` on multiple conditions")
detectLongsInput = input.bool(true, "Detect Longs")
detectShortsInput = input.bool(true, "Detect Shorts")

r = ta.rsi(close, 20)
// Detect crosses.
xUp = ta.crossover(r, 50)
xDn = ta.crossunder(r, 50)
// Only generate entries when the trade's direction is allowed in inputs.
enterLong = detectLongsInput and xUp
enterShort = detectShortsInput and xDn

plot(r)
plotchar(enterLong, "Go Long", "▲", location.bottom, color.lime, size =
size.tiny)
plotchar(enterShort, "Go Short", "▼", location.top, color.red, size =
size.tiny)
hline(50)
// Trigger the alert when one of the conditions is met.
alertcondition(enterLong or enterShort, "Compound alert", "Entry")
```

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

Placeholders

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

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

```
{{exchange}}
    Exchange of the symbol used in the alert (NASDAQ, NYSE, MOEX, etc.). Note that for
    delayed symbols, the exchange will end with "_DL" or "_DLY." For example,
    "NYMEX_DL."
{{interval}}
    Returns the timeframe of the chart the alert is created on. Note that Range charts are
    calculated based on 1m data, so the placeholder will always return "1" on any alert created on
    a Range chart.
{{open}}, {{high}}, {{low}}, {{close}}, {{volume}}
    Corresponding values of the bar on which the alert has been triggered.
{{plot_0}}, {{plot_1}}, [...], {{plot_19}}
```

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

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

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

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

```
//@version=5
indicator("")
r = ta.rsi(close, 14)
xUp = ta.crossover(r, 50)
plot(r, "RSI", display = display.none)
alertcondition(xUp, "xUp alert", message = 'RSI is bullish at:
{{plot("RSI")}}')
```

```
{{ticker}}
```

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

```
{{time}}
```

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

```
{{timenow}}
```

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

[Avoiding repainting with alerts](#)

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

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

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

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

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) → void
```

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

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

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

```
//@version=5
indicator("Session backgrounds", overlay = true)

// Default color constants using transparency of 25.
BLUE_COLOR   = #0050FF40
PURPLE_COLOR = #0000FF40
PINK_COLOR   = #5000FF40
NO_COLOR     = color(na)

// Allow user to change the colors.
preMarketColor = input.color(BLUE_COLOR, "Pre-market")
regSessionColor = input.color(PURPLE_COLOR, "Pre-market")
postMarketColor = input.color(PINK_COLOR, "Pre-market")

// Function returns `true` when the bar's time is
timeInRange(tf, session) =>
    time(tf, session) != 0

// Function prints a message at the bottom-right of the chart.
f_print(_text) =>
    var table _t = table.new(position.bottom_right, 1, 1)
    table.cell(_t, 0, 0, _text, bgcolor = color.yellow)

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

bgcolor(sessionColor)
```



Note that:

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

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

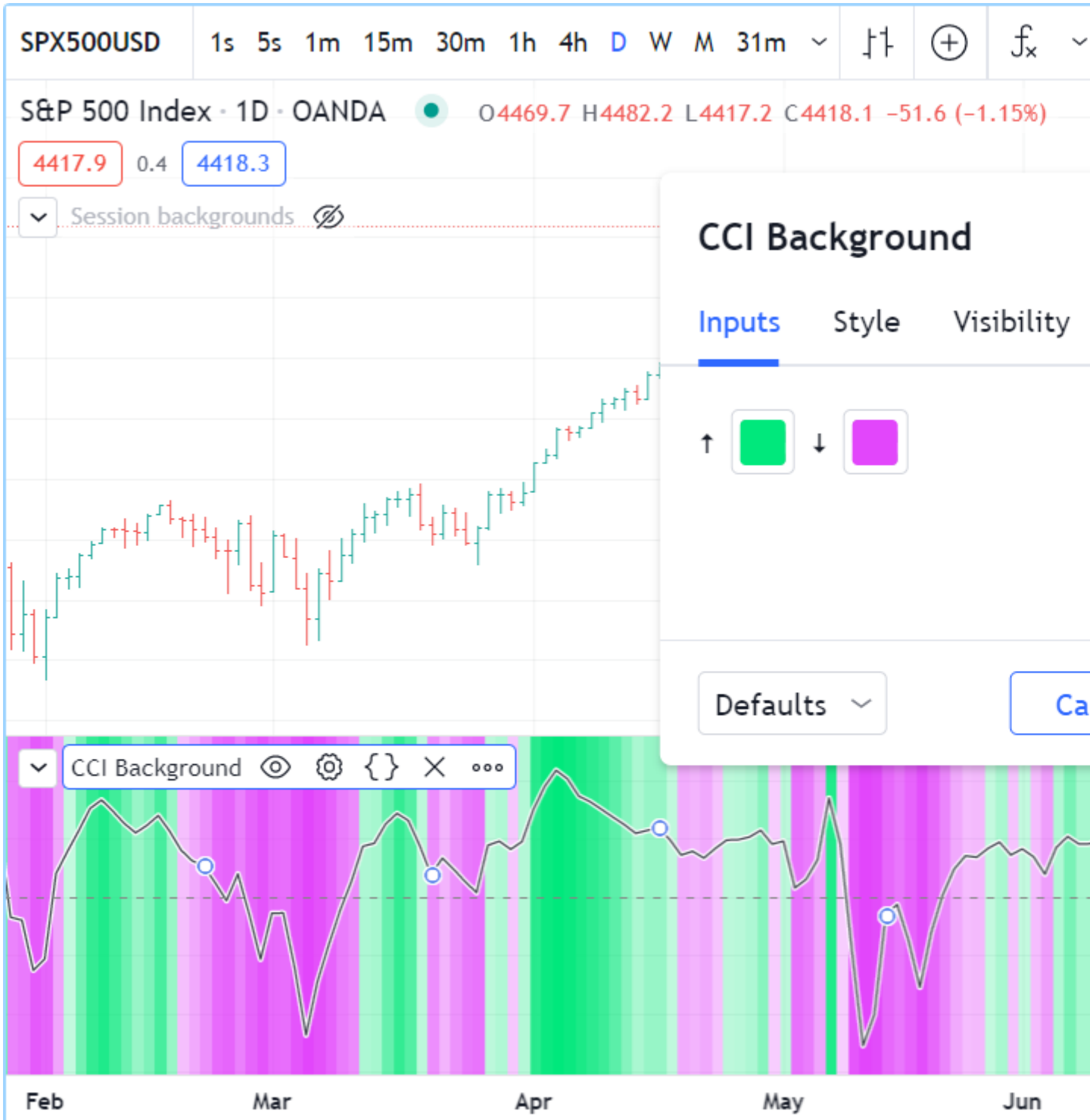
```
//@version=5
indicator("CCI Background")

bullColor = input.color(color.lime, "?", inline = "1")
bearColor = input.color(color.fuchsia, "?", inline = "1")

// Calculate CCI.
myCCI = ta.cci(hlc3, 20)
// Get relative position of CCI in last 100 bars, on a 0-100% scale.
myCCIPosition = ta.percentrank(myCCI, 100)
// Generate a bull gradient when position is 50-100%, bear gradient when
position is 0-50%.
backgroundColor = if myCCIPosition >= 50
    color.from_gradient(myCCIPosition, 50, 100, color.new(bullColor, 75),
```

```
bullColor)
else
  color.from_gradient(myCCIPosition, 0, 50, bearColor, color.new(bearColor,
75))

// Wider white line background.
plot(myCCI, "CCI", color.white, 3)
// Think black line.
plot(myCCI, "CCI", color.black, 1)
// Zero level.
hline(0)
// Gradient background.
bgcolor(backgroundColor)
```



Note that:

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

See the [Colors](#) page for more examples of backgrounds.

Bar coloring

The [barcolor\(\)](#) function lets you color chart bars. It is the only Pine Script[®] function that allows a script running in a pane to affect the chart.

The function's signature is:

```
barcolor(color, offset, editable, show_last, title) → 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:

Bar plotting

- [Introduction](#)
- [Plotting candles with `plotcandle\(\)`](#)
- [Plotting bars with `plotbar\(\)`](#)

Introduction

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

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

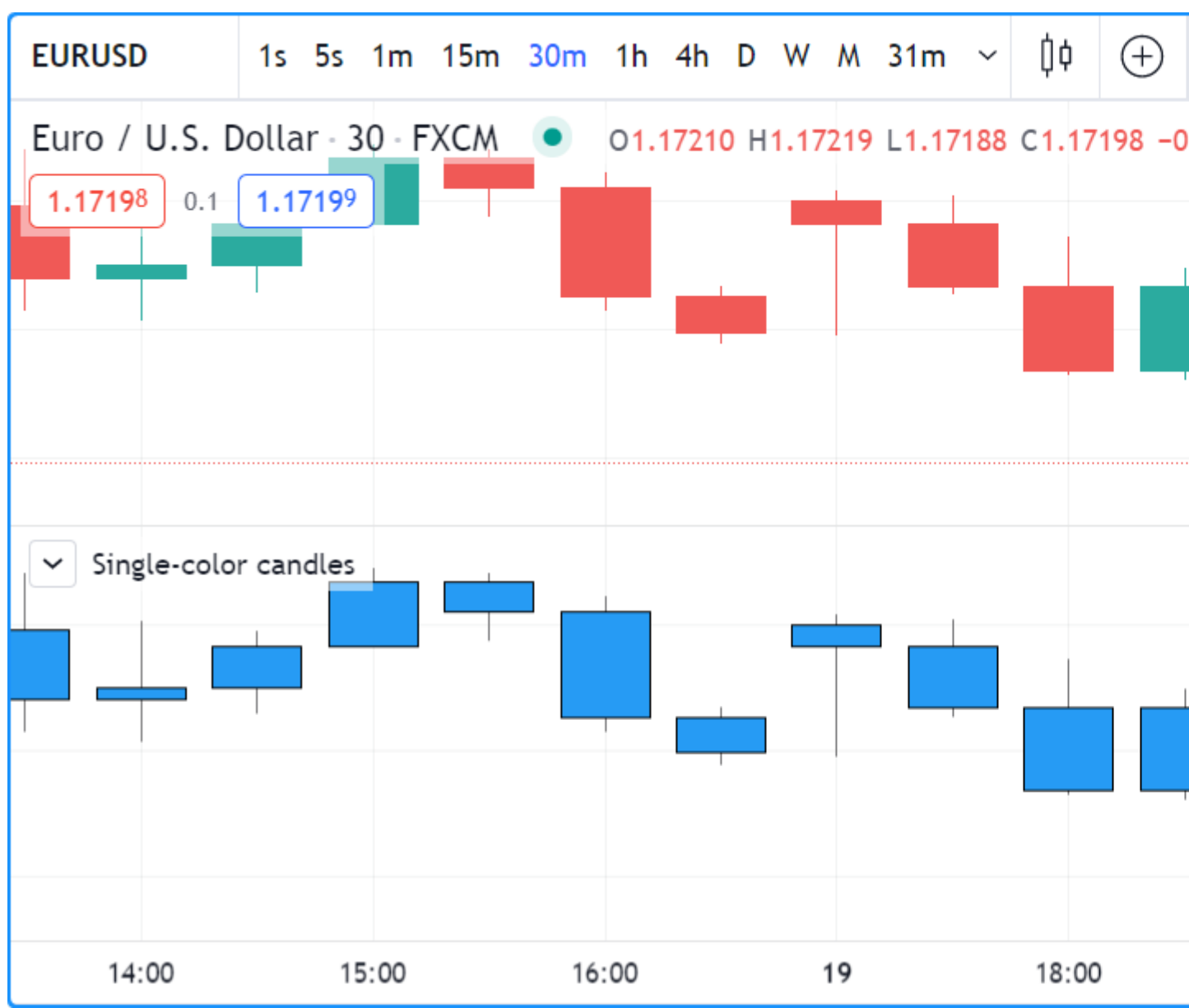
Plotting candles with `plotcandle()`

The signature of [plotcandle\(\)](#) is:


```
plotcandle(open, high, low, close, title, color, wickcolor, editable, show_last, bordercolor, display) → void
```

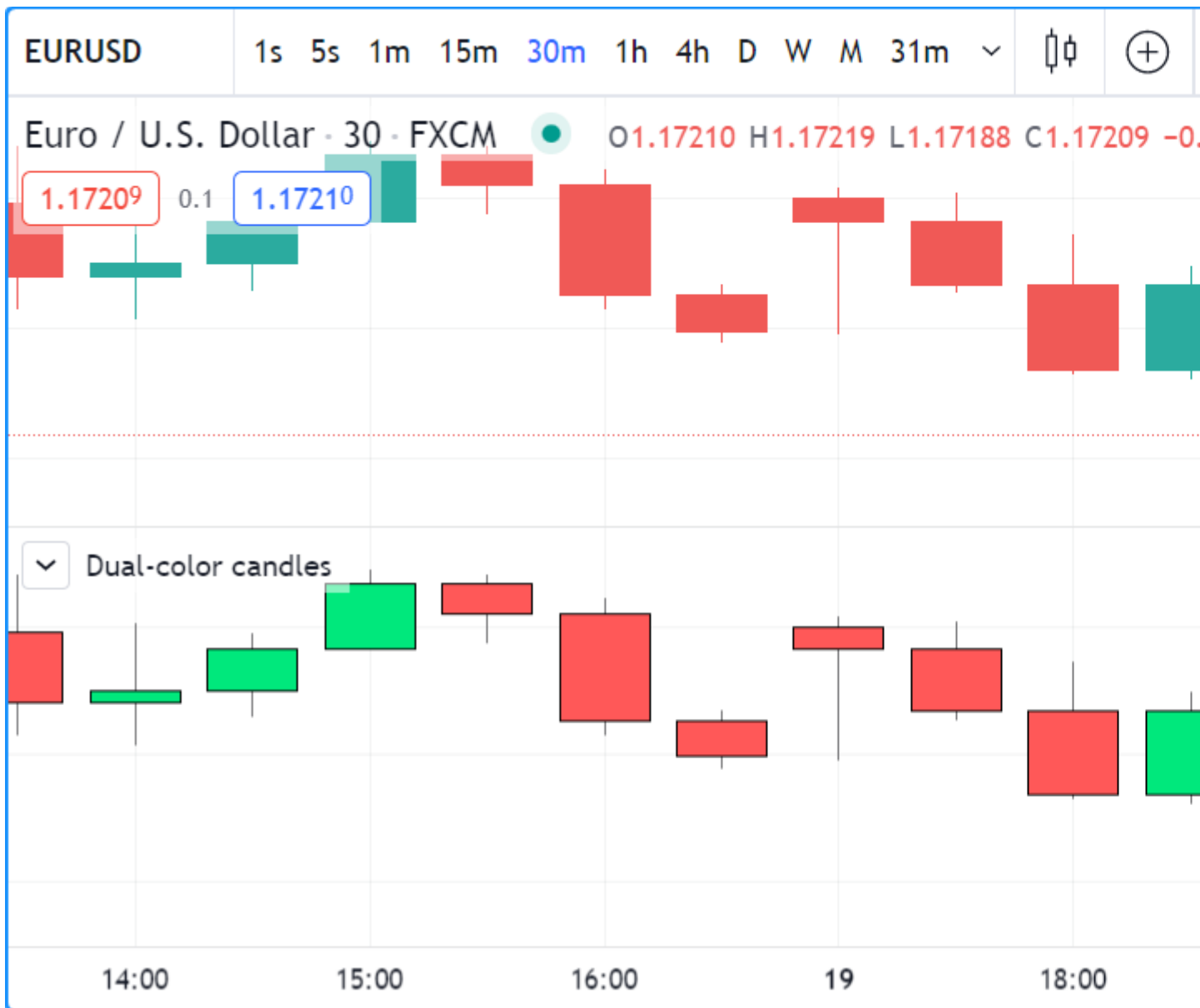
This plots simple candles, all in blue, using the habitual OHLC values, in a separate pane:

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



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

```
//@version=5  
indicator("Example 2")  
paletteColor = close >= open ? color.lime : color.red  
plotbar(open, high, low, close, color = paletteColor)
```



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

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

```
//@version=5
indicator("Smoothed candles", overlay = true)
lenInput = input.int(9)
smooth(source, length) =>
    ta.sma(source, length)
o = smooth(open, lenInput)
h = smooth(high, lenInput)
l = smooth(low, lenInput)
c = smooth(close, lenInput)
ourWickColor = close > c ? color.green : color.red
plotcandle(o, h, l, c, wickcolor = ourWickColor)
```

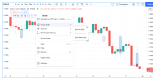


You may find it useful to plot OHLC values taken from a higher timeframe. You can, for example, plot daily bars on an intraday chart:

```
// NOTE: Use this script on an intraday chart.
//@version=5
indicator("Daily bars")

// Use gaps to only return data when the 1D timeframe completes, `na` otherwise.
[o, h, l, c] = request.security(syminfo.tickerid, "D", [open, high, low, close],
gaps = barmerge.gaps_on)

var color UP_COLOR = color.silver
var color DN_COLOR = color.blue
color wickColor = c >= o ? UP_COLOR : DN_COLOR
color bodyColor = c >= o ? color.new(UP_COLOR, 70) : color.new(DN_COLOR, 70)
// Only plot candles on intraday timeframes,
// and when non `na` values are returned by `request.security()` because a HTF
has completed.
plotcandle(timeframe.isintraday ? o : na, h, l, c, color = bodyColor, wickcolor
= wickColor)
```



Note that:

- We show the script's plot after having used "Visual Order/Bring to Front" from the script's "More" menu. This causes our script's candles to appear on top of the chart's candles.
- The script will only display candles when two conditions are met:
 - The chart is using an intraday timeframe (see the check on `timeframe.isintraday` in the [plotcandle\(\)](#) call). We do this because it's not useful to show a daily value on timeframes higher or equal to 1D.
 - The [request.security\(\)](#) function returns non [na](#) values (see `gaps = barmerge.gaps_on` in the function call).
- We use a tuple (`[open, high, low, close]`) with [request.security\(\)](#) to fetch four values in one call.
- We use [var](#) to declare our `UP_COLOR` and `DN_COLOR` color constants on bar zero only. We use constants because those colors are used in more than one place in our code. This way, if we need to change them, we need only do so in one place.
- We create a lighter transparency for the body of our candles in the `bodyColor` variable initialization, so they don't obstruct the chart's candles.

Plotting bars with `plotbar()`

The signature of [plotbar\(\)](#) is:

```
plotbar(open, high, low, close, title, color, editable, show_last, display) → void
```

Note that [plotbar\(\)](#) has no parameter for `bordercolor` or `wickcolor`, as there are no borders

or wicks on conventional bars.

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

```
//@version=5
indicator("Dual-color bars")
paletteColor = close >= open ? color.lime : color.red
plotbar(open, high, low, close, color = paletteColor)
```

Bar states

- [Introduction](#)
- [Bar state built-in variables](#)
 - [`barstate.isfirst`](#)
 - [`barstate.islast`](#)
 - [`barstate.ishistory`](#)
 - [`barstate.isrealtime`](#)
 - [`barstate.isnew`](#)
 - [`barstate.isconfirmed`](#)
 - [`barstate.islastconfirmedhistory`](#)
- [Example](#)

Introduction

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

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

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

Bar state built-in variables

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

```
//@version=5
strategy("S")
bgcolor(barstate.islast ? color.silver : na)
```

`barstate.isfirst`

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

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

```
// Declare array and set its values on the first bar only.
FILL_COLOR = color.green
var fillColors = array.new_color(0)
if barstate.isfirst
    // Initialize the array elements with progressively lighter shades of the
```

```

fill color.
    array.push(fillColors, color.new(FILL_COLOR, 70))
    array.push(fillColors, color.new(FILL_COLOR, 75))
    array.push(fillColors, color.new(FILL_COLOR, 80))
    array.push(fillColors, color.new(FILL_COLOR, 85))
    array.push(fillColors, color.new(FILL_COLOR, 90))

```

`barstate.islast`

[barstate.islast](#) is true if the current bar is the last one on the chart, whether that bar is a realtime bar or not.

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

```

//@version=5
indicator("", "", true)
// Create label on the first bar only.
var label hiLabel = label.new(na, na, "")
// Update the label's position and text on the last bar,
// including on all realtime bar updates.
if barstate.islast
    label.set_xy(hiLabel, bar_index, high)
    label.set_text(hiLabel, str.tostring(high, format.mintick))

```

`barstate.ishistory`

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

`barstate.isrealtime`

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

`barstate.isnew`

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

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

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

```

//@version=5
indicator("")
updateNo() =>
    varip int updateNo = na
    if barstate.isnew
        updateNo := 1

```

```

else
    updateNo += 1
plot(updateNo())

```

barstate.isconfirmed

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

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

```

//@version=5
indicator("")
myRSI = ta.rsi(close, 20)
plot(barstate.isconfirmed ? myRSI : na)

```

[barstate.isconfirmed](#) will not work when used in a [request.security\(\)](#) call.

barstate.islastconfirmedhistory

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

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

Example

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

```

//@version=5
indicator("Bar States", overlay = true, max_labels_count = 500)

stateText() =>
    string txt = ""
    txt += barstate.isfirst      ? "isfirst\n"      : ""
    txt += barstate.islast      ? "islast\n"       : ""
    txt += barstate.ishistory   ? "ishistory\n"     : ""
    txt += barstate.isrealtime  ? "isrealtime\n"   : ""
    txt += barstate.isnew       ? "isnew\n"        : ""
    txt += barstate.isconfirmed ? "isconfirmed\n"  : ""
    txt += barstate.islastconfirmedhistory ? "islastconfirmedhistory\n" : ""

labelColor = switch
    barstate.isfirst           => color.fuchsia
    barstate.islastconfirmedhistory => color.gray
    barstate.ishistory         => color.silver
    barstate.isconfirmed       => color.orange
    barstate.isnew             => color.red
    => color.yellow

label.new(bar_index, na, stateText(), yloc = yloc.abovebar, color = labelColor)

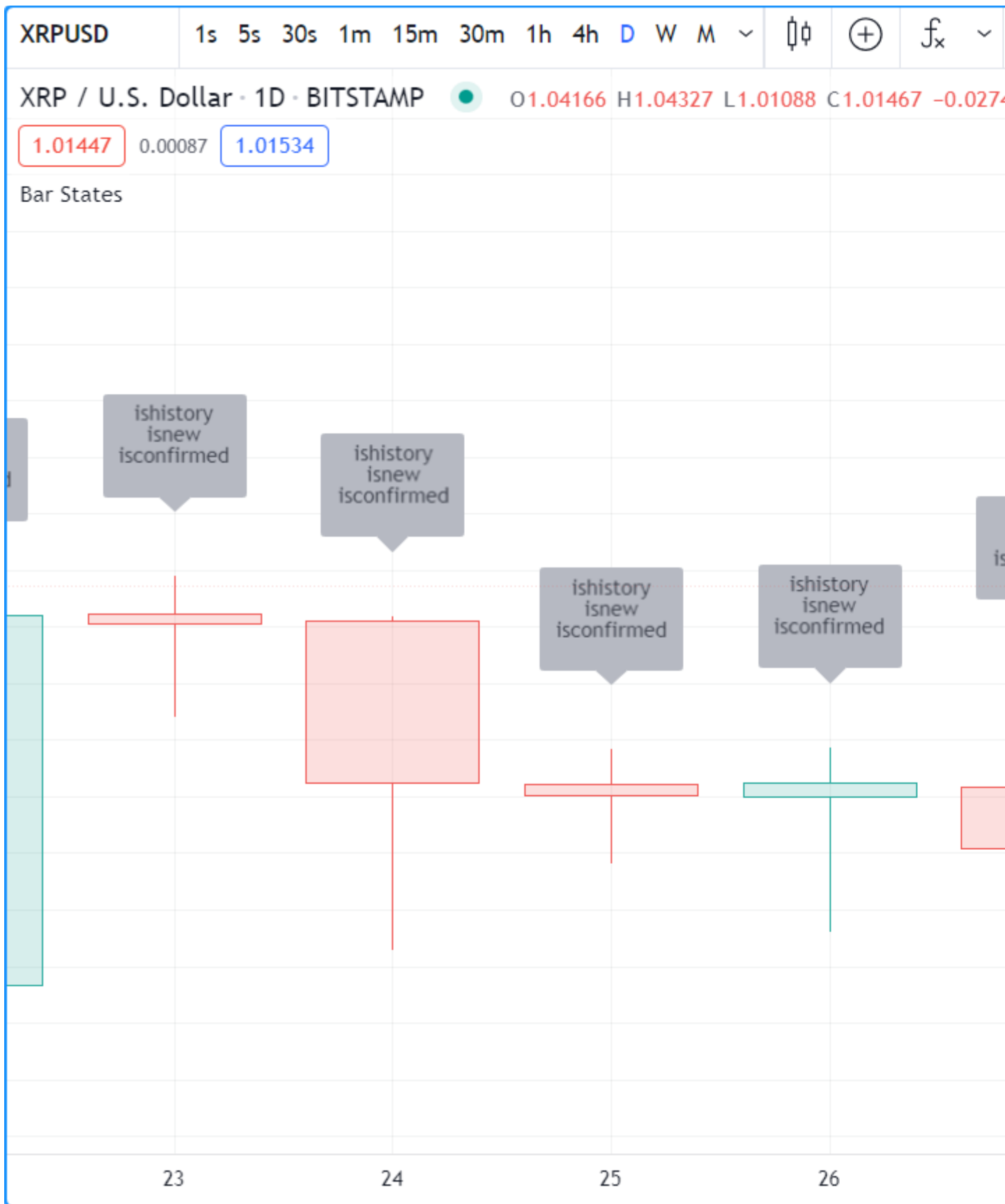
```

Note that:

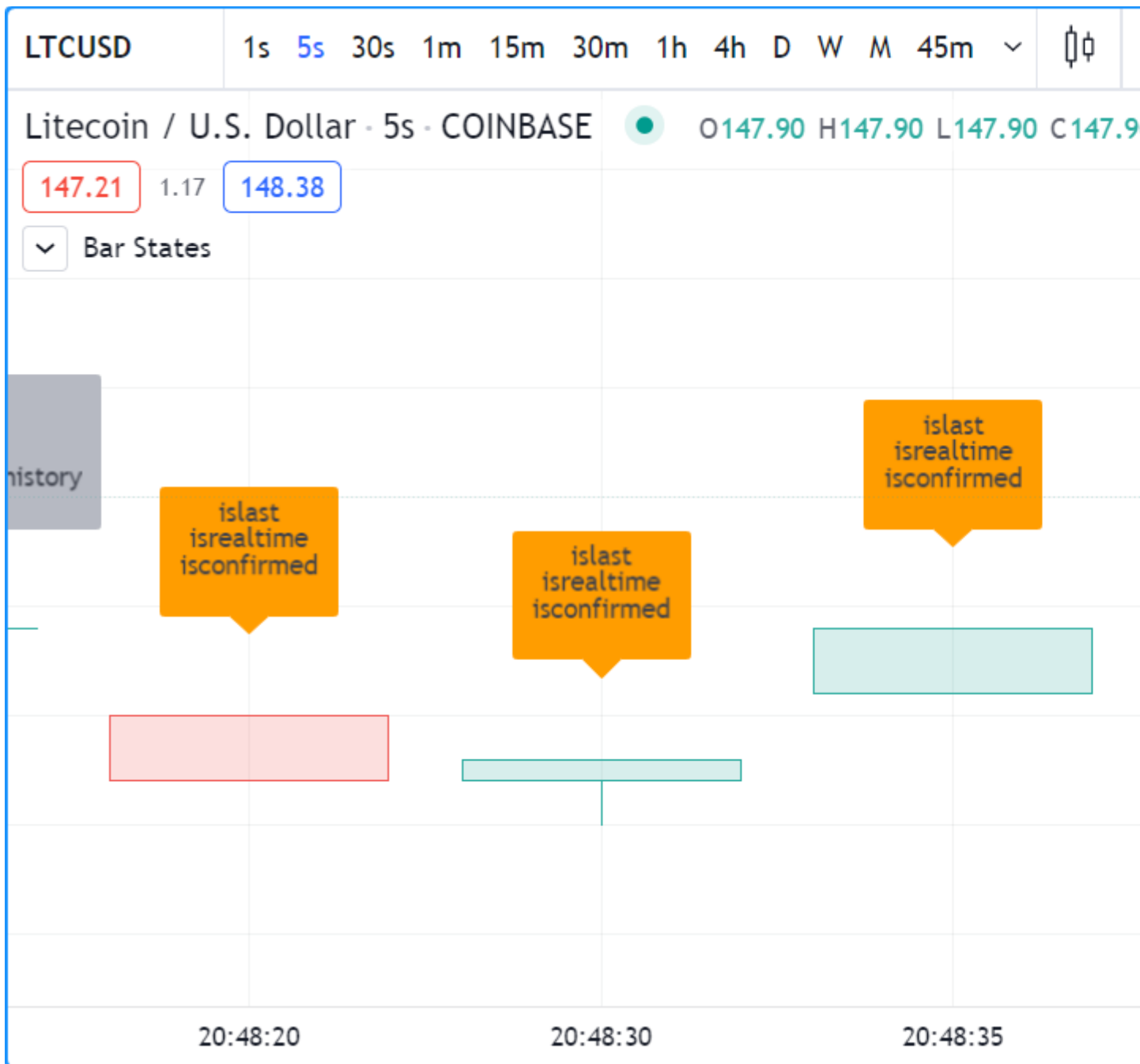
- Each state's name will appear in the label's text when it is `true`.

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

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



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



Note that:

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

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

Chart information

- [Introduction](#)
- [Prices and volume](#)
- [Symbol information](#)
- [Chart timeframe](#)
- [Session information](#)

Introduction

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

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

Prices and volume

The built-in variables for OHLCV values are:

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

Other values are available through:

- [hl2](#): the average of the bar's [high](#) and [low](#) values.
- [hlc3](#): the average of the bar's [high](#), [low](#) and [close](#) values.
- [ohlc4](#): the average of the bar's [open](#), [high](#), [low](#) and [close](#) values.

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

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

The `[]` [history-referencing operator](#) can be used to refer to past values of the built-in variables, e.g.,

`close[1]` refers to the value of [close](#) on the previous bar, relative to the particular bar the script is executing on.

Symbol information

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

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

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

```
//@version=5
indicator("`syminfo.*` built-ins", "", true)
printTable(txtLeft, txtRight) =>
    var table t = table.new(position.middle_right, 2, 1)
    table.cell(t, 0, 0, txtLeft, bgcolor = color.yellow, text_halign =
text.align_right)
    table.cell(t, 1, 0, txtRight, bgcolor = color.yellow, text_halign =
text.align_left)

nl = "\n"
left =
"syminfo.basecurrency: " + nl +
"syminfo.currency: " + nl +
"syminfo.description: " + nl +
"syminfo.mintick: " + nl +
```

```

"syminfo.pointvalue: "      + nl +
"syminfo.prefix: "        + nl +
"syminfo.root: "          + nl +
"syminfo.session: "       + nl +
"syminfo.ticker: "        + nl +
"syminfo.tickerid: "      + nl +
"syminfo.timezone: "      + nl +
"syminfo.type: "

right =
  syminfo.basecurrency      + nl +
  syminfo.currency          + nl +
  syminfo.description       + nl +
  str.toString(syminfo.mintick) + nl +
  str.toString(syminfo.pointvalue) + nl +
  syminfo.prefix           + nl +
  syminfo.root             + nl +
  syminfo.session          + nl +
  syminfo.ticker           + nl +
  syminfo.tickerid         + nl +
  syminfo.timezone         + nl +
  syminfo.type

printTable(left, right)

```

Chart timeframe

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

- [timeframe.isseconds](#)
- [timeframe.isminutes](#)
- [timeframe.isintraday](#)
- [timeframe.isdaily](#)
- [timeframe.isweekly](#)
- [timeframe.ismonthly](#)
- [timeframe.isdwm](#)

Two additional built-ins return more specific timeframe information:

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

See the page on [Timeframes](#) for more information.

Session information

Session information is available in different forms:

- The [syminfo.session](#) built-in variable returns a value that is either [session.regular](#) or [session.extended](#). It reflects the session setting on the chart for that symbol. If the “Chart settings/Symbol/Session” field is set to “Extended”, it will only return “extended” if the

symbol and the user's feed allow for extended sessions. It is used when a session type is expected, for example as the argument for the `session` parameter in [`ticker.new\(\)`](#).

- [Session state built-ins](#) provide information on the trading session a bar belongs to.

Colors

- [Introduction](#)
 - [Transparency](#)
 - [Z-index](#)
- [Constant colors](#)
- [Conditional coloring](#)
- [Calculated colors](#)
 - [`color.new\(\)`](#)
 - [`color.rgb\(\)`](#)
 - [`color.from_gradient\(\)`](#)
- [Mixing transparencies](#)
- [Tips](#)
 - [Designing usable colors schemes](#)
 - [Plot crisp lines](#)
 - [Customize gradients](#)
 - [Color selection through script settings](#)

[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
- Plots
- Hlines
- Fills
- Boxes
- Labels
- Lines
- 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

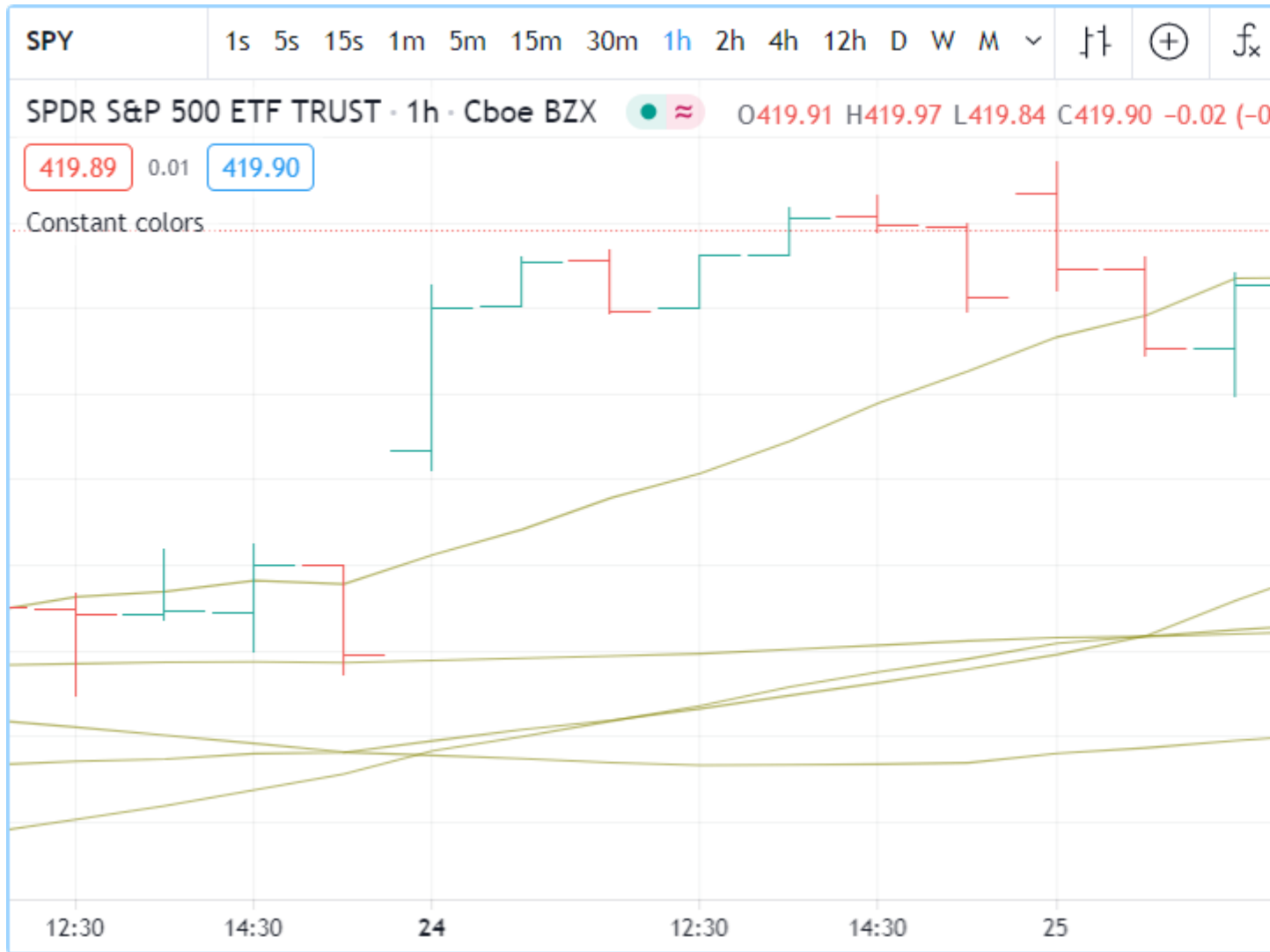
There are 17 built-in colors in Pine Script[®]. This table lists their names, hexadecimal equivalent, and RGB values as arguments to [color.rgb\(\)](#):

Name	Hex	RGB values
color.aqua	#00BCD4	color.rgb(0, 188, 212)

Name	Hex	RGB values
color.black	#363A45	color.rgb(54, 58, 69)
color.blue	#2196F3	color.rgb(33, 150, 243)
color.fuchsia	#E040FB	color.rgb(224, 64, 251)
color.gray	#787B86	color.rgb(120, 123, 134)
color.green	#4CAF50	color.rgb(76, 175, 80)
color.lime	#00E676	color.rgb(0, 230, 118)
color.maroon	#880E4F	color.rgb(136, 14, 79)
color.navy	#311B92	color.rgb(49, 27, 146)
color.olive	#808000	color.rgb(128, 128, 0)
color.orange	#FF9800	color.rgb(255, 152, 0)
color.purple	#9C27B0	color.rgb(156, 39, 176)
color.red	#FF5252	color.rgb(255, 82, 82)
color.silver	#B2B5BE	color.rgb(178, 181, 190)
color.teal	#00897B	color.rgb(0, 137, 123)
color.white	#FFFFFF	color.rgb(255, 255, 255)

Name	Hex	RGB values
color.yellow	#FFEB3B	color.rgb(255, 235, 59)

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



```
//@version=5
indicator("", "", true)
// ——— Transparency (#99) is included in the hex value.
plot(ta.sma(close, 10), "10", #80800099)
// ——— Transparency is included in the color-generating function's arguments.
plot(ta.sma(close, 30), "30", color.new(color.olive, 40))
plot(ta.sma(close, 50), "50", color.rgb(128, 128, 0, 40))
// ——— Use `transp` parameter (deprecated and advised against)
plot(ta.sma(close, 70), "70", color.olive, transp = 40)
plot(ta.sma(close, 90), "90", #808000, transp = 40)
```

Note

The last two `plot()` calls specify transparency using the `transp` parameter. This use should be avoided as the `transp` is deprecated in Pine Script® v5. Using the `transp` parameter to define transparency is not as flexible because it requires an argument of *input integer* type, which entails it

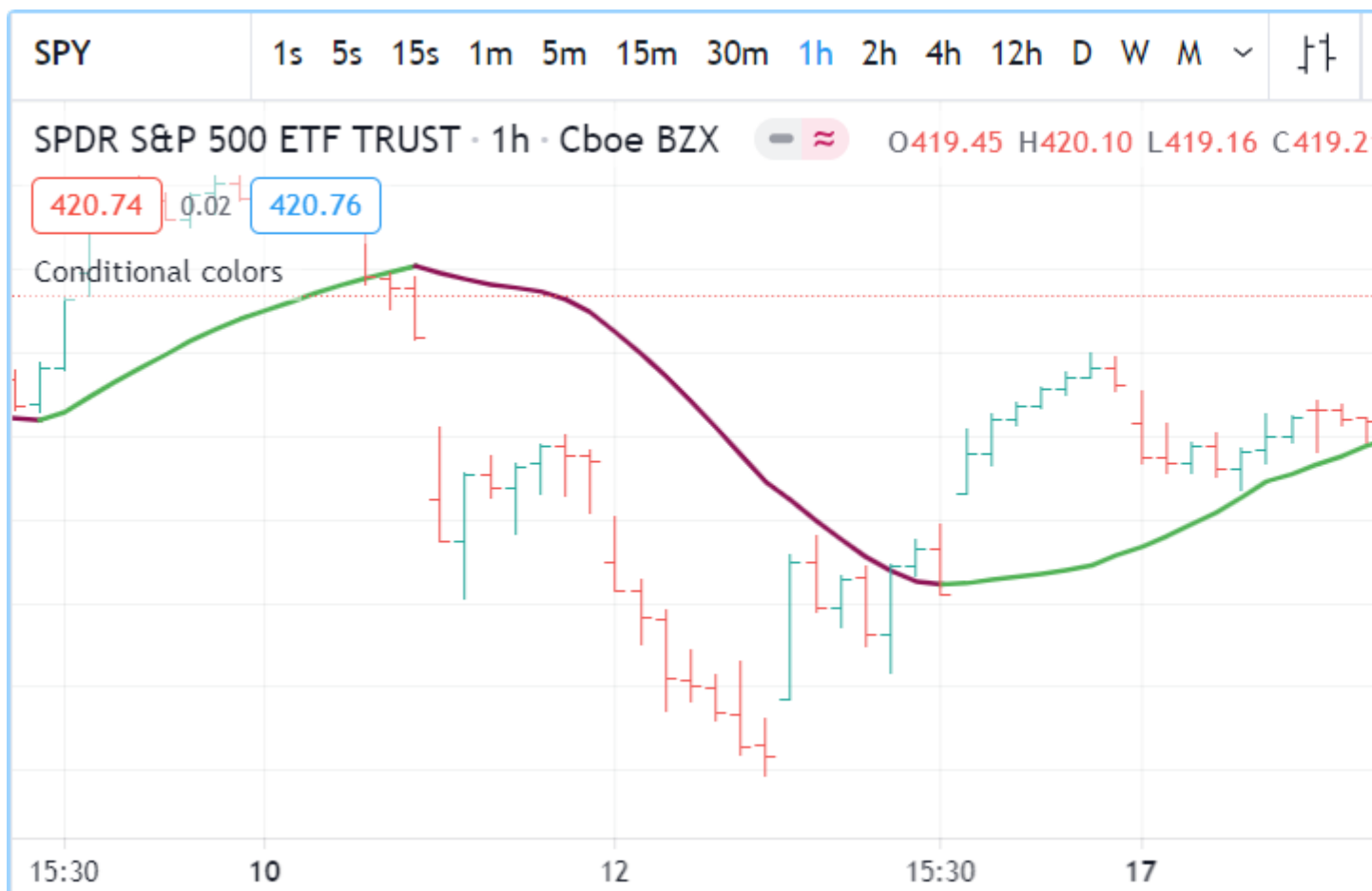
must be known before the script is executed, and so cannot be calculated dynamically, as your script executes bar to bar. Additionally, if you use a `color` argument that already includes transparency information, as is done in the next three `plot()` calls, any argument used for the `transp` parameter would have no effect. This is also true for other functions with a `transp` parameter.

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

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

Conditional coloring

Let's say you want to color a moving average in different colors, depending on some conditions you define. To do so, you can use a conditional statement that will select a different color for each of your states. Let's start by coloring a moving average in a bull color when it's rising, and in a bear color when it's not:



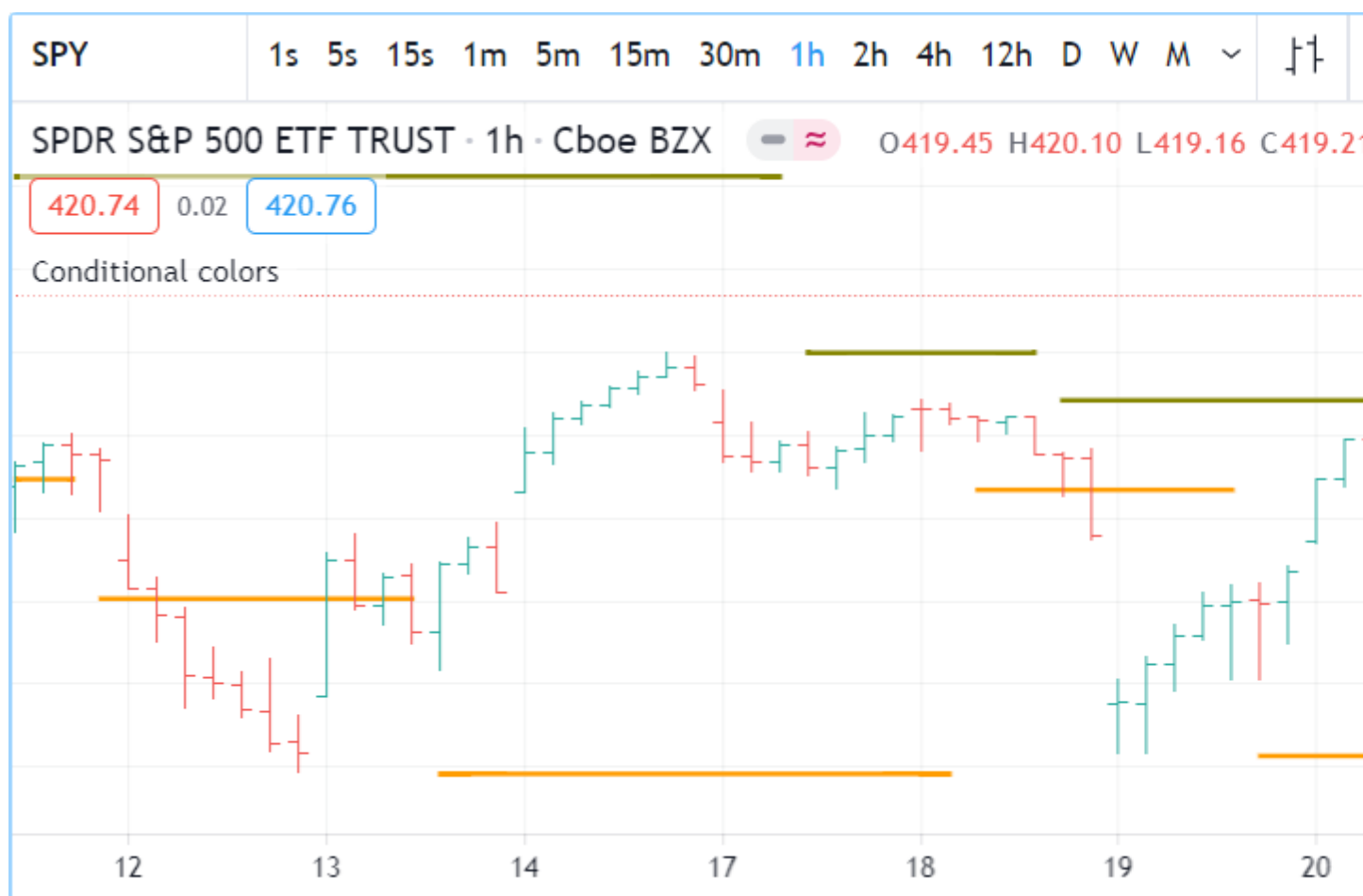
```
//@version=5
indicator("Conditional colors", "", true)
int lengthInput = input.int(20, "Length", minval = 2)
color maBullColorInput = input.color(color.green, "Bull")
color maBearColorInput = input.color(color.maroon, "Bear")
float ma = ta.sma(close, lengthInput)
// Define our states.
bool maRising = ta.rising(ma, 1)
```

```
// Build our color.
color c_ma = maRising ? maBullColorInput : maBearColorInput
plot(ma, "MA", c_ma, 2)
```

Note that:

- We provide users of our script a selection of colors for our bull/bear colors.
- We define an `maRising` boolean variable which will hold `true` when the moving average is higher on the current bar than it was on the last.
- We define a `c_ma` color variable that is assigned one of our two colors, depending on the value of the `maRising` boolean. We use the [?: ternary operator](#) to write our conditional statement.

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



```
//@version=5
indicator("Conditional colors", "", true)
int legsInput = input.int(5, "Pivot Legs", minval = 1)
color pHiColorInput = input.color(color.olive, "High pivots")
color pLoColorInput = input.color(color.orange, "Low pivots")
// Intialize the pivot level variables.
var float pHi = na
var float pLo = na
// When a new pivot is detected, save its value.
pHi := nz(ta.pivohigh(legsInput, legsInput), pHi)
pLo := nz(ta.pivotlow(legsInput, legsInput), pLo)
// When a new pivot is detected, do not plot a color.
```

```
plot(pHi, "High", ta.change(pHi) ? na : pHiColorInput, 2, plot.style_line)
plot(pLo, "Low", ta.change(pLo) ? na : pLoColorInput, 2, plot.style_line)
```

To understand how this code works, one must first know that [ta.pivohigh\(\)](#) 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 transparency components. While [color.rgb\(\)](#) creates a color, its sister functions [color.r\(\)](#), [color.g\(\)](#), [color.b\(\)](#) and [color.t\(\)](#) can be used to extract the red, green, blue or transparency values from a color, which can in turn be used to generate a variant.

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

color.new()

Let's put [color.new\(color, transp\)](#) to use to create different transparencies for volume columns using one of two bull/bear base colors:



```
//@version=5
indicator("Volume")
// We name our color constants to make them more readable.
var color GOLD_COLOR = #CCCC00ff
var color VIOLET_COLOR = #AA00FFff
color bullColorInput = input.color(GOLD_COLOR, "Bull")
color bearColorInput = input.color(VIOLET_COLOR, "Bear")
int levelsInput = input.int(10, "Gradient levels", minval = 1)
// We initialize only once on bar zero with `var`, otherwise the count would
reset to zero on each bar.
var float riseFallCnt = 0
// Count the rises/falls, clamping the range to: 1 to `i_levels`.
riseFallCnt := math.max(1, math.min(levelsInput, riseFallCnt + math.sign(volume
- nz(volume[1]))))
// Rescale the count on a scale of 80, reverse it and cap transparency to <80 so
that colors remains visible.
float transparency = 80 - math.abs(80 * riseFallCnt / levelsInput)
```

```
// Build the correct transparency of either the bull or bear color.
color volumeColor = color.new(close > open ? bullColorInput : bearColorInput,
transparency)
plot(volume, "Volume", volumeColor, 1, plot.style_columns)
```

Note that:

- In the next to last line of our script, we dynamically calculate the column color by varying both the base color used, depending on whether the bar is up or down, **and** the transparency level, which is calculated from the cumulative rises or falls of volume.
- We offer the script user control over not only the base bull/bear colors used, but also on the number of brightness levels we use. We use this value to determine the maximum number of rises or falls we will track. Giving users the possibility to manage this value allows them to adapt the indicator's visuals to the timeframe or market they use.
- We take care to control the maximum level of transparency we use so that it never goes higher than 80. This ensures our colors always retain some visibility.
- We also set the minimum value for the number of levels to 1 in the inputs. When the user selects 1, the volume columns will be either in bull or bear color of maximum brightness—or transparency zero.

color.rgb()

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



```
//@version=5
indicator("Holiday candles", "", true)
float r = math.random(0, 255)
float g = math.random(0, 255)
float b = math.random(0, 255)
float t = math.random(0, 100)
color holidayColor = color.rgb(r, g, b, t)
plotcandle(open, high, low, close, color = c_holiday, wickcolor = holidayColor,
bordercolor = c_holiday)
```

Note that:

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

color.from_gradient()

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

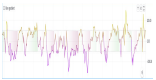


```
//@version=5
indicator(title="CCI line gradient", precision=2, timeframe="")
var color GOLD_COLOR = #CCCC00
var color VIOLET_COLOR = #AA00FF
var color BEIGE_COLOR = #9C6E1B
float srcInput = input.source(close, title="Source")
int lenInput = input.int(20, "Length", minval = 5)
color bullColorInput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

Note that:

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

The argument used for `value` in [color.from_gradient\(\)](#) does not necessarily have to be the value of the line we are calculating. Anything we want can be used, as long as arguments for `bottom_value` and `top_value` can be supplied. Here, we enhance our CCI indicator by coloring the band using the number of bars since the signal has been above/below the centerline:



```
//@version=5
indicator(title="CCI line gradient", precision=2, timeframe="")
var color GOLD_COLOR = #CCCC00
var color VIOLET_COLOR = #AA00FF
var color GREEN_BG_COLOR = color.new(color.green, 70)
var color RED_BG_COLOR = color.new(color.maroon, 70)
float srcInput = input.source(close, "Source")
int lenInput = input.int(20, "Length", minval = 5)
int stepsInput = input.int(50, "Gradient levels", minval = 1)
color bullColorInput = input.color(GOLD_COLOR, "Line: Bull", inline = "11")
color bearColorInput = input.color(VIOLET_COLOR, "Bear", inline = "11")
color bullBgColorInput = input.color(GREEN_BG_COLOR, "Background: Bull", inline = "12")
```

```

color bearBgColorInput = input.color(RED_BG_COLOR, "Bear", inline = "12")

// Plot colored signal line.
float signal = ta.cci(srcInput, lenInput)
color signalColor = color.from_gradient(signal, -200, 200,
color.new(bearColorInput, 0), color.new(bullColorInput, 0))
plot(signal, "CCI", signalColor, 2)

// Detect crosses of the centerline.
bool signalX = ta.cross(signal, 0)
// Count no of bars since cross. Capping it to the no of steps from inputs.
int gradientStep = math.min(stepsInput, nz(ta.barssince(signalX)))
// Choose bull/bear end color for the gradient.
color endColor = signal > 0 ? bullBgColorInput : bearBgColorInput
// Get color from gradient going from no color to `c_endColor`
color bandColor = color.from_gradient(gradientStep, 0, stepsInput, na, endColor)
bandTopPlotID = hline(100, "Upper Band", color.silver, hline.style_dashed)
bandBotPlotID = hline(-100, "Lower Band", color.silver, hline.style_dashed)
fill(bandTopPlotID, bandBotPlotID, bandColor, title = "Band")

```

Note that:

- The signal plot uses the same base colors and gradient as in our previous example. We have however increased the width of the line from the default 1 to 2. It is the most important component of our visuals; increasing its width is a way to give it more prominence, and ensure users are not distracted by the band, which has become busier than it was in its original, flat beige color.
- The fill must remain unobtrusive for two reasons. First, it is of secondary importance to the visuals, as it provides complementary information, i.e., the duration for which the signal has been in bull/bear territory. Second, since fills have a greater z-index than plots, the fill will cover the signal plot. For these reasons, we make the fill's base colors fairly transparent, at 70, so they do not mask the plots. The gradient used for the band starts with no color at all (see the [na](#) used as the argument to `bottom_color` in the `color.from_gradient()` call), and goes to the base bull/bear colors from the inputs, which the conditional, `c_endColor` color variable contains.
- We provide users with distinct bull/bear color selections for the line and the band.
- When we calculate the `gradientStep` variable, we use `nz()` on `ta.barssince()` because in early bars of the dataset, when the condition tested has not occurred yet, `ta.barssince()` will return [na](#). Because we use `nz()`, the value returned is replaced with zero in those cases.

Mixing transparencies

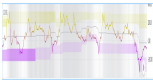
In this example we take our CCI indicator in another direction. We will build dynamically adjusting extremes zone buffers using a Donchian Channel (historical highs/lows) calculated from the CCI. We build the top/bottom bands by making them 1/4 the height of the DC. We will use a dynamically adjusting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

Our aim is to provide users of our indicator with:

- The CCI line colored using a bull/bear gradient, as we illustrated in our most recent examples.
- The top and bottom bands of the Donchian Channel, filled in such a way that their color darkens as a historical high/low becomes older and older.

- A way to appreciate the state of our volatility measure, which we will do by painting the background with one color whose intensity increases when volatility increases.

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



And with the dark theme:



```
//@version=5
indicator("CCI DC", precision = 6)
color GOLD_COLOR = #CCCC00ff
color VIOLET_COLOR = #AA00FFff
int lengthInput = input.int(20, "Length", minval = 5)
color bullColorInput = input.color(GOLD_COLOR, "Bull")
color bearColorInput = input.color(VIOLET_COLOR, "Bear")

// —— Function clamps `val` between `min` and `max`.
clamp(val, min, max) =>
    math.max(min, math.min(max, val))

// —— Volatility expressed as 0-100 value.
float v = ta.atr(lengthInput / 5) / ta.atr(lengthInput * 5)
float vPct = ta.percentrank(v, lengthInput * 5)

// —— Calculate dynamic lookback for DC. It increases/decreases on low/high
volatility.
bool highVolatility = vPct > 50
var int lookBackMin = lengthInput * 2
var int lookBackMax = lengthInput * 10
var float lookBack = math.avg(lookBackMin, lookBackMax)
lookBack += highVolatility ? -2 : 2
lookBack := clamp(lookBack, lookBackMin, lookBackMax)

// —— Dynamic lookback length Donchian channel of signal.
float signal = ta.cci(close, lengthInput)
// `lookBack` is a float; need to cast it to int to be used a length.
float hiTop = ta.highest(signal, int(lookBack))
float loBot = ta.lowest(signal, int(lookBack))
// Get margin of 25% of the DC height to build high and low bands.
float margin = (hiTop - loBot) / 4
float hiBot = hiTop - margin
float loTop = loBot + margin
// Center of DC.
float center = math.avg(hiTop, loBot)

// —— Create colors.
color signalColor = color.from_gradient(signal, -200, 200, bearColorInput,
bullColorInput)
// Bands: Calculate transparencies so the longer since the hi/lo has changed,
// the darker the color becomes. Cap highest transparency to 90.
float hiTransp = clamp(100 - (100 * math.max(1,
nz(ta.barssince(ta.change(hiTop)) + 1)) / 255), 60, 90)
float loTransp = clamp(100 - (100 * math.max(1,
nz(ta.barssince(ta.change(loBot)) + 1)) / 255), 60, 90)
color hiColor = color.new(bullColorInput, hiTransp)
color loColor = color.new(bearColorInput, loTransp)
// Background: Rescale the 0-100 range of `vPct` to 0-25 to create 75-100
transparencies.
color bgColor = color.new(color.gray, 100 - (vPct / 4))
```

```

// ——— Plots
// Invisible lines for band fills.
hiTopPlotID = plot(hiTop, color = na)
hiBotPlotID = plot(hiBot, color = na)
loTopPlotID = plot(loTop, color = na)
loBotPlotID = plot(loBot, color = na)
// Plot signal and centerline.
p_signal = plot(signal, "CCI", signalColor, 2)
plot(center, "Centerline", color.silver, 1)

// Fill the bands.
fill(hiTopPlotID, hiBotPlotID, hiColor)
fill(loTopPlotID, loBotPlotID, loColor)

// ——— Background.
bgcolor(bgColor)

```

Note that:

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

Tips

Designing usable colors schemes

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

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

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

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

Plot crisp lines

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

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



```
//@version=5
indicator("")
plot(high, "", color.new(color.orange, 80), 8)
plot(high, "", color.new(color.orange, 60), 4)
plot(high, "", color.new(color.orange, 00), 1)

plot(hl2, "", color.new(color.orange, 60), 4)
plot(hl2, "", color.new(color.orange, 00), 1)

plot(low, "", color.new(color.orange, 0), 1)
```

Customize gradients

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

Color selection through script settings

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



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

```
//@version=5
indicator("Calculated colors", "", true)
float ma = ta.sma(close, 20)
float maHeight = ta.percentrank(ma, 100)
float transparency = math.min(80, 100 - maHeight)
// This plot uses a calculated color.
plot(ma, "MA1", color.rgb(156, 39, 176, transparency), 2)
```

```
// This plot does not use a calculated color.  
plot(close, "Close", color.blue)
```

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

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



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



```
//@version=5  
indicator("Calculated colors", "", true)  
color maInput = input.color(color.purple, "MA")  
color closeInput = input.color(color.blue, "Close")  
float ma = ta.sma(close, 20)  
float maHeight = ta.percentrank(ma, 100)  
float transparency = math.min(80, 100 - maHeight)  
// This plot uses a calculated color.  
plot(ma, "MA1", color.new(maInput, transparency), 2)  
// This plot does not use a calculated color.  
plot(close, "Close", closeInput)
```

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



Fills

- [Introduction](#)
- [`plot\(\)` and `hline\(\)` fills](#)
- [Line fills](#)

Introduction

There are two different mechanisms dedicated to filling the space between Pine visuals:

- The [fill\(\)](#) function lets you color the background between either two plots plotted using [plot\(\)](#) or two horizontal lines plotted using [hline\(\)](#).
- The [linefill.new\(\)](#) function fills the space between lines created with [line.new\(\)](#).

`plot()` and `hline()` fills

The [fill\(\)](#) function has two signatures:

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

The arguments used for the `plot1`, `plot2`, `hline1` and `hline2` parameters must be the IDs returned by the [plot\(\)](#) and [hline\(\)](#) calls. The [fill\(\)](#) function is the only built-in function where these IDs are used.

See in this first example how the IDs returned by the [plot\(\)](#) and [hline\(\)](#) calls are captured in the `p1`, `p2`, `p3`, and `h1`, `h2`, `h3` and `h4` variables for reuse as [fill\(\)](#) arguments:



```
//@version=5  
indicator("Example 1")  
p1 = plot(math.sin(high))  
p2 = plot(math.cos(low))  
p3 = plot(math.sin(close))  
fill(p1, p3, color.new(color.red, 90))  
fill(p2, p3, color.new(color.blue, 90))  
h1 = hline(0)  
h2 = hline(1.0)  
h3 = hline(0.5)  
h4 = hline(1.5)  
fill(h1, h2, color.new(color.yellow, 90))  
fill(h3, h4, color.new(color.lime, 90))
```

Because [fill\(\)](#) requires two IDs from the same function, we sometimes need to use a [plot\(\)](#) call where we would have otherwise used an [hline\(\)](#) call, as in this example:



```
//@version=5  
indicator("Example 2")  
src = close  
ma = ta.sma(src, 10)  
osc = 100 * (ma - src) / ma  
oscPlotID = plot(osc)  
// An `hline()` would not work here because two `plot()` calls are needed.  
zeroPlotID = plot(0, "Zero", color.silver, 1, plot.style_circles)  
fill(oscPlotID, zeroPlotID, color.new(color.blue, 90))
```

Because a “series color” can be used as an argument for the `color` parameter in [fill\(\)](#), you can use constants like `color.red` or `#FF001A`, as well as expressions calculating the color on each bar, as in this example:



```
//@version=5
indicator("Example 3", "", true)
line1 = ta.sma(close, 5)
line2 = ta.sma(close, 20)
p1PlotID = plot(line1)
p2PlotID = plot(line2)
fill(p1PlotID, p2PlotID, line1 > line2 ? color.new(color.green, 90) :
color.new(color.red, 90))
```

Line fills

Linefills are objects that allow you to fill the space between two line drawings created via the [line.new\(\)](#) function. A linefill object is displayed on the chart when the [linefill.new\(\)](#) function is called. The function has the following signature:

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

The `line1` and `line2` arguments are the line IDs of the two lines to fill between. The `color` argument is the color of the fill. Any two-line pair can only have one linefill between them, so successive calls to [linefill.new\(\)](#) on the same pair of lines will replace the previous linefill with a new one. The function returns the ID of the `linefill` object it created, which can be saved in a variable for use in [linefill.set_color\(\)](#) call that will change the color of an existing linefill.

The behavior of linefills is dependent on the lines they are attached to. Linefills cannot be moved directly; their coordinates follow those of the lines they are tied to. If both lines extend in the same direction, the linefill will also extend.

Note that for line extensions to work correctly, a line's `x1` coordinate must be less than its `x2` coordinate. If a line's `x1` argument is greater than its `x2` argument and `extend.left` is used, the line will actually extend to the right because `x2` is assumed to be the rightmost `x` coordinate.

In the example below, our indicator draws two lines connecting the last two high and low pivot points of the chart. We extend the lines to the right to project the short-term movement of the chart, and fill the space between them to enhance the visibility of the channel the lines create:

Apple Inc - 1D - NASDAQ - TradingView



O145.76 H148.45 L145.56 C147.75 +2.88 (+1.99%)

Channel



```
//@version=5
indicator("Channel", overlay = true)

LEN_LEFT = 15
LEN_RIGHT = 5
pH = ta.pivohigh(LEN_LEFT, LEN_RIGHT)
pL = ta.pivotlow(LEN_LEFT, LEN_RIGHT)
```

```

// Bar indices of pivot points
pH_x1 = ta.valuewhen(pH, bar_index, 1) - LEN_RIGHT
pH_x2 = ta.valuewhen(pH, bar_index, 0) - LEN_RIGHT
pL_x1 = ta.valuewhen(pL, bar_index, 1) - LEN_RIGHT
pL_x2 = ta.valuewhen(pL, bar_index, 0) - LEN_RIGHT
// Price values of pivot points
pH_y1 = ta.valuewhen(pH, pH, 1)
pH_y2 = ta.valuewhen(pH, pH, 0)
pL_y1 = ta.valuewhen(pL, pL, 1)
pL_y2 = ta.valuewhen(pL, pL, 0)

if barstate.islastconfirmedhistory
    // Lines
    lH = line.new(pH_x1, pH_y1, pH_x2, pH_y2, extend = extend.right)
    lL = line.new(pL_x1, pL_y1, pL_x2, pL_y2, extend = extend.right)
    // Fill
    fillColor = switch
        pH_y2 > pH_y1 and pL_y2 > pL_y1 => color.green
        pH_y2 < pH_y1 and pL_y2 < pL_y1 => color.red
        => color.silver
    linefill.new(lH, lL, color.new(fillColor, 90))

```

Inputs

- [Introduction](#)
- [Input functions](#)
- [Input function parameters](#)
- [Input types](#)
 - [Simple input](#)
 - [Integer input](#)
 - [Float input](#)
 - [Boolean input](#)
 - [Color input](#)
 - [Timeframe input](#)
 - [Symbol input](#)
 - [Session input](#)
 - [Source input](#)
 - [Time input](#)
- [Other features affecting Inputs](#)
- [Tips](#)

Introduction

Inputs allow scripts to receive values that users can change. Using them for key values will make your scripts more adaptable to user preferences.

The following script plots a 20-period [simple moving average \(SMA\)](#) using `ta.sma(close, 20)`. While it is simple to write, it is not very flexible because that specific MA is all it will ever plot:

```

//@version=5
indicator("MA", "", true)

```

```
plot(ta.sma(close, 20))
```

If instead we write our script this way, it becomes much more flexible because its users will be able to select the source and the length they want to use for the MA's calculation:

```
//@version=5
indicator("MA", "", true)
sourceInput = input(close, "Source")
lengthInput = input(20, "Length")
plot(ta.sma(sourceInput, lengthInput))
```

Inputs can only be accessed when a script is running on the chart. Script users access them through the script's "Settings" dialog box, which can be reached by either:

- Double-clicking on the name of an on-chart indicator
- Right-clicking on the script's name and choosing the "Settings" item from the dropdown menu
- Choosing the "Settings" item from the "More" menu icon (three dots) that appears when one hovers over the indicator's name on the chart
- Double-clicking on the indicator's name from the Data Window (fourth icon down to the right of the chart)

The "Settings" dialog box always contains the "Style" and "Visibility" tabs, which allow users to specify their preferences about the script's visuals and the chart timeframes where it should be visible.

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

MACD



Inputs

Style

Visibility

Indicator Timeframe

Same as chart



Colors:



Fast Length

12

Slow Length

26

Source

close



Signal Smoothing

9

Simple MA(Oscillator)

Simple MA(Signal Line)

Defaults



Cancel

Ok

In the flow of a script's execution, inputs are processed when the script is already on a chart and a user changes values in the "Inputs" tab. The changes trigger a re-execution of the script on all the chart bars, so when a user changes an input value, your script recalculates using that new value.

Input functions

The following input functions are available:

- [input\(\)](#)
- [input.int\(\)](#)
- [input.float\(\)](#)
- [input.bool\(\)](#)
- [input.color\(\)](#)
- [input.string\(\)](#)
- [input.timeframe\(\)](#)
- [input.symbol\(\)](#)
- [input.price\(\)](#)
- [input.source\(\)](#)
- [input.session\(\)](#)
- [input.time\(\)](#)

A specific input *widget* is created in the "Inputs" tab to accept each type of input. Unless otherwise specified in the `input.*()` call, each input appears on a new line of the "Inputs" tab, in the order the `input.*()` calls appear in the script.

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

Input function definitions typically contain many parameters, which allow you to control the default value of inputs, their limits, and their organization in the "Inputs" tab.

An `input*.()` call being just another function call in Pine Script[®], its result can be combined with [arithmetic](#), comparison, [logical](#) or [ternary](#) operators to form an expression to be assigned to the variable. Here, we compare the result of our call to [input.string\(\)](#) to the string "On". The expression's result is then stored in the `plotDisplayInput` variable. Since that variable holds a true or false value, it is of "input bool" type:

```
//@version=5
indicator("Input in an expression", "", true)
bool plotDisplayInput = input.string("On", "Plot Display", options = ["On",
"Off"]) == "On"
plot(plotDisplayInput ? close : na)
```

All values returned by `input.*()` functions except "source" ones are of the "input" form (see the section on [forms](#) for more information).

Input function parameters

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

All these parameters expect arguments of "const" form (except if it's an input used for a "source", which returns a "series float" result). This means they must be known at compile time and cannot change during the script's execution. Because the result of `input.*()` function is always of "input" or "series" form, it follows that the result of one `input.*()` function call cannot be used

as an argument in a subsequent `input.*()` call because the “input” form is stronger than the “const” form.

Let’s go over each parameter:

- `defval` is the first parameter of all input functions. It is the default value that will appear in the input widget. It requires an argument of the type of input value the function is used for.
- `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. It used to group any number of inputs in the same 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"]`). It is used to create a dropdown menu offering the list’s elements in the form of menu selections. Only one menu item can be selected. When an `options` list is used, the `defval` value must be one of the list’s elements. When `options` is used in input functions allowing `minval`, `maxval` or `step`, those parameters cannot be used 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 affect 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.

Simple input

[input\(\)](#) is a simple, generic function that supports the fundamental Pine Script[®] types: “int”, “float”,

“bool”, “color” and “string”. It also supports “source” inputs, which are price-related values such as [close](#), [hl2](#), [hlc3](#), and [hlcc4](#), or which can be used to receive the output value of another script.

Its signature is:

```
input(defval, title, tooltip, inline, group) → input int/float/bool/color/string  
| series float
```

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

```
//@version=5  
indicator("`input()`", "", true)  
a = input(1, "input int")  
b = input(1.0, "input float")  
c = input(true, "input bool")  
d = input(color.orange, "input color")  
e = input("1", "input string")  
f = input(close, "series float")  
plot(na)
```

Integer input

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

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

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

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

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

```
//@version=5  
indicator("MA", "", true)  
maLengthInput = input.int(10, minval = 2)  
ma = ta.sma(close, maLengthInput)  
plot(ma)
```

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



Float input

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

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

```
input.int(defval, title, options, tooltip, inline, group, confirm) → input int
```

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

```
//@version=5
indicator("MA", "", true)
maLengthInput = input.int(10, minval = 1)
bbFactorInput = input.float(1.5, minval = 0, step = 0.5)
ma      = ta.sma(close, maLengthInput)
bbWidth = ta.stdev(ma, maLengthInput) * bbFactorInput
bbHi    = ma + bbWidth
bbLo    = ma - bbWidth
plot(ma)
plot(bbHi, "BB Hi", color.gray)
plot(bbLo, "BB Lo", color.gray)
```

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



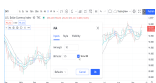
Boolean input

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

```
//@version=5
indicator("MA", "", true)
maLengthInput = input.int(10, "MA length", minval = 1)
bbFactorInput = input.float(1.5, "BB factor", inline = "01", minval = 0, step = 0.5)
showBBInput = input.bool(true, "Show BB", inline = "01")
ma = ta.sma(close, maLengthInput)
bbWidth = ta.stdev(ma, maLengthInput) * bbFactorInput
bbHi = ma + bbWidth
bbLo = ma - bbWidth
plot(ma, "MA", color.aqua)
plot(showBBInput ? bbHi : na, "BB Hi", color.gray)
plot(showBBInput ? bbLo : na, "BB Lo", color.gray)
```

Note that:

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



Color input

As is explained in the Color selection through script settings section of the "Colors" page, the color selections that usually appear in the "Settings/Style" tab are not always available. When that is the case, script users will have no means to change the colors your script uses. For those cases, it is essential to provide color inputs if you want your script's colors to be modifiable through the script's "Settings". Instead of using the "Settings/Style" tab to change colors, you will then allow your script users to change the colors using calls to [input.color\(\)](#).

Suppose we wanted to plot our BBs in a lighter shade when the [high](#) and [low](#) values are higher/lower than the BBs. You could use code like this to create your colors:

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

When using dynamic (or "series") color components like the transparency here, the color widgets in the "Settings/Style" will no longer appear. Let's create our own, which will appear in our "Inputs"

tab:

```
//@version=5
indicator("MA", "", true)
maLengthInput = input.int(10, "MA length", inline = "01", minval = 1)
maColorInput = input.color(color.aqua, "", inline = "01")
bbFactorInput = input.float(1.5, "BB factor", inline = "02", minval = 0,
step = 0.5)
bbColorInput = input.color(color.gray, "", inline = "02")
showBBInput = input.bool(true, "Show BB", inline = "02")
ma = ta.sma(close, maLengthInput)
bbWidth = ta.stdev(ma, maLengthInput) * bbFactorInput
bbHi = ma + bbWidth
bbLo = ma - bbWidth
bbHiColor = color.new(bbColorInput, high > bbHi ? 60 : 0)
bbLoColor = color.new(bbColorInput, low < bbLo ? 60 : 0)
plot(ma, "MA", maColorInput)
plot(showBBInput ? bbHi : na, "BB Hi", bbHiColor, 2)
plot(showBBInput ? bbLo : na, "BB Lo", bbLoColor, 2)
```

Note that:

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



Timeframe input

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

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

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

Note that:

- We use the [input.timeframe\(\)](#) function to receive the timeframe input.
- The function creates a dropdown widget where some standard timeframes are proposed. The list of timeframes also includes any you have favorated in the chart user interface.

- We use the `tfInput` in our [request.security\(\)](#) call. We also use `gaps = barmerge.gaps_on` in the call, so the function only returns data when the higher timeframe has completed.

BTCUSD

1s 5s 30s 1m 15m 30m 1h 4h D W M

Bitcoin / U.S. Dollar · 1h · BITSTAMP

O 47938.09 H 47952.82 L 47898.61

47898.61

14.26

47912.87

MA [Icons]

MA

Inputs

Style

Timeframe

1 d

San

1 m

3 m

Defaults

5 m

15 m

30 m

45 m

1 h

2 h

3 h

4 h

1 d



Symbol input

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

Let's add a symbol input to our script:

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

Note that:

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

Session input

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

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

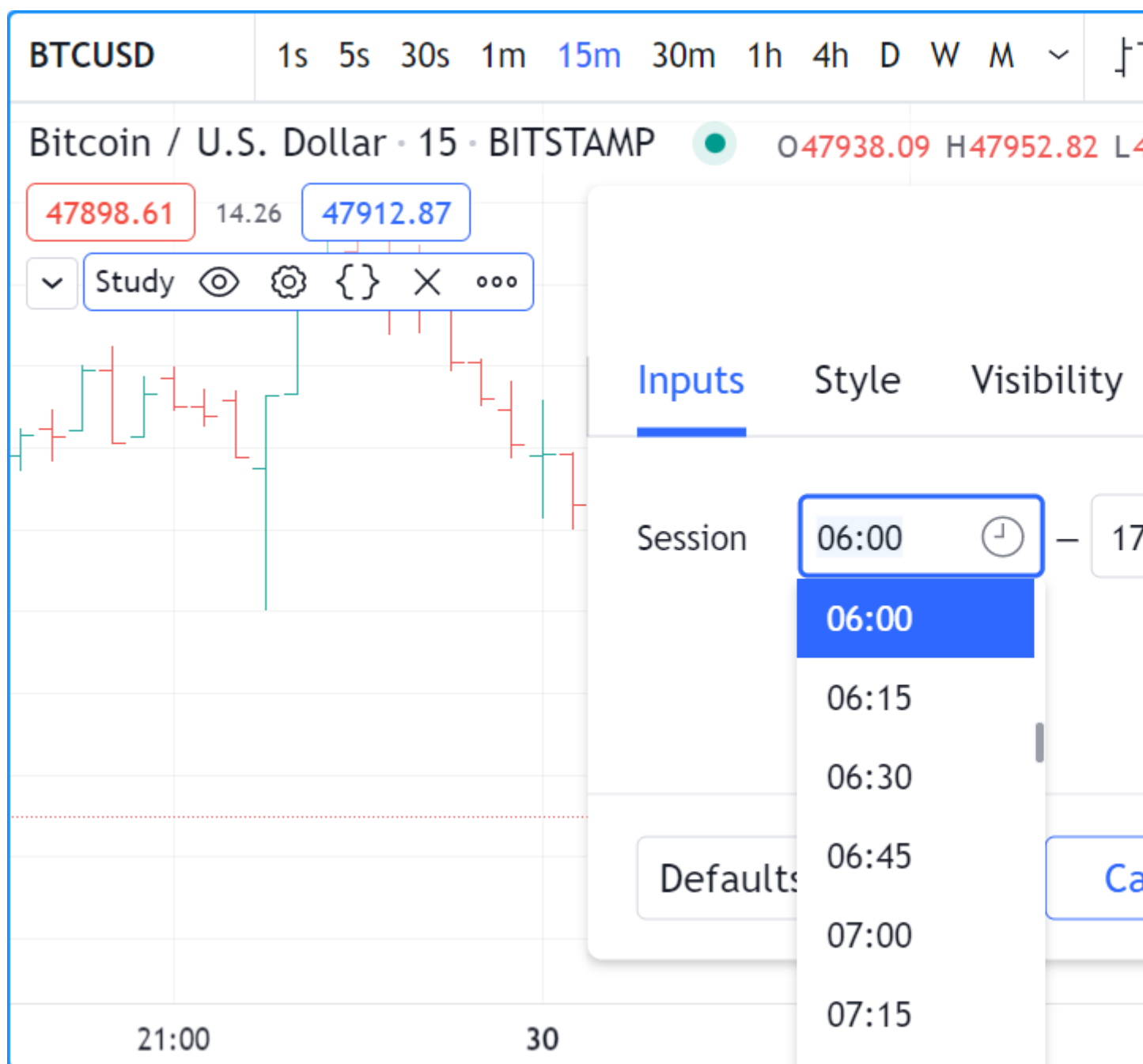
Session information can also contain information on the days where the session is valid. We use an [input.string\(\)](#) function call here to input that day information:

```
//@version=5
indicator("Session input", "", true)
string sessionInput = input.session("0600-1700", "Session")
string daysInput = input.string("1234567", tooltip = "1 = Sunday, 7 = Saturday")
sessionString = sessionInput + ":" + daysInput
inSession = not na(time(timeframe.period, sessionString))
bgcolor(inSession ? color.silver : na)
```

Note that:

- This script proposes a default session of “0600-1700”.
- The [input.string\(\)](#) call uses a tooltip to provide users with help on the format to use to enter day information.
- A complete session string is built by concatenating the two strings the script receives as inputs.
- We explicitly declare the type of our two inputs with the [string](#) keyword to make it clear those variables will contain a string.
- We detect if the chart bar is in the user-defined session by calling [time\(\)](#) with the session

string. If the current bar's [time](#) value (the time at the bar's [open](#)) is not in the session, [time\(\)](#) returns [na](#), so `inSession` will be true whenever [time\(\)](#) returns a value that is not [na](#).



Source input

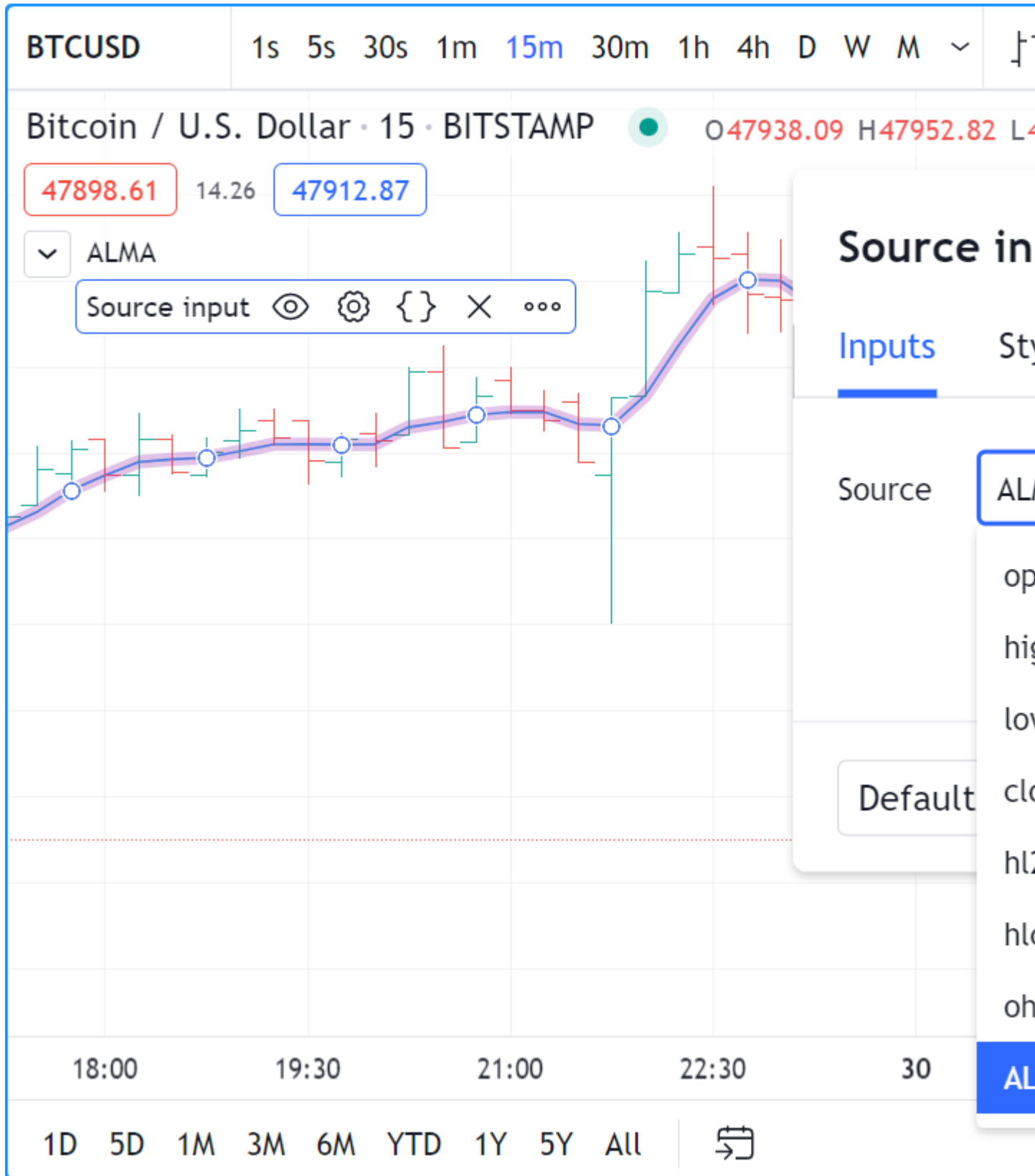
Source inputs are useful to provide a selection of two types of sources:

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

This script simply plots the user's selection of source. We propose the [high](#) as the default value:

```
//@version=5
indicator("Source input", "", true)
srcInput = input.source(high, "Source")
plot(srcInput, "Src", color.new(color.purple, 70), 6)
```

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



Time input

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

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

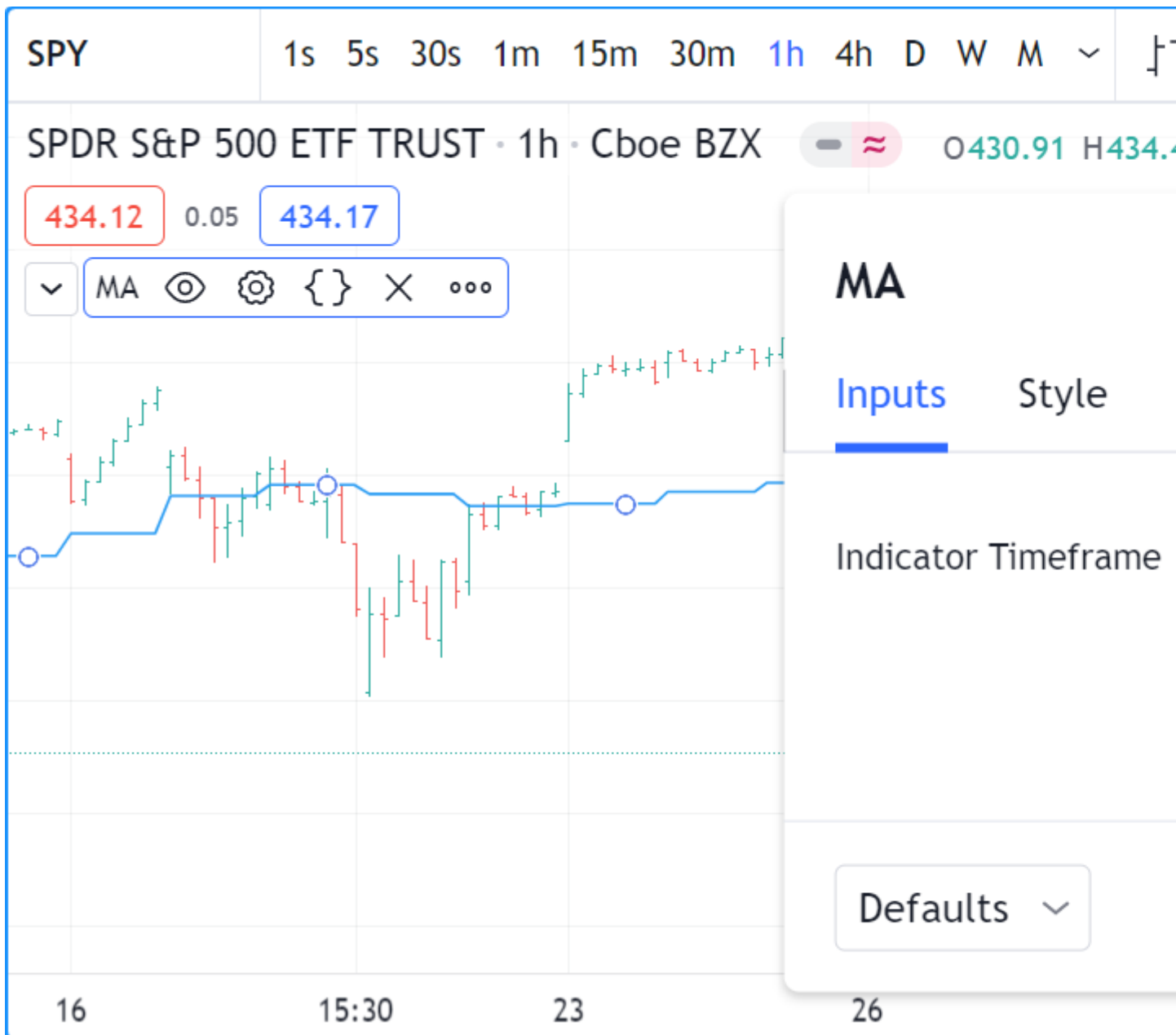
```
//@version=5
indicator("Time input", "T", true)
timeAndDateInput = input.time(timestamp("1 Aug 2021 00:00 +0300"), "Date and
time")
barIsLater = time > timeAndDateInput
plotchar(barIsLater, "barIsLater", "?", location.top, size = size.tiny)
```

Note that the defval value we use is a call to the [timestamp\(\)](#) function.

Other features affecting Inputs

Some parameters of the [indicator\(\)](#) function, when used, will populate the script's "Inputs" tab with a field. The parameters are `timeframe` and `timeframe_gaps`. An example:

```
//@version=5
indicator("MA", "", true, timeframe = "D", timeframe_gaps = false)
plot(ta.vwma(close, 10))
```



Tips

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

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

- Do the same for individual inputs **within** sections.

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

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

```
//@version=5
indicator("Aligned inputs", "", true)

var GRP1 = "Not aligned"
ma1SourceInput    = input(close, "MA source",      inline = "11", group = GRP1)
ma1LengthInput    = input(close, "Length",        inline = "11", group = GRP1)
long1SourceInput  = input(close, "Signal source",  inline = "12", group = GRP1)
long1LengthInput  = input(close, "Length",        inline = "12", group = GRP1)

var GRP2 = "Aligned"
// The three spaces after "MA source" are Unicode EN spaces (U+2002).
ma2SourceInput    = input(close, "MA source  ", inline = "21", group = GRP2)
ma2LengthInput    = input(close, "Length",      inline = "21", group = GRP2)
long2SourceInput  = input(close, "Signal source", inline = "22", group = GRP2)
long2LengthInput  = input(close, "Length",      inline = "22", group = GRP2)

plot(ta.vwma(close, 10))
```

Levels

- [`hline\(\)` levels](#)
- [Fills between levels](#)

[`hline\(\)` levels](#)

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

The function has the following signature:

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

[hline\(\)](#) has a few constraints when compared to [plot\(\)](#):

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

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

```
//@version=5
indicator("TSI")
myTSI = 100 * ta.tsi(close, 25, 13)
```

```
hline( 50, "+50", color.lime)
hline( 25, "+25", color.green)
hline(  0, "Zero", color.gray, linestyle = hline.style_dotted)
hline(-25, "-25", color.maroon)
hline(-50, "-50", color.red)
```

```
plot(myTSI)
```



Note that:

- We display 5 levels, each of a different color.
- We use a different line style for the zero centerline.

- We choose colors that will work well on both light and dark themes.
- The usual range for the indicator's values is +100 to -100. Since the `ta.tsi()` built-in returns values in the +1 to -1 range, we make the adjustment in our code.

Fills between levels

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

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

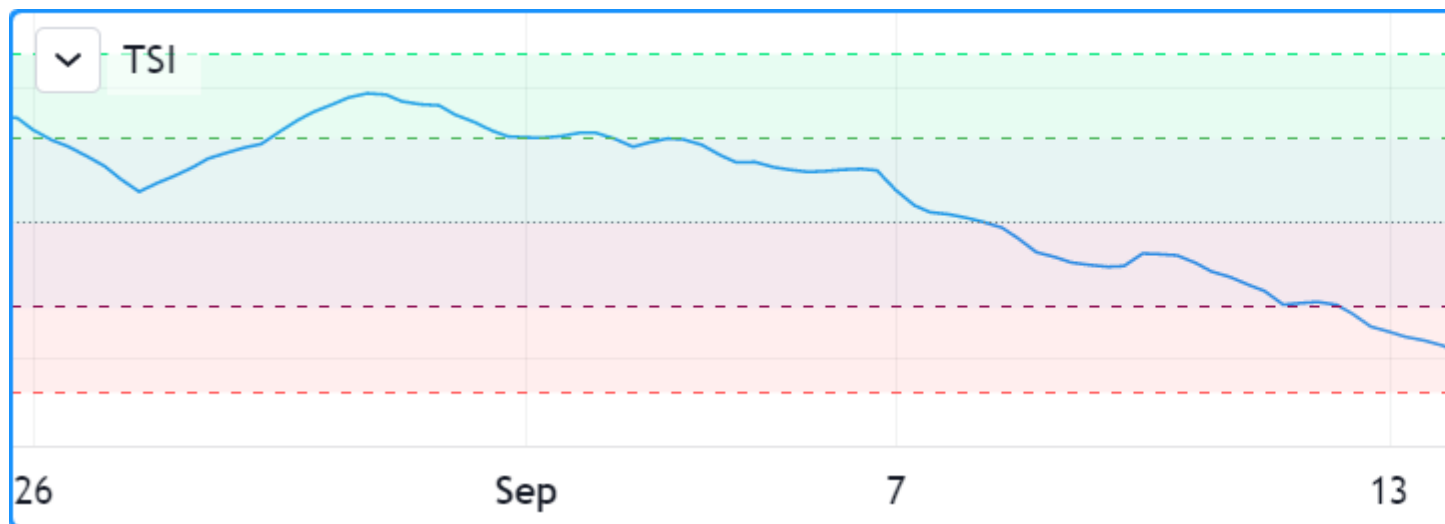
```
//@version=5
indicator("TSI")
myTSI = 100 * ta.tsi(close, 25, 13)

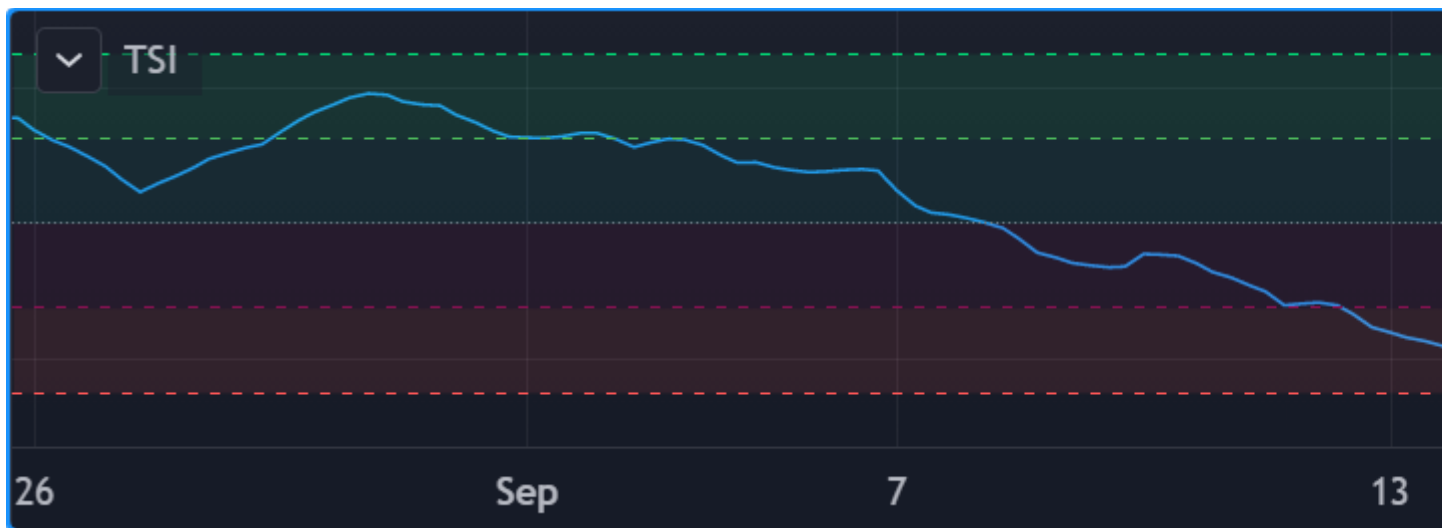
plus50Hline = hline( 50, "+50", color.lime)
plus25Hline = hline( 25, "+25", color.green)
zeroHline    = hline(  0, "Zero", color.gray, linestyle = hline.style_dotted)
minus25Hline = hline(-25, "-25", color.maroon)
minus50Hline = hline(-50, "-50", color.red)

// ——— Function returns a color in a light shade for use as a background.
fillColor(color col) =>
    color.new(col, 90)

fill(plus50Hline, plus25Hline, fillColor(color.lime))
fill(plus25Hline, zeroHline,    fillColor(color.teal))
fill(zeroHline,   minus25Hline, fillColor(color.maroon))
fill(minus25Hline, minus50Hline, fillColor(color.red))

plot(myTSI)
```





Note that:

- We have now used the return value of our `hline()` function calls, which is of the `hline` special type. We use the `plus50Hline`, `plus25Hline`, `zeroHline`, `minus25Hline` and `minus50Hline` variables to store those “hline” IDs because we will need them in our `fill()` calls later.
- To generate lighter color shades for the background colors, we declare a `fillColor()` function that accepts a color and returns its 90 transparency. We use calls to that function for the `color` arguments in our `fill()` calls.
- We make our `fill()` calls for each of the four different fills we want, between four different pairs of levels.
- We use `color.teal` in our second fill because it produces a green that fits the color scheme better than the `color.green` used for the 25 level.



SPY

1s 5s 30s 1m 15m 30m 1h 4h D W M

SPDR S&P 500 ETF TRUST · 1h · Cboe BZX

O 430.91 H 434.4

434.12

0.05

434.17



Aligned inputs



Aligned inp

Inputs

Style

NOT ALIGNED

MA source

close

Signal source

close

ALIGNED

MA source

close

Signal source

close

Defaults



16

15:30

23

26

Note that:

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



Libraries

- [Introduction](#)
- [Creating a library](#)
 - [Library functions](#)
 - [Argument form control](#)
 - [User-defined types and objects](#)
- [Publishing a library](#)
 - [House Rules](#)
- [Using a library](#)

Introduction

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

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

Library programmers should be familiar with Pine Script[®]'s typing nomenclature, scopes and user-defined functions. If you need to brush up on forms and types, see the User Manual's page on the [Type system](#). For more information on user-defined functions and scopes, see the [User-defined functions](#) page.

You can browse the library scripts published publicly by members in TradingView's [Community Scripts](#).

Creating a library

A library is a special kind of script that begins with the [library\(\)](#) declaration statement, rather than [indicator\(\)](#) or [strategy\(\)](#). A library contains exportable function definitions, which constitute the only visible part of the library when it is used by another script. Libraries can also use other Pine Script® code in their global scope, like a normal indicator. This code will typically serve to demonstrate how to use the library's functions.

A library script has the following structure, where one or more exportable functions must be defined:

```
//@version=5

// @description <library_description>
library(title, overlay)

<script_code>

// @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`, `// @function`, `// @param` and `// @returns` [compiler annotations](#) are optional but we highly recommend you use them. They serve a double purpose: 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 you would normally use in an indicator, including inputs or plots.

This is an example library:

```
//@version=5

// @description Provides functions calculating the all-time high/low of values.
library("AllTimeHighLow", true)

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

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

plot(hi())
```

```
plot(lo())
```

Library functions

Function definitions in libraries are slightly different than those of user-defined functions in indicators and strategies. There are constraints as to what can be included in the body of library functions.

In library function signatures (their first line):

- The [export](#) keyword is mandatory.
- The type of argument expected for each parameter must be explicitly mentioned.
- A [simple](#) or [series](#) form modifier can restrict the allowable forms of arguments (the next section explains their use).

These are the constraints imposed on library functions:

- They cannot use variables from the library's global scope unless they are of "const" form. This means you cannot use global variables initialized from script inputs, for example, or globally declared arrays.
- `request.*()` calls are not allowed.
- `input.*()` calls are not allowed.
- `plot*()`, `fill()` and `bgcolor()` calls are not allowed.

Library functions always return a result that is either of "simple" or "series" form. You cannot use them to calculate values where "const" or "input" forms are required, as is the case with some built-in functions. For example, a library function cannot be used to calculate an argument for the `show_last` parameter in a [plot\(\)](#) call, because an "input int" argument is required for `show_last`.

Argument form control

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

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

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

While library functions cannot return results of "const" or "input" forms, they can be written to produce a result of "simple" form. This makes them useful in more contexts than functions returning a result of "series" form, because some built-in functions do not allow "series" arguments. For example, [request.security\(\)](#) requires a "simple string" for its `symbol` parameter. If we wrote a library function to assemble the argument to `symbol` in the following way, the function's result would not work because it is of "series" form:

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

However, by restricting the form of its parameters to "simple", we could 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” result, no “series” values can be used in its calculation; otherwise the result will be of “series” form.

One can also use the [series](#) keyword to prefix the type of a library function parameter. However, because arguments are by default cast to the “series” form, using the [series](#) modifier is redundant; it exists more for completeness.

User-defined types and objects

You can export [user-defined types \(UDTs\)](#) from libraries, and library functions can return [objects](#).

To export a UDT, prefix its definition with the [export](#) keyword just as you would export a function:

```
//@version=5
library("Point")

export type point
  int x
  float y
  bool isHi
  bool wasBreached = false
```

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

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

Note that:

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

UDTs used in a library **must** be exported if any of its exported functions use a parameter or returns a result of that user-defined type.

When a library only uses a UDT internally, it does not have to be exported. The following library uses the `point` UDT internally, but only its `drawPivots()` function is exported, which does not use a parameter nor return a result of `point` type:

```
//@version=5
library("PivotLabels", true)

// We use this `point` UDT in the library, but it does NOT require exporting
because:
// 1. The exported function's parameters do not use the UDT.
// 2. The exported function does not return a UDT result.
type point
  int x
  float y
  bool isHi
```

```

bool wasBreached = false

fillPivotsArray(qtyLabels, leftLegs, rightLegs) =>
    // Create an array of the specified qty of pivots to maintain.
    var pivotsArray = array.new<point>(math.max(qtyLabels, 0))

    // Detect pivots.
    float pivotHi = ta.pivohigh(leftLegs, rightLegs)
    float pivotLo = ta.pivotlow(leftLegs, rightLegs)

    // Create a new `point` object when a pivot is found.
    point foundPoint = switch
        pivotHi => point.new(time[rightLegs], pivotHi, true)
        pivotLo => point.new(time[rightLegs], pivotLo, false)
    => na

    // Add new pivot info to the array and remove the oldest pivot.
    if not na(foundPoint)
        array.push(pivotsArray, foundPoint)
        array.shift(pivotsArray)

    array<point> result = pivotsArray

detectBreaches(pivotsArray) =>
    // Detect breaches.
    for [i, eachPoint] in pivotsArray
        if not na(eachPoint)
            if not eachPoint.wasBreached
                bool hiWasBreached = eachPoint.isHi and high[1] <=
eachPoint.y and high > eachPoint.y
                bool loWasBreached = not eachPoint.isHi and low[1] >=
eachPoint.y and low < eachPoint.y
                if hiWasBreached or loWasBreached
                    // This pivot was breached; change its `wasBreached` field.
                    point p = array.get(pivotsArray, i)
                    p.wasBreached := true
                    array.set(pivotsArray, i, p)

drawLabels(pivotsArray) =>
    for eachPoint in pivotsArray
        if not na(eachPoint)
            label.new(
                eachPoint.x,
                eachPoint.y,
                str.toString(eachPoint.y, format.mintick),
                xloc.bar_time,
                color = eachPoint.wasBreached ? color.gray : eachPoint.isHi ?
color.teal : color.red,
                style = eachPoint.isHi ? label.style_label_down:
label.style_label_up,
                textcolor = eachPoint.wasBreached ? color.silver : color.white)

// @function      Displays a label for each of the last `qtyLabels` pivots.
//               Colors high pivots in green, low pivots in red, and breached
pivots in gray.
// @param qtyLabels (simple int) Quantity of last labels to display.
// @param leftLegs (simple int) Left pivot legs.
// @param rightLegs (simple int) Right pivot legs.
// @returns      Nothing.
export drawPivots(int qtyLabels, int leftLegs, int rightLegs) =>

```

```

// Gather pivots as they occur.
pointsArray = fillPivotsArray(qtyLabels, leftLegs, rightLegs)

// Mark breached pivots.
detectBreaches(pointsArray)

// Draw labels once.
if barstate.islastconfirmedhistory
    drawLabels(pointsArray)

// Example use of the function.
drawPivots(20, 10, 5)

```

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

```

//@version=5
indicator("")
import TradingView/PivotLabels/1 as dpl
dpl.drawPivots(20, 10, 10)

```

Publishing a library

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

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

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



Note that:

- We leave the library's title as is (the `title` argument in our [library\(\)](#) declaration statement is used as the default). While you can change the publication's title, it is preferable to keep its default value because the `title` argument is used to reference imported libraries in the [import](#) statement. It makes life easier for library users when your publication's title matches the actual name of the library.
- A default description is built from the [compiler annotations](#) we used in our library. We will publish the library without 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>]
```

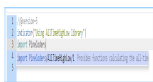
where:

- 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_name>()`. 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 + space` “Auto-complete” command to display a popup providing selections that match the available libraries:



This is an indicator that reuses our library:

```
//@version=5
indicator("Using AllTimeHighLow library", "", true)
import PineCoders/AllTimeHighLow/1 as allTime

plot(allTime.hi())
plot(allTime.lo())
plot(allTime.hi(close))
```

Note that:

- We have chosen to use the “allTime” alias for the library’s instance in our script. When typing that alias in the Editor, a popup will appear to help you select the particular function

you want to use from the library.

- We use the library's `hi()` and `lo()` functions without an argument, so the default [high](#) and [low](#) built-in variables will be used for their series, respectively.
- We use a second call to `allTime.hi()`, but this time using [close](#) as its argument, to plot the highest [close](#) in the chart's history.

Lines and boxes

- [Introduction](#)
- [Lines](#)
 - [Creating lines](#)
 - [Modifying lines](#)
 - [Line styles](#)
 - [Getting line properties](#)
 - [Cloning lines](#)
 - [Deleting lines](#)
- [Boxes](#)
 - [Creating boxes](#)
 - [Modifying boxes](#)
 - [Box styles](#)
 - [Getting box properties](#)
 - [Cloning boxes](#)
 - [Deleting boxes](#)
- [Realtime behavior](#)
- [Limitations](#)
 - [Total number of objects](#)
 - [Future references with ``xloc.bar_index``](#)
 - [Additional securities](#)
 - [Historical buffer and ``max_bars_back``](#)
- [Examples](#)
 - [Pivot Points Standard](#)
 - [Pivot Points High/Low](#)
 - [Linear Regression](#)
 - [Zig Zag](#)

[Introduction](#)

Lines and boxes are only available in v4 and higher versions of Pine Script[®]. They are useful to draw support and resistance levels, trend lines, price ranges. Multiple small line segments are also useful to draw complex geometric forms.

The flexibility lines and boxes allow in their positioning mechanism makes them particularly well-suited to drawing objects at points in the past that are detected a variable number of bars after the fact.

Lines and boxes are objects, like [labels](#) and [tables](#). Like them, they are referred to using an ID, which acts like a pointer. Line IDs are of “line” type, and box IDs are of “box” type. As with other

objects, lines and box IDs are “time series” and all the functions used to manage them accept “series” arguments, which makes them very flexible.

Note

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

Lines can be horizontal or at an angle, while boxes are always rectangular. Both share many common characteristics:

- They can start and end from any point on the chart, including the future.
- The functions used to manage them can be placed in conditional or loop structures, making it easier to control their behavior.
- They can be extended to infinity, left or right of their anchoring coordinates.
- Their attributes can be changed during the script’s execution.
- The *x* coordinates used to position them can be expressed as a bar index or a time value.
- In the *x* coordinate, they start and stop on the middle of the bar.
- Different pre-defined styles can be used for line patterns and end points, and box borders.
- A maximum of 500 of each can be drawn on the chart at any given time. The default is ~50, but you can use the `max_lines_count` and `max_boxes_count` parameters in your [indicator\(\)](#) or [strategy\(\)](#) declaration statement to specify up to 500. Lines and boxes, like labels, are managed using a garbage collection mechanism which deletes the oldest ones on the chart, such that only the most recently displayed are visible.

This script draws both lines and boxes:



```
//@version=5
indicator("Opening bar's range", "", true)
string tfInput = input.timeframe("D", "Timeframe")
// Initialize variables on bar zero only, so they preserve their values across
bars.
var hi = float(na)
var lo = float(na)
var line hiLine = na
var line loLine = na
var box hiLoBox = na
// Detect changes in timeframe.
bool newTF = ta.change(time(tfInput))
if newTF
    // New bar in higher timeframe; reset values and create new lines and box.
    hi := high
    lo := low
    hiLine := line.new(bar_index - 1, hi, bar_index, hi, color = color.green,
width = 2)
    loLine := line.new(bar_index - 1, lo, bar_index, lo, color = color.red,
width = 2)
    hiLoBox := box.new(bar_index - 1, hi, bar_index, lo, border_color = na,
bgcolor = color.silver)
    int(na)
else
    // On other bars, extend the right coordinate of lines and box.
    line.set_x2(hiLine, bar_index)
    line.set_x2(loLine, bar_index)
```

```

    box.set_right(hiLoBox, bar_index)
    // Change the color of the boxes' background depending on whether high/low
is higher/lower than the box.
    boxColor = high > hi ? color.green : low < lo ? color.red : color.silver
    box.set_bgcolor(hiLoBox, color.new(boxColor, 50))
    int(na)

```

Note that:

- We are detecting the first bar of a user-defined higher timeframe and saving its [high](#) and [low](#) values.
- We draw the `hi` and `low` levels using one line for each.
- We fill the space in between with a box.
- Every time we create two new lines and a box, we save their ID in variables `hiLine`, `loLine` and `hiLoBox`, which we then use in the calls to the setter functions to prolong these objects as new bars come in during the higher timeframe.
- We change the color of the boxes' background (`boxColor`) using the position of the bar's [high](#) and [low](#) with relative to the opening bar's same values. This entails that our script is repainting, as the boxes' color on past bars will change, depending on the current bar's values.
- We artificially make the return type of both branches of our `if` structure `int(na)` so the compiler doesn't complain about them not returning the same type. This occurs because [box.new\(\)](#) in the first branch returns a result of type "box", while [box.set_bgcolor\(\)](#) in the second branch returns type "void". See the [Matching local block type requirement](#) section for more information.

Lines

Lines are managed using built-in functions in the `line` namespace. They include:

- [line.new\(\)](#) to create them.
- `line.set_*` functions to modify the properties of an line.
- `line.get_*` functions to read the properties of an existing line.
- [line.copy\(\)](#) to clone them.
- [line.delete\(\)](#) to delete them.
- The [line.all](#) array which always contains the IDs of all the visible lines on the chart. The array's size will depend on the maximum line count for your script and how many of those you have drawn. `array.size(line.all)` will return the array's size.

Creating lines

The [line.new\(\)](#) function creates a new line. It has the following signature:

```
line.new(x1, y1, x2, y2, xloc, extend, color, style, width) → series line
```

Lines are positioned on the chart according to x (bars) and y (price) coordinates. Five parameters affect this behavior: `x1`, `y1`, `x2`, `y2` and `xloc`:

`x1` and `x2`

They are the x coordinates of the line's start and end points. They are either a bar index or a time value, as determined by the argument used for `xloc`. When a bar index is used, the value can be offset in the past (maximum of 5000 bars) or in the future (maximum of 500 bars). Past or future offsets can also be calculated when using time values. The `x1` and `x2` values of an existing line can be modified using [line.set_x1\(\)](#), [line.set_x2\(\)](#), [line.set_xy1\(\)](#) or

[line.set_xy2\(\)](#).

`xloc`

Is either [xloc.bar_index](#) (the default) or [xloc.bar_time](#). It determines which type of argument must be used with `x1` and `x2`. With [xloc.bar_index](#), `x1` and `x2` must be absolute bar indices. With [xloc.bar_time](#), `x1` and `x2` must be a UNIX timestamp in milliseconds corresponding to the [time](#) value of a bar's [open](#). The `xloc` value of an existing line can be modified using [line.set_xloc\(\)](#).

`y1` and `y2`

They are the `y` coordinates of the line's start and end points. While they are called price levels, they must be of values that make sense in the script's visual space. For an RSI indicator, they would typically be between 0 and 100, for example. When an indicator is running as an overlay, then the price scale will usually be that of the chart's symbol. The `y1` and `y2` values of an existing line can be modified using [line.set_y1\(\)](#), [line.set_y2\(\)](#), [line.set_xy1\(\)](#) or [line.set_xy2\(\)](#).

The remaining four parameters in [line.new\(\)](#) control the visual appearance of lines:

`extend`

Determines if the line is extended past its coordinates. It can be [extend.none](#), [extend.left](#), [extend.right](#) or [extend.both](#).

`color`

Is the line's color.

`style`

Is the style of line. See this page's [Line styles](#) section.

`width`

Determines the width of the line in pixels.

This is how you can create lines in their simplest form. We connect the preceding bar's [high](#) to the current bar's [low](#):



```
//@version=5
indicator("", "", true)
line.new(bar_index - 1, high[1], bar_index, low, width = 3)
```

Note that:

- We use a different `x1` and `x2` value: `bar_index - 1` and `bar_index`. This is necessary, otherwise no line would be created.
- We make the width of our line 3 pixels using `width = 3`.
- No logic controls our [line.new\(\)](#) call, so lines are created on every bar.
- Only approximately the last 50 lines are shown because that is the default value for the `max_lines_count` parameter in [indicator\(\)](#), which we haven't specified.
- Lines persist on bars until your script deletes them using [line.delete\(\)](#), or garbage collection removes them.

In this next example, we use lines to create probable travel paths for price. We draw a user-selected quantity of lines from the previous bar's center point between its [close](#) and [open](#) values. The lines project one bar after the current bar, after having been distributed along the [close](#) and [open](#) range of the current bar:



```
//@version=5
indicator("Price path projection", "PPP", true, max_lines_count = 100)
qtyOfLinesInput = input.int(10, minval = 1)

y2Increment = (close - open) / qtyOfLinesInput
// Starting point of the fan in y.
lineY1 = math.avg(close[1], open[1])
// Loop creating the fan of lines on each bar.
for i = 0 to qtyOfLinesInput
    // End point in y if line stopped at current bar.
    lineY2 = open + (y2Increment * i)
    // Extrapolate necessary y position to the next bar because we extend lines
    one bar in the future.
    lineY2 := lineY2 + (lineY2 - lineY1)
    lineColor = lineY2 > lineY1 ? color.lime : color.fuchsia
    line.new(bar_index - 1, lineY1, bar_index + 1, lineY2, color = lineColor)
```

Note that:

- We are creating a set of lines from within a [for](#) structure.
- We use the default `xloc = xloc.bar_index`, so our `x1` and `x2` values are bar indices.
- We want to start lines on the previous bar, so we use `bar_index - 1` for `x1` and `bar_index + 1` for `x2`.
- We use a “series color” value (its value can change in any of the loop’s iterations) for the line’s color. When the line is going up we make it lime; if not we make it fuchsia.
- The script will repaint in realtime because it is using the [close](#) and [open](#) values of the realtime bar to calculate line projections. Once the realtime bar closes, the lines drawn on elapsed realtime bars will no longer update.
- We use `max_lines_count = 100` in our [indicator\(\)](#) call to preserve the last 100 lines.

Modifying lines

The *setter* functions allowing you to change a line’s properties are:

- [line.set_x1\(\)](#)
- [line.set_y1\(\)](#)
- [line.set_xy1\(\)](#)
- [line.set_x2\(\)](#)
- [line.set_y2\(\)](#)
- [line.set_xy2\(\)](#)
- [line.set_xloc\(\)](#)
- [line.set_extend\(\)](#)
- [line.set_color\(\)](#)
- [line.set_style\(\)](#)
- [line.set_width\(\)](#)

They all have a similar signature. The one for [line.set_color\(\)](#) is:

```
line.set_color(id, color) → void
```

where:

- `id` is the ID of the line whose property is to be modified.
- The next parameter is the property of the line to modify. It depends on the setter function

used. [line.set_xy1\(\)](#) and [line.set_xy2\(\)](#) change two properties, so they have two such parameters.

In the next example we display a line showing the highest [high](#) value in the last `lookbackInput` bars. We will be using setter functions to modify an existing line:



```
//@version=5
MAX_BARS_BACK = 500
indicator("Last high", "", true, max_bars_back = MAX_BARS_BACK)

repaintInput = input.bool(false, "Position bars in the past")
lookbackInput = input.int(50, minval = 1, maxval = MAX_BARS_BACK)

// Keep track of highest `high` and detect when it changes.
hi = ta.highest(lookbackInput)
newHi = ta.change(hi)
// Find the offset to the highest `high` in last 50 bars. Change it's sign so it
is positive.
highestBarOffset = - ta.highestbars(lookbackInput)
// Create label on bar zero only.
var lbl = label.new(na, na, "", color = color(na), style =
label.style_label_left)
var lin = line.new(na, na, na, na, xloc = xloc.bar_time, style =
line.style_arrow_right)
// When a new high is found, move the label there and update its text and
tooltip.
if newHi
    // Build line.
    lineX1 = time[highestBarOffset + 1]
    // Get the `high` value at that offset. Note that `highest(50)` would be
equivalent,
    // but it would require evaluation on every bar, prior to entry into this
`if` structure.
    lineY = high[highestBarOffset]
    // Determine line's starting point with user setting to plot in past or not.
    line.set_xy1(lin, repaintInput ? lineX1 : time[1], lineY)
    line.set_xy2(lin, repaintInput ? lineX1 : time, lineY)

    // Reposition label and display new high's value.
    label.set_xy(lbl, bar_index, lineY)
    label.set_text(lbl, str.toString(lineY, format.mintick))
else
    // Update line's right end point and label to current bar's.
    line.set_x2(lin, time)
    label.set_x(lbl, bar_index)

// Show a blue dot when a new high is found.
plotchar(newHi, "newHighFound", "•", location.top, size = size.tiny)
```



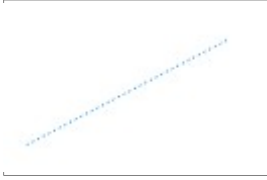

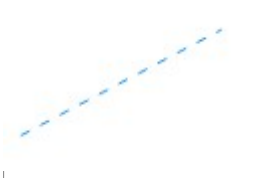

Note that:

- We plot the line starting on the bar preceding the point where the new high is found. We draw the line from the preceding bar so that we see a one bar line when a new high is found.
- We only start the line in the past, from the actual highest point, when the user explicitly chooses to do so through the script's inputs. This gives the user control over the repainting behavior of the script. It also avoids misleading traders into thinking that our script is prescient and can know in advance if a high point will still be the high point in the lookback period n bars later.

- We manage the historical buffer to avoid runtime errors when referring to bars too far away in the past. We do two things for this: we use the `max_bars_back` parameter in our [indicator\(\)](#) call, and we cap the input for `lookbackInput` using `maxval` in our [input.int\(\)](#) call. Rather than use the 500 literal in two places, we create a `MAX_BARS_BACK` constant for it.
- We create our line and label on the first bar only, using [var](#). From that point on, we only need to update their properties, so we are moving the same line and label along, resetting their position and the label's text when a new high is found, and then only updating their x coordinates as new bars come in. We use the [line.set_xy1\(\)](#) and [line.set_xy1\(\)](#) functions when we find a new high, and [line.set_x2\(\)](#) on other bars, to extend the line.
- We use time values for $x1$ and $x2$ because our [line.new\(\)](#) call specifies `xloc = xloc.bar_time`.
- We use `style = line.style_arrow_right` in our [line.new\(\)](#) call to display a right arrow line style.
- Even though our label's background is not visible, we use `style = label.style_label_left` in our [label.new\(\)](#) call so that the price value is positioned to the right of the chart's last bar.
- To better visualize on which bars a new high is found, we plot a blue dot using [plotchar\(\)](#). Note that this does not necessarily entail the bar where it appears is the new highest value. While this may happen, a new highest value can also be calculated because a long-standing high has dropped off from the lookback length and been replaced by another high that may not be on the bar where the blue dot appears.
- Our chart cursor points to the bar with the highest value in the last 50 bars.
- When the user does not choose to plot in the past, our script does not repaint.

Line styles

Various styles can be applied to lines with either the [line.new\(\)](#) or [line.set_style\(\)](#) functions:

Argument	Line	Argument	Line
<code>line.style_solid</code>		<code>line.style_arrow_left</code>	
<code>line.style_dotted</code>		<code>line.style_arrow_right</code>	
<code>line.style_dashed</code>		<code>line.style_arrow_both</code>	

Getting line properties

The following *getter* functions are available for lines:

- [line.get_price\(\)](#)
- [line.get_x1\(\)](#)

- [line.get_y1\(\)](#)
- [line.get_x2\(\)](#)
- [line.get_y2\(\)](#)

The signature for [line.get_price\(\)](#) is:

```
line.get_price(id, x) → series float
```

where:

- `id` is the line whose `x1` value is to be retrieved
- `x` is the bar index of the point on the line whose `y` coordinate is to be returned.

The last four functions all have a similar signature. The one for [line.get_x1\(\)](#) is:

```
line.get_x1(id) → series int
```

where `id` is the ID of the line whose `x1` value is to be retrieved.

Cloning lines

The [line.copy\(\)](#) function is used to clone lines. Its syntax is:

```
line.copy(id) → void
```

Deleting lines

The [line.delete\(\)](#) function is used to delete lines. Its syntax is:

```
line.delete(id) → void
```

To keep only a user-defined quantity of lines on the chart, one could use code like this, where we are drawing a level every time RSI rises/falls for a user-defined quantity of consecutive bars:

LTCUSD 1s 5s 30s 1m 15m 30m 1h 4h D W M ▾ ⌵ ⊕

Litecoin / U.S. Dollar - 1 - COINBASE ● O142.49 H142.56 L142.47 C142.56

142.50 0.15 142.65



RSI levels 51.49 0.00 0.00



```
//@version=5
int MAX_LINES_COUNT = 500
indicator("RSI levels", max_lines_count = MAX_LINES_COUNT)
```

```

int linesToKeepInput = input.int(10, minval = 1, maxval = MAX_LINES_COUNT)
int sensitivityInput = input.int(5, minval = 1)

float myRSI = ta.rsi(close, 20)
bool myRSIRises = ta.rising(myRSI, sensitivityInput)
bool myRSIFalls = ta.falling(myRSI, sensitivityInput)
if myRSIRises or myRSIFalls
    color lineColor = myRSIRises ? color.new(color.green, 70) :
color.new(color.red, 70)
    line.new(bar_index, myRSI, bar_index + 1, myRSI, color = lineColor, width =
2)
    // Once the new line is created, delete the oldest one if we have too many.
    if array.size(line.all) > linesToKeepInput
        line.delete(array.get(line.all, 0))
    int(na)
else
    // Extend all visible lines.
    int lineNo = 0
    while lineNo < array.size(line.all)
        line.set_x2(array.get(line.all, lineNo), bar_index)
        lineNo += 1
    int(na)

plot(myRSI)
hline(50)
// Plot markers to show where our triggering conditions are `true`.
plotchar(myRSIRises, "myRSIRises", "?", location.top, color.green, size =
size.tiny)
plotchar(myRSIFalls, "myRSIFalls", "?", location.bottom, color.red, size =
size.tiny)

```

Note that:

- We define a `MAX_LINES_COUNT` constant to hold the maximum quantity of lines a script can accommodate. We use that value to set the `max_lines_count` parameter's value in our [indicator\(\)](#) call, and also as the `maxval` value in our [input.int\(\)](#) call, to cap the user value.
- We use the `myRSIRises` and `myRSIFalls` variables to hold the states determining when we create a new level. After that, we delete the oldest line in the [line.all](#) built-in array that is automatically maintained by the Pine Script® runtime and contains the ID of all the visible lines drawn by our script. We use the [array.get\(\)](#) function to retrieve the array element at index zero (the oldest visible line ID). We then use [line.delete\(\)](#) to delete the line referenced by that ID.
- Again, we need to artificially return `int(na)` in both local blocks of our [if](#) structure so the compiler doesn't complain. See the [Matching local block type requirement](#) section for more information.
- This time, we mention the type of variables explicitly when we declare them, as in `float myRSI = ta.rsi(close, 20)`. The declarations are functionally redundant, but they can help make your intention clear to readers of your code — you being the one who will read it the most frequently.

Boxes

Boxes are managed using built-in functions in the `box` namespace. They include:

- [box.new\(\)](#) to create them.

- `box.set_*()` functions to modify the properties of a box.
- `box.get_*()` functions to read some of the properties of an existing box.
- [box.copy\(\)](#) to clone them.
- [box.delete\(\)](#) to delete them.
- The [box.all](#) array which always contains the IDs of all the visible boxes on the chart. The array's size will depend on the maximum box count for your script and how many of those you have drawn. `array.size(box.all)` will return the array's size.

Creating boxes

The [box.new\(\)](#) function creates a new line. It has the following signature:

```
box.new(left, top, right, bottom, border_color, border_width, border_style,
extend, xloc, bgcolor) → series box
```

Boxes are positioned on the chart according to *x* (bars) and *y* (price) coordinates. Five parameters affect this behavior: `left`, `top`, `right`, `bottom` and `xloc`:

`left` and `right`

They are the *x* coordinates of the line's start and end points. They are either a bar index or a time value, as determined by the argument used for `xloc`. When a bar index is used, the value can be offset in the past (maximum of 5000 bars) or in the future (maximum of 500 bars). Past or future offsets can also be calculated when using time values. The `left` and `right` values of an existing line can be modified using [box.set_left\(\)](#), [box.set_right\(\)](#), [box.set_lefttop\(\)](#) or [box.set_rightbottom\(\)](#).

`xloc`

Is either [xloc.bar_index](#) (the default) or [xloc.bar_time](#). It determines which type of argument must be used with `left` and `right`. With [xloc.bar_index](#), `left` and `right` must be absolute bar indices. With [xloc.bar_time](#), `left` and `right` must be a UNIX timestamp in milliseconds corresponding to a value between the bar's [time](#) (opening time) and [time_close](#) (closing time) values.

`top` and `bottom`

They are the *y* coordinates of the boxes' top and bottom levels (boxes are always rectangular). While they are called price levels, they must be of values that make sense in the script's visual space. For an RSI indicator, they would typically be between 0 and 100, for example. When an indicator is running as an overlay, then the price scale will usually be that of the chart's symbol. The `top` and `bottom` values of an existing line can be modified using [box.set_top\(\)](#), [box.set_bottom\(\)](#), [box.set_lefttop\(\)](#) or [box.set_rightbottom\(\)](#).

The remaining five parameters in [box.new\(\)](#) control the visual appearance of boxes:

`border_color`

Is the border's color. It defaults to [color.blue](#).

`border_width`

Determines the width of the border in pixels.

`border_style`

Is the style of border. See this page's [Box styles](#) section.

`extend`

Determines if the borders is extended past the box's coordinates. It can be [extend.none](#), [extend.left](#), [extend.right](#) or [extend.both](#).

`bgcolor`

Is the background color of the box. It defaults to [color.blue](#).

Let's create simple boxes:



```
//@version=5
indicator("", "", true)
box.new(bar_index, high, bar_index + 1, low, border_color = color.gray, bgcolor
= color.new(color.silver, 60))
```

Note that:

- The start and end points of boxes, like lines, are always the **horizontal center** of bars.
- We start these boxes at `bar_index` and end them on `bar_index + 1` (the following bar in the future) so that we get an actual box. If we used `bar_index` for both coordinates, only a vertical line would be drawn in the center of the bar.
- No logic controls our `box.new()` call, so boxes are created on every bar.
- Only approximately the last 50 boxes are shown because that is the default value for the `max_boxes_count` parameter in `indicator()`, which we haven't specified.
- Boxes persist on bars until your script deletes them using `box.delete()`, or garbage collection removes them.

Modifying boxes

The available *setter* functions for box drawings are:

- [box.set_left\(\)](#)
- [box.set_top\(\)](#)
- [box.set_lefttop\(\)](#)
- [box.set_right\(\)](#)
- [box.set_bottom\(\)](#)
- [box.set_rightbottom\(\)](#)
- [box.set_border_color\(\)](#)
- [box.set_border_width\(\)](#)
- [box.set_border_style\(\)](#)
- [box.set_extend\(\)](#)
- [box.set_bgcolor\(\)](#)

Note that contrary to lines, there is no setter function to modify `xloc` for boxes.

This script uses setter functions to update boxes. It detects the largest up and down volume bars during a user-defined timeframe and draws boxes with the [high](#) and [low](#) levels of those bars. If a higher volume bar comes in, the timeframe's box is redrawn using the new bar's [high](#) and [low](#) levels:



```
//@version=5
indicator("High volume bar boxes", "", true)

string tfInput      = input.timeframe("D", "Resetting timeframe")
int    transpInput  = 100 - input.int(100, "Line Brightness", minval = 0, maxval
= 100, step = 5, inline = "1", tooltip = "100 is brightest")
int    widthInput   = input.int(2, "Width", minval = 0, maxval = 100, step = 5,
inline = "1")
color  upColorInput = input.color(color.lime, "?", inline = "1")
```

```

color dnColorInput = input.color(color.fuchsia, "?", inline = "1")

bool newTF = ta.change(time(tfInput))
bool barUp = close > open

// These keep track of highest up/dn volume found during the TF.
var float hiVolUp = na
var float hiVolDn = na
// These always hold the IDs of the current TFs boxes.
var box boxUp = na
var box boxDn = na

if newTF and not na(volume)
    // New TF begins; create new boxes, one of which will be invisible.
    if barUp
        hiVolUp := volume
        hiVolDn := na
        boxUp := box.new(bar_index, high, bar_index + 1, low, border_color =
color.new(upColorInput, transpInput), border_width = widthInput, bgcolor = na)
        boxDn := box.new(na, na, na, na, border_color = color.new(dnColorInput,
transpInput), border_width = widthInput, bgcolor = na)
    else
        hiVolDn := volume
        hiVolUp := na
        boxDn := box.new(bar_index, high, bar_index + 1, low, border_color =
color.new(dnColorInput, transpInput), border_width = widthInput, bgcolor = na)
        boxUp := box.new(na, na, na, na, border_color = color.new(upColorInput,
transpInput), border_width = widthInput, bgcolor = na)
    int(na)
else
    // On bars during the HTF, keep tracks of highest up/dn volume bar.
    if barUp
        hiVolUp := math.max(nz(hiVolUp), volume)
    else
        hiVolDn := math.max(nz(hiVolDn), volume)
    // If a new bar has higher volume, reset its box.
    if hiVolUp > nz(hiVolUp[1])
        box.set_lefttop(boxUp, bar_index, high)
        box.set_rightbottom(boxUp, bar_index + 1, low)
    else if hiVolDn > nz(hiVolDn[1])
        box.set_lefttop(boxDn, bar_index, high)
        box.set_rightbottom(boxDn, bar_index + 1, low)
    int(na)

// On all bars, extend right side of both boxes.
box.set_right(boxUp, bar_index + 1)
box.set_right(boxDn, bar_index + 1)
// Plot circle mark on TF changes.
plotchar(newTF, "newTF", "•", location.top, size = size.tiny)

```

Note that:

- We use the `inline` parameter in the inputs relating to the boxes' visual appearance to place them on the same line.
- We subtract the 0-100 brightness level given by the user from 100 to generate the correct transparency for our box borders. We do this because it is more intuitive for users to specify a brightness level where 100 represents maximum brightness. We provide a tooltip to explain the scale.
- When a new higher timeframe bar comes in and the symbol's feed contains volume data, we reset our information. If the timeframe's first bar is up, we create a new visible `boxUp` box and an invisible `boxDn` box. We do the inverse if the first bar's polarity is down. We take

care to reassign the IDs of the newly created boxes to `boxUp` and `boxDown` so we will be able to update those boxes later in the script. This is possible because we have declared those variables with [var](#). See the section on the [var declaration mode](#) for more information.

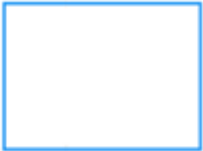


- On all other chart bars belonging to the same higher timeframe bar, we monitor volume values to keep track of the highest. If a new higher volume bar is encountered, we reset the corresponding box's coordinates on that new bar using [box.set_lefttop\(\)](#) and [box.set_rightbottom\(\)](#).
- On all bars, we extend the right side of the timeframe's two boxes using [box.set_right\(\)](#).
- Approximately the last 50 boxes will be visible on the chart because we do not use `max_boxes_count` in our [indicator\(\)](#) call to change its default value.

This is our script's "Settings/Inputs" tab:



Box styles

Various styles can be applied to boxes with either the [box.new\(\)](#) or [box.set_border_style\(\)](#) functions:

Argument	Box
<code>line.style_solid</code>	
<code>line.style_dotted</code>	
<code>line.style_dashed</code>	

Getting box properties

The following *getter* functions are available for boxes:

- [box.get_bottom\(\)](#)
- [box.get_left\(\)](#)
- [box.get_right\(\)](#)
- [box.get_top\(\)](#)

The signature for [box.get_top\(\)](#) is:

```
box.get_top(id) → series float
```

where `id` is the ID of the box whose `top` value is to be retrieved.

Cloning boxes

The `box.copy()` function is used to clone boxes. Its syntax is:

```
box.copy(id) → void
```

Deleting boxes

The `box.delete()` function is used to delete boxes. Its signature is:

```
box.delete(id) → void
```

Realtime behavior

Lines and boxes are subject to both *commit* and *rollback* actions, which affect the behavior of a script when it executes in the realtime bar. See the page on Pine Script®'s [Execution model](#).

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

```
//@version=5
indicator("My Script", overlay = true)
line.new(bar_index, high, bar_index, low, width = 6)
```

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

Limitations

Total number of objects

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

This code creates a line on every bar:

```
//@version=5
indicator("", "", true)
line.new(bar_index, high, bar_index, low, width = 6)
```

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



You can change the drawing limit to a value in range from 1 to 500 using the `max_lines_count` and `max_boxes_count` parameters in the `indicator()` or `strategy()` functions:

```
//@version=5
indicator("", "", true, max_lines_count = 100)
line.new(bar_index, high, bar_index, low, width = 6)
```


Future references with `xloc.bar_index`

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

Additional securities

Lines and boxes cannot be managed in functions sent with `request.security()` calls. While they can use values fetched through `request.security()`, they must be drawn in the main symbol's context.

This is also the reason why line and box drawing code will not work in scripts using the `timeframe` parameter in `indicator()`.

Historical buffer and `max_bars_back`

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

```
//@version=5
indicator("My Script", overlay = true)

if barstate.isrealtime
    label.new(bar_index[300], na, text = "Label", yloc = yloc.abovebar)
```

It will, however, fail at runtime. The reason for the error is that the script cannot determine the buffer size for historical values of the `time` plot, even though the `time` built-in variable isn't mentioned in the code. This is due to the fact that the built-in variable `bar_index` uses the `time` series in its inner workings. Accessing the value of the bar index 300 bars back requires that the history buffer size of the `time` series be of size 300 or more.

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

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

```
//@version=5
indicator("My Script", overlay = true)

max_bars_back(time, 300)

if barstate.isrealtime
    label.new(bar_index[300], na, text = "Label", yloc = yloc.abovebar)
```

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

Examples

Pivot Points Standard



```

//@version=5
indicator("Pivot Points Standard", overlay = true)
higherTFInput = input.timeframe("D")
prevCloseHTF = request.security(syminfo.tickerid, higherTFInput, close[1],
lookahead = barmerge.lookahead_on)
prevOpenHTF = request.security(syminfo.tickerid, higherTFInput, open[1],
lookahead = barmerge.lookahead_on)
prevHighHTF = request.security(syminfo.tickerid, higherTFInput, high[1],
lookahead = barmerge.lookahead_on)
prevLowHTF = request.security(syminfo.tickerid, higherTFInput, low[1], lookahead
= barmerge.lookahead_on)

pLevel = (prevHighHTF + prevLowHTF + prevCloseHTF) / 3
r1Level = pLevel * 2 - prevLowHTF
s1Level = pLevel * 2 - prevHighHTF

var line r1Line = na
var line pLine = na
var line s1Line = na

if pLevel[1] != pLevel
    line.set_x2(r1Line, bar_index)
    line.set_x2(pLine, bar_index)
    line.set_x2(s1Line, bar_index)
    line.set_extend(r1Line, extend.none)
    line.set_extend(pLine, extend.none)
    line.set_extend(s1Line, extend.none)
    r1Line := line.new(bar_index, r1Level, bar_index, r1Level, extend =
extend.right)
    pLine := line.new(bar_index, pLevel, bar_index, pLevel, width=3, extend =
extend.right)
    s1Line := line.new(bar_index, s1Level, bar_index, s1Level, extend =
extend.right)
    label.new(bar_index, r1Level, "R1", style = label.style_none)
    label.new(bar_index, pLevel, "P", style = label.style_none)
    label.new(bar_index, s1Level, "S1", style = label.style_none)

if not na(pLine) and line.get_x2(pLine) != bar_index
    line.set_x2(r1Line, bar_index)
    line.set_x2(pLine, bar_index)
    line.set_x2(s1Line, bar_index)

```

Pivot Points High/Low



```

//@version=5
indicator("Pivot Points High Low", "Pivots HL", true)

int lenHInput = input.int(10, "Length High", minval = 1)
int lenLInput = input.int(10, "Length Low", minval = 1)

float pivotHigh = ta.pivohigh(high, lenHInput, lenHInput)
float pivotLow = ta.pivotlow(low, lenLInput, lenLInput)

float pivot = 0.0
if not na(pivotHigh)
    pivot := nz(high[lenHInput])
    label.new(nz(bar_index[lenHInput]), pivot, str.tostring(pivot,
format.mintick), style = label.style_label_down, yloc = yloc.abovebar, color =
color.lime)

```

```

if not na(pivotLow)
    pivot := nz(low[lenLInput])
    label.new(nz(bar_index[lenLInput]), pivot, str.tostring(pivot,
format.mintick), style = label.style_label_up, yloc = yloc.belowbar, color =
color.red)

```

Linear Regression



```

//@version=5
indicator('Linear Regression', shorttitle='LinReg', overlay=true)

upperMult = input(title='Upper Deviation', defval=2)
lowerMult = input(title='Lower Deviation', defval=-2)

useUpperDev = input(title='Use Upper Deviation', defval=true)
useLowerDev = input(title='Use Lower Deviation', defval=true)
showPearson = input(title='Show Pearson\'s R', defval=true)
extendLines = input(title='Extend Lines', defval=false)

len = input(title='Count', defval=100)
src = input(title='Source', defval=close)

extend = extendLines ? extend.right : extend.none

calcSlope(src, len) =>
    if not barstate.islast or len <= 1
        [float(na), float(na), float(na)]
    else
        sumX = 0.0
        sumY = 0.0
        sumXSqr = 0.0
        sumXY = 0.0
        for i = 0 to len - 1 by 1
            val = src[i]
            per = i + 1.0
            sumX := sumX + per
            sumY := sumY + val
            sumXSqr := sumXSqr + per * per
            sumXY := sumXY + val * per
        sumXY
        slope = (len * sumXY - sumX * sumY) / (len * sumXSqr - sumX * sumX)
        average = sumY / len
        intercept = average - slope * sumX / len + slope
        [slope, average, intercept]

[s, a, intercpt] = calcSlope(src, len)

startPrice = intercpt + s * (len - 1)
endPrice = intercpt
var line baseLine = na

if na(baseLine) and not na(startPrice)
    baseLine := line.new(bar_index - len + 1, startPrice, bar_index, endPrice,
width = 1, extend=extend, color = color.red)
    baseLine
else
    line.set_xy1(baseLine, bar_index - len + 1, startPrice)
    line.set_xy2(baseLine, bar_index, endPrice)
    na

```

```

calcDev(src, len, slope, average, intercept) =>
  upDev = 0.0
  dnDev = 0.0
  stdDevAcc = 0.0
  dsxx = 0.0
  dsyy = 0.0
  dsxy = 0.0

  periods = len - 1

  daY = intercept + slope * periods / 2
  val = intercept

  for i = 0 to periods by 1
    price = high[i] - val
    if price > upDev
      upDev := price
      upDev

    price := val - low[i]
    if price > dnDev
      dnDev := price
      dnDev

    price := src[i]
    dxt = price - average
    dyt = val - daY

    price := price - val
    stdDevAcc := stdDevAcc + price * price
    dsxx := dsxx + dxt * dxt
    dsyy := dsyy + dyt * dyt
    dsxy := dsxy + dxt * dyt
    val := val + slope
    val

  stdDev = math.sqrt(stdDevAcc / (periods == 0 ? 1 : periods))
  pearsonR = dsxx == 0 or dsyy == 0 ? 0 : dsxy / math.sqrt(dsxx * dsyy)
  [stdDev, pearsonR, upDev, dnDev]

[stdDev, pearsonR, upDev, dnDev] = calcDev(src, len, s, a, intercpt)

upperStartPrice = startPrice + (useUpperDev ? upperMult * stdDev : upDev)
upperEndPrice = endPrice + (useUpperDev ? upperMult * stdDev : upDev)
var line upper = na

lowerStartPrice = startPrice + (useLowerDev ? lowerMult * stdDev : -dnDev)
lowerEndPrice = endPrice + (useLowerDev ? lowerMult * stdDev : -dnDev)
var line lower = na

if na(upper) and not na(upperStartPrice)
  upper := line.new(bar_index - len + 1, upperStartPrice, bar_index,
  upperEndPrice, width=1, extend=extend, color=#0000ff)
  upper
else
  line.set_xy1(upper, bar_index - len + 1, upperStartPrice)
  line.set_xy2(upper, bar_index, upperEndPrice)
  na

if na(lower) and not na(lowerStartPrice)
  lower := line.new(bar_index - len + 1, lowerStartPrice, bar_index,
  lowerEndPrice, width=1, extend=extend, color=#0000ff)
  lower

```

```

else
    line.set_xy1(lower, bar_index - len + 1, lowerStartPrice)
    line.set_xy2(lower, bar_index, lowerEndPrice)
    na

// Pearson's R
var label r = na
transparent = color.new(color.white, 100)
label.delete(r[1])
if showPearson and not na(pearsonR)
    r := label.new(bar_index - len + 1, lowerStartPrice, str.tostring(pearsonR,
'#.#####'), color=transparent, textcolor=#0000ff, size=size.normal,
style=label.style_label_up)
    r

```

Zig Zag



```

//@version=5
indicator('Zig Zag', overlay = true)

float dev_threshold = input.float(title = 'Deviation (%)', defval = 5, minval =
1, maxval = 100)
int depth          = input.int(title = 'Depth', defval = 10, minval = 1)

type Point
    int     index
    float   price

type Pivot
    line    ln
    bool    isHigh
    Point   point

var pivotArray = array.new<Pivot>()
int length     = math.floor(depth / 2)
float pH       = ta.pivohigh(high, length, length)
float pL       = ta.pivotlow(low, length, length)

calcDeviation(base_price, price) =>
    100 * math.abs(price - base_price) / base_price

newPivot(Point lastPoint, bool isHigh, int index, float price) =>
    line    ln      = line.new(lastPoint.index, lastPoint.price, index, price,
color = color.red, width = 2)
    Pivot   pivot   = Pivot.new(ln, isHigh, Point.new(index, price))
    array.push(pivotArray, pivot)
    pivot

updatePivot(Pivot pivot, int index, float price) =>
    line ln = pivot.ln
    line.set_xy2(ln, index, price)
    pivot.point.index := index
    pivot.point.price := price
    pivot

isPivotFound(bool isHigh, float price) =>
    bool result = false
    int index = bar_index[length]
    int size = array.size(pivotArray)

```

```

    Pivot prevPivot = size >= 1 ? array.get(pivotArray, size - 1) :
newPivot(Point.new(index, price), isHigh, index, price)

    if prevPivot.isHigh and not na(prevPivot.ln)
        m = isHigh ? 1 : -1
        if price * m > prevPivot.point.price * m
            updatePivot(prevPivot, index, price)
            result := true
        else if na(prevPivot.ln) or math.abs(calcDeviation(prevPivot.point.price,
price)) >= dev_threshold
            newPivot(prevPivot.point, isHigh, index, price)
            result := true
        result

isPivotFound(true, pH)
isPivotFound(false, pL)

```

Non-standard charts data

- [Introduction](#)
- [`ticker.heikinashi\(\)`](#)
- [`ticker.renko\(\)`](#)
- [`ticker.linebreak\(\)`](#)
- [`ticker.kagi\(\)`](#)
- [`ticker.pointfigure\(\)`](#)

Introduction

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

`ticker.heikinashi()`

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

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

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



```

//@version=5
indicator("HA Close", "", true)
haTicker = ticker.heikinashi(syminfo.tickerid)
haClose = request.security(haTicker, timeframe.period, close)
plot(haClose, "HA Close", color.black, 3)

```

Note that:

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

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



```
//@version=5
indicator("HA Close", "", true)
regularSessionTicker = ticker.new(syminfo.prefix, syminfo.ticker,
session.regular)
haTicker = ticker.heikinashi(regularSessionTicker)
haClose = request.security(haTicker, timeframe.period, close, gaps =
barmerge.gaps_on)
plot(haClose, "HA Close", color.black, 3, plot.style_linebr)
```

Note that:

- We use the [ticker.new\(\)](#) function first, to create a ticker without extended session information.
- We use that ticker instead of [syminfo.tickerid](#) in our [ticker.heikinashi\(\)](#) call.
- In our [request.security\(\)](#) call, we set the `gaps` parameter’s value to `barmerge.gaps_on`. This instructs the function not to use previous values to fill slots where data is absent. This makes it possible for it to return [na](#) values outside of regular sessions.
- To be able to see this on the chart, we also need to use a special `plot.style_linebr` style, which breaks the plots on [na](#) values.

This script plots Heikin-Ashi candles under the chart:



```
//@version=5
indicator("Heikin-Ashi candles")
CANDLE_GREEN = #26A69A
CANDLE_RED = #EF5350

haTicker = ticker.heikinashi(syminfo.tickerid)
[haO, haH, haL, haC] = request.security(haTicker, timeframe.period, [open, high,
low, close])
candleColor = haC >= haO ? CANDLE_GREEN : CANDLE_RED
plotcandle(haO, haH, haL, haC, color = candleColor)
```

Note that:

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

[`ticker.renko\(\)](#)

Renko bars only plot price movements, without taking time or volume into consideration. They look like bricks stacked in adjacent columns [\[1\]](#). A new brick is only drawn after the price passes the top or bottom by a predetermined amount. The [ticker.renko\(\)](#) function creates a ticker id which can be

used with [request.security\(\)](#) to fetch Renko values, but there is no Pine Script® function to draw Renko bars on the chart:

```
//@version=5
indicator("", "", true)
renkoTicker = ticker.renko(syminfo.tickerid, "ATR", 10)
renkoLow = request.security(renkoTicker, timeframe.period, low)
plot(renkoLow)
```

[`ticker.linebreak\(\)`](#)

The *Line Break* chart type displays a series of vertical boxes that are based on price changes [1]. The [ticker.linebreak\(\)](#) function creates a ticker id which can be used with [request.security\(\)](#) to fetch “Line Break” values, but there is no Pine Script® function to draw such bars on the chart:

```
//@version=5
indicator("", "", true)
lineBreakTicker = ticker.linebreak(syminfo.tickerid, 3)
lineBreakClose = request.security(lineBreakTicker, timeframe.period, close)
plot(lineBreakClose)
```

[`ticker.kagi\(\)`](#)

Kagi charts are made of a continuous line that changes directions. The direction changes when the price changes [1] beyond a predetermined amount. The [ticker.kagi\(\)](#) function creates a ticker id which can be used with [request.security\(\)](#) to fetch “Kagi” values, but there is no Pine Script® function to draw such bars on the chart:

```
//@version=5
indicator("", "", true)
kagiBreakTicker = ticker.linebreak(syminfo.tickerid, 3)
kagiBreakClose = request.security(kagiBreakTicker, timeframe.period, close)
plot(kagiBreakClose)
```

[`ticker.pointfigure\(\)`](#)

Point and Figure (PnF) charts only plot price movements [1], without taking time into consideration. A column of X’s is plotted as the price rises, and O’s are plotted when price drops. The [ticker.pointfigure\(\)](#) function creates a ticker id which can be used with [request.security\(\)](#) to fetch “PnF” values, but there is no Pine Script® function to draw such bars on the chart. Every column of X’s or O’s is represented with four numbers. You may think of them as synthetic OHLC PnF values:

```
//@version=5
indicator("", "", true)
pnfTicker = ticker.pointfigure(syminfo.tickerid, "h1", "ATR", 14, 3)
[pnfO, pnfC] = request.security(pnfTicker, timeframe.period, [open, close],
barmerge.gaps_on)
plot(pnfO, "PnF Open", color.green, 4, plot.style_linebr)
plot(pnfC, "PnF Close", color.red, 4, plot.style_linebr)
```

Footnotes

[1] ([1](#), [2](#), [3](#), [4](#)) On TradingView, Renko, Line Break, Kagi and PnF chart types are generated from OHLC values from a lower timeframe. These chart types thus represent only an approximation

of what they would be like if they were generated from tick data.

Plots

- [Introduction](#)
- [`plot\(\)` parameters](#)
- [Plotting conditionally](#)
 - [Value control](#)
 - [Color control](#)
- [Levels](#)
- [Offsets](#)
- [Plot count limit](#)
- [Scale](#)
 - [Merging two indicators](#)

Introduction

The [plot\(\)](#) function is the most frequently used function used to display information calculated using Pine scripts. It is versatile and can plot different styles of lines, histograms, areas, columns (like volume columns), fills, circles or crosses.

The use of [plot\(\)](#) to create fills is explained in the page on [Fills](#).

This script showcases a few different uses of [plot\(\)](#) in an overlay script:



```
//@version=5
indicator("`plot()`", "", true)
plot(high, "Blue `high` line")
plot(math.avg(close, open), "Crosses in body center", close > open ?
color.lime : color.purple, 6, plot.style_cross)
plot(math.min(open, close), "Navy step line on body low point", color.navy, 3,
plot.style_stepline)
plot(low, "Gray dot on `low`", color.gray, 3, plot.style_circles)

color VIOLET = #AA00FF
color GOLD   = #CCCC00
ma = ta.alma(hl2, 40, 0.85, 6)
var almaColor = color.silver
almaColor := ma > ma[2] ? GOLD : ma < ma[2] ? VIOLET : almaColor
plot(ma, "Two-color ALMA", almaColor, 2)
```

Note that:

- The first [plot\(\)](#) call plots a 1-pixel blue line across the bar highs.
- The second plots crosses at the mid-point of bodies. The crosses are colored lime when the bar is up and purple when it is down. The argument used for `linewidth` is 6 but it is not a pixel value; just a relative size.
- The third call plots a 3-pixel wide step line following the low point of bodies.
- The fourth call plot a gray circle at the bars' [low](#).
- The last plot requires some preparation. We first define our bull/bear colors, calculate an

[Arnaud Legoux Moving Average](#), then make our color calculations. We initialize our color variable on bar zero only, using `var`. We initialize it to `color.silver`, so on the dataset's first bars, until one of our conditions causes the color to change, the line will be silver. The conditions that change the color of the line require it to be higher/lower than its value two bars ago. This makes for less noisy color transitions than if we merely looked for a higher/lower value than the previous one.

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



```
//@version=5
indicator("Volume change", format = format.volume)

color GREEN          = #008000
color GREEN_LIGHT    = color.new(GREEN, 50)
color GREEN_LIGHTER  = color.new(GREEN, 85)
color PINK           = #FF0080
color PINK_LIGHT     = color.new(PINK, 50)
color PINK_LIGHTER   = color.new(PINK, 90)

bool barUp = ta.rising(close, 1)
bool barDn = ta.falling(close, 1)
float volumeChange = ta.change(volume)

volumeColor = barUp ? GREEN_LIGHTER : barDn ? PINK_LIGHTER : color.gray
plot(volume, "Volume columns", volumeColor, style = plot.style_columns)

volumeChangeColor = barUp ? volumeChange > 0 ? GREEN : GREEN_LIGHT :
volumeChange > 0 ? PINK : PINK_LIGHT
plot(volumeChange, "Volume change columns", volumeChangeColor, 12,
plot.style_histogram)

plot(0, "Zero line", color.gray)
```

Note that:

- We are plotting normal `volume` values as wide columns above the zero line (see the `style = plot.style_columns` in our `plot()` call).
- Before plotting the columns we calculate our `volumeColor` by using the values of the `barUp` and `barDn` boolean variables. They become respectively `true` when the current bar's `close` is higher/lower than the previous one. Note that the "Volume" built-in does not use the same condition; it identifies an up bar with `close > open`. We use the `GREEN_LIGHTER` and `PINK_LIGHTER` colors for the volume columns.
- Because the first plot plots columns, we do not use the `linewidth` parameter, as it has no effect on columns.
- Our script's second plot is the **change** in volume, which we have calculated earlier using `ta.change(volume)`. This value is plotted as a histogram, for which the `linewidth` parameter controls the width of the column. We make this width 12 so that histogram elements are thinner than the columns of the first plot. Positive/negative `volumeChange` values plot above/below the zero line; no manipulation is required to achieve this effect.
- Before plotting the histogram of `volumeChange` values, we calculate its color value, which can be one of four different colors. We use the bright `GREEN` or `PINK` colors when the bar is up/down AND the volume has increased since the last bar (`volumeChange > 0`). Because `volumeChange` is positive in this case, the histogram's element will be plotted above the zero line. We use the bright `GREEN_LIGHT` or `PINK_LIGHT` colors

when the bar is up/down AND the volume has NOT increased since the last bar. Because `volumeChange` is negative in this case, the histogram's element will be plotted below the zero line.

- Finally, we plot a zero line. We could just as well have used `hline(0)` there.
- We use `format = format.volume` in our `indicator()` call so that large values displayed for this script are abbreviated like those of the built-in "Volume" indicator.

`plot()` calls must always be placed in a line's first position, which entails they are always in the script's global scope. They can't be placed in user-defined functions or structures like `if`, `for`, etc. Calls to `plot()` can, however, be designed to plot conditionally in two ways, which we cover in the Conditional plots section of this page.

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

`plot()` parameters

The `plot()` function has the following signature:

```
plot(series, title, color, linewidth, style, trackprice, histbase, offset, join,
editable, show_last, display) → plot
```

The parameters of `plot()` are:

`series`

It is the only mandatory parameter. Its argument must be of "series int/float" type. Note that because the auto-casting rules in Pine Script[®] convert in the `int ? float ? bool` direction, a "bool" type variable cannot be used as is; it must be converted to an "int" or a "float" for use as an argument. For example, if `newDay` is of "bool" type, then `newDay ? 1 : 0` can be used to plot 1 when the variable is `true`, and zero when it is `false`.

`title`

Requires a "const string" argument, so it must be known at compile time. The string appears:

- In the script's scale when the "Chart settings/Scales/Indicator Name Label" field is checked.
- In the Data Window.
- In the "Settings/Style" tab.
- In the dropdown of `input.source()` fields.
- In the "Condition" field of the "Create Alert" dialog box, when the script is selected.
- As the column header when exporting chart data to a CSV file.

`color`

Accepts "series color", so can be calculated on the fly, bar by bar. Plotting with `na` as the color, or any color with a transparency of 100, is one way to hide plots when they are not needed.

`linewidth`

Is the plotted element's size, but it does not apply to all styles. When a line is plotted, the unit is pixels. It has no impact when `plot.style_columns` is used.

`style`

The available arguments are:

- `plot.style_line` (the default): It plots a continuous line using the `linewidth` argument in pixels for its width. `na` values will not plot as a line, but they will be bridged when a

value that is not `na` comes in. Non-`na` values are only bridged if they are visible on the chart.

- [plot.style_linebr](#): Allows the plotting of discontinuous lines by not plotting on `na` values, and not joining gaps, i.e., bridging over `na` values.
- [plot.style_stepline](#): Plots using a staircase effect. Transitions between changes in values are done using a vertical line drawn in middle of bars, as opposed to a point-to-point diagonal joining the midpoints of bars. Can also be used to achieve an effect similar to that of [plot.style_linebr](#), but only if care is taken to plot no color on `na` values.
- [plot.style_area](#): plots a line of `linewidth` width, filling the area between the line and the `histbase`. The `color` argument is used for both the line and the fill. You can make the line a different color by using another [plot\(\)](#) call. Positive values are plotted above the `histbase`, negative values below it.
- [plot.style_areabr](#): This is similar to [plot.style_area](#) but it doesn't bridge over `na` values. Another difference is how the indicator's scale is calculated. Only the plotted values serve in the calculation of the `y` range of the script's visual space. If only high values situated far away from the `histbase` are plotted, for example, those values will be used to calculate the `y` scale of the script's visual space. Positive values are plotted above the `histbase`, negative values below it.
- [plot.style_columns](#): Plots columns similar to those of the "Volume" built-in indicator. The `linewidth` value does **not** affect the width of the columns. Positive values are plotted above the `histbase`, negative values below it. Always includes the value of `histbase` in the `y` scale of the script's visual space.
- [plot.style_histogram](#): Plots columns similar to those of the "Volume" built-in indicator, except that the `linewidth` value is used to determine the width of the histogram's bars in pixels. Note that since `linewidth` requires an "input int" value, the width of the histogram's bars cannot vary bar to bar. Positive values are plotted above the `histbase`, negative values below it. Always includes the value of `histbase` in the `y` scale of the script's visual space.
- [plot.style_circles](#) and [plot.style_cross](#): These plot a shape that is not joined across bars unless `join = true` is also used. For these styles, the `linewidth` argument becomes a relative sizing measure — its units are not pixels.

`trackprice`

The default value of this is `false`. When it is `true`, a dotted line made up of small squares will be plotted the full width of the script's visual space. It is often used in conjunction with `show_last = 1, offset = -99999` to hide the actual plot and only leave the residual dotted line.

`histbase`

It is the reference point used with [plot.style_area](#), [plot.style_columns](#) and [plot.style_histogram](#). It determines the level separating positive and negative values of the series argument. It cannot be calculated dynamically, as an "input int/float" is required.

`offset`

This allows shifting the plot in the past/future using a negative/positive offset in bars. The value cannot change during the script's execution.

`join`

This only affect styles [plot.style_circles](#) or [plot.style_cross](#). When `true`, the shapes are joined by a one-pixel line.

`editable`

This boolean parameter controls whether or not the plot's properties can be edited in the "Settings/Style" tab. Its default value is `true`.

show_last

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

display

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

```
//@version=5
indicator("")
r = ta.rsi(close, 14)
xUp = ta.crossover(r, 50)
plot(r, "RSI", display = display.none)
alertcondition(xUp, "xUp alert", message = 'RSI is bullish at:
{{plot("RSI")}}')
```

Plotting conditionally

[plot\(\)](#) calls cannot be used in conditional structures such as [if](#), but they can be controlled by varying their plotted values, or their color. When no plot is required, you can either plot [na](#) values, or plot values using [na](#) color or any color with 100 transparency (which also makes it invisible).

Value control

One way to control the display of plots is to plot [na](#) values when no plot is needed. Sometimes, values returned by functions such as [request.security\(\)](#) will return [na](#) values, when `barmerge.gaps_on` is used, for example. In both these cases it is sometimes useful to plot discontinuous lines. This script shows a few ways to do it:



```
//@version=5
indicator("Discontinuous plots", "", true)
bool plotValues = bar_index % 3 == 0
plot(plotValues ? high : na, color = color.fuchsia, linewidth = 6, style =
plot.style_linebr)
plot(plotValues ? high : na)
plot(plotValues ? math.max(open, close) : na, color = color.navy, linewidth = 6,
style = plot.style_cross)
plot(plotValues ? math.min(open, close) : na, color = color.navy, linewidth = 6,
style = plot.style_circles)
plot(plotValues ? low : na, color = plotValues ? color.green : na, linewidth =
6, style = plot.style_stepline)
```

Note that:

- We define the condition determining when we plot using `bar_index % 3 == 0`, which becomes `true` when the remainder of the division of the bar index by 3 is zero. This will happen every three bars.
- In the first plot, we use [plot.style_linebr](#), which plots the fuchsia line on highs. It is centered on the bar’s horizontal midpoint.
- The second plot shows the result of plotting the same values, but without using special care

to break the line. What's happening here is that the thin blue line of the plain `plot()` call is automatically bridged over `na` values (or *gaps*), so the plot does not interrupt.

- We then plot navy blue crosses and circles on the body tops and bottoms. The `plot.style_circles` and `plot.style_cross` style are a simple way to plot discontinuous values, e.g., for stop or take profit levels, or support & resistance levels.
- The last plot in green on the bar lows is done using `plot.style_stepline`. Note how its segments are wider than the fuchsia line segments plotted with `plot.style_linebr`. Also note how on the last bar, it only plots halfway until the next bar comes in.
- The plotting order of each plot is controlled by their order of appearance in the script. See

This script shows how you can restrict plotting to bars after a user-defined date. We use the `input.time()` function to create an input widget allowing script users to select a date and time, using Jan 1st 2021 as its default value:

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

Color control

The [Conditional coloring](#) section of the page on colors discusses color control for plots. We'll look here at a few examples.

The value of the `color` parameter in `plot()` can be a constant, such as one of the built-in [constant colors](#) or a [color literal](#). In Pine Script[®], the form-type of such colors is called “**const color**” (see the [Type system](#) page). They are known at compile time:

```
//@version=5
indicator("", "", true)
plot(close, color = color.gray)
```

The color of a plot can also be determined using information that is only known when the script begins execution on the first historical bar of a chart (bar zero, i.e., `bar_index == 0` or `barstate.isfirst == true`), as will be the case when the information needed to determine a color depends on the chart the script is running on. Here, we calculate a plot color using the `syminfo.type` built-in variable, which returns the type of the chart's symbol. The form-type of `plotColor` in this case will be “**simple color**”:

```
//@version=5
indicator("", "", true)
plotColor = switch syminfo.type
    "stock"      => color.purple
    "futures"    => color.red
    "index"      => color.gray
    "forex"      => color.fuchsia
    "crypto"     => color.lime
    "fund"       => color.orange
    "dr"         => color.aqua
    "cfd"        => color.blue
plot(close, color = plotColor)
printTable(txt) => var table t = table.new(position.middle_right, 1, 1),
table.cell(t, 0, 0, txt, bgcolor = color.yellow)
printTable(syminfo.type)
```

Plot colors can also be chosen through a script's inputs. In this case, the `lineColorInput` variable is of form-type “**input color**”:

```
//@version=5
indicator("", "", true)
color lineColorInput = input(#1848CC, "Line color")
plot(close, color = lineColorInput)
```

Finally, plot colors can also be a *dynamic* value, i.e., a calculated value that is only known on each bar. These are of form-type “**series color**”:

```
//@version=5
indicator("", "", true)
plotColor = close >= open ? color.lime : color.red
plot(close, color = plotColor)
```

When plotting pivot levels, one common requirement is to avoid plotting level transitions. Using [lines](#) is one alternative, but you can also use [plot\(\)](#) like this:



```
//@version=5
indicator("Pivot plots", "", true)
pivotHigh = fixnan(ta.pivohigh(3,3))
plot(pivotHigh, "High pivot", ta.change(pivotHigh) ? na : color.olive, 3)
plotchar(ta.change(pivotHigh), "ta.change(pivotHigh)", "•", location.top, size = size.small)
```

Note that:

- We use `pivotHigh = fixnan(ta.pivohigh(3,3))` to hold our pivot values. Because [ta.pivohigh\(\)](#) only returns a value when a new pivot is found, we use [fixnan\(\)](#) to fill the gaps with the last pivot value returned. The gaps here refer to the [na](#) values [ta.pivohigh\(\)](#) returns when no new pivot is found.
- Our pivots are detected three bars after they occur because we use the argument 3 for both the `leftbars` and `rightbars` parameters in our [ta.pivohigh\(\)](#) call.
- The last plot is plotting a continuous value, but it is setting the plot’s color to [na](#) when the pivot’s value changes, so the plot isn’t visible then. Because of this, a visible plot will only appear on the bar following the one where we plotted using [na](#) color.
- The blue dot indicates when a new high pivot is detected and no plot is drawn between the preceding bar and that one. Note how the pivot on the bar indicated by the arrow has just been detected in the realtime bar, three bars later, and how no plot is drawn. The plot will only appear on the next bar, making the plot visible **four bars** after the actual pivot.

Levels

Pine Script[®] has an [hline\(\)](#) function to plot horizontal lines (see the page on [Levels](#)). [hline\(\)](#) is useful because it has some line styles unavailable with [plot\(\)](#), but it also has some limitations, namely that it does not accept “series color”, and that its `price` parameter requires an “input int/float”, so cannot vary during the script’s execution.

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



```
//@version=5
indicator("CCI levels with `plot()`)")
plot(ta.cci(close, 20))
```

```

plot(0, "Zero", color.gray, 1, plot.style_circles)
plot(bar_index % 2 == 0 ? 100 : na, "100", color.lime, 1, plot.style_linebr)
plot(bar_index % 2 == 0 ? -100 : na, "-100", color.fuchsia, 1,
plot.style_linebr)
plot(200, "200", color.green, 2, trackprice = true, show_last = 1, offset =
-99999)
plot(-200, "-200", color.red, 2, trackprice = true, show_last = 1, offset =
-99999)
plot(300, "300", color.new(color.green, 50), 1)
plot(-300, "-300", color.new(color.red, 50), 1)

```

Note that:

- The zero level is plotted using [plot.style_circles](#).
- The 100 levels are plotted using a conditional value that only plots every second bar. In order to prevent the [na](#) values from being bridged, we use the [plot.style_linebr](#) line style.
- The 200 levels are plotted using `trackprice = true` to plot a distinct pattern of small squares that extends the full width of the script's visual space. The `show_last = 1` in there displays only the last plotted value, which would appear as a one-bar straight line if the next trick wasn't also used: the `offset = -99999` pushes that one-bar segment far away in the past so that it is never visible.
- The 300 levels are plotted using a continuous line, but a lighter transparency is used to make them less prominent.

Offsets

The `offset` parameter specifies the shift used when the line is plotted (negative values shift in the past, positive values shift into the future. For example:

```

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

```



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

Plot count limit

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

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

```

plot(close, color = color.green)

```

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

```

plot(close, color = syminfo.mintick > 0.0001 ? color.green : color.red) //?
"simple color"
plot(close, color = input.color(color.purple)) //? "input color"

```



```
plot(close, color = close > open ? color.green : color.red) //? "series color"
plot(close, color = color.new(color.silver, close > open ? 40 : 0)) //? "series
color"
```

Scale

Not all values can be plotted everywhere. Your script's visual space is always bound by upper and lower limits that are dynamically adjusted with the values plotted. An [RSI](#) indicator will plot values between 0 and 100, which is why it is usually displayed in a distinct *pane* — or area — above or below the chart. If RSI values were plotted as an overlay on the chart, the effect would be to distort the symbol's normal price scale, unless it just happened to be close to RSI's 0 to 100 range. This shows an RSI signal line and a centerline at the 50 level, with the script running in a separate pane:



```
//@version=5
indicator("RSI")
myRSI = ta.rsi(close, 20)
bullColor = color.from_gradient(myRSI, 50, 80, color.new(color.lime, 70),
color.new(color.lime, 0))
bearColor = color.from_gradient(myRSI, 20, 50, color.new(color.red, 0),
color.new(color.red, 70))
myRSIColor = myRSI > 50 ? bullColor : bearColor
plot(myRSI, "RSI", myRSIColor, 3)
hline(50)
```

Note that the *y* axis of our script's visual space is automatically sized using the range of values plotted, i.e., the values of RSI. See the page on [Colors](#) for more information on the [color.from_gradient\(\)](#) function used in the script.

If we try to plot the symbol's [close](#) values in the same space by adding the following line to our script:

```
plot(close)
```

This is what happens:



The chart is on the BTCUSD symbol, whose [close](#) prices are around 40000 during this period. Plotting values in the 40000 range makes our RSI plots in the 0 to 100 range indiscernible. The same distorted plots would occur if we placed the [RSI](#) indicator on the chart as an overlay.

Merging two indicators

If you are planning to merge two signals in one script, first consider the scale of each. It is impossible, for example, to correctly plot an [RSI](#) and a [MACD](#) in the same script's visual space because RSI has a fixed range (0 to 100) while MACD doesn't, as it plots moving averages calculated on price.

If both your indicators used fixed ranges, you can shift the values of one of them so they do not overlap. We could, for example, plot both [RSI](#) (0 to 100) and the [True Strength Indicator \(TSI\)](#) (-100 to +100) by displacing one of them. Our strategy here will be to compress and shift the [TSI](#) values so they plot over [RSI](#):



```
//@version=5
indicator("RSI and TSI")
myRSI = ta.rsi(close, 20)
bullColor = color.from_gradient(myRSI, 50, 80, color.new(color.lime, 70),
color.new(color.lime, 0))
bearColor = color.from_gradient(myRSI, 20, 50, color.new(color.red, 0),
color.new(color.red, 70))
myRSIColor = myRSI > 50 ? bullColor : bearColor
plot(myRSI, "RSI", myRSIColor, 3)
hline(100)
hline(50)
hline(0)

// 1. Compress TSI's range from -100/100 to -50/50.
// 2. Shift it higher by 150, so its -50 min value becomes 100.
myTSI = 150 + (100 * ta.tsi(close, 13, 25) / 2)
plot(myTSI, "TSI", color.blue, 2)
plot(ta.ema(myTSI, 13), "TSI EMA", #FF006E)
hline(200)
hline(150)
```

Note that:

- We have added levels using [hline](#) to situate both signals.
- In order for both signal lines to oscillate on the same range of 100, we divide the [TSI](#) value by 2 because it has a 200 range (-100 to +100). We then shift this value up by 150 so it oscillates between 100 and 200, making 150 its centerline.
- The manipulations we make here are typical of the compromises required to bring two indicators with different scales in the same visual space, even when their values, contrary to [MACD](#), are bounded in a fixed range.

Repainting

- [Introduction](#)
 - [For script users](#)
 - [For Pine Script[®] programmers](#)
- [Historical vs realtime calculations](#)
 - [Fluid data values](#)
 - [Repainting `request.security\(\)` calls](#)
 - [Using `request.security\(\)` at lower timeframes](#)
 - [Future leak with `request.security\(\)`](#)
 - [`varip`](#)
 - [Bar state built-ins](#)
 - [`timenow`](#)
 - [Strategies](#)
- [Plotting in the past](#)
- [Dataset variations](#)
 - [Starting points](#)

- [Revision of historical data](#)

Introduction

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

Repainting behavior is widespread and can be caused by many factors. Following our definition, our estimate is that more than 95% of indicators in existence repaint. Widely used indicators like MACD and RSI, for example, exhibit one form of repainting because they show one value on historical bars, yet when running in realtime they will produce results that constantly fluctuate until the realtime bar closes. They thus behave differently on historical and realtime bars. This does not necessarily make them less useful in all contexts, nor prevent knowledgeable traders from using them. Who would think of discrediting a volume profile indicator, for example because it updates in real time, and so repaints?

The different types of repainting we discuss in this page can be divided this way:

- **Widespread and often acceptable:** recalculation during the realtime bar (most classic indicators like MACD, RSI, and the vast majority of indicators in the [Community Scripts](#), scripts using repainting [request.security\(\)](#) calls, etc.). There is often nothing wrong in using such scripts, provided you understand how they work. If you elect to use these scripts to issue alerts or trade orders, however, then you should know if they are being generated using the realtime or confirmed values, and decide for yourself if the script's behavior meets your requirements.
- **Misleading:** plotting in the past, calculating results in realtime that cannot be replicated on historical bars, relocating past events (Ichimoku, most pivot scripts, most strategies using `calc_on_vey_tick = true`, scripts using repainting [request.security\(\)](#) calls when their values are plotted on historical bars, as their behavior will not be the same in realtime, most scripts using [varip](#), most scripts using [timenow](#), some scripts using `barstate.*` variables).
- **Unacceptable:** scripts using future information, strategies running on non-standard charts, scripts using realtime intrabar timeframes to generate alerts or orders.
- **Unavoidable:** revision of historical feeds by data suppliers, varying starting bar on historical bars.

The first two types of repainting can be perfectly acceptable if:

1. You are aware of the behavior.
2. You can live with it, or
3. You can circumvent it.

It should now be clear to you that not **all** repainting behavior is inherently wrong and should be avoided at all cost. In many situations, repainting indicators are exactly what's needed. What's important is to know when repainting behavior is **not** acceptable to you. To avoid such situations, you must understand exactly how the tools you use work, or how you should design the ones you build. If you publish scripts, any potentially misleading repainting behavior should be mentioned along with the other limitations of your script.

Note

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

For script users

You can very well decide to use repainting indicators if you understand how they behave, and that behavior meets your trading methodology's requirements. Don't be one of those newcomers to trading who slap "repaint" sentences on published scripts as if it discredits them. Doing so only reveals your incomprehension of the subject.

The question "Does it repaint?" means nothing. Consequently, it cannot be answered in a meaningful way. Why? Because it needs to be qualified. Instead, one could ask:

- Do you wait for the realtime bar to close before displaying your entry/exit markers?
- Do alerts wait for the end of the realtime bar before triggering?
- Do the higher timeframe plots repaint (which means they won't plot the same way on realtime bars as they do on historical bars)?
- Does your script plot in the past (as most pivot or zigzag scripts will do)?
- Does your strategy use `calc_on_every_tick = true`?
- Will your indicator display in realtime the same way it does on historical bars?
- Are you fetching future information with your `request.security()` calls?

What's important is that you understand how the tools you use work, and if their behavior is compatible with your objectives, repainting or not. As you will learn if you read this page, repainting is a complex matter. It has many faces and many causes. Even if you don't program in Pine Script[®], this page will help you understand the array of causes that can lead to repainting, and hopefully enable more meaningful discussions with script authors.

For Pine Script[®] programmers

As we discussed in the previous section, not all types of repainting behavior need to be avoided at all costs, and as we will see in the following text, some can't. We hope this page helps you better understand the dynamics at play, so that you can make better design decisions concerning your trading tools. This page's content should help you avoid making the most common coding mistakes that lead to repainting or misleading plots.

Whatever your design decisions are, if you publish your script, you should explain them to traders so they can understand how your script behaves.

We will survey three broad categories of repainting causes:

- Historical vs realtime calculations
- Plotting in the past
- Dataset variations

Historical vs realtime calculations

Fluid data values

Historical data does not include records of intermediary price movements on bars; only [open](#), [high](#), [low](#) and [close](#) values (OHLC).

On realtime bars (bars running when the instrument's market is open), however, the [high](#), [low](#) and [close](#) values are not fixed; they can change values many times before the realtime bar closes and its HLC values are fixed. They are *fluid*. This leads to a script sometimes working differently on historical data and in real time, where only the [open](#) price will not change during the bar.

Any script using values like [high](#), [low](#) and [close](#) in realtime is subject to producing calculations that

may not be repeatable on historical bars — thus repaint.

Let's look at this simple script. It detects crosses of the [close](#) value (in the realtime bar, this corresponds to the current price of the instrument) over and under an [EMA](#):

```
//@version=5
indicator("Repainting", "", true)
ma = ta.ema(close, 5)
xUp = ta.crossover(close, ma)
xDn = ta.crossunder(close, ma)
plot(ma, "MA", color.black, 2)
bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) :
na)
```



Note that:

- The script uses [bgcolor\(\)](#) to color the background green when [close](#) crosses over the EMA, and red on crosses under the EMA.
- The screen snapshot shows the script in realtime on a 30sec chart. A cross over the EMA has been detected, thus the background of the realtime bar is green.
- The problem here is that nothing guarantees this condition will hold true until the end of the realtime bar. The arrow points to the timer showing that 21 seconds remain in the realtime bar, and anything could happen until then.
- We are witnessing a repainting script.

To prevent this repainting, we must rewrite our script so that it does not use values that fluctuate during the realtime bar. This will require using values from a bar that has elapsed (typically the preceding bar), or the [open](#) price, which does not vary in realtime.

We can achieve this in many ways. This method adds a `and barstate.isconfirmed` condition to our cross detections, which requires the script to be executing on the bar's last iteration, when it closes and prices are confirmed. It is a simple way to avoid repainting:

```
//@version=5
indicator("Repainting", "", true)
ma = ta.ema(close, 5)
xUp = ta.crossover(close, ma) and barstate.isconfirmed
xDn = ta.crossunder(close, ma) and barstate.isconfirmed
plot(ma, "MA", color.black, 2)
bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) :
na)
```

This uses the crosses detected on the previous bar:

```
//@version=5
indicator("Repainting", "", true)
ma = ta.ema(close, 5)
xUp = ta.crossover(close, ma)[1]
xDn = ta.crossunder(close, ma)[1]
plot(ma, "MA", color.black, 2)
bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) :
na)
```

This uses only confirmed [close](#) and EMA values for its calculations:

```
//@version=5
indicator("Repainting", "", true)
ma = ta.ema(close[1], 5)
xUp = ta.crossover(close[1], ma)
```

```
xDn = ta.crossunder(close[1], ma)
plot(ma, "MA", color.black, 2)
bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) :
na)
```

This detects crosses between the realtime bar's [open](#) and the value of the EMA from the previous bars. Notice that the EMA is calculated using [close](#), so it repaints. We must ensure we use a confirmed value to detect crosses, thus `ma[1]` in the cross detection logic:

```
//@version=5
indicator("Repainting", "", true)
ma = ta.ema(close, 5)
xUp = ta.crossover(open, ma[1])
xDn = ta.crossunder(open, ma[1])
plot(ma, "MA", color.black, 2)
bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) :
na)
```

Note that all these methods have one thing in common: while they prevent repainting, they will also trigger signals later than repainting scripts. This is an inevitable compromise if one wants to avoid repainting. You just can't have your cake and eat it too.

[Repainting `request.security\(\)` calls](#)

The data fetched with [request.security\(\)](#) will differ on historical and realtime bars if the function is not used in the correct manner. Repainting [request.security\(\)](#) calls will produce historical data and plots that cannot be replicated in realtime. Let's look at a script showing the difference between repainting and non-repainting [request.security\(\)](#) calls:

```
//@version=5
indicator("Repainting vs non-repainting `request.security()`", "", true)
var BLACK_MEDIUM = color.new(color.black, 50)
var ORANGE_LIGHT = color.new(color.orange, 80)

tfInput = input.timeframe("1")

repaintingClose = request.security(syminfo.tickerid, tfInput, close)
plot(repaintingClose, "Repainting close", BLACK_MEDIUM, 8)

indexHighTF = barstate.isrealtime ? 1 : 0
indexCurrTF = barstate.isrealtime ? 0 : 1
nonRepaintingClose = request.security(syminfo.tickerid, tfInput,
close[indexHighTF])[indexCurrTF]
plot(nonRepaintingClose, "Non-repainting close", color.fuchsia, 3)

if ta.change(time(tfInput))
    label.new(bar_index, na, "U", yloc = yloc.abovebar, style =
label.style_none, textcolor = color.black, size = size.large)
bgcolor(barstate.isrealtime ? ORANGE_LIGHT : na)
```

This is what its output looks like on a 5sec chart that has been running the script for a few minutes:



Note that:

- The orange background identifies the realtime bar, and elapsed realtime bars.
- A black curved arrow indicates when a new higher timeframe comes in.
- The thick gray line shows the repainting [request.security\(\)](#) call used to initialize

- repaintingClose.
- The fuchsia line shows the non-repainting `request.security()` call used to initialize `nonRepaintingClose`.
 - The behavior of the repainting line is completely different on historical bars and in realtime. On historical bars, it shows the new value of a completed timeframe on the `close` of the bar where it completes. It then stays stable until another timeframe completes. The problem is that in realtime, it follows the **current** `close` price, so it moves all the time and changes on each bar.
 - The behavior of the non-repainting fuchsia line, in contrast, behaves exactly the same way on historical bars and in realtime. It updates on the bar following the completion of the higher timeframe, and doesn't move until the bar after another higher timeframe completes. It is more reliable and does not mislead script users. Note that while new higher timeframe data comes in at the `close` of historical bars, it will be available on the `open` of the same bar in realtime.

This script shows a `nonRepaintingSecurity()` function that can be used to do the same as our non-repainting code in the previous example:

```
//@version=5
indicator("Non-repainting `nonRepaintingSecurity()`", "", true)

tfInput = input.timeframe("1")

nonRepaintingSecurity(sym, tf, src) =>
    request.security(sym, tf, src[barstate.isrealtime ? 1 : 0])
    [barstate.isrealtime ? 0 : 1]

nonRepaintingClose = nonRepaintingSecurity(sym.info.tickerid, "1", close)
plot(nonRepaintingClose, "Non-repainting close", color.fuchsia, 3)
```

Another way to produce non-repainting higher timeframe data is this, which uses an offset of [1] on the series, and `lookahead`:

```
nonRepaintingSecurityAlternate(sym, tf, src) =>
    request.security(sym, tf, src[1], lookahead = barmerge.lookahead_on)
```

It will produce the same non-repainting behavior as `nonRepaintingSecurity()`. Note that the [1] offset to the series and the use of `lookahead = barmerge.lookahead_on` are interdependent. One **cannot** be removed without compromising the functionality of the function. Also note that occasional one-bar variations between when the `nonRepaintingSecurity()` and `nonRepaintingSecurityAlternate()` values come in on historical bars are to be expected.

Using `request.security()` at lower timeframes

Some scripts use `request.security()` to request data from a timeframe **lower** than the chart's timeframe. This can be useful when functions specifically designed to handle intrabars at lower timeframes are sent down the timeframe. When this type of user-defined function requires the detection of the intrabars' first bar, as most do, the technique will only work on historical bars. This is due to the fact that realtime intrabars are not yet sorted. The impact of this is that such scripts cannot reproduce in real time their behavior on historical bars. Any logic generating alerts, for example, will be flawed, and constant refreshing will be required to recalculate elapsed realtime bars as historical bars.

When used at lower timeframes than the chart's without specialized functions able to distinguish between intrabars, `request.security()` will only return the value of the **last** intrabar in the dilation of

the chart's bar, which is usually not useful, and will also not reproduce in real time, so lead to repainting.

For all these reasons, unless you understand the subtleties of using [request.security\(\)](#) at lower timeframes than the chart's, it is best to avoid using the function at those timeframes. Higher-quality scripts will have logic to detect such anomalies and prevent the display of results which would be invalid when a lower timeframe is used.

Future leak with `request.security()`

When [request.security\(\)](#) is used with `lookahead = barmerge.lookahead_on` to fetch prices without offsetting the series by `[1]`, it will return data from the future on historical bars, which is dangerously misleading.

While historical bars will magically display future prices before they should be known, no lookahead is possible in realtime because the future there is unknown, as it should, so no future bars exist.

This is an example:

```
// FUTURE LEAK! DO NOT USE!  
//@version=5  
indicator("Future leak", "", true)  
futureHigh = request.security(syminfo.tickerid, "D", high, lookahead =  
barmerge.lookahead_on)  
plot(futureHigh)
```



Note how the higher timeframe line is showing the timeframe's [high](#) value before it occurs. The solution is to use the function like we do in our `nonRepaintingSecurity()` shown earlier.

Public scripts using this misleading technique will be moderated.

`varip`

Scripts using the [varip](#) declaration mode for variables (see our section on [varip](#) for more information) save information across realtime updates, which cannot be reproduced on historical bars where only OHLC information is available. Such scripts may be useful in realtime, including to generate alerts, but their logic cannot be backtested, nor can their plots on historical bars reflect calculations that will be done in realtime.

Bar state built-ins

Scripts using [bar states](#) may or may not repaint. As we have seen in the previous section, using [barstate.isconfirmed](#) is actually one way to **avoid** repainting that **will** reproduce on historical bars, which are always “confirmed”. Uses of other bar states such as [barstate.isnew](#), however, will lead to repainting. The reason is that on historical bars, [barstate.isnew](#) is `true` on the bar's [close](#), yet in realtime, it is `true` on the bar's [open](#). Using the other bar state variables will usually cause some type of behavioral discrepancy between historical and realtime bars.

`timenow`

The [timenow](#) built-in returns the current time. Scripts using this variable cannot show consistent historical and realtime behavior, so they necessarily repaint.

Strategies

Strategies using `calc_on_every_tick = true` execute on each realtime update, while strategies run on the [close](#) of historical bars. They will most probably not generate the same order executions, and so repaint. Note that when this happens, it also invalidates backtesting results, as they are not representative of the strategy's behavior in realtime.

Plotting in the past

Scripts detecting pivots after 5 bars have elapsed will often go back in the past to plot pivot levels or values on the actual pivot, 5 bars in the past. This will often cause unsuspecting traders looking at plots on historical bars to infer that when the pivot happens in realtime, the same plots will appear on the pivot when it occurs, as opposed to when it is detected.

Let's look at a script showing the price of high pivots by placing the price in the past, 5 bars after the pivot was detected:

```
//@version=5
indicator("Plotting in the past", "", true)
pHi = ta.pivohigh(5, 5)
if not na(pHi)
    label.new(bar_index[5], na, str.tostring(pHi, format.mintick) + "\n?", yloc
= yloc.abovebar, style = label.style_none, textcolor = color.black, size =
size.normal)
```



Note that:

- This script repaints because an elapsed realtime bar showing no price may get a price placed on it if it is identified as a pivot, 5 bars after the actual pivot occurs.
- The display looks great, but it can be misleading.

The best solution to this problem when developing script for others is to plot **without** an offset by default, but give the option for script users to turn on plotting in the past through inputs, so they are necessarily aware of what the script is doing, e.g.:

```
//@version=5
indicator("Plotting in the past", "", true)
plotInThePast = input(false, "Plot in the past")
pHi = ta.pivohigh(5, 5)
if not na(pHi)
    label.new(bar_index[plotInThePast ? 5 : 0], na, str.tostring(pHi,
format.mintick) + "\n?", yloc = yloc.abovebar, style = label.style_none,
textcolor = color.black, size = size.normal)
```

Dataset variations

Starting points

Scripts begin executing on the chart's first historical bar, and then execute on each bar sequentially, as is explained in this manual's page on Pine Script[®]'s [execution model](#). If the first bar changes, then the script will often not calculate the same way it did when the dataset began at a different point in time.

The following factors have an impact on the quantity of bars you see on your charts, and their

starting point:

- The type of account you hold
- The historical data available from the data supplier
- The alignment requirements of the dataset, which determine its *starting point*

These are the account-specific bar limits:

- 20000 historical bars for the Premium plan.
- 10000 historical bars for Pro and Pro+ plans.
- 5000 historical bars for other plans.

Starting points are determined using the following rules, which depend on the chart's timeframe:

- **1, 5, 10, 15, 30 seconds**: aligns to the beginning of a day.
- **1 - 14 minutes**: aligns to the beginning of a week.
- **15 - 29 minutes**: aligns to the beginning of a month.
- **30 - 1439 minutes**: aligns to the beginning of a year.
- **1440 minutes and higher**: aligns to the first available historical data point.

As time goes by, these factors cause your chart's history to start at different points in time. This often has an impact on your scripts calculations, because changes in calculation results in early bars can ripple through all the other bars in the dataset. Using functions like [ta.valuewhen\(\)](#), [ta.barssince\(\)](#) or [ta.ema\(\)](#), for example, will yield results that vary with early history.

Revision of historical data

Historical and realtime bars are built using two different data feeds supplied by exchanges/brokers: historical data, and realtime data. When realtime bars elapse, exchanges/brokers sometimes make what are usually small adjustments to bar prices, which are then written to their historical data. When the chart is refreshed or the script is re-executed on those elapsed realtime bars, they will then be built and calculated using the historical data, which will contain those usually small price revisions, if any have been made.

Historical data may also be revised for other reasons, e.g., for stock splits.

Sessions

- [Introduction](#)
- [Session strings](#)
 - [Session string specifications](#)
 - [Using session strings](#)
- [Session states](#)
- [Using sessions with `request.security\(\)`](#)

Introduction

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

1. **Session strings** containing from-to start times and day information that can be used in functions such as [time\(\)](#) and [time_close\(\)](#) to detect when bars are in a particular time period, with the option of limiting valid sessions to specific days. The [input.session\(\)](#) function provides a way to allow script users to define session values through a script's "Inputs" tab

- (see the [Session input](#) section for more information).
2. **Session states** built-in variables such as [session.ismarket](#) can identify which session a bar belongs to.
 3. When fetching data with [request.security\(\)](#) you can also choose to return data from *regular* sessions only or *extended* sessions. In this case, the definition of **regular and extended sessions** is that of the exchange. It is part of the instrument's properties — not user-defined, as in point #1. This notion of *regular* and *extended* sessions is the same one used in the chart's interface, in the “Chart Settings/Symbol/Session” field, for example.

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

Note that:

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

Session strings

Session string specifications

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

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

Where:

- `<time_period>` uses times in “hhmm” format, with “hh” in 24-hour format, so 1700 for 5PM. The time periods are in the “hhmm-hhmm” format, and a comma can separate multiple time periods to specify combinations of discrete periods.

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

Note

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

These are examples of session strings:

```
"24x7"
```

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

```
"0000-0000:1234567"
```

Equivalent to the previous example.

```
"0000-0000"
```

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

```
"0000-0000:23456"
```

The same as the previous example, but only Monday to Friday.

```
"2000-1630:1234567"
```

An overnight session that begins at 20:00 and ends at 16:30 the next day. It is valid on all days of the week.

```
"0930-1700:146"
```

A session that begins at 9:30 and ends at 17:00 on Sundays (1), Wednesdays (4), and Fridays

(6).

```
"1700-1700:23456"
```

An *overnight session*. The Monday session starts Sunday at 17:00 and ends Monday at 17:00. It is valid Monday through Friday.

```
"1000-1001:26"
```

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

```
"0900-1600,1700-2000"
```

A session that begins at 9:00, breaks from 16:00 to 17:00, and continues until 20:00. Applies to every day of the week.

Using session strings

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

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

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



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

Note that:

- We use a session input to allow users to specify the time they want to detect. We are only looking for the session's beginning time on bars, so we use a five-minute gap between the beginning and end time of our "0930-0935" default value.
- We create a `sessionBegins()` function to detect the beginning of a session. Its `time("", sess)` call uses an empty string for the function's `timeframe` parameter, which means it uses the chart's timeframe, whatever that is. The function returns `true` when:
 - The chart uses an intraday timeframe (seconds or minutes).

- The script isn't on the chart's first bar, which we ensure with `(not barstate.isfirst)`. This check prevents the code from always detecting a session beginning on the first bar because `na(t[1])` and `not na(t)` is always `true` there.
- The `time()` call has returned `na` on the previous bar because it wasn't in the session's time period, and it has returned a value that is not `na` on the current bar, which means the bar is **in** the session's time period.

Session states

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

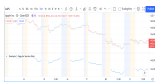
- `session.ismarket` returns `true` when the bar belongs to regular trading hours.
- `session.ispremarket` returns `true` when the bar belongs to the extended session preceding regular trading hours.
- `session.ispostmarket` returns `true` when the bar belongs to the extended session following regular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

Using sessions with `request.security()`

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

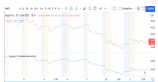
- **regular** (which does not include pre- and post-market data), and
- **extended** (which includes pre- and post-market data).

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



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

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



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

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

The [ticker.new\(\)](#) function has the following signature:

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

Where:

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

Our first example could be rewritten as:

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

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

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

Strategies

- [Introduction](#)
- [A simple strategy example](#)
- [Applying a strategy to a chart](#)
- [Strategy tester](#)
 - [Overview](#)
 - [Performance summary](#)
 - [List of trades](#)
 - [Properties](#)
- [Broker emulator](#)
 - [Bar magnifier](#)
- [Orders and entries](#)
 - [Order types](#)
 - [Market orders](#)
 - [Limit orders](#)
 - [Stop and stop-limit orders](#)
 - [Order placement commands](#)
 - [`strategy.entry\(\)`](#)

- [`strategy.order\(\)`](#)
- [`strategy.exit\(\)`](#)
- [`strategy.close\(\)`](#) and [`strategy.close_all\(\)`](#)
- [`strategy.cancel\(\)`](#) and [`strategy.cancel_all\(\)`](#)
- [Position sizing](#)
- [Closing a market position](#)
- [OCA groups](#)
 - [`strategy.oca.cancel`](#)
 - [`strategy.oca.reduce`](#)
 - [`strategy.oca.none`](#)
- [Currency](#)
- [Altering calculation behavior](#)
 - [`calc_on_every_tick`](#)
 - [`calc_on_order_fills`](#)
 - [`process_orders_on_close`](#)
- [Simulating trading costs](#)
 - [Commission](#)
 - [Slippage and unfilled limits](#)
- [Risk management](#)
- [Margin](#)
- [Strategy Alerts](#)
- [Notes on testing strategies](#)
 - [Backtesting and forward testing](#)
 - [Lookahead bias](#)
 - [Selection bias](#)
 - [Overfitting](#)

Introduction

Pine Script[®] strategies simulate the execution of trades on historical and real-time data to facilitate the backtesting and forward testing of trading systems. They include many of the same capabilities as Pine Script[®] indicators while providing the ability to place, modify, and cancel hypothetical orders and analyze the results.

When a script uses the [strategy\(\)](#) function for its declaration, it gains access to the `strategy.*` namespace, where it can call functions and variables for simulating orders and accessing essential strategy information. Additionally, the script will display information and simulated results externally in the “Strategy Tester” tab.

A simple strategy example

The following script is a simple strategy that simulates the entry of long or short positions upon the crossing of two moving averages:

```
//@version=5
strategy("test", overlay = true)

// Calculate two moving averages with different lengths.
float fastMA = ta.sma(close, 14)
float slowMA = ta.sma(close, 28)

// Enter a long position when `fastMA` crosses over `slowMA`.
```

```

if ta.crossover(fastMA, slowMA)
    strategy.entry("buy", strategy.long)

// Enter a short position when `fastMA` crosses under `slowMA`.
if ta.crossunder(fastMA, slowMA)
    strategy.entry("sell", strategy.short)

// Plot the moving averages.
plot(fastMA, "Fast MA", color.aqua)
plot(slowMA, "Slow MA", color.orange)

```

Note that:

- The `strategy("test" overlay = true)` line declares that the script is a strategy named “test” with visual outputs overlaid on the main chart pane.
- [strategy.entry\(\)](#) is the command that the script uses to simulate “buy” and “sell” orders. When the script places an order, it also plots the order id on the chart and an arrow to indicate the direction.
- Two [plot\(\)](#) functions plot the moving averages with two different colors for visual reference.

Applying a strategy to a chart

To test a strategy, apply it to the chart. You can use a built-in strategy from the “Indicators & Strategies” dialog box or write your own in the Pine Editor. Click “Add to chart” from the “Pine Editor” tab to apply a script to the chart:



After a strategy script is compiled and applied to a chart, it will plot order marks on the main chart pane and display simulated performance results in the “Strategy Tester” tab below:



Note

The results from a strategy applied to non-standard charts ([Heikin Ashi](#), [Renko](#), [Line Break](#), [Kagi](#), [Point & Figure](#), and [Range](#)) do not reflect actual market conditions by default. Strategy scripts will use the synthetic price values from these charts during simulation, which often do not align with actual market prices and will thus produce unrealistic backtest results. We therefore highly recommend using standard chart types for backtesting strategies. Alternatively, on Heikin Ashi charts, users can simulate orders using actual prices by enabling the “Fill orders using standard OHLC” option in the [Strategy properties](#) or by using `fill_orders_on_standard_ohlc = true` in the [strategy\(\)](#) function call.

Strategy tester

The Strategy Tester module is available to all scripts declared with the [strategy\(\)](#) function. Users can access this module from the “Strategy Tester” tab below their charts, where they can conveniently visualize their strategies and analyze hypothetical performance results.

Overview

The [Overview](#) tab of the Strategy Tester presents essential performance metrics and equity and

drawdown curves over a simulated sequence of trades, providing a quick look at strategy performance without diving into granular detail. The chart in this section shows the strategy's [equity curve](#) as a baseline plot centered at the initial value, the [buy and hold equity curve](#) as a line plot, and the [drawdown curve](#) as a histogram plot. Users can toggle these plots and scale them as absolute values or percentages using the options below the chart.



Note that:

- The overview chart uses two scales; the left is for the equity curves, and the right is for the drawdown curve.
- When a user clicks a point on these plots, this will direct the main chart view to the point where the trade was closed.

[Performance summary](#)

The [Performance Summary](#) tab of the module presents a comprehensive overview of a strategy's performance metrics. It displays three columns: one for all trades, one for all longs, and one for all shorts, to provide traders with more detailed insights on a strategy's long, short, and overall simulated trading performance.

[List of trades](#)

The [List of Trades](#) tab provides a granular look at the trades simulated by a strategy with essential information, including the date and time of execution, the type of order used (entry or exit), the number of contracts/shares/lots/units traded, and the price, as well as some key trade performance metrics.

Note that:

- Users can navigate the times of specific trades on their charts by clicking on them in this list.
- By clicking the "Trade #" field above the list, users can organize the trades in ascending order starting from the first or descending order starting from the last.

[Properties](#)

The Properties tab provides detailed information about a strategy's configuration and the dataset to which it is applied. It includes the strategy's date range, symbol information, script settings, and strategy properties.

- **Date Range** - Includes the range of dates with simulated trades and the total available backtesting range.
- **Symbol Info** - Contains the symbol name and broker/exchange, the chart's timeframe and type, the tick size, the point value for the chart, and the base currency.
- **Strategy Inputs** - Outlines the various parameters and variables used in the strategy script available in the "Inputs" tab of the script settings.

- **Strategy Properties** - Provides an overview of the configuration of the trading strategy. It includes essential details such as the initial capital, base currency, order size, margin, pyramiding, commission, and slippage. Additionally, this section highlights any modifications made to strategy calculation behavior.



Broker emulator

TradingView utilizes a *broker emulator* to simulate the performance of trading strategies. Unlike in real-life trading, the emulator strictly uses available chart prices for order simulation. Consequently, the simulation can only place historical trades after a bar closes, and it can only place real-time trades on a new price tick. For more information on this behavior, please refer to the Pine Script® [Execution model](#).

Since the emulator can only use chart data, it makes assumptions about intrabar price movement. It uses a bar's open, high, and low prices to infer intrabar activity while calculating order fills with the following logic:

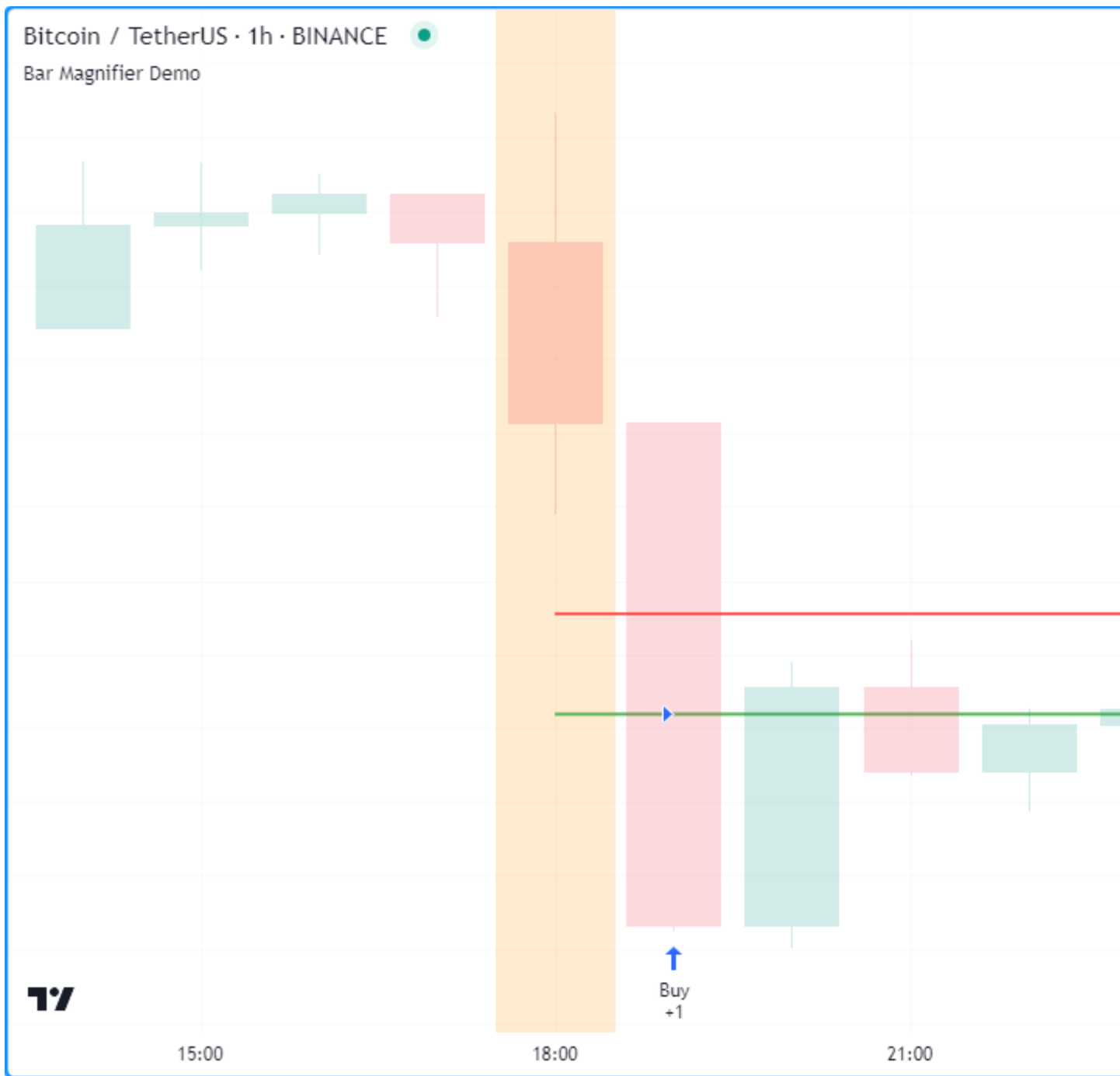
- If the high price is closer to the opening price than the low price, it assumes that the price moved in this order on the bar: open → high → low → close.
- If the low price is closer to the opening price than the high price, it assumes that the price moved in this order on the bar: open → low → high → close.
- The broker emulator assumes no gaps exist between prices within bars; in the “eyes” of the emulator, the full range of intrabar prices is available for order execution.



Bar magnifier

Premium account holders can override the broker emulator's intrabar assumptions via the `use_bar_magnifier` parameter of the `strategy()` function or the “Use bar magnifier” input in the “Properties” tab of the script settings. The [Bar Magnifier](#) inspects data on timeframes smaller than the chart's to obtain more granular information about price action within a bar, thus allowing more precise order fills during simulation.

To demonstrate, the following script places a “Buy” limit order at the `entryPrice` and an “Exit” limit order at the `exitPrice` when the `time` value crosses the `orderTime`, and draws two horizontal lines to visualize the order prices. The script also highlights the background using the `orderColor` to indicate when the strategy placed the orders:



```

//@version=5
strategy("Bar Magnifier Demo", overlay = true, use_bar_magnifier = false)

//@variable The UNIX timestamp to place the order at.
int orderTime = timestamp("UTC", 2023, 3, 22, 18)

//@variable Returns `color.orange` when `time` crosses the `orderTime`, false
otherwise.
color orderColor = na

// Entry and exit prices.
float entryPrice = hl2 - (high - low)
float exitPrice = entryPrice + (high - low) * 0.25

// Entry and exit lines.
var line entryLine = na
var line exitLine = na

```

```

if ta.cross(time, orderTime)
    // Draw new entry and exit lines.
    entryLine := line.new(bar_index, entryPrice, bar_index + 1, entryPrice,
color = color.green, width = 2)
    exitLine := line.new(bar_index, exitPrice, bar_index + 1, exitPrice, color
= color.red, width = 2)

    // Update order highlight color.
    orderColor := color.new(color.orange, 80)

    // Place limit orders at the `entryPrice` and `exitPrice`.
    strategy.entry("Buy", strategy.long, limit = entryPrice)
    strategy.exit("Exit", "Buy", limit = exitPrice)

// Update lines while the position is open.
else if strategy.position_size > 0.0
    entryLine.set_x2(bar_index + 1)
    exitLine.set_x2(bar_index + 1)

bgcolor(orderColor)

```

As we see in the chart above, the broker emulator assumed that intrabar prices moved from open to high, then high to low, then low to close on the bar the “Buy” order filled on, meaning the emulator assumed that the “Exit” order couldn’t fill on the same bar. However, after including `use_bar_magnifier = true` in the declaration statement, we see a different story:



Note

The maximum amount of intrabars that a script can request is 100,000. Some symbols with lengthier history may not have full intrabar coverage for their beginning chart bars with this limitation, meaning that simulated trades on those bars will not be affected by the bar magnifier.

Orders and entries

Just like in real-life trading, Pine strategies use orders to manage positions. In this context, an *order* is a command to simulate a market action, and a *trade* is the result after the order fills. Thus, to enter or exit positions using Pine, users must create orders with parameters that specify how they’ll behave.

To take a closer look at how orders work and how they become trades, let’s write a simple strategy script:

```

//@version=5
strategy("My strategy", overlay = true, margin_long = 100, margin_short = 100)

//@function Displays text passed to `txt` when called.
debugLabel(txt) =>
    label.new(
        bar_index, high, text = txt, color=color.lime, style =
label.style_label_lower_right,
        textcolor = color.black, size = size.large
    )

longCondition = bar_index % 20 == 0 // true on every 20th bar
if (longCondition)
    debugLabel("Long entry order created")
    strategy.entry("My Long Entry Id", strategy.long)

```

```
strategy.close_all()
```

In this script, we've defined a `longCondition` that is true whenever the `bar_index` is divisible by 20, i.e., every 20th bar. The strategy uses this condition within an `if` structure to simulate an entry order with `strategy.entry()` and draws a label at the entry price with the user-defined `debugLabel()` function. The script calls `strategy.close_all()` from the global scope to simulate a market order that closes any open position. Let's see what happens once we add the script to our chart:



The blue arrows on the chart indicate entry locations, and the purple ones mark the points where the strategy closed positions. Notice that the labels precede the actual entry point rather than occurring on the same bar - this is orders in action. By default, Pine strategies wait for the next available price tick before filling orders, as filling an order on the same tick isn't realistic. Also, they recalculate on the close of every historical bar, meaning the next available tick to fill an order at is the open of the next bar in this case. As a result, by default, all orders are delayed by one chart bar.

It's important to note that although the script calls `strategy.close_all()` from the global scope, forcing execution on every bar, the function call does nothing if the strategy isn't simulating an open position. If there is an open position, the command issues a market order to close it, which executes on the next available tick. For example, when the `longCondition` is true on bar 20, the strategy places an entry order to fill at the next tick, which is at the open of bar 21. Once the script recalculates its values on that bar's close, the function places an order to close the position, which fills at the open of bar 22.

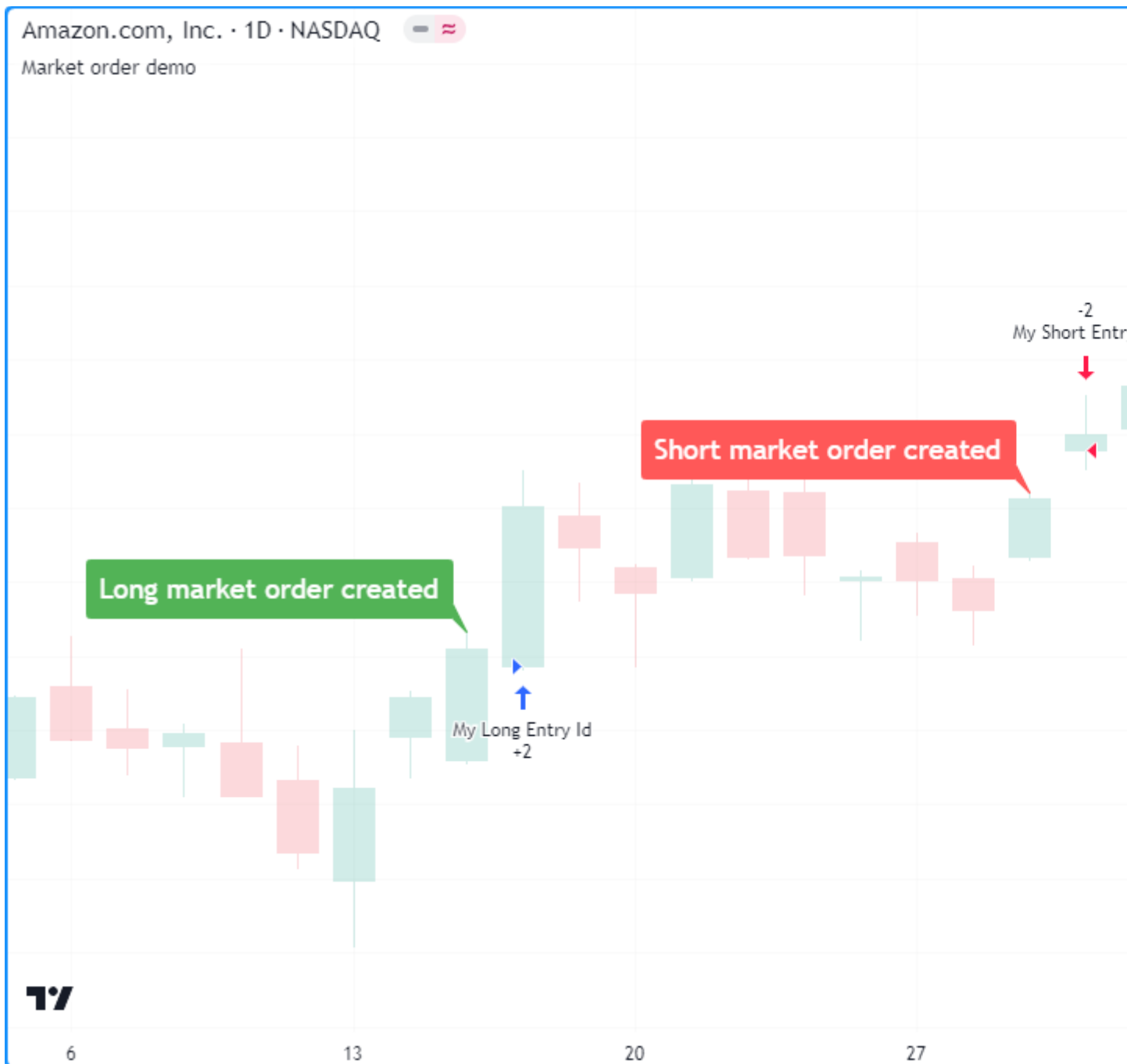
Order types

Pine Script® strategies allow users to simulate different order types for their particular needs. The main notable types are *market*, *limit*, *stop*, and *stop-limit*.

Market orders

Market orders are the most basic type of orders. They command a strategy to buy or sell a security as soon as possible, regardless of the price. Consequently, they always execute on the next available price tick. By default, all `strategy.*()` functions that generate orders specifically produce market orders.

The following script simulates a long market order when the `bar_index` is divisible by $2 * cycleLength$. Otherwise, it simulates a short market order when the `bar_index` is divisible by `cycleLength`, resulting in a strategy with alternating long and short trades once every `cycleLength` bars:



```

//@version=5
strategy("Market order demo", overlay = true, margin_long = 100, margin_short = 100)

//@variable Number of bars between long and short entries.
cycleLength = input.int(10, "Cycle length")

//@function Displays text passed to `txt` when called.
debugLabel(txt, lblColor) => label.new(
    bar_index, high, text = txt, color = lblColor, textcolor = color.white,
    style = label.style_label_lower_right, size = size.large
)

//@variable Returns `true` every `2 * cycleLength` bars.
longCondition = bar_index % (2 * cycleLength) == 0
//@variable Returns `true` every `cycleLength` bars.
shortCondition = bar_index % cycleLength == 0

```

```
// Generate a long market order with a `color.green` label on `longCondition`.
if longCondition
    debugLabel("Long market order created", color.green)
    strategy.entry("My Long Entry Id", strategy.long)
// Otherwise, generate a short market order with a `color.red` label on
`shortCondition`.
else if shortCondition
    debugLabel("Short market order created", color.red)
    strategy.entry("My Short Entry Id", strategy.short)
```

Limit orders

Limit orders command a strategy to enter a position at a specific price or better (lower than specified for long orders and higher for short ones). When the current market price is better than the order command's `limit` parameter, the order will fill without waiting for the market price to reach the limit level.

To simulate limit orders in a script, pass a price value to an order placement command with a `limit` parameter. The following example places a limit order 800 ticks below the bar close 100 bars before the `last_bar_index`:



```
//@version=5
strategy("Limit order demo", overlay = true, margin_long = 100, margin_short =
100)

//@function Displays text passed to `txt` and a horizontal line at `price` when
called.
debugLabel(price, txt) =>
    label.new(
        bar_index, price, text = txt, color = color.teal, textcolor =
color.white,
        style = label.style_label_lower_right, size = size.large
    )
    line.new(
        bar_index, price, bar_index + 1, price, color = color.teal, extend =
extend.right,
        style = line.style_dashed
    )

// Generate a long limit order with a label and line 100 bars before the
`last_bar_index`.
if last_bar_index - bar_index == 100
    limitPrice = close - syminfo.mintick * 800
    debugLabel(limitPrice, "Long Limit order created")
    strategy.entry("Long", strategy.long, limit = limitPrice)
```

Note how the script placed the label and started the line several bars before the trade. As long as the price remained above the `limitPrice` value, the order could not fill. Once the market price reached the limit, the strategy executed the trade mid-bar. If we had set the `limitPrice` to 800 ticks *above* the bar close rather than *below*, the order would fill immediately since the price is already at a better value:



```
//@version=5
strategy("Limit order demo", overlay = true, margin_long = 100, margin_short =
```

```

100)

//@function Displays text passed to `txt` and a horizontal line at `price` when
called.
debugLabel(price, txt) =>
  label.new(
    bar_index, price, text = txt, color = color.teal, textcolor =
color.white,
    style = label.style_label_lower_right, size = size.large
  )
  line.new(
    bar_index, price, bar_index + 1, price, color = color.teal, extend =
extend.right,
    style = line.style_dashed
  )

// Generate a long limit order with a label and line 100 bars before the
`last_bar_index`.
if last_bar_index - bar_index == 100
  limitPrice = close + syminfo.mintick * 800
  debugLabel(limitPrice, "Long Limit order created")
  strategy.entry("Long", strategy.long, limit = limitPrice)

```

Stop and stop-limit orders

Stop orders command a strategy to simulate another order after price reaches the specified `stop` price or a worse value (higher than specified for long orders and lower for short ones). They are essentially the opposite of limit orders. When the current market price is worse than the `stop` parameter, the strategy will trigger the subsequent order without waiting for the current price to reach the stop level. If the order placement command includes a `limit` argument, the subsequent order will be a limit order at the specified value. Otherwise, it will be a market order.

The script below places a stop order 800 ticks above the `close` 100 bars ago. In this example, the strategy entered a long position when the market price crossed the `stop` price some bars after it placed the order. Notice that the initial price at the time of the order was better than the one passed to `stop`. An equivalent limit order would have filled on the following chart bar:



```

//@version=5
strategy("Stop order demo", overlay = true, margin_long = 100, margin_short =
100)

//@function Displays text passed to `txt` when called and shows the `price`
level on the chart.
debugLabel(price, txt) =>
  label.new(
    bar_index, high, text = txt, color = color.teal, textcolor =
color.white,
    style = label.style_label_lower_right, size = size.large
  )
  line.new(bar_index, high, bar_index, price, style = line.style_dotted, color
= color.teal)
  line.new(
    bar_index, price, bar_index + 1, price, color = color.teal, extend =
extend.right,
    style = line.style_dashed
  )

// Generate a long stop order with a label and lines 100 bars before the last

```



```

bar.
if last_bar_index - bar_index == 100
    stopPrice = close + syminfo.mintick * 800
    debugLabel(stopPrice, "Long Stop order created")
    strategy.entry("Long", strategy.long, stop = stopPrice)

```

Order placement commands that use both `limit` and `stop` arguments produce stop-limit orders. This order type waits for the price to cross the stop level, then places a limit order at the specified limit price.

Let's modify our previous script to simulate and visualize a stop-limit order. In this example, we use the `low` value from 100 bars ago as the `limit` price in the entry command. This script also displays a label and price level to indicate when the strategy crosses the `stopPrice`, i.e., when the strategy activates the limit order. Notice how the market price initially reaches the limit level, but the strategy doesn't simulate a trade because the price must cross the `stopPrice` to place the pending limit order at the `limitPrice`:



```

//@version=5
strategy("Stop-Limit order demo", overlay = true, margin_long = 100,
margin_short = 100)

//@function Displays text passed to `txt` when called and shows the `price`
level on the chart.
debugLabel(price, txt, lblColor, lineWidth = 1) =>
    label.new(
        bar_index, high, text = txt, color = lblColor, textcolor = color.white,
        style = label.style_label_lower_right, size = size.large
    )
    line.new(bar_index, close, bar_index, price, style = line.style_dotted,
color = lblColor, width = lineWidth)
    line.new(
        bar_index, price, bar_index + 1, price, color = lblColor, extend =
extend.right,
        style = line.style_dashed, width = lineWidth
    )

var float stopPrice = na
var float limitPrice = na

// Generate a long stop-limit order with a label and lines 100 bars before the
last bar.
if last_bar_index - bar_index == 100
    stopPrice := close + syminfo.mintick * 800
    limitPrice := low
    debugLabel(limitPrice, "", color.gray)
    debugLabel(stopPrice, "Long Stop-Limit order created", color.teal)
    strategy.entry("Long", strategy.long, stop = stopPrice, limit = limitPrice)

// Draw a line and label once the strategy activates the limit order.
if high >= stopPrice
    debugLabel(limitPrice, "Limit order activated", color.green, 2)
    stopPrice := na

```

Order placement commands

Pine Script® strategies feature several functions to simulate the placement of orders, known as *order placement commands*. Each command serves a unique purpose and behaves differently from

the others.

[`strategy.entry\(\)`](#)

This command simulates entry orders. By default, strategies place market orders when calling this function, but they can also create stop, limit, and stop-limit orders when utilizing the `stop` and `limit` parameters.

To simplify opening positions, [strategy.entry\(\)](#) features several unique behaviors. One such behavior is that this command can reverse an open market position without additional function calls. When an order placed using [strategy.entry\(\)](#) fills, the function will automatically calculate the amount the strategy needs to close the open market position and trade `qty` contracts/shares/lots/units in the opposite direction by default. For example, if a strategy has an open position of 15 shares in the [strategy.long](#) direction and calls [strategy.entry\(\)](#) to place a market order in the [strategy.short](#) direction, the amount the strategy will trade to place the order is 15 shares plus the `qty` of the new short order.

The example below demonstrates a strategy that uses only [strategy.entry\(\)](#) calls to place entry orders. It creates a long market order with a `qty` value of 15 shares once every 100 bars and a short market order with a `qty` of 5 once every 25 bars. The script highlights the background blue and red for occurrences of the respective `buyCondition` and `sellCondition`:



```
//@version=5
strategy("Entry demo", "test", overlay = true)

//@variable Is `true` on every 100th bar.
buyCondition = bar_index % 100 == 0
//@variable Is `true` on every 25th bar except for those that are divisible by
100.
sellCondition = bar_index % 25 == 0 and not buyCondition

if buyCondition
    strategy.entry("buy", strategy.long, qty = 15)
if sellCondition
    strategy.entry("sell", strategy.short, qty = 5)

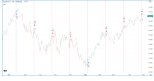
bgcolor(buyCondition ? color.new(color.blue, 90) : na)
bgcolor(sellCondition ? color.new(color.red, 90) : na)
```

As we see in the chart above, the order marks show that the strategy traded 20 shares on each order fill rather than 15 and 5. Since [strategy.entry\(\)](#) automatically reverses positions, unless otherwise specified via the [strategy.risk.allow_entry_in\(\)](#) function, it adds the open position size (15 for long entries) to the new order's size (5 for short entries) when it changes the direction, resulting in a traded quantity of 20 shares.

Notice that in the above example, although the `sellCondition` occurs three times before another `buyCondition`, the strategy only places a “sell” order on the first occurrence. Another unique behavior of the [strategy.entry\(\)](#) command is that it's affected by a script's *pyramiding* setting. Pyramiding specifies the number of consecutive orders the strategy can fill in the same direction. Its value is 1 by default, meaning the strategy only allows one consecutive order to fill in either direction. Users can set the strategy pyramiding values via the `pyramiding` parameter of the [strategy\(\)](#) function call or the “Pyramiding” input in the “Properties” tab of the script settings.

If we add `pyramiding = 3` to our previous script's declaration statement, the strategy will allow up to three consecutive trades in the same direction, meaning it can simulate new market orders on

each occurrence of `sellCondition`:



`strategy.order()`

This command simulates a basic order. Unlike most order placement commands, which contain internal logic to simplify interfacing with strategies, [strategy.order\(\)](#) uses the specified parameters without accounting for most additional strategy settings. Orders placed by [strategy.order\(\)](#) can open new positions and modify or close existing ones.

The following script uses only [strategy.order\(\)](#) calls to create and modify entries. The strategy simulates a long market order for 15 units every 100 bars, then three short orders for five units every 25 bars. The script highlights the background blue and red to indicate when the strategy simulates “buy” and “sell” orders:



```
//@version=5
strategy("Order demo", "test", overlay = true)

//@variable Is `true` on every 100th bar.
buyCond = bar_index % 100 == 0
//@variable Is `true` on every 25th bar except for those that are divisible by 100.
sellCond = bar_index % 25 == 0 and not buyCond

if buyCond
    strategy.order("buy", strategy.long, qty = 15) // Enter a long position of 15 units.
if sellCond
    strategy.order("sell", strategy.short, qty = 5) // Exit 5 units from the long position.

bgcolor(buyCond ? color.new(color.blue, 90) : na)
bgcolor(sellCond ? color.new(color.red, 90) : na)
```

This particular strategy will never simulate a short position, as unlike [strategy.entry\(\)](#), [strategy.order\(\)](#) does not automatically reverse positions. When using this command, the resulting market position is the net sum of the current market position and the filled order quantity. After the strategy fills the “buy” order for 15 units, it executes three “sell” orders that reduce the open position by five units each, and $15 - 5 * 3 = 0$. The same script would behave differently using [strategy.entry\(\)](#), as per the example shown in the [section above](#).

`strategy.exit()`

This command simulates exit orders. It’s unique in that it allows a strategy to exit a market position or form multiple exits in the form of stop-loss, take-profit, and trailing stop orders via the `loss`, `stop`, `profit`, `limit`, and `trail_*` parameters.

The most basic use of the [strategy.exit\(\)](#) command is the creation of levels where the strategy will exit a position due to losing too much money (stop-loss), earning enough money (take-profit), or both (bracket).

The stop-loss and take-profit functionalities of this command are associated with two parameters. The function’s `loss` and `profit` parameters specify stop-loss and take-profit values as a defined number of ticks away from the entry order’s price, while its `stop` and `limit` parameters provide

specific stop-loss and take-profit price values. The absolute parameters in the function call supersede the relative ones. If a `strategy.exit()` call contains `profit` and `limit` arguments, the command will prioritize the `limit` value and ignore the `profit` value. Likewise, it will only consider the `stop` value when the function call contains `stop` and `loss` arguments.

Note

Despite sharing the same names with parameters from `strategy.entry()` and `strategy.order()` commands, the `limit` and `stop` parameters work differently in `strategy.exit()`. In the first case, using `limit` and `stop` in the command will create a single stop-limit order that opens a limit order after crossing the stop price. In the second case, the command will create a separate limit and stop order to exit from an open position.

All exit orders from `strategy.exit()` with a `from_entry` argument are bound to the `id` of a corresponding entry order; strategies cannot simulate exit orders when there is no open market position or active entry order associated with a `from_entry` ID.

The following strategy places a “buy” entry order via `strategy.entry()` and a stop-loss and take-profit order via the `strategy.exit()` command every 100 bars. Notice that the script calls `strategy.exit()` twice. The “exit1” command references a “buy1” entry order, and “exit2” references the “buy” order. The strategy will only simulate exit orders from “exit2” because “exit1” references an order ID that doesn’t exist:



```
//@version=5
strategy("Exit demo", "test", overlay = true)

//@variable Is `true` on every 100th bar.
buyCondition = bar_index % 100 == 0

//@variable Stop-loss price for exit commands.
var float stopLoss = na
//@variable Take-profit price for exit commands.
var float takeProfit = na

// Place orders upon `buyCondition`.
if buyCondition
    if strategy.position_size == 0.0
        stopLoss := close * 0.99
        takeProfit := close * 1.01
        strategy.entry("buy", strategy.long)
        strategy.exit("exit1", "buy1", stop = stopLoss, limit = takeProfit) // Does
        nothing. "buy1" order doesn't exist.
        strategy.exit("exit2", "buy", stop = stopLoss, limit = takeProfit)

// Set `stopLoss` and `takeProfit` to `na` when price touches either, i.e., when
the strategy simulates an exit.
if low <= stopLoss or high >= takeProfit
    stopLoss := na
    takeProfit := na

plot(stopLoss, "SL", color.red, style = plot.style_circles)
plot(takeProfit, "TP", color.green, style = plot.style_circles)
```

Note that:

- Limit and stop orders from each exit command do not necessarily fill at the specified prices. Strategies can fill limit orders at better prices and stop orders at worse prices, depending on the range of values available to the broker emulator.

If a user does not provide a `from_entry` argument in the `strategy.exit()` call, the function will create exit orders for each open entry.

In this example, the strategy creates “buy1” and “buy2” entry orders and calls `strategy.exit()` without a `from_entry` argument every 100 bars. As we can see from the order marks on the chart, once the market price reaches the `stopLoss` or `takeProfit` values, the strategy fills an exit order for both “buy1” and “buy2” entries:



```
//@version=5
strategy("Exit all demo", "test", overlay = true, pyramiding = 2)

//@variable Is `true` on every 100th bar.
buyCondition = bar_index % 100 == 0

//@variable Stop-loss price for exit commands.
var float stopLoss = na
//@variable Take-profit price for exit commands.
var float takeProfit = na

// Place orders upon `buyCondition`.
if buyCondition
    if strategy.position_size == 0.0
        stopLoss := close * 0.99
        takeProfit := close * 1.01
        strategy.entry("buy1", strategy.long)
        strategy.entry("buy2", strategy.long)
        strategy.exit("exit", stop = stopLoss, limit = takeProfit) // Places orders
to exit all open entries.

// Set `stopLoss` and `takeProfit` to `na` when price touches either, i.e., when
the strategy simulates an exit.
if low <= stopLoss or high >= takeProfit
    stopLoss := na
    takeProfit := na

plot(stopLoss, "SL", color.red, style = plot.style_circles)
plot(takeProfit, "TP", color.green, style = plot.style_circles)
```

It is possible for a strategy to exit from the same entry ID more than once, which facilitates the formation of multi-level exit strategies. When performing multiple exit commands, each order’s quantity must be a portion of the traded quantity, with their sum not exceeding the open position. If the `qty` of the function is less than the size of the current market position, the strategy will simulate a partial exit. If the `qty` value exceeds the open position quantity, it will reduce the order since it cannot fill more contracts/shares/lots/units than the open position.

In the example below, we’ve modified our previous “Exit demo” script to simulate two stop-loss and take-profit orders per entry. The strategy places a “buy” order with a `qty` of two shares, “exit1” stop-loss and take-profit orders with a `qty` of one share, and “exit2” stop-loss and take profit orders with a `qty` of three shares:



```
//@version=5
strategy("Multiple exit demo", "test", overlay = true)

//@variable Is `true` on every 100th bar.
buyCondition = bar_index % 100 == 0
```

```

//@variable Stop-loss price for "exit1" commands.
var float stopLoss1 = na
//@variable Stop-loss price for "exit2" commands.
var float stopLoss2 = na
//@variable Take-profit price for "exit1" commands.
var float takeProfit1 = na
//@variable Take-profit price for "exit2" commands.
var float takeProfit2 = na

// Place orders upon `buyCondition`.
if buyCondition
    if strategy.position_size == 0.0
        stopLoss1 := close * 0.99
        stopLoss2 := close * 0.98
        takeProfit1 := close * 1.01
        takeProfit2 := close * 1.02
        strategy.entry("buy", strategy.long, qty = 2)
        strategy.exit("exit1", "buy", stop = stopLoss1, limit = takeProfit1, qty =
1)
        strategy.exit("exit2", "buy", stop = stopLoss2, limit = takeProfit2, qty =
3)

// Set `stopLoss1` and `takeProfit1` to `na` when price touches either.
if low <= stopLoss1 or high >= takeProfit1
    stopLoss1 := na
    takeProfit1 := na
// Set `stopLoss2` and `takeProfit2` to `na` when price touches either.
if low <= stopLoss2 or high >= takeProfit2
    stopLoss2 := na
    takeProfit2 := na

plot(stopLoss1, "SL1", color.red, style = plot.style_circles)
plot(stopLoss2, "SL2", color.red, style = plot.style_circles)
plot(takeProfit1, "TP1", color.green, style = plot.style_circles)
plot(takeProfit2, "TP2", color.green, style = plot.style_circles)

```

As we can see from the order marks on the chart, the strategy filled “exit2” orders despite the specified qty value exceeding the traded amount. Rather than using this quantity, the script reduced the orders’ sizes to match the remaining position.

Note that:

- All orders generated from a [strategy.exit\(\)](#) call belong to the same [strategy.oca.reduce](#) group, meaning that when either order fills, the strategy reduces all others to match the open position.

It’s important to note that orders produced by this command reserve a portion of the open market position to exit. [strategy.exit\(\)](#) cannot place an order to exit a portion of the position already reserved for exit by another exit command.

The following script simulates a “buy” market order for 20 shares 100 bars ago with “limit” and “stop” orders of 19 and 20 shares respectively. As we see on the chart, the strategy executed the “stop” order first. However, the traded quantity was only one share. Since the script placed the “limit” order first, the strategy reserved its qty (19 shares) to close the open position, leaving only one share to be closed by the “stop” order:



```

//@version=5
strategy("Reserved exit demo", "test", overlay = true)

```

```

//@variable "stop" exit order price.
var float stop = na
//@variable "limit" exit order price
var float limit = na
//@variable Is `true` 100 bars before the `last_bar_index`.
longCondition = last_bar_index - bar_index == 100

if longCondition
    stop := close * 0.99
    limit := close * 1.01
    strategy.entry("buy", strategy.long, 20)
    strategy.exit("limit", limit = limit, qty = 19)
    strategy.exit("stop", stop = stop, qty = 20)

bool showPlot = strategy.position_size != 0
plot(showPlot ? stop : na, "Stop", color.red, 2, plot.style_linebr)
plot(showPlot ? limit : na, "Limit 1", color.green, 2, plot.style_linebr)

```

Another key feature of the [strategy.exit\(\)](#) function is that it can create *trailing stops*, i.e., stop-loss orders that trail behind the market price by a specified amount whenever the price moves to a better value in the favorable direction. These orders have two components: the activation level and the trail offset. The activation level is the value the market price must cross to activate the trailing stop calculation, expressed in ticks via the `trail_points` parameter or as a price value via the `trail_price` parameter. If an exit call contains both arguments, the `trail_price` argument takes precedence. The trail offset is the distance the stop will follow behind the market price, expressed in ticks via the `trail_offset` parameter. For [strategy.exit\(\)](#) to create and activate trailing stops, the function call must contain `trail_offset` and either `trail_price` or `trail_points` arguments.

The example below shows a trailing stop in action and visualizes its behavior. The strategy simulates a long entry order on the bar 100 bars before the last bar on the chart, then a trailing stop on the next bar. The script has two inputs: one controls the activation level offset (i.e., the amount past the entry price required to activate the stop), and the other controls the trail offset (i.e., the distance to follow behind the market price when it moves to a better value in the desired direction).

The green dashed line on the chart shows the level the market price must cross to trigger the trailing stop order. After the price crosses this level, the script plots a blue line to signify the trailing stop. When the price rises to a new high value, which is favorable for the strategy since it means the position's value is increasing, the stop also rises to maintain a distance of `trailingStopOffset` ticks behind the current price. When the price decreases or doesn't reach a new high point, the stop value stays the same. Eventually, the price crosses below the stop, triggering the exit:



```

//@version=5
strategy("Trailing stop order demo", overlay = true, margin_long = 100,
margin_short = 100)

//@variable Offset used to determine how far above the entry price (in ticks)
the activation level will be located.
activationLevelOffset = input(1000, "Activation Level Offset (in ticks)")
//@variable Offset used to determine how far below the high price (in ticks) the
trailing stop will trail the chart.
trailingStopOffset = input(2000, "Trailing Stop Offset (in ticks)")

//@function Displays text passed to `txt` when called and shows the `price`

```

```

level on the chart.
debugLabel(price, txt, lblColor, hasLine = false) =>
    label.new(
        bar_index, price, text = txt, color = lblColor, textcolor =
color.white,
        style = label.style_label_lower_right, size = size.large
    )
    if hasLine
        line.new(
            bar_index, price, bar_index + 1, price, color = lblColor, extend =
extend.right,
            style = line.style_dashed
        )

//@variable The price at which the trailing stop activation level is located.
var float trailPriceActivationLevel = na
//@variable The price at which the trailing stop itself is located.
var float trailingStop = na
//@variable Caclulates the value that Trailing Stop would have if it were active
at the moment.
theoreticalStopPrice = high - trailingStopOffset * syminfo.mintick

// Generate a long market order to enter 100 bars before the last bar.
if last_bar_index - bar_index == 100
    strategy.entry("Long", strategy.long)

// Generate a trailing stop 99 bars before the last bar.
if last_bar_index - bar_index == 99
    trailPriceActivationLevel := open + syminfo.mintick * activationLevelOffset
    strategy.exit(
        "Trailing Stop", from_entry = "Long", trail_price =
trailPriceActivationLevel,
        trail_offset = trailingStopOffset
    )
    debugLabel(trailPriceActivationLevel, "Trailing Stop Activation Level",
color.green, true)

// Visualize the trailing stop mechanic in action.
// If there is an open trade, check whether the Activation Level has been
achieved.
// If it has been achieved, track the trailing stop by assigning its value to a
variable.
if strategy.opentrades == 1
    if na(trailingStop) and high > trailPriceActivationLevel
        debugLabel(trailPriceActivationLevel, "Activation level crossed",
color.green)
        trailingStop := theoreticalStopPrice
        debugLabel(trailingStop, "Trailing Stop Activated", color.blue)

    else if theoreticalStopPrice > trailingStop
        trailingStop := theoreticalStopPrice

// Visualize the movement of the trailing stop.
plot(trailingStop, "Trailing Stop")

```

[`strategy.close\(\)` and `strategy.close_all\(\)`](#)

These commands simulate exit positions using market orders. The functions close trades upon being called rather than at a specific price.

The example below demonstrates a simple strategy that places a “buy” order via [strategy.entry\(\)](#) once every 50 bars that it closes with a market order using [strategy.close\(\)](#) 25 bars afterward:



```

//@version=5
strategy("Close demo", "test", overlay = true)

//@variable Is `true` on every 50th bar.
buyCond = bar_index % 50 == 0
//@variable Is `true` on every 25th bar except for those that are divisible by
50.
sellCond = bar_index % 25 == 0 and not buyCond

if buyCond
    strategy.entry("buy", strategy.long)
if sellCond
    strategy.close("buy")

bgcolor(buyCond ? color.new(color.blue, 90) : na)
bgcolor(sellCond ? color.new(color.red, 90) : na)

```

Unlike most other order placement commands, the `id` parameter of [strategy.close\(\)](#) references an existing entry ID to close. If the specified `id` does not exist, the command will not execute an order. If a position was formed from multiple entries with the same ID, the command will exit all entries simultaneously.

To demonstrate, the following script places a “buy” order once every 25 bars. The script closes all “buy” entries once every 100 bars. We’ve included `pyramiding = 3` in the [strategy\(\)](#) declaration statement to allow the strategy to simulate up to three orders in the same direction:



```
//@version=5
strategy("Multiple close demo", "test", overlay = true, pyramiding = 3)

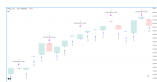
//@variable Is `true` on every 100th bar.
sellCond = bar_index % 100 == 0
//@variable Is `true` on every 25th bar except for those that are divisible by 100.
buyCond = bar_index % 25 == 0 and not sellCond

if buyCond
    strategy.entry("buy", strategy.long)
if sellCond
    strategy.close("buy")

bgcolor(buyCond ? color.new(color.blue, 90) : na)
bgcolor(sellCond ? color.new(color.red, 90) : na)
```

For cases where a script has multiple entries with different IDs, the [strategy.close_all\(\)](#) command can come in handy since it closes all entries, irrespective of their IDs.

The script below places “A”, “B”, and “C” entry orders sequentially based on the number of open trades, then closes all of them with a single market order:



```
//@version=5
strategy("Close multiple ID demo", "test", overlay = true, pyramiding = 3)

switch strategy.opentrades
    0 => strategy.entry("A", strategy.long)
    1 => strategy.entry("B", strategy.long)
    2 => strategy.entry("C", strategy.long)
    3 => strategy.close_all()
```

[`strategy.cancel\(\)` and `strategy.cancel_all\(\)`](#)

These commands allow a strategy to cancel pending orders, i.e., those generated by [strategy.exit\(\)](#) or by [strategy.order\(\)](#) or [strategy.entry\(\)](#) when they use `limit` or `stop` arguments.

The following strategy simulates a “buy” limit order 500 ticks below the closing price 100 bars ago, then cancels the order on the next bar. The script draws a horizontal line at the `limitPrice` and colors the background green and orange to indicate when the limit order is placed and canceled respectively. As we can see, nothing happened once the market price crossed the `limitPrice` because the strategy already canceled the order:



```

//@version=5
strategy("Cancel demo", "test", overlay = true)

//@variable Draws a horizontal line at the `limit` price of the "buy" order.
var line limitLine = na

//@variable Returns `color.green` when the strategy places the "buy" order,
`color.orange` when it cancels the order.
color bgColor = na

if last_bar_index - bar_index == 100
    float limitPrice = close - syminfo.mintick * 500
    strategy.entry("buy", strategy.long, limit = limitPrice)
    limitLine := line.new(bar_index, limitPrice, bar_index + 1, limitPrice,
extend = extend.right)
    bgColor := color.new(color.green, 50)

if last_bar_index - bar_index == 99
    strategy.cancel("buy")
    bgColor := color.new(color.orange, 50)

bgcolor(bgColor)

```

As with [strategy.close\(\)](#), the `id` parameter of [strategy.cancel\(\)](#) refers to the ID of an existing entry. This command will do nothing if the `id` parameter references an ID that doesn't exist. When there are multiple pending orders with the same ID, this command will cancel all of them at once.

In this example, we've modified the previous script to place a "buy" limit order on three consecutive bars starting from 100 bars ago. The strategy cancels all of them after the `bar_index` is 97 bars away from the most recent bar, resulting in it doing nothing when the price crosses any of the lines:



```

//@version=5
strategy("Multiple cancel demo", "test", overlay = true, pyramiding = 3)

//@variable Draws a horizontal line at the `limit` price of the "buy" order.
var line limitLine = na

//@variable Returns `color.green` when the strategy places the "buy" order,
`color.orange` when it cancels the order.
color bgColor = na

if last_bar_index - bar_index <= 100 and last_bar_index - bar_index >= 98
    float limitPrice = close - syminfo.mintick * 500
    strategy.entry("buy", strategy.long, limit = limitPrice)
    limitLine := line.new(bar_index, limitPrice, bar_index + 1, limitPrice,
extend = extend.right)
    bgColor := color.new(color.green, 50)

if last_bar_index - bar_index == 97
    strategy.cancel("buy")
    bgColor := color.new(color.orange, 50)

bgcolor(bgColor)

```

Note that:

- We added `pyramiding = 3` to the script's declaration statement to allow three [strategy.entry\(\)](#) orders to fill. Alternatively, the script would achieve the same output

by using [strategy.order\(\)](#) since it isn't sensitive to the pyramiding setting.

It's important to note that neither [strategy.cancel\(\)](#) nor [strategy.cancel_all\(\)](#) can cancel *market* orders, as the strategy executes them immediately upon the next tick. Strategies cannot cancel orders after they've been filled. To close an open position, use [strategy.close\(\)](#) or [strategy.close_all\(\)](#).

This example simulates a "buy" market order 100 bars ago, then attempts to cancel all pending orders on the next bar. Since the strategy already filled the "buy" order, the [strategy.cancel_all\(\)](#) command does nothing in this case, as there are no pending orders to cancel:



```
//@version=5
strategy("Cancel market demo", "test", overlay = true)

//@variable Returns `color.green` when the strategy places the "buy" order,
`color.orange` when it tries to cancel.
color bgColor = na

if last_bar_index - bar_index == 100
    strategy.entry("buy", strategy.long)
    bgColor := color.new(color.green, 50)

if last_bar_index - bar_index == 99
    strategy.cancel_all()
    bgColor := color.new(color.orange, 50)

bgcolor(bgColor)
```

Position sizing

Pine Script® strategies feature two ways to control the sizes of simulated trades:

- Set a default fixed quantity type and value for all orders using the `default_qty_type` and `default_qty_value` arguments in the [strategy\(\)](#) function, which also sets the default values in the "Properties" tab of the script settings.
- Specify the `qty` argument when calling [strategy.entry\(\)](#). When a user supplies this argument to the function, the script ignores the strategy's default quantity value and type.

The following example simulates "Buy" orders of `longAmount` size whenever the low price equals the lowest value, and "Sell" orders of `shortAmount` size when the high price equals the highest value:



```
//@version=5
strategy("Buy low, sell high", overlay = true, default_qty_type = strategy.cash,
default_qty_value = 5000)

int length = input.int(20, "Length")
float longAmount = input.float(4.0, "Long Amount")
float shortAmount = input.float(2.0, "Short Amount")

float highest = ta.highest(length)
float lowest = ta.lowest(length)
```

```

switch
  low == lowest  => strategy.entry("Buy", strategy.long, longAmount)
  high == highest => strategy.entry("Sell", strategy.short, shortAmount)

```

Notice that in the above example, although we've specified the `default_qty_type` and `default_qty_value` arguments in the declaration statement, the script does not use these defaults for the simulated orders. Instead, it sizes them as a `longAmount` and `shortAmount` of units. If we want the script to use the default type and value, we must remove the `qty` specification from the [strategy.entry\(\)](#) calls:



```

//@version=5
strategy("Buy low, sell high", overlay = true, default_qty_type = strategy.cash,
default_qty_value = 5000)

int length = input.int(20, "Length")

float highest = ta.highest(length)
float lowest  = ta.lowest(length)

switch
  low == lowest  => strategy.entry("Buy", strategy.long)
  high == highest => strategy.entry("Sell", strategy.short)

```

Closing a market position

Although it is possible to simulate an exit from a specific entry order shown in the List of Trades tab of the [Strategy Tester](#) module, all orders are linked according to FIFO (first in, first out) rules. If the user does not specify the `from_entry` parameter of a [strategy.exit\(\)](#) call, the strategy will exit the open market position starting from the first entry order that opened it.

The following example simulates two orders sequentially: "Buy1" at the market price for the last 100 bars and "Buy2" once the position size matches the size of "Buy1". The strategy only places an exit order when the `positionSize` is 15 units. The script does not supply a `from_entry` argument to the [strategy.exit\(\)](#) command, so the strategy places exit orders for all open positions each time it calls the function, starting with the first. It plots the `positionSize` in a separate pane for visual reference:



```

//@version=5
strategy("Exit Demo", pyramiding = 2)

float positionSize = strategy.position_size

if positionSize == 0 and last_bar_index - bar_index <= 100
  strategy.entry("Buy1", strategy.long, 5)
else if positionSize == 5
  strategy.entry("Buy2", strategy.long, 10)
else if positionSize == 15
  strategy.exit("bracket", loss = 10, profit = 10)

plot(positionSize == 0 ? na : positionSize, "Position Size", color.lime, 4,
plot.style_histogram)

```

Note that:

- We included `pyramiding = 2` in our script's declaration statement to allow it to simulate two consecutive orders in the same direction.

Suppose we wanted to exit "Buy2" before "Buy1". Let's see what happens if we instruct the strategy to close "Buy2" before "Buy1" when it fills both orders:



```
//@version=5
strategy("Exit Demo", pyramiding = 2)

float positionSize = strategy.position_size

if positionSize == 0 and last_bar_index - bar_index <= 100
    strategy.entry("Buy1", strategy.long, 5)
else if positionSize == 5
    strategy.entry("Buy2", strategy.long, 10)
else if positionSize == 15
    strategy.close("Buy2")
    strategy.exit("bracket", "Buy1", loss = 10, profit = 10)

plot(positionSize == 0 ? na : positionSize, "Position Size", color.lime, 4,
plot.style_histogram)
```

As we can see in the Strategy Tester's "List of Trades" tab, rather than closing the "Buy2" position with `strategy.close()`, it closes the quantity of "Buy1" first, which is half the quantity of the close order, then closes half of the "Buy2" position, as the broker emulator follows FIFO rules by default. Users can change this behavior by specifying `close_entries_rule = "ANY"` in the `strategy()` function.

OCA groups

One-Cancels-All (OCA) groups allow a strategy to fully or partially cancel other orders upon the execution of order placement commands, including `strategy.entry()` and `strategy.order()`, with the same `oca_name`, depending on the `oca_type` that the user provides in the function call.

`strategy.oca.cancel``

The `strategy.oca.cancel` OCA type cancels all orders with the same `oca_name` upon the fill or partial fill of an order from the group.

For example, the following strategy executes orders upon `ma1` crossing `ma2`. When the `strategy.position_size` is 0, it places long and short stop orders on the `high` and `low` of the bar. Otherwise, it calls `strategy.close_all()` to close all open positions with a market order. Depending on the price action, the strategy may fill both orders before issuing a close order. Additionally, if the broker emulator's intrabar assumption supports it, both orders may fill on the same bar. The `strategy.close_all()` command does nothing in such cases, as the script cannot invoke the action until after already executing both orders:



```
//@version=5
strategy("OCA Cancel Demo", overlay=true)
```

```

float ma1 = ta.sma(close, 5)
float ma2 = ta.sma(close, 9)

if ta.cross(ma1, ma2)
    if strategy.position_size == 0
        strategy.order("Long", strategy.long, stop = high)
        strategy.order("Short", strategy.short, stop = low)
    else
        strategy.close_all()

plot(ma1, "Fast MA", color.aqua)
plot(ma2, "Slow MA", color.orange)

```

To eliminate scenarios where the strategy fills long and short orders before a close order, we can instruct it to cancel one order after it executes the other. In this example, we've set the `oca_name` for both [strategy.order\(\)](#) commands to "Entry" and their `oca_type` to `strategy.oca.cancel`:



```

//@version=5
strategy("OCA Cancel Demo", overlay=true)

float ma1 = ta.sma(close, 5)
float ma2 = ta.sma(close, 9)

if ta.cross(ma1, ma2)
    if strategy.position_size == 0
        strategy.order("Long", strategy.long, stop = high, oca_name = "Entry",
oca_type = strategy.oca.cancel)
        strategy.order("Short", strategy.short, stop = low, oca_name = "Entry",
oca_type = strategy.oca.cancel)
    else
        strategy.close_all()

plot(ma1, "Fast MA", color.aqua)
plot(ma2, "Slow MA", color.orange)

```

[strategy.oca.reduce](#)

The [strategy.oca.reduce](#) OCA type does not cancel orders. Instead, it reduces the size of orders with the same `oca_name` upon each new fill by the number of closed contracts/shares/lots/units, which is particularly useful for exit strategies.

The following example demonstrates an attempt at a long-only exit strategy that generates a stop-loss order and two take-profit orders for each new entry. Upon the crossover of two moving averages, it simulates a "Long" entry order using [strategy.entry\(\)](#) with a `qty` of 6 units, then simulates stop/limit orders for 6, 3, and 3 units using [strategy.order\(\)](#) at the `stop`, `limit1`, and `limit2` prices respectively.

After adding the strategy to our chart, we see it doesn't work as intended. The issue with this script is that [strategy.order\(\)](#) doesn't belong to an OCA group by default, unlike [strategy.exit\(\)](#). Since we have not explicitly assigned the orders to an OCA group, the strategy does not cancel or reduce them when it fills one, meaning it's possible to trade a greater quantity than the open position and reverse the direction:

Euro / U.S. Dollar · 1D · OANDA ● O1.07694 H1.08586 L1.07591 C1.08474 +0.00780 (+0.72%)

1.08466 1.6 1.08482

Multiple TP Demo 0.97926 0.99904 1.00893



1D 5D 1M 3M 6M YTD 1Y 5Y All

Forex Screener Pine Editor Strategy Tester Trading Panel

Multiple TP Demo

Overview Performance Summary List of Trades Properties

Trade # ↓	Type	Signal	Date/Time
5	Exit Short	Open	
	Entry Short	Stop	2002-07-30
4	Exit Long	Limit 2	2002-07-15
	Entry Long	Long	2002-07-11

```
//@version=5
strategy("Multiple TP Demo", overlay = true)

var float stop = na
var float limit1 = na
var float limit2 = na

bool longCondition = ta.crossover(ta.sma(close, 5), ta.sma(close, 9))
```



```

if longCondition and strategy.position_size == 0
    stop := close * 0.99
    limit1 := close * 1.01
    limit2 := close * 1.02
    strategy.entry("Long", strategy.long, 6)
    strategy.order("Stop", strategy.short, stop = stop, qty = 6)
    strategy.order("Limit 1", strategy.short, limit = limit1, qty = 3)
    strategy.order("Limit 2", strategy.short, limit = limit2, qty = 3)

bool showPlot = strategy.position_size != 0
plot(showPlot ? stop : na, "Stop", color.red, style = plot.style_linebr)
plot(showPlot ? limit1 : na, "Limit 1", color.green, style = plot.style_linebr)
plot(showPlot ? limit2 : na, "Limit 2", color.green, style = plot.style_linebr)

```

For our strategy to work as intended, we must instruct it to reduce the number of units for the other stop-loss/take-profit orders so that they do not exceed the size of the remaining open position.

In the example below, we've set the `oca_name` for each order in our exit strategy to "Bracket" and the `oca_type` to [strategy.oca.reduce](#). These settings tell the strategy to reduce the `qty` values of orders in the "Bracket" group by the `qty` filled when it executes one of them, preventing it from trading an excessive number of units and causing a reversal:



```
//@version=5
strategy("Multiple TP Demo", overlay = true)

var float stop = na
var float limit1 = na
var float limit2 = na

bool longCondition = ta.crossover(ta.sma(close, 5), ta.sma(close, 9))
if longCondition and strategy.position_size == 0
    stop := close * 0.99
    limit1 := close * 1.01
    limit2 := close * 1.02
    strategy.entry("Long", strategy.long, 6)
    strategy.order("Stop", strategy.short, stop = stop, qty = 6, oca_name =
"Bracket", oca_type = strategy.oca.reduce)
    strategy.order("Limit 1", strategy.short, limit = limit1, qty = 3, oca_name
= "Bracket", oca_type = strategy.oca.reduce)
    strategy.order("Limit 2", strategy.short, limit = limit2, qty = 6, oca_name
= "Bracket", oca_type = strategy.oca.reduce)
```

```

bool showPlot = strategy.position_size != 0
plot(showPlot ? stop : na, "Stop", color.red, style = plot.style_linebr)
plot(showPlot ? limit1 : na, "Limit 1", color.green, style = plot.style_linebr)
plot(showPlot ? limit2 : na, "Limit 2", color.green, style = plot.style_linebr)

```

Note that:

- We changed the `qty` of the “Limit 2” order to 6 instead of 3 because the strategy will reduce its value by 3 when it fills the “Limit 1” order. Keeping the `qty` value of 3 would cause it to drop to 0 and never fill after filling the first limit order.

[`strategy.oca.none`](#)

The [strategy.oca.none](#) OCA type specifies that an order executes independently of any OCA group. This value is the default `oca_type` for [strategy.order\(\)](#) and [strategy.entry\(\)](#) order placement commands.

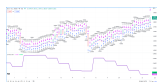
Note

If two order placement commands have the same `oca_name` but different `oca_type` values, the strategy considers them to be from two distinct groups. i.e., OCA groups cannot combine [strategy.oca.cancel](#), [strategy.oca.reduce](#), and [strategy.oca.none](#) OCA types.

Currency

Pine Script® strategies can use different base currencies than the instruments they calculate on. Users can specify the simulated account’s base currency by including a `currency.*` variable as the `currency` argument in the [strategy\(\)](#) function, which will change the script’s [strategy.account_currency](#) value. The default `currency` value for strategies is `currency.NONE`, meaning that the script uses the base currency of the instrument on the chart.

When a strategy script uses a specified base currency, it multiplies the simulated profits by the `FX_IDC` conversion rate from the previous trading day. For example, the strategy below places an entry order for a standard lot (100,000 units) with a profit target and stop-loss of 1 point on each of the last 500 chart bars, then plots the net profit alongside the inverted daily close of the symbol in a separate pane. We have set the base currency to `currency.EUR`. When we add this script to `FX_IDC:EURUSD`, the two plots align, confirming the strategy uses the previous day’s rate from this symbol for its calculations:



```

//@version=5
strategy("Currency Test", currency = currency.EUR)

if last_bar_index - bar_index < 500
    strategy.entry("LE", strategy.long, 100000)
    strategy.exit("LX", "LE", profit = 1, loss = 1)
plot(math.abs(ta.change(strategy.netprofit)), "1 Point profit", color =
color.fuchsia, linewidth = 4)
plot(request.security(syminfo.tickerid, "D", 1 / close)[1], "Previous day's
inverted price", color = color.lime)

```

Note that:

- When trading on timeframes higher than daily, the strategy will use the closing price from one trading day before the bar closes for cross-rate calculation on historical bars.

For example, on a weekly timeframe, it will base the cross-rate on the previous Thursday's closing value, though the strategy will still use the daily closing rate for real-time bars.

Altering calculation behavior

Strategies execute on all historical bars available from a chart, then automatically continue their calculations in real-time as new data is available. By default, strategy scripts only calculate once per confirmed bar. We can alter this behavior by changing the parameters of the [strategy\(\)](#) function or clicking the checkboxes in the "Recalculate" section of the script's "Properties" tab.

`calc_on_every_tick`

`calc_on_every_tick` is an optional setting that controls the calculation behavior on real-time data. When this parameter is enabled, the script will recalculate its values on each new price tick. By default, its value is false, meaning the script only executes calculations after a bar is confirmed.

Enabling this calculation behavior may be particularly useful when forward testing since it facilitates granular, real-time strategy simulation. However, it's important to note that this behavior introduces a data difference between real-time and historical simulations, as historical bars do not contain tick information. Users should exercise caution with this setting, as the data difference may cause a strategy to repaint its history.

The following script will simulate a new order each time that `close` reaches the highest or lowest value over the input length. Since `calc_on_every_tick` is enabled in the strategy declaration, the script will simulate new orders on each new real-time price tick after compilation:

```
//@version=5
strategy("Donchian Channel Break", overlay = true, calc_on_every_tick = true,
pyramiding = 20)

int length = input.int(15, "Length")

float highest = ta.highest(close, length)
float lowest = ta.lowest(close, length)

if close == highest
    strategy.entry("Buy", strategy.long)
if close == lowest
    strategy.entry("Sell", strategy.short)

//@variable The starting time for real-time bars.
var realTimeStart = timenow

// Color the background of real-time bars.
bgcolor(time_close >= realTimeStart ? color.new(color.orange, 80) : na)

plot(highest, "Highest", color = color.lime)
plot(lowest, "Lowest", color = color.red)
```

Note that:

- The script uses a `pyramiding` value of 20 in its declaration, which allows the strategy to simulate a maximum of 20 trades in the same direction.
- To visually demarcate what bars are processed as real-time bars by the strategy, the script colors the background for all bars since the [timenow](#) when it was last compiled.

After applying the script to the chart and letting it calculate on some real-time bars, we may see an

output like the following:



The script placed “Buy” orders on each new real-time tick the condition was valid on, resulting in multiple orders per bar. However, it may surprise users unfamiliar with this behavior to see the strategy’s outputs change after recompiling the script, as the bars that it previously executed real-time calculations on are now historical bars, which do not hold tick information:



`calc_on_order_fills`

The optional `calc_on_order_fills` setting enables the recalculation of a strategy immediately after simulating an order fill, which allows the script to use more granular prices and place additional orders without waiting for a bar to be confirmed.

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

The example below shows a simple strategy declared with `calc_on_order_fills` enabled that simulates a “Buy” order when the `strategy.position_size` is 0. The script uses the `strategy.position_avg_price` to calculate a `stopLoss` and `takeProfit` and simulates “Exit” orders when the price crosses them, regardless of whether the bar is confirmed. As a result, as soon as an exit is triggered, the strategy recalculates and places a new entry order because the `strategy.position_size` is once again equal to 0. The strategy places the order once the exit happens and executes it on the next tick after the exit, which will be one of the bar’s OHLC values, depending on the emulated intrabar movement:



```
//@version=5
strategy("Intrabar exit", overlay = true, calc_on_order_fills = true)

float stopSize = input.float(5.0, "SL %", minval = 0.0) / 100.0
float profitSize = input.float(5.0, "TP %", minval = 0.0) / 100.0

if strategy.position_size == 0.0
    strategy.entry("Buy", strategy.long)

float stopLoss = strategy.position_avg_price * (1.0 - stopSize)
float takeProfit = strategy.position_avg_price * (1.0 + profitSize)

strategy.exit("Exit", stop = stopLoss, limit = takeProfit)
```

Note that:

- With `calc_on_order_fills` turned off, the same strategy will only ever enter one bar after it triggers an exit order. First, the mid-bar exit will happen, but no entry order. Then, the strategy will simulate an entry order once the bar closes, which it will fill on the next tick after that, i.e., the open of the next bar.

It’s important to note that enabling `calc_on_order_fills` may produce unrealistic strategy results, as the broker emulator may assume order prices that are not possible when trading in real-

time. Users must exercise caution with this setting and carefully consider the logic in their scripts.

The following example simulates a “Buy” order after each new order fill and bar confirmation over a 25-bar window from the [last_bar_index](#) when the script loaded on the chart. With the setting enabled, the strategy simulates four entries per bar since the emulator considers each bar to have four ticks (open, high, low, close), which is unrealistic behavior, as it’s not typically possible for an order to fill at the exact high or low of a bar:



```
//@version=5
strategy("buy on every fill", overlay = true, calc_on_order_fills = true,
pyramiding = 100)

if last_bar_index - bar_index <= 25
    strategy.entry("Buy", strategy.long)
```

[`process_orders_on_close`](#)

The default strategy behavior simulates orders at the close of each bar, meaning that the earliest opportunity to fill the orders and execute strategy calculations and alerts is upon the opening of the following bar. Traders can change this behavior to process a strategy using the closing value of each bar by enabling the `process_orders_on_close` setting.

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

Note that:

- It’s crucial to be aware that using strategies with `process_orders_on_close` in a live trading environment may lead to a repainting strategy, as alerts on the close of a bar still occur when the market closes, and orders may not fill until the next market open.

[Simulating trading costs](#)

For a strategy performance report to contain relevant, meaningful data, traders should strive to account for potential real-world costs in their strategy results. Neglecting to do so may give traders an unrealistic view of strategy performance and undermine the credibility of test results. Without modeling the potential costs associated with their trades, traders may overestimate a strategy’s historical profitability, potentially leading to suboptimal decisions in live trading. Pine Script® strategies include inputs and parameters for simulating trading costs in performance results.

[Commission](#)

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

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

lowest value:



```
//@version=5
strategy("Commission Demo", overlay=true, default_qty_value = 2,
default_qty_type = strategy.percent_of_equity)

length = input.int(10, "Length")

float highest = ta.highest(close, length)
float lowest = ta.lowest(close, length)

switch close
    highest => strategy.entry("Long", strategy.long)
    lowest => strategy.close("Long")

plot(highest, color = color.new(color.lime, 50))
plot(lowest, color = color.new(color.red, 50))
```

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



```
//@version=5
strategy(
    "Commission Demo", overlay=true, default_qty_value = 2, default_qty_type =
strategy.percent_of_equity,
    commission_type = strategy.commission.percent, commission_value = 1
)

length = input.int(10, "Length")

float highest = ta.highest(close, length)
float lowest = ta.lowest(close, length)

switch close
    highest => strategy.entry("Long", strategy.long)
    lowest => strategy.close("Long")

plot(highest, color = color.new(color.lime, 50))
plot(lowest, color = color.new(color.red, 50))
```

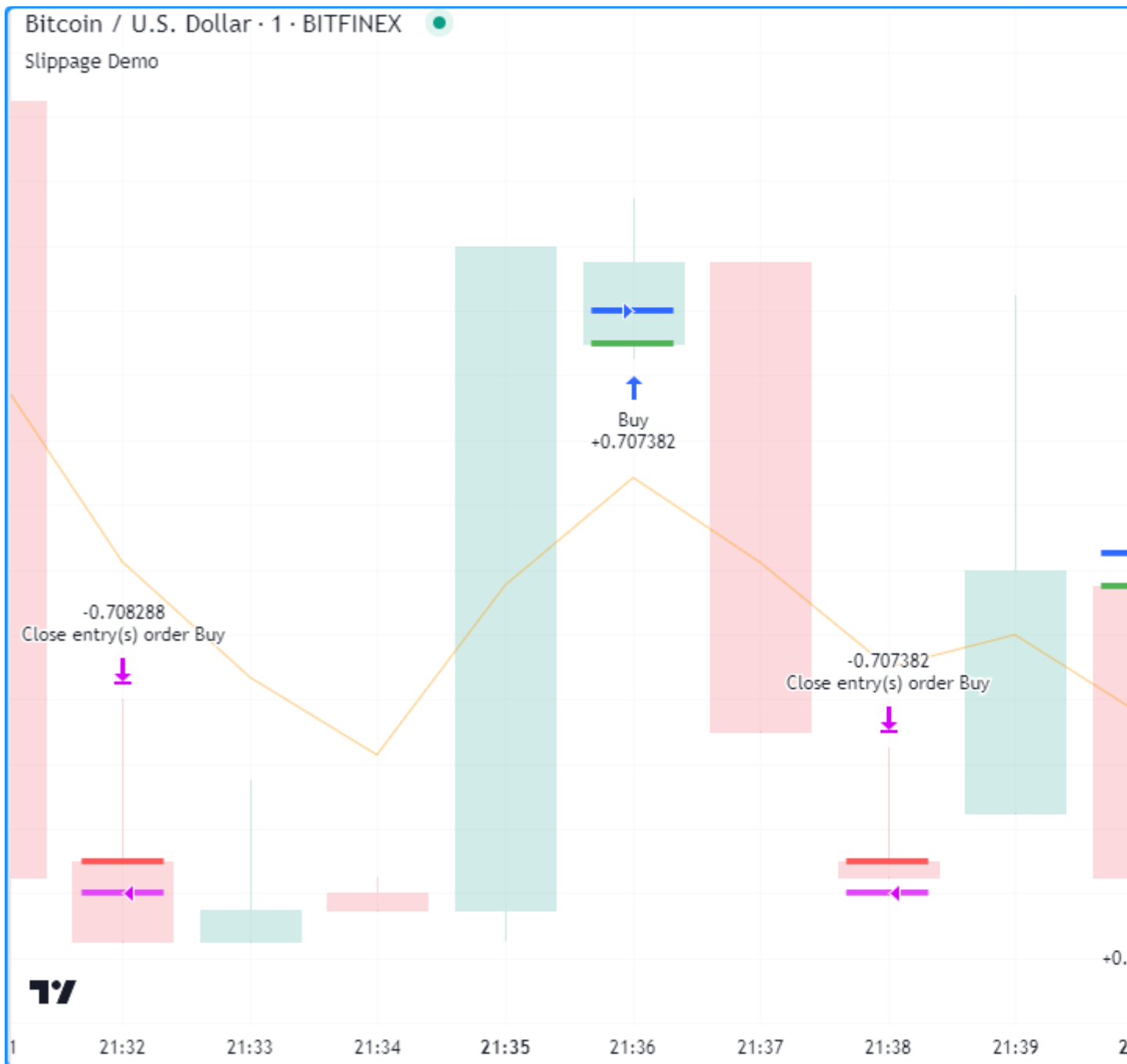
As we can see in the example above, after applying a 1% commission to the backtest, the strategy simulated a significantly reduced net profit of only 1.42% and a more volatile equity curve with an elevated max drawdown, highlighting the impact commission simulation can have on a strategy's test results.

Slippage and unfilled limits

In real-life trading, a broker/exchange may fill orders at slightly different prices than a trader intended due to volatility, liquidity, order size, and other market factors, which can profoundly impact a strategy's performance. The disparity between expected prices and the actual prices at

which the broker/exchange executes trades is what we refer to as slippage. Slippage is dynamic and unpredictable, making it impossible to simulate precisely. However, factoring in a small amount of slippage on each trade during a backtest or forward test may help the results better align with reality. Users can model slippage in their strategy results, sized as a fixed number of ticks, by including a `slippage` argument in the [strategy\(\)](#) declaration or by setting the “Slippage” input in the “Properties” tab of the strategy settings.

The following example demonstrates how slippage simulation affects the fill prices of market orders in a strategy test. The script below places a “Buy” market order of 2% equity when the market price is above an EMA while the EMA is rising and closes the position when the price dips below the EMA while it’s falling. We’ve included `slippage = 20` in the [strategy\(\)](#) function, which declares that the price of each simulated order will slip 20 ticks in the direction of the trade. The script uses [strategy.opentrades.entry_bar_index\(\)](#) and [strategy.closedtrades.exit_bar_index\(\)](#) to get the `entryIndex` and `exitIndex`, which it utilizes to obtain the `fillPrice` of the order. When the bar index is at the `entryIndex`, the `fillPrice` is the first [strategy.opentrades.entry_price\(\)](#) value. At the `exitIndex`, `fillPrice` is the [strategy.closedtrades.exit_price\(\)](#) value from the last closed trade. The script plots the expected fill price along with the simulated fill price after slippage to visually compare the difference:



```

//@version=5
strategy(
    "Slippage Demo", overlay = true, slippage = 20,
    default_qty_value = 2, default_qty_type = strategy.percent_of_equity
)

int length = input.int(5, "Length")

//@variable Exponential moving average with an input `length`.
float ma = ta.ema(close, length)

//@variable Returns `true` when `ma` has increased and `close` is greater than
it, `false` otherwise.
bool longCondition = close > ma and ma > ma[1]
//@variable Returns `true` when `ma` has decreased and `close` is less than it,
`false` otherwise.
bool shortCondition = close < ma and ma < ma[1]

```

```

// Enter a long market position on `longCondition`, close the position on
`shortCondition`.
if longCondition
    strategy.entry("Buy", strategy.long)
if shortCondition
    strategy.close("Buy")

//@variable The `bar_index` of the position's entry order fill.
int entryIndex = strategy.opentrades.entry_bar_index(0)
//@variable The `bar_index` of the position's close order fill.
int exitIndex = strategy.closedtrades.exit_bar_index(strategy.closedtrades - 1)

//@variable The fill price simulated by the strategy.
float fillPrice = switch bar_index
    entryIndex => strategy.opentrades.entry_price(0)
    exitIndex => strategy.closedtrades.exit_price(strategy.closedtrades - 1)

//@variable The expected fill price of the open market position.
float expectedPrice = fillPrice ? open : na

color expectedColor = na
color filledColor = na

if bar_index == entryIndex
    expectedColor := color.green
    filledColor := color.blue
else if bar_index == exitIndex
    expectedColor := color.red
    filledColor := color.fuchsia

plot(ma, color = color.new(color.orange, 50))

plotchar(fillPrice ? open : na, "Expected fill price", "-", location.absolute,
expectedColor)
plotchar(fillPrice, "Fill price after slippage", "-", location.absolute,
filledColor)

```

Note that:

- Since the strategy applies constant slippage to all order fills, some orders can fill outside the candle range in the simulation. Thus users should exercise caution with this setting, as excessive simulated slippage can produce unrealistically worse testing results.

Some traders may assume that they can avoid the adverse effects of slippage by using limit orders, as unlike market orders, they cannot execute at a worse price than the specified value. However, depending on the state of the real-life market, even if the market price reaches an order price, there's a chance that a limit order may not fill, as limit orders can only fill if a security has sufficient liquidity and price action around the value. To account for the possibility of unfilled orders in a backtest, users can specify the `backtest_fill_limits_assumption` value in the declaration statement or use the "Verify price for limit orders" input in the "Properties" tab to instruct the strategy to fill limit orders only after prices move a defined number of ticks past order prices.

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



```

//@version=5
strategy(
    "Verify price for limits example", overlay = true,
    default_qty_type = strategy.percent_of_equity, default_qty_value = 2
)

int length = input.int(25, title = "Length")

//@variable Draws a line at the limit price of the most recent entry order.
var line limitLine = na

// Highest high and lowest low
highest = ta.highest(length)
lowest  = ta.lowest(length)

// Place an entry order and draw a new line when the the `high` equals the
`highest` value and `limitLine` is `na`.
if high == highest and na(limitLine)
    float limitPrice = hlcc4
    strategy.entry("Long", strategy.long, limit = limitPrice)
    limitLine := line.new(bar_index, limitPrice, bar_index + 1, limitPrice)

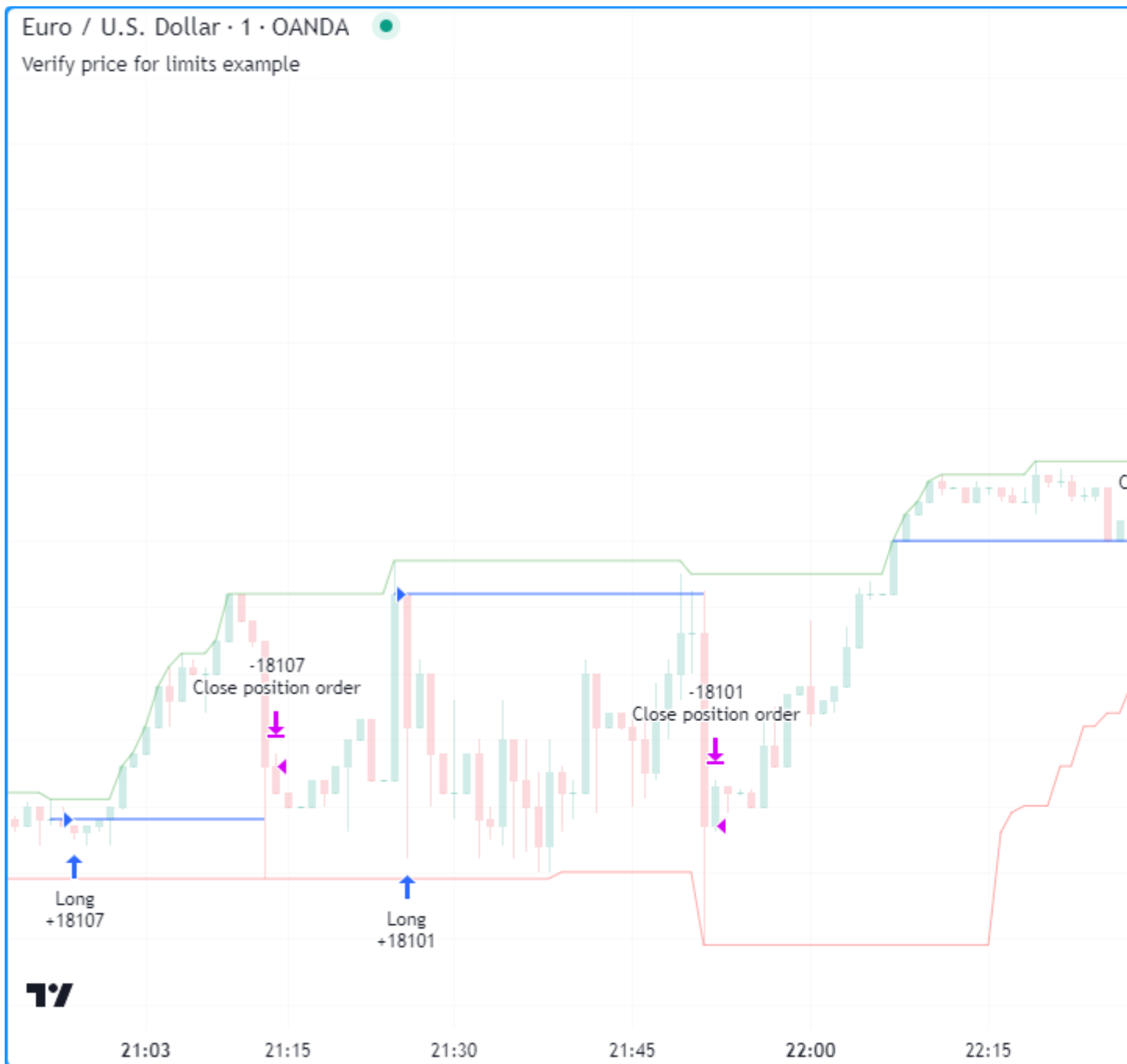
// Close the open market position, cancel orders, and set `limitLine` to `na`
when the `low` equals the `lowest` value.
if low == lowest
    strategy.cancel_all()
    limitLine := na
    strategy.close_all()

// Update the `x2` value of `limitLine` if it isn't `na`.
if not na(limitLine)
    limitLine.set_x2(bar_index + 1)

plot(highest, "Highest High", color = color.new(color.green, 50))
plot(lowest, "Lowest Low", color = color.new(color.red, 50))

```

By default, the script assumes that all limit orders are guaranteed to fill. However, this is often not the case in real-life trading. Let's add price verification to our limit orders to account for potentially unfilled ones. In this example, we've included `backtest_fill_limits_assumption = 3` in the [strategy\(\)](#) function call. As we can see, using limit verification omits some simulated order fills and changes the times of others since the entry orders can now only fill after the price penetrates the limit price by three ticks:



Note

It's important to notice that although the limit verification changed the *times* of some order fills, the strategy simulated them at the same *prices*. This “time-warping” effect is a compromise that preserves the prices of verified limit orders, but it can cause the strategy to simulate their fills at times that wouldn't necessarily be possible in the real world. Users should exercise caution with this setting and understand its limitations when analyzing strategy results.

Risk management

Designing a strategy that performs well, let alone one that does so in a broad class of markets, is a challenging task. Most are designed for specific market patterns/conditions and may produce uncontrollable losses when applied to other data. Therefore, a strategy's risk management qualities can be critical to its performance. Users can set risk management criteria in their strategy scripts using the special commands with the `strategy.risk` prefix.

Strategies can incorporate any number of risk management criteria in any combination. All risk management commands execute on every tick and order execution event, regardless of any changes to the strategy's calculation behavior. There is no way to disable any of these commands at a script's runtime. Irrespective of the risk rule's location, it will always apply to the strategy unless the user removes the call from the code.

[strategy.risk.allow_entry_in\(\)](#)

This command overrides the market direction allowed for [strategy.entry\(\)](#) commands. When a user specifies the trade direction with this function (e.g., [strategy.direction.long](#)), the strategy will only enter trades in that direction. However, it's important to note that if a script calls an entry command in the opposite direction while there's an open market position, the strategy will simulate a market order to exit the position.

[strategy.risk.max_cons_loss_days\(\)](#)

This command cancels all pending orders, closes the open market position, and stops all additional trade actions after the strategy simulates a defined number of trading days with consecutive losses.

[strategy.risk.max_drawdown\(\)](#)

This command cancels all pending orders, closes the open market position, and stops all additional trade actions after the strategy's drawdown reaches the amount specified in the function call.

[strategy.risk.max_intraday_filled_orders\(\)](#)

This command specifies the maximum number of filled orders per trading day (or per chart bar if the timeframe is higher than daily). Once the strategy executes the maximum number of orders for the day, it cancels all pending orders, closes the open market position, and halts trading activity until the end of the current session.

[strategy.risk.max_intraday_loss\(\)](#)

This command controls the maximum loss the strategy will tolerate per trading day (or per chart bar if the timeframe is higher than daily). When the strategy's losses reach this threshold, it will cancel all pending orders, close the open market position, and stop all trading activity until the end of the current session.

[strategy.risk.max_position_size\(\)](#)

This command specifies the maximum possible position size when using [strategy.entry\(\)](#) commands. If the quantity of an entry command results in a market position that exceeds this threshold, the strategy will reduce the order quantity so that the resulting position does not exceed the limitation.

Margin

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

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

1. Calculate the amount of capital spent on the position: $\text{Money Spent} = \text{Quantity} * \text{Price}$

- Entry Price
2. Calculate the Market Value of Security (MVS): $MVS = \text{Position Size} * \text{Current Price}$
 3. Calculate the Open Profit as the difference between MVS and Money Spent. If the position is short, we multiply this by -1.
 4. Calculate the strategy's equity value: $\text{Equity} = \text{Initial Capital} + \text{Net Profit} + \text{Open Profit}$
 5. Calculate the margin ratio: $\text{Margin Ratio} = \text{Margin Percent} / 100$
 6. Calculate the margin value, which is the cash required to cover the trader's portion of the position: $\text{Margin} = MVS * \text{Margin Ratio}$
 7. Calculate the available funds: $\text{Available Funds} = \text{Equity} - \text{Margin}$
 8. Calculate the total amount of money the trader has lost: $\text{Loss} = \text{Available Funds} / \text{Margin Ratio}$
 9. Calculate how many contracts/shares/lots/units the trader would need to liquidate to cover the loss. We truncate this value to the same decimal precision as the minimum position size for the current symbol: $\text{Cover Amount} = \text{TRUNCATE}(\text{Loss} / \text{Current Price})$.
 10. Calculate how many units the broker will liquidate to cover the loss: $\text{Margin Call} = \text{Cover Amount} * 4$

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



The first entry happened at the bar's opening price on 16 Sep 2010. The strategy bought 682,438 shares (Position size) at 4.43 USD (Entry price). Then, on 23 Sep 2010, when the price dipped to 3.9 (Current price), the emulator forcibly liquidated 111,052 shares via margin call.

Money spent: $682438 * 4.43 = 3023200.34$
MVS: $682438 * 3.9 = 2661508.2$
Open Profit: -361692.14
Equity: $1000000 + 0 - 361692.14 = 638307.86$
Margin Ratio: $25 / 100 = 0.25$
Margin: $2661508.2 * 0.25 = 665377.05$
Available Funds: $638307.86 - 665377.05 = -27069.19$
Money Lost: $-27069.19 / 0.25 = -108276.76$
Cover Amount: $\text{TRUNCATE}(-108276.76 / 3.9) = \text{TRUNCATE}(-27763.27) = -27763$
Margin Call Size: $-27763 * 4 = -111052$

Strategy Alerts

Regular Pine Script[®] indicators have two different mechanisms to set up custom alert conditions: the [alertcondition\(\)](#) function, which tracks one specific condition per function call, and the [alert\(\)](#) function, which tracks all its calls simultaneously, but provides greater flexibility in the number of calls, alert messages, etc.

Pine Script[®] strategies do not work with [alertcondition\(\)](#) calls, but they do support the generation of custom alerts via the [alert\(\)](#) function. Along with this, each function that creates orders also comes with its own built-in alert functionality that does not require any additional code to implement. As such, any strategy that uses an order placement command can issue alerts upon order execution. The precise mechanics of such built-in strategy alerts are described in the Order Fill events section of the [Alerts](#) page in our User Manual.

When a strategy uses functions that create orders and the `alert()` function together, the alert creation dialogue provides a choice between the conditions that it will trigger upon: it can trigger on `alert()` events, order fill events, or both.

For many trading strategies, the latency between a triggered condition and a live trade can be a critical performance factor. By default, strategy scripts can only execute `alert()` function calls on the close of real-time bars, considering them to use `alert.freq_once_per_bar_close`, regardless of the `freq` argument in the call. Users can change the alert frequency by also including `calc_on_every_tick = true` in the `strategy()` call or selecting the “Recalculate on every tick” option in the “Properties” tab of the strategy settings before creating the alert. However, depending on the script, this may also adversely impact a strategy’s behavior, so exercise caution and be aware of the limitations when using this approach.

When sending alerts to a third party for strategy automation, we recommend using order fill alerts rather than the `alert()` function since they don’t suffer the same limitations; alerts from order fill events execute immediately, unaffected by a script’s `calc_on_every_tick` setting. Users can set the default message for order fill alerts via the `@strategy_alert_message` compiler annotation. The text provided with this annotation will populate the “Message” field for order fills in the alert creation dialogue.

The following script shows a simple example of a default order fill alert message. Above the `strategy()` declaration statement, it uses `@strategy_alert_message` with *placeholders* for the trade action, position size, ticker, and fill price values in the message text:

```
//@version=5
//@strategy_alert_message {{strategy.order.action}} {{strategy.position_size}}
{{ticker}} @ {{strategy.order.price}}
strategy("Alert Message Demo", overlay = true)
float fastMa = ta.sma(close, 5)
float slowMa = ta.sma(close, 10)

if ta.crossover(fastMa, slowMa)
    strategy.entry("buy", strategy.long)

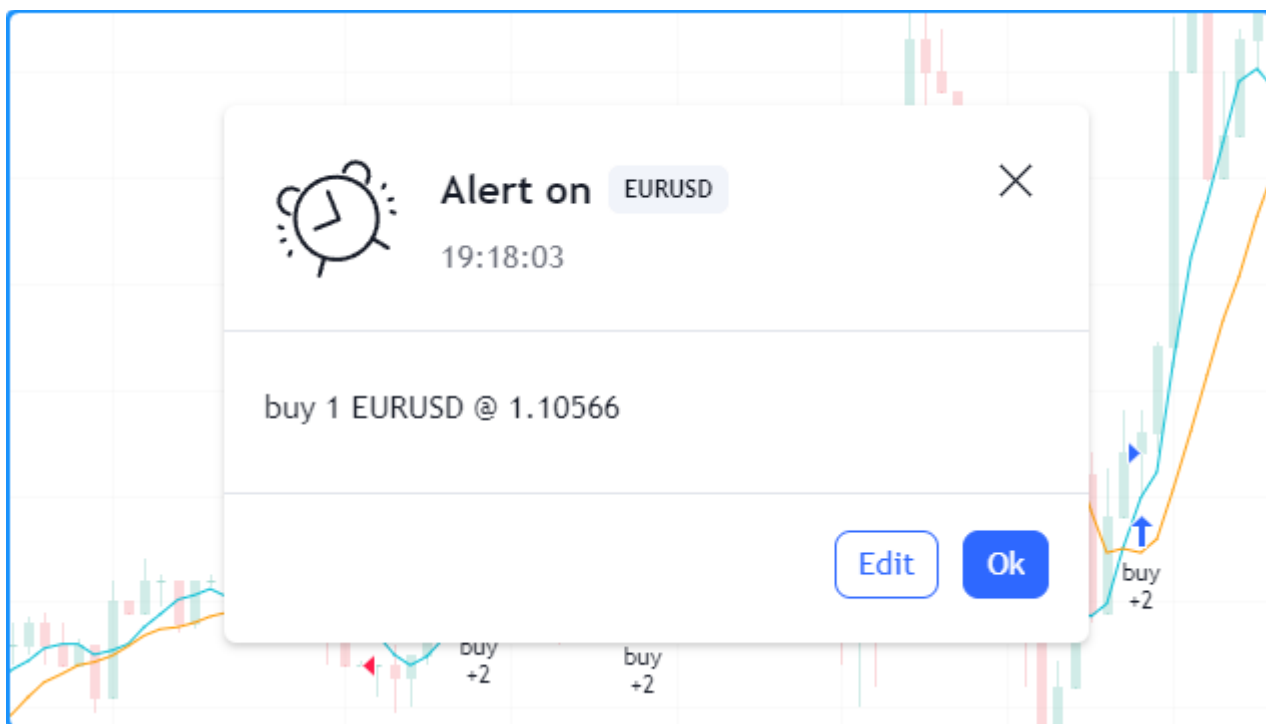
if ta.crossunder(fastMa, slowMa)
    strategy.entry("sell", strategy.short)

plot(fastMa, "Fast MA", color.aqua)
plot(slowMa, "Slow MA", color.orange)
```

This script will populate the alert creation dialogue with its default message when the user selects its name from the “Condition” dropdown tab:



Upon the alert trigger, the strategy will populate the placeholders in the alert message with their corresponding values. For example:



Notes on testing strategies

It's common for traders to test and tune their strategies in historical and real-time market conditions because many believe that analyzing the results may provide valuable insight into a strategy's characteristics, potential weaknesses, and possibly its future potential. However, traders should always be aware of the biases and limitations of simulated strategy results, especially when using the results to support live trading decisions. This section outlines some caveats associated with strategy validation and tuning and possible solutions to mitigate their effects.

Note

While testing strategies on existing data may give traders helpful information about a strategy's qualities, it's important to note that neither the past nor the present guarantees the future. Financial markets can change rapidly and unpredictably, which may cause a strategy to sustain uncontrollable losses. Additionally, simulated results may not fully account for other real-world factors that can impact trading performance. Therefore, we recommend that traders thoroughly understand the limitations and risks when evaluating backtests and forward tests and consider them "parts of the whole" in their validation processes rather than basing decisions solely on the results.

Backtesting and forward testing

Backtesting is a technique that traders use to evaluate the historical performance of a trading strategy or model by simulating and analyzing its past results on historical market data; this technique assumes that analysis of a strategy's results on past data may provide insight into its strengths and weaknesses. When backtesting, many traders tweak the parameters of a strategy in an attempt to optimize its results. Analysis and optimization of historical results may help traders to gain a deeper understanding of a strategy. However, traders should always understand the risks and limitations when basing their decisions on optimized backtest results.

Parallel to backtesting, prudent trading system development often also involves incorporating real-time analysis as a tool for evaluating a trading system on a forward-looking basis. Forward testing aims to gauge the performance of a strategy in real-time, real-world market conditions, where factors such as trading costs, slippage, and liquidity can meaningfully affect its performance.

Forward testing has the distinct advantage of not being affected by certain types of biases (e.g., lookahead bias or “future data leakage”) but carries the disadvantage of being limited in the quantity of data to test. Therefore, it’s not typically a standalone solution for strategy validation, but it can provide helpful insights into a strategy’s performance in current market conditions.

Backtesting and forward testing are two sides of the same coin, as both approaches aim to validate the effectiveness of a strategy and identify its strengths and weaknesses. By combining backtesting and forward testing, traders may be able to compensate for some limitations and gain a clearer perspective on their strategy’s performance. However, it’s up to traders to sanitize their strategies and evaluation processes to ensure that insights align with reality as closely as possible.

Lookahead bias

One typical issue in backtesting some strategies, namely ones that request alternate timeframe data, use repainting variables such as [timenow](#), or alter calculation behavior for intrabar order fills, is the leakage of future data into the past during evaluation, which is known as lookahead bias. Not only is this bias a common cause of unrealistic strategy results since the future is never actually knowable beforehand, but it is also one of the typical causes of strategy repainting. Traders can often confirm this bias by forward testing their systems, as lookahead bias does not apply to real-time data where no known data exists beyond the current bar. Users can eliminate this bias in their strategies by ensuring that they don’t use repainting variables that leak the future into the past, `request.*()` functions don’t include [barmerge.lookahead_on](#) without offsetting the data series as described on [this](#) section of our page on [repainting](#), and they use realistic calculation behavior.

Selection bias

Selection bias is a common issue that many traders experience when testing their strategies. It occurs when a trader only analyzes results on specific instruments or timeframes while ignoring others. This bias can result in a distorted perspective of the strategy’s robustness, which may impact trading decisions and performance optimizations. Traders can reduce the effects of selection bias by evaluating their strategies on multiple, ideally diverse, symbols and timeframes, making it a point not to ignore poor performance results in their analysis or cherry-pick testing ranges.

Overfitting

A common pitfall when optimizing a backtest is the potential for overfitting (“curve fitting”), which occurs when the strategy is tailored for specific data and fails to generalize well on new, unseen data. One widely-used approach to help reduce the potential for overfitting and promote better generalization is to split an instrument’s data into two or more parts to test the strategy outside the sample used for optimization, otherwise known as “in-sample” (IS) and “out-of-sample” (OOS) backtesting. In this approach, traders use the IS data for strategy optimization, while the OOS portion is used for testing and evaluating IS-optimized performance on new data without further optimization. While this and other, more robust approaches may provide a glimpse into how a strategy might fare after optimization, traders should exercise caution, as the future is inherently unknowable. No trading strategy can guarantee future performance, regardless of the data used for testing and optimization.



Tables

- [Introduction](#)
- [Creating tables](#)
 - [Placing a single value in a fixed position](#)
 - [Coloring the chart's background](#)
 - [Creating a display panel](#)
 - [Displaying a heatmap](#)
- [Tips](#)

[Introduction](#)

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

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

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

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

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

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

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

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

- Use the [var](#) keyword to declare tables.
- Enclose all other calls inside an [if barstate.islast](#) block.

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

Creating tables

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

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

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

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

Placing a single value in a fixed position

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

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



Note that:

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

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

```
//@version=5
indicator("ATR", "", true)
atrPeriodInput = input.int(14, "ATR period", minval = 1, tooltip = "Using a
period of 1 yields True Range.")

var table atrDisplay = table.new(position.top_right, 1, 1, bgcolor = color.gray,
frame_width = 2, frame_color = color.black)
myAtr = ta.atr(atrPeriodInput)
if barstate.islast
    table.cell(atrDisplay, 0, 0, str.tostring(myAtr, format.mintick), text_color
= color.white)
```



Note that:

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

Coloring the chart's background

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

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

colorChartBg(bullColorInput, bearColorInput)
```

Note that:

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

width and height of the indicator's space, so it covers the whole chart.

Creating a display panel

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

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

BTCUSD

1s 5s 15s 1m 5m 15m 30m 1h 2h 4h 12h D W M

Bitcoin / U.S. Dollar - 5 - BITSTAMP

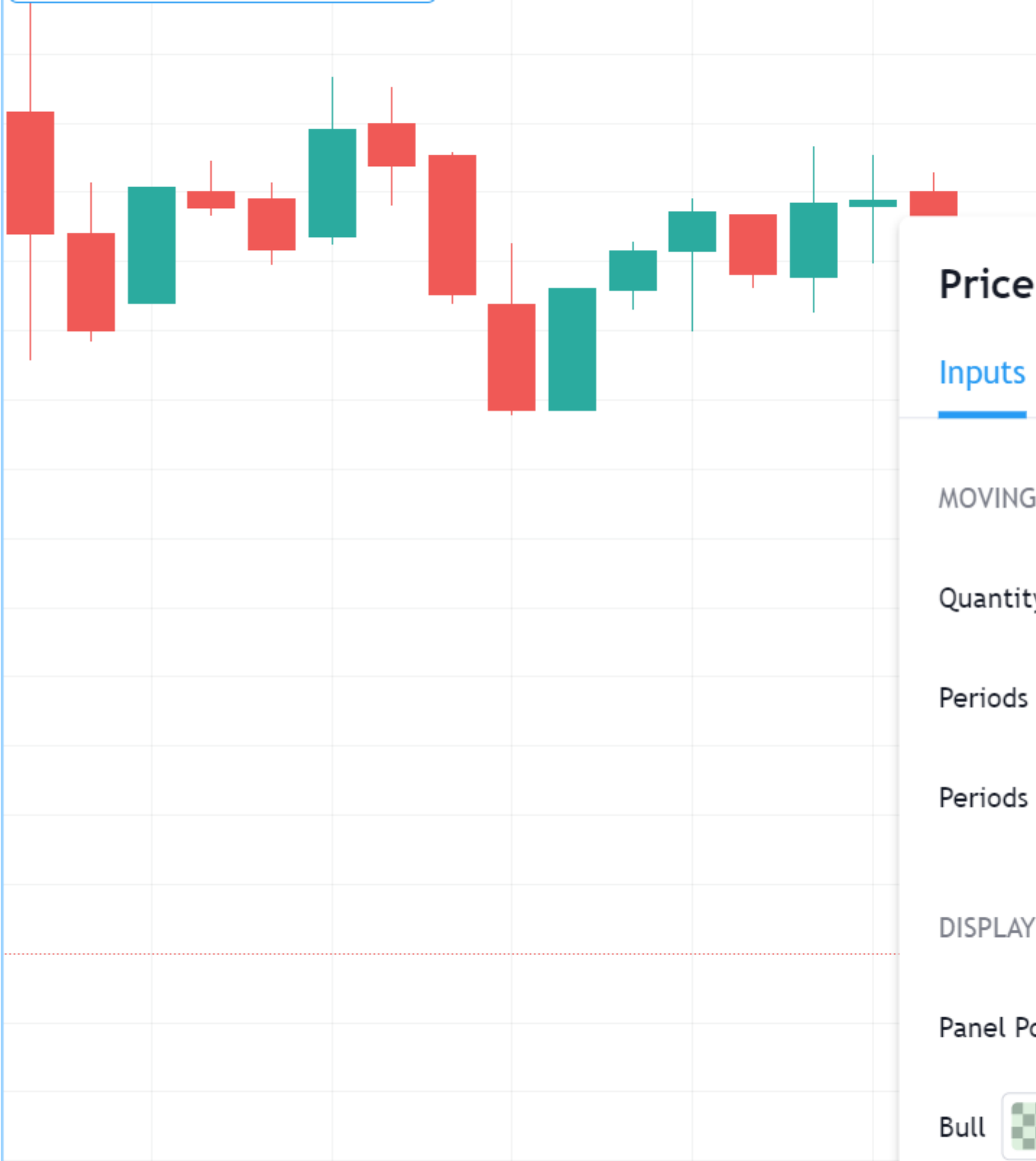
O39956.13 H40010.77 L39593.10 C3

39734.41

64.14

39798.55

Price vs MA     



Price

Inputs

MOVING

Quantity

Periods

Periods

DISPLAY

Panel Po

Bull 

```

//@version=5
indicator("Price vs MA", "", true)

var string GP1 = "Moving averages"
int   masQtyInput   = input.int(20, "Quantity", minval = 1, maxval = 40,
group = GP1, tooltip = "1-40")
int   masStartInput = input.int(20, "Periods begin at", minval = 2, maxval =
200, group = GP1, tooltip = "2-200")
int   masStepInput  = input.int(20, "Periods increase by", minval = 1, maxval =
100, group = GP1, tooltip = "1-100")

var string GP2 = "Display"
string tableYposInput = input.string("top", "Panel position", inline = "11",
options = ["top", "middle", "bottom"], group = GP2)
string tableXposInput = input.string("right", "", inline = "11", options =
["left", "center", "right"], group = GP2)
color  bullColorInput = input.color(color.new(color.green, 30), "Bull", inline
= "12", group = GP2)
color  bearColorInput = input.color(color.new(color.red, 30), "Bear", inline =
"12", group = GP2)
color  neutColorInput = input.color(color.new(color.gray, 30), "Neutral",
inline = "12", group = GP2)

var table panel = table.new(tableYposInput + "_" + tableXposInput, 2,
masQtyInput + 1)
if barstate.islast
    // Table header.
    table.cell(panel, 0, 0, "MA", bgcolor = neutColorInput)
    table.cell(panel, 1, 0, "Value", bgcolor = neutColorInput)

int period = masStartInput
for i = 1 to masQtyInput
    // —— Call MAs on each bar.
    float ma = ta.sma(close, period)
    // —— Only execute table code on last bar.
    if barstate.islast
        // Period in left column.
        table.cell(panel, 0, i, str.tostring(period), bgcolor = neutColorInput)
        // If MA is between the open and close, use neutral color. If close is
lower/higher than MA, use bull/bear color.
        bgColor = close > ma ? open < ma ? neutColorInput : bullColorInput :
bearColorInput
        // MA value in right column.
        table.cell(panel, 1, i, str.tostring(ma, format.mintick), text_color =
color.black, bgcolor = bgColor)
        period += masStepInput

```

Note that:

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

Displaying a heatmap

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



```
//@version=5
indicator("Price vs Past", "", true)

var int MAX_LOOKBACK = 300

int    lookBackInput  = input.int(150, minval = 1, maxval = MAX_LOOKBACK, step
= 10)
color  bullColorInput = input.color(#00FF00ff, "Bull", inline = "11")
color  bearColorInput = input.color(#FF0080ff, "Bear", inline = "11")

// ——— Function draws a heatmap showing the position of the current `_src`
relative to its past `_lookBack` values.
drawHeatmap(src, lookBack) =>
    // float src      : evaluated price series.
    // int   lookBack: number of past bars evaluated.
```

```

// Dependency: MAX_LOOKBACK

// Force historical buffer to a sufficient size.
max_bars_back(src, MAX_LOOKBACK)
// Only run table code on last bar.
if barstate.islast
    var heatmap = table.new(position.bottom_center, lookBack, 1)
    for i = 1 to lookBackInput
        float transp = 100. * i / lookBack
        if src > src[i]
            table.cell(heatmap, lookBack - i, 0, bgcolor =
color.new(bullColorInput, transp))
        else
            table.cell(heatmap, lookBack - i, 0, bgcolor =
color.new(bearColorInput, transp))

drawHeatmap(high, lookBackInput)

```

Note that:

- We define a maximum lookback period as a `MAX_LOOKBACK` constant. This is an important value and we use it for two purposes: to specify the number of columns we will create in our one-row table, and to specify the lookback period required for the `_src` argument in our function, so that we force Pine Script® to create a historical buffer size that will allow us to refer to the required quantity of past values of `_src` in our `for` loop.
- We offer users the possibility of configuring the bull/bear colors in the inputs and we use `inline` to place the color selections on the same line.
- Inside our function, we enclose our table-creation code in an `if barstate.islast` construct so that it only runs on the last bar of the chart.
- The initialization of the table is done inside the `if` statement. Because of that, and the fact that it uses the `var` keyword, initialization only occurs the first time the script executes on a last bar. Note that this behavior is different from the usual `var` declarations in the script's global scope, where initialization occurs on the first bar of the dataset, at `bar_index` zero.
- We do not specify an argument to the `text` parameter in our `table.cell()` calls, so an empty string is used.
- We calculate our transparency in such a way that the intensity of the colors decreases as we go further in history.
- We use dynamic color generation to create different transparencies of our base colors as needed.
- Contrary to other objects displayed in Pine scripts, this heatmap's cells are not linked to chart bars. The configured lookback period determines how many table cells the heatmap contains, and the heatmap will not change as the chart is panned horizontally, or scaled.
- The maximum number of cells that can be displayed in the script's visual space will depend on your viewing device's resolution and the portion of the display used by your chart. Higher resolution screens and wider windows will allow more table cells to be displayed.

Tips

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

necessary bars only. This saves server resources and your charts will display faster, so everybody wins.



Text and shapes

- [Introduction](#)
- [`plotchar\(\)`](#)
- [`plotshape\(\)`](#)
- [`plotarrow\(\)`](#)
- [Labels](#)
 - [Creating and modifying labels](#)
 - [Positioning labels](#)
 - [Reading label properties](#)
 - [Cloning labels](#)
 - [Deleting labels](#)
 - [Realtime behavior](#)

Introduction

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

- [plotchar\(\)](#)
- [plotshape\(\)](#)
- [plotarrow\(\)](#)
- Labels created with [label.new\(\)](#)
- Tables created with [table.new\(\)](#) (see [Tables](#))

Which one to use depends on your needs:

- Tables can display text in various relative positions on charts that will not move as users scroll or zoom the chart horizontally. Their content is not tethered to bars. In contrast, text displayed with [plotchar\(\)](#), [plotshape\(\)](#) or [label.new\(\)](#) is always tethered to a specific bar, so it will move with the bar's position on the chart. See the page on [Tables](#) for more information on them.
- Three functions include are able to display pre-defined shapes: [plotshape\(\)](#), [plotarrow\(\)](#) and Labels created with [label.new\(\)](#).
- [plotarrow\(\)](#) cannot display text, only up or down arrows.
- [plotchar\(\)](#) and [plotshape\(\)](#) can display non-dynamic (not of "series" form) text on any bar or all bars of the chart.
- [plotchar\(\)](#) can only display one character while [plotshape\(\)](#) can display strings, including line breaks.
- [label.new\(\)](#) can display a maximum of 500 labels on the chart. Its text **can** contain dynamic text, or "series strings". Line breaks are also supported in label text.
- While [plotchar\(\)](#) and [plotshape\(\)](#) can display text at a fixed offset in the past or the future, which cannot change during the script's execution, each [label.new\(\)](#) call can use a "series" offset that can be calculated on the fly.

These are a few things to keep in mind concerning Pine Script[®] strings:

- Since the `text` parameter in both [plotchar\(\)](#) and [plotshape\(\)](#) require a “const string” argument, it cannot contain values such as prices that can only be known on the bar (“series string”).
- To include “series” values in text displayed using [label.new\(\)](#), they will first need to be converted to strings using [str.tostring\(\)](#).
- The concatenation operator for strings in Pine is `+`. It is used to join string components into one string, e.g., `msg = "Chart symbol: " + syminfo.tickerid` (where [syminfo.tickerid](#) is a built-in variable that returns the chart’s exchange and symbol information in string format).
- Characters displayed by all these functions can be Unicode characters, which may include Unicode symbols. See this [Exploring Unicode](#) script to get an idea of what can be done with Unicode characters.
- The color or size of text can sometimes be controlled using function parameters, but no inline formatting (bold, italics, monospace, etc.) is possible.
- Text from Pine scripts always displays on the chart in the Trebuchet MS font, which is used in many TradingView texts, including this one.

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

```
//@version=5
indicator("Four displays of text", overlay = true)
plotchar(ta.rising(close, 5), "`plotchar()`", "?", location.belowbar,
color.lime, size = size.small)
plotshape(ta.falling(close, 5), "`plotchar()`", location = location.abovebar,
color = na, text = "•`plotshape()`•`n?", textcolor = color.fuchsia, size =
size.huge)

if bar_index % 25 == 0
    label.new(bar_index, na, "•LABEL•`nHigh = " + str.tostring(high,
format.mintick) + "`n?", yloc = yloc.abovebar, style = label.style_none,
textcolor = color.black, size = size.normal)

printTable(txt) => var table t = table.new(position.middle_right, 1, 1),
table.cell(t, 0, 0, txt, bgcolor = color.yellow)
printTable("•TABLE•`n" + str.tostring(bar_index + 1) + " bars`nin the dataset")
```



Note that:

- The method used to display each text string is shown with the text, except for the lime up arrows displayed using [plotchar\(\)](#), as it can only display one character.
- Label and table calls can be inserted in conditional structures to control when they are executed, whereas [plotchar\(\)](#) and [plotshape\(\)](#) cannot. Their conditional plotting must be controlled using their first argument, which is a “series bool” whose `true` or `false` value determines when the text is displayed.
- Numeric values displayed in the table and labels is first converted to a string using [str.tostring\(\)](#).
- We use the `+` operator to concatenate string components.
- [plotshape\(\)](#) is designed to display a shape with accompanying text. Its `size` parameter controls the size of the shape, not of the text. We use [na](#) for its `color` argument so that the shape is not visible.
- Contrary to other texts, the table text will not move as you scroll or scale the chart.
- Some text strings contain the ? Unicode arrow (U+1F807).

- Some text strings contain the `\n` sequence that represents a new line.

`plotchar()`

This function is useful to display a single character on bars. It has the following syntax:

```
plotchar(series, title, char, location, color, offset, text, textcolor,
editable, size, show_last, display) → void
```

See the [Reference Manual entry for plotchar\(\)](#) for details on its parameters.

As explained in the [When the script's scale must be preserved](#) section of our page on [Debugging](#), the function can be used to display and inspect values in the Data Window or in the indicator values displayed to the right of the script's name on the chart:

```
//@version=5
indicator("", "", true)
plotchar(bar_index, "Bar index", "", location.top)
```



Note that:

- The cursor is on the chart's last bar.
- The value of `bar_index` on **that** bar is displayed in indicator values (1) and in the Data Window (2).
- We use `location.top` because the default `location.abovebar` will put the price into play in the script's scale, which will often interfere with other plots.

`plotchar()` also works well to identify specific points on the chart or to validate that conditions are `true` when we expect them to be. This example displays an up arrow under bars where `close`, `high` and `volume` have all been rising for two bars:

```
//@version=5
indicator("", "", true)
bool longSignal = ta.rising(close, 2) and ta.rising(high, 2) and (na(volume) or
ta.rising(volume, 2))
plotchar(longSignal, "Long", "▲", location.belowbar, color = na(volume) ?
color.gray : color.blue, size = size.tiny)
```



Note that:

- We use `(na(volume) or ta.rising(volume, 2))` so our script will work on symbols without `volume` data. If we did not make provisions for when there is no `volume` data, which is what `na(volume)` does by being `true` when there is no volume, the `longSignal` variable's value would never be `true` because `ta.rising(volume, 2)` yields `false` in those cases.
- We display the arrow in gray when there is no volume, to remind us that all three base conditions are not being met.
- Because `plotchar()` is now displaying a character on the chart, we use `size = size.tiny` to control its size.
- We have adapted the `location` argument to display the character under bars.

If you don't mind plotting only circles, you could also use `plot()` to achieve a similar effect:

```
//@version=5
indicator("", "", true)
longSignal = ta.rising(close, 2) and ta.rising(high, 2) and (na(volume) or
ta.rising(volume, 2))
plot(longSignal ? low - ta.tr : na, "Long", color.blue, 2, plot.style_circles)
```

This method has the inconvenience that, since there is no relative positioning mechanism with [plot\(\)](#) one must shift the circles down using something like [ta.tr](#) (the bar's "True Range"):



[plotshape\(\)](#)

This function is useful to display pre-defined shapes and/or text on bars. It has the following syntax:

```
plotshape(series, title, style, location, color, offset, text, textcolor,
editable, size, show_last, display) → void
```

See the [Reference Manual entry for plotshape\(\)](#) for details on its parameters.

Let's use the function to achieve more or less the same result as with our second example of the previous section:

```
//@version=5
indicator("", "", true)
longSignal = ta.rising(close, 2) and ta.rising(high, 2) and (na(volume) or
ta.rising(volume, 2))
plotshape(longSignal, "Long", shape.arrowup, location.belowbar)
```

Note that here, rather than using an arrow character, we are using the `shape.arrowup` argument for the `style` parameter.



























It is possible to use different [plotshape\(\)](#) calls to superimpose text on bars. You will need to use `\n` followed by a special non-printing character that doesn't get stripped out to preserve the newline's functionality. Here we're using a Unicode Zero-width space (U+200E). While you don't see it in the following code's strings, it is there and can be copy/pasted. The special Unicode character needs to be the **last** one in the string for text going up, and the **first** one when you are plotting under the bar and text is going down:

```
//@version=5
indicator("Lift text", "", true)
plotshape(true, "", shape.arrowup, location.abovebar, color.green, text =
"A")
plotshape(true, "", shape.arrowup, location.abovebar, color.lime, text =
"B\n")
plotshape(true, "", shape.arrowdown, location.belowbar, color.red, text =
"C")
plotshape(true, "", shape.arrowdown, location.belowbar, color.maroon, text =
"\nD")
```



The available shapes you can use with the `style` parameter are:

Argument	Shape	With Text	Argument	Shape	With Text
shape.xcross		text 	shape.arrowup		text 
shape.cross		text 	shape.arrowdown		text 
shape.circle		text 	shape.square		text 
shape.triangleup		text 	shape.diamond		text 
shape.triangledown		text 	shape.labelup		text 
shape.flag		text 	shape.labeldown		text 

plotarrow()

The [plotarrow](#) function displays up or down arrows of variable length, based on the relative value of the series used in the function's first argument. It has the following syntax:

```
plotarrow(series, title, colorup, colordown, offset, minheight, maxheight,
editable, show_last, display) → void
```

See the [Reference Manual entry for plotarrow\(\)](#) for details on its parameters.

The `series` parameter in [plotarrow\(\)](#) is not a “series bool” as in [plotchar\(\)](#) and [plotsshape\(\)](#); it is a “series int/float” and there's more to it than a simple `true` or `false` value determining when the arrows are plotted. This is the logic governing how the argument supplied to `series` affects the behavior of [plotarrow\(\)](#):

- `series > 0`: An up arrow is displayed, the length of which will be proportional to the relative value of the series on that bar in relation to other series values.
- `series < 0`: A down arrow is displayed, proportionally-sized using the same rules.
- `series == 0` or `na(series)`: No arrow is displayed.

The maximum and minimum possible sizes for the arrows (in pixels) can be controlled using the `minheight` and `maxheight` parameters.

Here is a simple script illustrating how [plotarrow\(\)](#) works:

```
//@version=5
indicator("", "", true)
body = close - open
plotarrow(body, colorup = color.teal, colordown = color.orange)
```



Note how the height of arrows is proportional to the relative size of the bar bodies.

You can use any series to plot the arrows. Here we use the value of the “Chaikin Oscillator” to control the location and size of the arrows:

```
//@version=5
indicator("Chaikin Oscillator Arrows", overlay = true)
fastLengthInput = input.int(3, minval = 1)
slowLengthInput = input.int(10, minval = 1)
osc = ta.ema(ta.accdist, fastLengthInput) - ta.ema(ta.accdist, slowLengthInput)
plotarrow(osc)
```



Note that we display the actual “Chaikin Oscillator” in a pane below the chart, so you can see what values are used to determine the position and size of the arrows.

Labels

Labels are only available in v4 and higher versions of Pine Script[®]. They work very differently than [plotchar\(\)](#) and [plotshape\(\)](#).

Labels are objects, like [lines and boxes](#), or [tables](#). Like them, they are referred to using an ID, which acts like a pointer. Label IDs are of “label” type. As with other objects, labels IDs are “time series” and all the functions used to manage them accept “series” arguments, which makes them very flexible.

Note

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

Labels are advantageous because:

- They allow “series” values to be converted to text and placed on charts. This means they are ideal to display values that cannot be known before time, such as price values, support and resistance levels, of any other values that your script calculates.
- Their positioning options are more flexible than those of the `plot*()` functions.
- They offer more display modes.
- Contrary to `plot*()` functions, label-handling functions can be inserted in conditional or loop structures, making it easier to control their behavior.
- You can add tooltips to labels.

One drawback to using labels versus [plotchar\(\)](#) and [plotshape\(\)](#) is that you can only draw a limited quantity of them on the chart. The default is ~50, but you can use the `max_labels_count` parameter in your [indicator\(\)](#) or [strategy\(\)](#) declaration statement to specify up to 500. Labels, like [lines and boxes](#), are managed using a garbage collection mechanism which deletes the oldest ones on the chart, such that only the most recently drawn labels are visible.

Your toolbox of built-ins to manage labels are all in the `label` namespace. They include:

- [label.new\(\)](#) to create labels.
- `label.set_*()` functions to modify the properties of an existing label.
- `label.get_*()` functions to read the properties of an existing label.
- [label.delete\(\)](#) to delete labels
- The [label.all](#) array which always contains the IDs of all the visible labels on the chart. The array's size will depend on the maximum label count for your script and how many of those you have drawn. `array.size(label.all)` will return the array's size.

Creating and modifying labels

The [label.new\(\)](#) function creates a new label. It has the following signature:

```
label.new(x, y, text, xloc, yloc, color, style, textcolor, size, textalign,
tooltip) → series label
```

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

- [label.set_x\(\)](#)
- [label.set_y\(\)](#)
- [label.set_xy\(\)](#)
- [label.set_text\(\)](#)
- [label.set_xloc\(\)](#)
- [label.set_yloc\(\)](#)
- [label.set_color\(\)](#)
- [label.set_style\(\)](#)
- [label.set_textcolor\(\)](#)
- [label.set_size\(\)](#)
- [label.set_textalign\(\)](#)
- [label.set_tooltip\(\)](#)

They all have a similar signature. The one for [label.set_color\(\)](#) is:

```
label.set_color(id, color) → void
```

where:

- `id` is the ID of the label whose property is to be modified.
- The next parameter is the property of the label to modify. It depends on the setter function used. [label.set_xy\(\)](#) changes two properties, so it has two such parameters.

This is how you can create labels in their simplest form:

```
//@version=5
indicator("", "", true)
label.new(bar_index, high)
```



Note that:

- The label is created with the parameters `x = bar_index` (the index of the current bar, [bar_index](#)) and `y = high` (the bar's [high](#) value).
- We do not supply an argument for the function's `text` parameter. Its default value being an empty string, no text is displayed.
- No logic controls our [label.new\(\)](#) call, so labels are created on every bar.

- Only the last 54 labels are displayed because our [indicator\(\)](#) call does not use the `max_labels_count` parameter to specify a value other than the ~50 default.
- Labels persist on bars until your script deletes them using [label.delete\(\)](#), or garbage collection removes them.

In the next example we display a label on the bar with the highest [high](#) value in the last 50 bars:

```
//@version=5
indicator("", "", true)

// Find the highest `high` in last 50 bars and its offset. Change it's sign so
it is positive.
LOOKBACK = 50
hi = ta.highest(LOOKBACK)
highestBarOffset = - ta.highestbars(LOOKBACK)

// Create label on bar zero only.
var lbl = label.new(na, na, "", color = color.orange, style =
label.style_label_lower_left)
// When a new high is found, move the label there and update its text and
tooltip.
if ta.change(hi)
    // Build label and tooltip strings.
    labelText = "High: " + str.tostring(hi, format.mintick)
    tooltipText = "Offest in bars: " + str.tostring(highestBarOffset) + "\nLow:
" + str.tostring(low[highestBarOffset], format.mintick)
    // Update the label's position, text and tooltip.
    label.set_xy(lbl, bar_index[highestBarOffset], hi)
    label.set_text(lbl, labelText)
    label.set_tooltip(lbl, tooltipText)
```



Note that:

- We create the label on the first bar only by using the [var](#) keyword to declare the `lbl` variable that contains the label's ID. The `x`, `y` and `text` arguments in that [label.new\(\)](#) call are irrelevant, as the label will be updated on further bars. We do, however, take care to use the `color` and `style` we want for the labels, so they don't need updating later.
- On every bar, we detect if a new high was found by testing for changes in the value of `hi`
- When a change in the high value occurs, we update our label with new information. To do this, we use three `label.set*()` calls to change the label's relevant information. We refer to our label using the `lbl` variable, which contains our label's ID. The script is thus maintaining the same label throughout all bars, but moving it and updating its information when a new high is detected.

Here we create a label on each bar, but we set its properties conditionally, depending on the bar's polarity:

```
//@version=5
indicator("", "", true)
lbl = label.new(bar_index, na)
if close >= open
    label.set_text( lbl, "green")
    label.set_color(lbl, color.green)
    label.set_yloc( lbl, yloc.belowbar)
    label.set_style(lbl, label.style_label_up)
else
    label.set_text( lbl, "red")
    label.set_color(lbl, color.red)
```

```
label.set_yloc( lbl, yloc.abovebar)
label.set_style(lbl, label.style_label_down)
```



Positioning labels

Labels are positioned on the chart according to x (bars) and y (price) coordinates. Five parameters affect this behavior: x , y , $xloc$, $yloc$ and $style$:

x

Is either a bar index or a time value. When a bar index is used, the value can be offset in the past or in the future (maximum of 500 bars in the future). Past or future offsets can also be calculated when using time values. The x value of an existing label can be modified using [label.set_x\(\)](#) or [label.set_xy\(\)](#).

$xloc$

Is either [xloc.bar_index](#) (the default) or [xloc.bar_time](#). It determines which type of argument must be used with x . With [xloc.bar_index](#), x must be an absolute bar index. With [xloc.bar_time](#), x must be a UNIX time in milliseconds corresponding to the [time](#) value of a bar's [open](#). The $xloc$ value of an existing label can be modified using [label.set_xloc\(\)](#).

y

Is the price level where the label is positioned. It is only taken into account with the default $yloc$ value of [yloc.price](#). If $yloc$ is [yloc.abovebar](#) or [yloc.belowbar](#) then the y argument is ignored. The y value of an existing label can be modified using [label.set_y\(\)](#) or [label.set_xy\(\)](#).

$yloc$



























Can be [yloc.price](#) (the default), [yloc.abovebar](#) or [yloc.belowbar](#). The argument used for y is only taken into account with [yloc.price](#). The $yloc$ value of an existing label can be modified using [label.set_yloc\(\)](#).

$style$

The argument used has an impact on the visual appearance of the label and on its position relative to the reference point determined by either the y value or the top/bottom of the bar when [yloc.abovebar](#) or [yloc.belowbar](#) are used. The $style$ of an existing label can be modified using [label.set_style\(\)](#).

These are the available $style$ arguments:

Argument	Label	Label with text	Argument	Label	Label with text
<code>label.style_xcross</code>			<code>label.style_label_up</code>		
<code>label.style_cross</code>			<code>label.style_label_down</code>		
<code>label.style_flag</code>			<code>label.style_label_left</code>		

Argument	Label	Label with text	Argument	Label	Label with text
label.style_circle			label.style_label_right		
label.style_square			label.style_label_lower_left		
label.style_diamond			label.style_label_lower_right		
label.style_triangleup			label.style_label_upper_left		
label.style_triangledown			label.style_label_upper_right		
label.style_arrowup			label.style_label_center		
label.style_arrowdown			label.style_none		text

When using [xloc.bar_time](#), the x value must be a UNIX timestamp in milliseconds. See the page on [Time](#) for more information. The start time of the current bar can be obtained from the [time](#) built-in variable. The bar time of previous bars is `time[1]`, `time[2]` and so on. Time can also be set to an absolute value with the [timestamp](#) function. You may add or subtract periods of time to achieve relative time offset.

Let's position a label one day ago from the date on the last bar:

```
//@version=5
indicator("")
daysAgoInput = input.int(1, tooltip = "Use negative values to offset in the future")
if barstate.islast
    MS_IN_ONE_DAY = 24 * 60 * 60 * 1000
    oneDayAgo = time - (daysAgoInput * MS_IN_ONE_DAY)
    label.new(oneDayAgo, high, xloc = xloc.bar_time, style =
label.style_label_right)
```

Note that because of varying time gaps and missing bars when markets are closed, the positioning of the label may not always be exact. Time offsets of the sort tend to be more reliable on 24x7 markets.

You can also offset using a bar index for the x value, e.g.:

```
label.new(bar_index + 10, high)
label.new(bar_index - 10, high[10])
label.new(bar_index[10], high[10])
```

Reading label properties

The following *getter* functions are available for labels:

- [label.get_x\(\)](#)
- [label.get_y\(\)](#)
- [label.get_text\(\)](#)

They all have a similar signature. The one for [label.get_text\(\)](#) is:

```
label.get_text(id) → series string
```

where `id` is the label whose text is to be retrieved.

Cloning labels

The [label.copy\(\)](#) function is used to clone labels. Its syntax is:

```
label.copy(id) → void
```

Deleting labels

The [label.delete\(\)](#) function is used to delete labels. Its syntax is:

```
label.delete(id) → void
```

To keep only a user-defined quantity of labels on the chart, one could use code like this:

```
//@version=5
MAX_LABELS = 500
indicator("", max_labels_count = MAX_LABELS)
qtyLabelsInput = input.int(5, "Labels to keep", minval = 0, maxval = MAX_LABELS)
myRSI = ta.rsi(close, 20)
if myRSI > ta.highest(myRSI, 20)[1]
    label.new(bar_index, myRSI, str.toString(myRSI, "%.00"), style =
label.style_none)
    if array.size(label.all) > qtyLabelsInput
        label.delete(array.get(label.all, 0))
plot(myRSI)
```



Note that:

- We define a `MAX_LABELS` constant to hold the maximum quantity of labels a script can accommodate. We use that value to set the `max_labels_count` parameter's value in our [indicator\(\)](#) call, and also as the `maxval` value in our [input.int\(\)](#) call to cap the user value.
- We create a new label when our RSI breaches its highest value of the last 20 bars. Note the offset of `[1]` we use in `if myRSI > ta.highest(myRSI, 20)[1]`. This is necessary. Without it, the value returned by [ta.highest\(\)](#) would always include the current value of `myRSI`, so `myRSI` would never be higher than the function's return value.
- After that, we delete the oldest label in the [label.all](#) array that is automatically maintained by the Pine Script® runtime and contains the ID of all the visible labels drawn by our script. We use the [array.get\(\)](#) function to retrieve the array element at index zero (the oldest visible label ID). We then use [label.delete\(\)](#) to delete the label linked with that ID.

Note that if one wants to position a label on the last bar only, it is unnecessary and inefficient to

create and delete the label as the script executes on all bars, so that only the last label remains:

```
// INEFFICIENT!
//@version=5
indicator("", "", true)
lbl = label.new(bar_index, high, str.tostring(high, format.mintick))
label.delete(lbl[1])
```

This is the efficient way to realize the same task:

```
//@version=5
indicator("", "", true)
if barstate.islast
    // Create the label once, the first time the block executes on the last bar.
    var lbl = label.new(na, na)
    // On all iterations of the script on the last bar, update the label's
information.
    label.set_xy(lbl, bar_index, high)
    label.set_text(lbl, str.tostring(high, format.mintick))
```

Realtime behavior

Labels are subject to both *commit* and *rollback* actions, which affect the behavior of a script when it executes in the realtime bar. See the page on Pine Script®'s Execution model.

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

```
//@version=5
indicator("", "", true)
label.new(bar_index, high)
```

On realtime bars, [label.new\(\)](#) creates a new label on every script update, but because of the rollback process, the label created on the previous update on the same bar is deleted. Only the last label created before the realtime bar's close will be committed, and thus persist.



Time

- [Introduction](#)
 - [Four references](#)
 - [Time built-ins](#)
 - [Time zones](#)
 - [Time zone strings](#)
- [Time variables](#)
 - [`time` and `time_close`](#)
 - [`time_tradingday`](#)
 - [`timenow`](#)
 - [Calendar dates and times](#)
 - [`syminfo.timezone\(\)`](#)
- [Time functions](#)
 - [`time\(\)` and `time_close\(\)`](#)
 - [Testing for sessions](#)
 - [Testing for changes in higher timeframes](#)
 - [Calendar dates and times](#)

- ``timestamp()``
- [Formatting dates and time](#)

Introduction

Four references

Four different references come into play when using date and time values in Pine Script®:

1. **UTC time zone:** The native format for time values in Pine Script® is the **Unix time in milliseconds**. Unix time is the time elapsed since the **Unix Epoch on January 1st, 1970**. See here for the [current Unix time in seconds](#) and here for more information on [Unix Time](#). A value for the Unix time is called a *timestamp*. Unix timestamps are always expressed in the UTC (or “GMT”, or “GMT+0”) time zone. They are measured from a fixed reference, i.e., the Unix Epoch, and do not vary with time zones. Some built-ins use the UTC time zone as a reference.
2. **Exchange time zone:** A second time-related key reference for traders is the time zone of the exchange where an instrument is traded. Some built-ins like [hour](#) return values in the exchange’s time zone by default.
3. `timezone` parameter: Some functions that normally return values in the exchange’s time zone, such as [hour\(\)](#) include a `timezone` parameter that allows you to adapt the function’s result to another time zone. Other functions like [time\(\)](#) include both `session` and `timezone` parameters. In those cases, the `timezone` argument applies to how the `session` argument is interpreted — not to the time value returned by the function.
4. **Chart’s time zone:** This is the time zone chosen by the user from the chart using the “Chart Settings/Symbol/Time Zone” field. This setting only affects the display of dates and times on the chart. It does not affect the behavior of Pine scripts, and they have no visibility over this setting.

When discussing variables or functions, we will note if they return dates or times in UTC or exchange time zone. Scripts do not have visibility on the user’s time zone setting on his chart.

Time built-ins

Pine Script® has built-in **variables** to:

- Get timestamp information from the current bar (UTC time zone): [time](#) and [time_close](#)
- Get timestamp information for the beginning of the current trading day (UTC time zone): [time_tradingday](#)
- Get the current time in one-second increments (UTC time zone): [timenow](#)
- Retrieve calendar and time values from the bar (exchange time zone): [year](#), [month](#), [weekofyear](#), [dayofmonth](#), [dayofweek](#), [hour](#), [minute](#) and [second](#)
- Return the time zone of the exchange of the chart’s symbol with [syminfo.timezone](#)

There are also built-in **functions** that can:

- Return timestamps of bars from other timeframes with [time\(\)](#) and [time_close\(\)](#), without the need for a [request.security\(\)](#) call
- Retrieve calendar and time values from any timestamp, which can be offset with a time zone: [year\(\)](#), [month\(\)](#), [weekofyear\(\)](#), [dayofmonth\(\)](#), [dayofweek\(\)](#), [hour\(\)](#), [minute\(\)](#) and [second\(\)](#)
- Create a timestamp using [timestamp\(\)](#)
- Convert a timestamp to a formatted date/time string for display, using [str.format\(\)](#)

- Input data and time values. See the section on [Inputs](#).
- Work with [session information](#).

Time zones

TradingView users can change the time zone used to display bar times on their charts. Pine scripts have no visibility over this setting. While there is a [syminfo.timezone](#) variable to return the time zone of the exchange where the chart's instrument is traded, there is **no** `chart.timezone` equivalent.

When displaying times on the chart, this shows one way of providing users a way of adjusting your script's time values to those of their chart. This way, your displayed times can match the time zone used by traders on their chart:



```
//@version=5
indicator("Time zone control")
MS_IN_1H = 1000 * 60 * 60
TOOLTIP01 = "Enter your time zone's offset (+ or -), including a decimal
fraction if needed."
hoursOffsetInput = input.float(0.0, "Timezone offset (in hours)", minval =
-12.0, maxval = 14.0, step = 0.5, tooltip = TOOLTIP01)

printTable(txt) =>
    var table t = table.new(position.middle_right, 1, 1)
    table.cell(t, 0, 0, txt, text_halign = text.align_right, bgcolor =
color.yellow)

msOffsetInput = hoursOffsetInput * MS_IN_1H
printTable(
    str.format("Last bar's open time UTC: {0,date,HH:mm:ss yyyy.MM.dd}", time) +
    str.format("\nLast bar's close time UTC: {0,date,HH:mm:ss yyyy.MM.dd}",
time_close) +
    str.format("\n\nLast bar's open time EXCHANGE: {0,date,HH:mm:ss yyyy.MM.dd}",
time(timeframe.period, syminfo.session, syminfo.timezone)) +
    str.format("\nLast bar's close time EXCHANGE: {0,date,HH:mm:ss yyyy.MM.dd}",
time_close(timeframe.period, syminfo.session, syminfo.timezone)) +
    str.format("\n\nLast bar's open time OFFSET ({0}): {1,date,HH:mm:ss
yyyy.MM.dd}", hoursOffsetInput, time + msOffsetInput) +
    str.format("\nLast bar's close time OFFSET ({0}): {1,date,HH:mm:ss
yyyy.MM.dd}", hoursOffsetInput, time_close + msOffsetInput) +
    str.format("\n\nCurrent time OFFSET ({0}): {1,date,HH:mm:ss yyyy.MM.dd}",
hoursOffsetInput, timenow + msOffsetInput))
```

Note that:

- We convert the user offset expressed in hours to milliseconds with `msOffsetInput`. We then add that offset to a timestamp in UTC format before converting it to display format, e.g., `time + msOffsetInput` and `timenow + msOffsetInput`.
- We use a tooltip to provide instructions to users.
- We provide `minval` and `maxval` values to protect the input field, and a `step` value of 0.5 so that when they use the field's up/down arrows, they can intuitively figure out that fractions can be used.
- The [str.format\(\)](#) function formats our time values, namely the last bar's time and the current time.

Some functions that normally return values in the exchange's time zone provide means to adapt

their result to another time zone through the `timezone` parameter. This script illustrates how to do this with `hour()`:



```
//@version=5
indicator('`hour(time, "GMT+0")` in orange')
color BLUE_LIGHT = #0000FF30
plot(hour, "", BLUE_LIGHT, 8)
plot(hour(time, syminfo.timezone))
plot(hour(time, "GMT+0"), "UTC", color.orange)
```

Note that:

- The `hour` variable and the `hour()` function normally returns a value in the exchange's time zone. Accordingly, plots in blue for both `hour` and `hour(time, syminfo.timezone)` overlap. Using the function form with `syminfo.timezone` is thus redundant if the exchange's hour is required.
- The orange line plotting `hour(time, "GMT+0")`, however, returns the bar's hour at UTC, or "GMT+0" time, which in this case is four hours less than the exchange's time, since MSFT trades on the NASDAQ whose time zone is UTC-4.

Time zone strings

The argument used for the `timezone` parameter in functions such as `time()`, `timestamp()`, `hour()`, etc., can be in different formats, which you can find in the [IANA time zone database name](#) reference page. Contents from the "TZ database name", "UTC offset ±hh:mm" and "UTC DST offset ±hh:mm" columns of that page's table can be used.

To express an offset of +5.5 hours from UTC, these strings found in the reference page are all equivalent:

- "GMT+05:30"
- "Asia/Calcutta"
- "Asia/Colombo"
- "Asia/Kolkata"

Non-fractional offsets can be expressed in the "GMT+5" form. "GMT+5.5" is not allowed.

Time variables

`time` and `time_close`

Let's start by plotting `time` and `time_close`, the Unix timestamp in milliseconds of the bar's opening and closing time:



```
//@version=5
indicator("`time` and `time_close` values on bars")
plot(time, "`time`")
plot(time_close, "`time_close`")
```

Note that:

- The [time](#) and [time_close](#) variables returns a timestamp in [UNIX time](#), which is independent of the timezone selected by the user on his chart. In this case, the **chart's** time zone setting is the exchange time zone, so whatever symbol is on the chart, its exchange time zone will be used to display the date and time values on the chart's cursor. The NASDAQ's time zone is UTC-4, but this only affects the chart's display of date/time values; it does not impact the values plotted by the script.
- The last [time](#) value for the plot shown in the scale is the number of milliseconds elapsed from 00:00:00 UTC, 1 January, 1970, until the bar's opening time. It corresponds to 17:30 on the 27th of September 2021. However, because the chart uses the UTC-4 time zone (the NASDAQ's time zone), it displays the 13:30 time, four hours earlier than UTC time.
- The difference between the two values on the last bar is the number of milliseconds in one hour ($1000 * 60 * 60 = 3,600,000$) because we are on a 1H chart.

[`time_tradingday`](#)

[time_tradingday](#) is useful when a symbol trades on overnight sessions that start and close on different calendar days. For example, this happens in forex markets where a session can open Sunday at 17:00 and close Monday at 17:00.

The variable returns the time of the beginning of the trading day in [UNIX time](#) when used at timeframes of 1D and less. When used on timeframes higher than 1D, it returns the starting time of the last trading day in the bar (e.g., at 1W, it will return the starting time of the last trading day of the week).

[`timenow`](#)

[timenow](#) returns the current time in [UNIX time](#). It works in realtime, but also when a script executes on historical bars. In realtime, your scripts will only perceive changes when they execute on feed updates. When no updates occur, the script is idle, so it cannot update its display. See the page on Pine Script[®]'s [execution model](#) for more information.

This script uses the values of [timenow](#) and [time_close](#) to calculate a realtime countdown for intraday bars. Contrary to the countdown on the chart, this one will only update when a feed update causes the script to execute another iteration:

```
//@version=5
indicator("", "", true)

printTable(txt) =>
    var table t = table.new(position.middle_right, 1, 1)
    table.cell(t, 0, 0, txt, text_halign = text.align_right, bgcolor =
color.yellow)

printTable(str.format("{0,time,HH:mm:ss.SSS}", time_close - timenow))
```

[Calendar dates and times](#)

Calendar date and time variables such as [year](#), [month](#), [weekofyear](#), [dayofmonth](#), [dayofweek](#), [hour](#), [minute](#) and [second](#) can be useful to test for specific dates or times, and as arguments to [timestamp\(\)](#).

When testing for specific dates or times, ones needs to account for the possibility that the script will be executing on timeframes where the tested condition cannot be detected, or for cases where a bar with the specific requirement will not exist. Suppose, for example, we wanted to detect the first trading day of the month. This script shows how using only [dayofmonth](#) will not work when a weekly chart is used or when no trading occurs on the 1st of the month:



```
//@version=5
indicator("", "", true)
firstDayIncorrect = dayofmonth == 1
firstDay = ta.change(time("M"))
plotchar(firstDayIncorrect, "firstDayIncorrect", "•", location.top, size =
size.small)
bgcolor(firstDay ? color.silver : na)
```

Note that:

- Using `ta.change(time("M"))` is more robust as it works on all months (#1 and #2), displayed as the silver background, whereas the blue dot detected using `dayofmonth == 1` does not work (#1) when the first trading day of September occurs on the 2nd.
- The `dayofmonth == 1` condition will be `true` on all intrabars of the first day of the month, but `ta.change(time("M"))` will only be `true` on the first.

If you wanted your script to only display for years 2020 and later, you could use:

```
//@version=5
indicator("", "", true)
plot(year >= 2020 ? close : na, linewidth = 3)
```

[`syminfo.timezone\(\)`](#)

[`syminfo.timezone`](#) returns the time zone of the chart symbol's exchange. It can be helpful when a `timezone` parameter is available in a function, and you want to mention that you are using the exchange's timezone explicitly. It is usually redundant because when no argument is supplied to `timezone`, the exchange's time zone is assumed.

[Time functions](#)

[`time\(\)`](#) and [`time_close\(\)`](#)

The [`time\(\)`](#) and [`time_close\(\)`](#) functions have the following signature:

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

They accept three arguments:

`timeframe`

A string in [`timeframe.period`](#) format.

`session`

An optional string in session specification format: `"hhmm-hhmm[:days]"`, where the `[:days]` part is optional. See the page on [sessions](#) for more information.

`timezone`

An optional value that qualifies the argument for `session` when one is used.

See the [`time\(\)`](#) and [`time_close\(\)`](#) entries in the Reference Manual for more information.

The [`time\(\)`](#) function is most often used to:

1. Test if a bar is in a specific time period, which will require using the `session` parameter.

In those cases, `timeframe.period`, i.e., the chart's timeframe, will often be used for the first parameter. When using the function this way, we rely on the fact that it will return [na](#) when the bar is not part of the period specified in the `session` argument.

2. Detecting changes in higher timeframes than the chart's by using the higher timeframe for the `timeframe` argument. When using the function for this purpose, we are looking for changes in the returned value, which means the higher timeframe bar has changed. This will usually require using [ta.change\(\)](#) to test, e.g., `ta.change(time("D"))` will return the change in time when a new higher timeframe bar comes in, so the expression's result will cast to a "bool" value when used in a conditional expression. The "bool" result will be `true` when there is a change and `false` when there is no change.

Testing for sessions

Let's look at an example of the first case where we want to determine if a bar's starting time is part of a period between 11:00 and 13:00:



```
//@version=5
indicator("Session bars", "", true)
inSession = not na(time(timeframe.period, "1100-1300"))
bgcolor(inSession ? color.silver : na)
```

Note that:

- We use `time(timeframe.period, "1100-1300")`, which says: "Check the chart's timeframe if the current bar's opening time is between 11:00 and 13:00 inclusively". The function returns its opening time if the bar is in the session. If it is **not**, the function returns [na](#).
- We are interested in identifying the instances when [time\(\)](#) does not return [na](#) because that means the bar is in the session, so we test for `not na(...)`. We do not use the actual return value of [time\(\)](#) when it is not [na](#); we are only interested in whether it returns [na](#) or not.

Testing for changes in higher timeframes

It is often helpful to detect changes in a higher timeframe. For example, you may want to detect trading day changes while on intraday charts. For these cases, you can use the fact that `time("D")` returns the opening time of the 1D bar, even if the chart is at an intraday timeframe such as 1H:



```
//@version=5
indicator("", "", true)
bool newDay = ta.change(time("D"))
bgcolor(newDay ? color.silver : na)

newExchangeDay = ta.change(dayofmonth)
plotchar(newExchangeDay, "newExchangeDay", "?", location.top, size = size.small)
```

Note that:

- The `newDay` variable detects changes in the opening time of 1D bars, so it follows the conventions for the chart's symbol, which uses overnight sessions of 17:00 to 17:00. It changes values when a new session comes in.

- Because `newExchangeDay` detects change in [dayofmonth](#) in the calendar day, it changes when the day changes on the chart.
- The two change detection methods only coincide on the chart when there are days without trading. On Sundays here, for example, both detection methods will detect a change because the calendar day changes from the last trading day (Friday) to the first calendar day of the new week, Sunday, which is when Monday's overnight session begins at 17:00.

Calendar dates and times

Calendar date and time functions such as [year\(\)](#), [month\(\)](#), [weekofyear\(\)](#), [dayofmonth\(\)](#), [dayofweek\(\)](#), [hour\(\)](#), [minute\(\)](#) and [second\(\)](#) can be useful to test for specific dates or times. They all have signatures similar to the ones shown here for [dayofmonth\(\)](#):

```
dayofmonth(time) → series int
dayofmonth(time, timezone) → series int
```

This will plot the day of the opening of the bar where the January 1st, 2021 at 00:00 time falls between its [time](#) and [time_close](#) values:

```
//@version=5
indicator("")
exchangeDay = dayofmonth(timestamp("2021-01-01"))
plot(exchangeDay)
```

The value will be the 31st or the 1st, depending on the calendar day of when the session opens on the chart's symbol. The date for symbols traded 24x7 at exchanges using the UTC time zone will be the 1st. For symbols trading on exchanges at UTC-4, the date will be the 31st.

`timestamp()`

The [timestamp\(\)](#) function has a few different signatures:

```
timestamp(year, month, day, hour, minute, second) → simple/series int
timestamp(timezone, year, month, day, hour, minute, second) → simple/series int
timestamp(dateString) → const int
```

The only difference between the first two is the `timezone` parameter. Its default value is [syminfo.timezone](#). See the [Time zone strings](#) section of this page for valid values.

The third form is used as a default value in [input.time\(\)](#). See the [timestamp\(\)](#) entry in the Reference Manual for more information.

[timestamp\(\)](#) is useful to generate a timestamp for a specific date. To generate a timestamp for Jan 1, 2021, use either one of these methods:

```
//@version=5
indicator("")
yearBeginning1 = timestamp("2021-01-01")
yearBeginning2 = timestamp(2021, 1, 1, 0, 0)
printTable(txt) => var table t = table.new(position.middle_right, 1, 1),
table.cell(t, 0, 0, txt, bgcolor = color.yellow)
printTable(str.format("yearBeginning1: {0,date,yyyy.MM.dd
hh:mm}\nyearBeginning2: {1,date,yyyy.MM.dd hh:mm}", yearBeginning1,
yearBeginning2))
```

You can use offsets in [timestamp\(\)](#) arguments. Here, we subtract 2 from the value supplied for its day parameter to get the date/time from the chart's last bar two days ago. Note that because of different bar alignments on various instruments, the bar identified on the chart may not always be exactly 48 hours away, although the function's return value is correct:

```
//@version=5
indicator("")
twoDaysAgo = timestamp(year, month, dayofmonth - 2, hour, minute)
printTable(txt) => var table t = table.new(position.middle_right, 1, 1),
table.cell(t, 0, 0, txt, bgcolor = color.yellow)
printTable(str.format("{0,date,yyyy.MM.dd hh:mm}", twoDaysAgo))
```

Formatting dates and time

Timestamps can be formatted using [str.format\(\)](#). These are examples of various formats:



```
//@version=5
indicator("", "", true)

print(txt, styl) =>
    var alignment = styl == label.style_label_right ? text.align_right :
text.align_left
    var lbl = label.new(na, na, "", xloc.bar_index, yloc.price, color(na), styl,
color.black, size.large, alignment)
    if barstate.islast
        label.set_xy(lbl, bar_index, hl2[1])
        label.set_text(lbl, txt)

var string format =
    "{0,date,yyyy.MM.dd hh:mm:ss}\n" +
    "{1,date,short}\n" +
    "{2,date,medium}\n" +
    "{3,date,long}\n" +
    "{4,date,full}\n" +
    "{5,date,h a z (zzzz)}\n" +
    "{6,time,short}\n" +
    "{7,time,medium}\n" +
    "{8,date,'Month 'MM, 'Week' ww, 'Day 'DD}\n" +
    "{9,time,full}\n" +
    "{10,time,hh:mm:ss}\n" +
    "{11,time,HH:mm:ss}\n" +
    "{12,time,HH:mm:ss} Left in bar\n"

print(format, label.style_label_right)
print(str.format(format,
    time, time, time, time, time, time, time,
    timenow, timenow, timenow, timenow,
    timenow - time, time_close - timenow), label.style_label_left)
```

Timeframes

- [Introduction](#)
- [Timeframe string specifications](#)
- [Comparing timeframes](#)

Introduction

The *timeframe* of a chart is sometimes also referred to as its *interval* or *resolution*. It is the unit of

time represented by one bar on the chart. All standard chart types use a timeframe: “Bars”, “Candles”, “Hollow Candles”, “Line”, “Area” and “Baseline”. One non-standard chart type also uses timeframes: “Heikin Ashi”.

Programmers interested in accessing data from multiple timeframes will need to become familiar with how timeframes are expressed in Pine Script[®], and how to use them.

Timeframe strings come into play in different contexts:

- They must be used in [request.security\(\)](#) when requesting data from another symbol and/or timeframe. See the page on [Other timeframes and data](#) to explore the use of [request.security\(\)](#).
- They can be used as an argument to [time\(\)](#) and [time_close\(\)](#) functions, to return the time of a higher timeframe bar. This, in turn, can be used to detect changes in higher timeframes from the chart’s timeframe without using [request.security\(\)](#). See the [Testing for changes in higher timeframes](#) section to see how to do this.
- The [input.timeframe\(\)](#) function provides a way to allow script users to define a timeframe through a script’s “Inputs” tab (see the [Timeframe input](#) section for more information).
- The [indicator\(\)](#) declaration statement has an optional `timeframe` parameter that can be used to provide multi-timeframe capabilities to simple scripts without using [request.security\(\)](#).
- Many built-in variables provide information on the timeframe used by the chart the script is running on. See the [Chart timeframe](#) section for more information on them, including [timeframe.period](#) which returns a string in Pine Script[®]’s timeframe specification format.

[Timeframe string specifications](#)

Timeframe strings follow these rules:

- They are composed of the multiplier and the timeframe unit, e.g., “1S”, “30” (30 minutes), “1D” (one day), “3M” (three months).
- The unit is represented by a single letter, with no letter used for minutes: “S” for seconds, “D” for days, “W” for weeks and “M” for months.
- When no multiplier is used, 1 is assumed: “S” is equivalent to “1S”, “D” to “1D”, etc. If only “1” is used, it is interpreted as “1min”, since no unit letter identifier is used for minutes.
- There is no “hour” unit; “1H” is **not** valid. The correct format for one hour is “60” (remember no unit letter is specified for minutes).
- The valid multipliers vary for each timeframe unit:
 - For seconds, only the discrete 1, 5, 10, 15 and 30 multipliers are valid.
 - For minutes, 1 to 1440.
 - For days, 1 to 365.
 - For weeks, 1 to 52.
 - For months, 1 to 12.

[Comparing timeframes](#)

It can be useful to compare different timeframe strings to determine, for example, if the timeframe used on the chart is lower than the higher timeframes used in the script, as using timeframes lower than the chart is usually not a good idea. See the [Requesting data of a lower timeframe](#) section for more information on the subject.

Converting timeframe strings to a representation in fractional minutes provides a way to compare them using a universal unit. This script uses the [timeframe.in_seconds\(\)](#) function to convert a timeframe into float seconds and then converts the result into minutes:

```
//@version=5
indicator("Timeframe in minutes example", "", true)
string tfInput = input.timeframe(defval = "", title = "Input TF")

float chartTFInMinutes = timeframe.in_seconds() / 60
float inputTFInMinutes = timeframe.in_seconds(tfInput) / 60

var table t = table.new(position.top_right, 1, 1)
string txt = "Chart TF: " + str.tostring(chartTFInMinutes, "#.##### minutes")
+
"\nInput TF: " + str.tostring(inputTFInMinutes, "#.##### minutes")
if barstate.isfirst
    table.cell(t, 0, 0, txt, bgcolor = color.yellow)
else if barstate.islast
    table.cell_set_text(t, 0, 0, txt)

if chartTFInMinutes > inputTFInMinutes
    runtime.error("The chart's timeframe must not be higher than the input's
timeframe.")
```

Note that:

- We use the built-in [timeframe.in_seconds\(\)](#) function to convert the chart and the [input.timeframe\(\)](#) function into seconds, then divide by 60 to convert into minutes.
- We use two calls to the [timeframe.in_seconds\(\)](#) function in the initialization of the `chartTFInMinutes` and `inputTFInMinutes` variables. In the first instance, we do not supply an argument for its `timeframe` parameter, so the function returns the chart's timeframe in seconds. In the second call, we supply the timeframe selected by the script's user through the call to [input.timeframe\(\)](#).
- Next, we validate the timeframes to ensure that the input timeframe is equal to or higher than the chart's timeframe. If it is not, we generate a runtime error.
- We finally print the two timeframe values converted to minutes.

Style guide

- [Introduction](#)
- [Naming Conventions](#)
- [Script organization](#)
 - [<license>](#)
 - [<version>](#)
 - [<declaration_statement>](#)
 - [<import_statements>](#)
 - [<constant_declarations>](#)
 - [<inputs>](#)
 - [<function_declarations>](#)
 - [<calculations>](#)
 - [<strategy_calls>](#)
 - [<visuals>](#)

- [<alerts>](#)
- [Spacing](#)
- [Line wrapping](#)
- [Vertical alignment](#)
- [Explicit typing](#)

Introduction

This style guide provides recommendations on how to name variables and organize your Pine scripts in a standard way that works well. Scripts that follow our best practices will be easier to read, understand and maintain.

You can see scripts using these guidelines published from the [TradingView](#) and [PineCoders](#) accounts on the platform.

Naming Conventions

We recommend the use of:

- camelCase for all identifiers, i.e., variable or function names: ma, maFast, maLengthInput, maColor, roundedOHLC(), pivotHi().
- All caps SNAKE_CASE for constants: BULL_COLOR, BEAR_COLOR, MAX_LOOKBACK.
- The use of qualifying suffixes when it provides valuable clues about the type or provenance of a variable: maShowInput, bearColor, bearColorInput, volumesArray, maPlotID, resultsTable, levelsColorArray.

Script organization

The Pine Script[®] compiler is quite forgiving of the positioning of specific statements or the version [compiler annotation](#) in the script. While other arrangements are syntactically correct, this is how we recommend organizing scripts:

```
<license>
<version>
<declaration_statement>
<import_statements>
<constant_declarations>
<inputs>
<function_declarations>
<calculations>
<strategy_calls>
<visuals>
<alerts>
```

<license>

If you publish your open-source scripts publicly on TradingView (scripts can also be published privately), your open-source code is by default protected by the Mozilla license. You may choose any other license you prefer.

The reuse of code from those scripts is governed by our [House Rules on Script Publishing](#) which preempt the author's license.

The standard license comments appearing at the beginning of scripts are:

```
// This source code is subject to the terms of the Mozilla Public License 2.0 at
https://mozilla.org/MPL/2.0/
// © username
```

<version>

This is the [compiler annotation](#) defining the version of Pine Script[®] the script will use. If none is present, v1 is used. For v5, use:

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

<declaration_statement>

This is the mandatory declaration statement which defines the type of your script. It must be a call to either [indicator\(\)](#), [strategy\(\)](#), or [library\(\)](#).

<import_statements>

If your script uses one or more Pine Script[®] [libraries](#), your [import](#) statements belong here.

<constant_declarations>

While there is a “constant” form in Pine Script[®], there is no formal “constant” type. We nonetheless use “constant” to denote variables of any type meeting these criteria:

- They are initialized using a literal (e.g., 100 or "AAPL") or a built-in of “const” form (e.g., `color.green`).
- Their value does not change during the script’s execution, meaning their value is never redefined using `:=`.

We use SNAKE_CASE to name these variables and group their declaration near the top of the script. For example:

```
// ----- Constants
int      MS_IN_MIN    = 60 * 1000
int      MS_IN_HOUR  = MS_IN_MIN * 60
int      MS_IN_DAY    = MS_IN_HOUR * 24

color    GRAY         = #808080ff
color    LIME          = #00FF00ff
color    MAROON        = #800000ff
color    ORANGE        = #FF8000ff
color    PINK          = #FF0080ff
color    TEAL          = #008080ff
color    BG_DIV        = color.new(ORANGE, 90)
color    BG_RESETS     = color.new(GRAY, 90)

string   RST1          = "No reset; cumulate since the beginning of the chart"
string   RST2          = "On a stepped higher timeframe (HTF)"
string   RST3          = "On a fixed HTF"
string   RST4          = "At a fixed time"
string   RST5          = "At the beginning of the regular session"
string   RST6          = "At the first visible chart bar"
string   RST7          = "Fixed rolling period"

string   LTF1          = "Least precise, covering many chart bars"
string   LTF2          = "Less precise, covering some chart bars"
string   LTF3          = "More precise, covering less chart bars"
```

```

string LTF4          = "Most precise, 1min intrabars"

string TT_TOTVOL     = "The 'Bodies' value is the transparency of the total
volume candle bodies. Zero is opaque, 100 is transparent."
string TT_RST_HTF    = "This value is used when '" + RST3 +"' is selected."
string TT_RST_TIME   = "These values are used when '" + RST4 +"' is selected.
    A reset will occur when the time is greater or equal to the bar's open time,
and less than its close time.\nHour: 0-23\nMinute: 0-59"
string TT_RST_PERIOD = "This value is used when '" + RST7 +"' is selected."

```

In this example:

- The RST* and LTF* constants will be used as tuple elements in the options argument of input.*() calls.
- The TT_* constants will be used as tooltip arguments in input.*() calls. Note how we use a line continuation for long string literals.
- We do not use [var](#) to initialize constants. The Pine Script® runtime is optimized to handle declarations on each bar, but using [var](#) to initialize a variable only the first time it is declared incurs a minor penalty on script performance because of the maintenance that [var](#) variables require on further bars.

Note that:

- Literals used in more than one place in a script should always be declared as a constant. Using the constant rather than the literal makes it more readable if it is given a meaningful name, and the practice makes code easier to maintain. Even though the quantity of milliseconds in a day is unlikely to change in the future, MS_IN_DAY is more meaningful than 1000 * 60 * 60 * 24.
- Constants only used in the local block of a function or [if](#), [while](#), etc., statement for example, can be declared in that local block.

<inputs>

It is **much** easier to read scripts when all their inputs are in the same code section. Placing that section at the beginning of the script also reflects how they are processed at runtime, i.e., before the rest of the script is executed.

Suffixing input variable names with input makes them more readily identifiable when they are used later in the script: maLengthInput, bearColorInput, showAvgInput, etc.

```

// —— Inputs
string resetInput          = input.string(RST2,          "CVD Resets",
inline = "00", options = [RST1, RST2, RST3, RST4, RST5, RST6, RST7])
string fixedTfInput       = input.timeframe("D",        " Fixed HTF: ",
tooltip = TT_RST_HTF)
int    hourInput          = input.int(9,                " Fixed time
hour: ",          inline = "01", minval = 0, maxval = 23)
int    minuteInput        = input.int(30,              "minute",
inline = "01", minval = 0, maxval = 59, tooltip = TT_RST_TIME)
int    fixedPeriodInput   = input.int(20,              " Fixed
period: ",          inline = "02", minval = 1, tooltip = TT_RST_PERIOD)
string ltfModeInput        = input.string(LTF3,         "Intrabar
precision",        inline = "03", options = [LTF1, LTF2, LTF3, LTF4])

```

<function_declarations>

All user-defined functions must be defined in the script's global scope; nested function definitions

are not allowed in Pine Script®.

Optimal function design should minimize the use of global variables in the function's scope, as they undermine function portability. When it can't be avoided, those functions must follow the global variable declarations in the code, which entails they can't always be placed in the `<function_declarations>` section. Such dependencies on global variables should ideally be documented in the function's comments.

It will also help readers if you document the function's objective, parameters and result. The same syntax used in [libraries](#) can be used to document your functions. This can make it easier to port your functions to a library should you ever decide to do so.

```
//@version=5
indicator("<function_declarations>", "", true)

string SIZE_LARGE = "Large"
string SIZE_NORMAL = "Normal"
string SIZE_SMALL = "Small"

string sizeInput = input.string(SIZE_NORMAL, "Size", options = [SIZE_LARGE,
SIZE_NORMAL, SIZE_SMALL])

// @function      Used to produce an argument for the `size` parameter in
built-in functions.
// @param userSize (simple string) User-selected size.
// @returns       One of the `size.*` built-in constants.
// Dependencies:  SIZE_LARGE, SIZE_NORMAL, SIZE_SMALL
getSize(simple string userSize) =>
    result =
        switch userSize
            SIZE_LARGE => size.large
            SIZE_NORMAL => size.normal
            SIZE_SMALL => size.small
            => size.auto

if ta.rising(close, 3)
    label.new(bar_index, na, yloc = yloc.abovebar, style = label.style_arrowup,
size = getSize(sizeInput))
```

[<calculations>](#)

This is where the script's core calculations and logic should be placed. Code can be easier to read when variable declarations are placed near the code segment using the variables. Some programmers prefer to place all their non-constant variable declarations at the beginning of this section, which is not always possible for all variables, as some may require some calculations to have been executed before their declaration.

[<strategy_calls>](#)

Strategies are easier to read when strategy calls are grouped in the same section of the script.

[<visuals>](#)

This section should ideally include all the statements producing the script's visuals, whether they be plots, drawings, background colors, candle-plotting, etc. See the Pine Script® User Manual's section on [here](#) for more information on how the relative depth of visuals is determined.

<alerts>

Alert code will usually require the script's calculations to have executed before it, so it makes sense to put it at the end of the script.

Spacing

A space should be used on both sides of all operators, except unary operators (-1). A space is also recommended after all commas and when using named function arguments, as in `plot(series = close)`

```
int a = close > open ? 1 : -1
var int newLen = 2
newLen := min(20, newlen + 1)
float a = -b
float c = d > e ? d - e : d
int index = bar_index % 2 == 0 ? 1 : 2
plot(close, color = color.red)
```

Line wrapping

Line wrapping can make long lines easier to read. Line wraps are defined by using an indentation level that is not a multiple of four, as four spaces or a tab are used to define local blocks. Here we use two spaces:

```
plot(
  series = close,
  title = "Close",
  color = color.blue,
  show_last = 10
)
```

Vertical alignment

Vertical alignment using tabs or spaces can be useful in code sections containing many similar lines such as constant declarations or inputs. They can make mass edits much easier using the Pine Script® Editor's multi-cursor feature (`ctrl + alt + ?/?`):

```
// Colors used as defaults in inputs.
color COLOR_AQUA = #0080FFff
color COLOR_BLACK = #000000ff
color COLOR_BLUE = #013BCAff
color COLOR_CORAL = #FF8080ff
color COLOR_GOLD = #CCCC00ff
```

Explicit typing

Including the type of variables when declaring them is not required and is usually overkill for small scripts; we do not systematically use it. It can be useful to make the type of a function's result clearer, and to distinguish a variable's declaration (using `=`) from its reassignments (using `:=`). Using explicit typing can also make it easier for readers to find their way in larger scripts.

Debugging

- [Introduction](#)
- [The lay of the land](#)
- [Displaying numeric values](#)
 - [When the script's scale is unimportant](#)
 - [When the script's scale must be preserved](#)
- [Displaying strings](#)
 - [Labels on each bar](#)
 - [Labels on last bar](#)
- [Debugging conditions](#)
 - [Single conditions](#)
 - [Compound conditions](#)
- [Debugging from inside functions](#)
- [Debugging from inside `for` loops](#)
 - [Extracting a single value](#)
 - [Using lines and labels](#)
 - [Extracting multiple values](#)
- [Tips](#)

Introduction

TradingView's close integration between the Pine Script[®] Editor and charts allows for efficient and interactive debugging of Pine Script[®] code. Once a programmer understands the most appropriate technique to use in each situation, they will be able to debug scripts quickly and thoroughly. This page demonstrates the most useful techniques to debug Pine Script[®] code.

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

The lay of the land

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

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



Note the following in the preceding screenshot:

- The chart's cursor is on the dataset's first bar, where [bar_index](#) is zero. That value is reflected next to the indicator's name and in the Data Window. **Moving your cursor on other bars would update those values so they always represent the value of the plot on that bar.** This is a good way to inspect the value of a variable as the script's execution

- progresses from bar to bar.
- The `title` argument of our `plot()` call, “Bar Index”, is used as the value’s legend in the Data Window.
- The precision of the values displayed in the Data Window is dependent on the chart symbol’s tick value. You can modify it in two ways:
 - By changing the value of the “Precision” field in the script’s “Settings/Style” tab. You can obtain up to eight digits of precision using this method.
 - By using the `precision` parameter in your script’s `indicator()` or `strategy()` declaration statement. This method allows specifying up to 16 digits precision.
- The `plot()` call in our script plots the value of `bar_index` in the indicator’s pane, which shows the increasing value of the variable.
- The scale of the script’s pane is automatically sized to accommodate the smallest and largest values plotted by all `plot()` calls in the script.

Displaying numeric values

When the script’s scale is unimportant

The script in the preceding screenshot used the simplest way to inspect numerical values: a `plot()` call, which plots a line corresponding to the variable’s value in the script’s display area. Our example script plotted the value of the `bar_index` built-in variable, which contains the bar’s number, a value beginning at zero on the dataset’s first bar and increased by one on each subsequent bar. We used a `plot()` call to plot the variable to inspect because our script was not plotting anything else; we were not preoccupied with preserving the scale for other plots to continue to plot normally. This is the script we used:

```
//@version=5
indicator("Plot `bar_index`")
plot(bar_index, "Bar Index")
```

When the script’s scale must be preserved

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

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

```
//@version=5
indicator("Plot RSI and `bar_index`")
r = ta.rsi(close, 20)
plot(r, "RSI", color.black)
plot(bar_index, "Bar Index")
```

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



where:

1. The RSI line in black is flat because it varies between zero and 100, but the indicator’s pane is scaled to show the maximum value of `bar_index`, which is 25692.0000.

2. The value of `bar_index` on the bar the cursor is on is displayed next to the indicator's name, and its blue plot in the script's pane is flat.
3. The 25692.0000 value of `bar_index` shown in the scale represents its value on the last bar, so the dataset contains 25693 bars.
4. The value of `bar_index` on the bar the cursor is on is also displayed in the Data Window, along with that bar's value for RSI just above it.

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

```
//@version=5
indicator("Plot RSI and `bar_index`")
r = ta.rsi(close, 20)
plot(r, "RSI", color.black)
plotchar(bar_index, "Bar index", "", location.top)
```



where:

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

Displaying strings

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

Labels on each bar

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

```
//@version=5
indicator("Simple label", "", true)
label.new(bar_index, high, syminfo.ticker)
```



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

```
indicator("Simple label", "", true, max_labels_count = 500)
```

Labels on last bar

As strings manipulated in Pine scripts often do not change bar to bar, the method most frequently used to visualize them is to draw a label on the dataset's last bar. Here, we use a function to create a

label that only appears on the chart's last bar. Our `f_print()` function has only one parameter, the text string to be displayed:

```
//@version=5
indicator("print()", "", true)
print(txt) =>
    // Create label on the first bar.
    var lbl = label.new(bar_index, na, txt, xloc.bar_index, yloc.price,
color(na), label.style_none, color.gray, size.large, text.align_left)
    // On next bars, update the label's x and y position, and the text it
displays.
    label.set_xy(lbl, bar_index, ta.highest(10)[1])
    label.set_text(lbl, txt)

print("Multiplier = " + str.tostring(timeframe.multiplier) + "\nPeriod = " +
timeframe.period + "\nHigh = " + str.tostring(high))
print("Hello world!\n\n\n\n")
```



Note the following in our last code example:

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

Debugging conditions

Single conditions

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

```
//@version=5
indicator("Single conditions")
r = ta.rsi(close, 20)
rIsLow = r < 30
hline(30)

// Method #1: Change the plot's color.
plot(r, "RSI", rIsLow ? color.fuchsia : color.black)
// Method #2: Plot a character in the bottom region of the display.
plotchar(rIsLow, "rIsLow char at bottom", "▲", location.bottom, size =
size.small)
// Method #3: Plot a character on the RSI line.
plotchar(rIsLow ? r : na, "rIsLow char on line", "•", location.absolute,
```

```

color.red, size = size.small)
// Method #4: Plot a shape in the top region of the display.
plotshape(rIsLow, "rIsLow shape", shape.arrowup, location.top)
// Method #5: Plot an arrow.
plotarrow(rIsLow ? 1 : na, "rIsLow arrow")
// Method #6: Change the background's color.
bgcolor(rIsLow ? color.new(color.green, 90) : na)

```



Note that:

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

Compound conditions

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

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

```

//@version=5
indicator("Compound conditions")
periodInput      = input.int(20)
bullLevelInput   = input.int(55)

r = ta.rsi(close, periodInput)

// Condition #1.

```

```

rsiBull = r > bullLevelInput
// Condition #2.
hiChannel = ta.highest(r, periodInput * 2)[1]
aboveHiChannel = r > hiChannel
// Condition #3.
channelIsOld = hiChannel >= hiChannel[periodInput]
// Condition #4.
historyIsBull = math.sum(rsiBull ? 1 : -1, periodInput * 3) > 0
// Compound condition.
bull = rsiBull and aboveHiChannel and channelIsOld and historyIsBull

hline(bullLevelInput)
plot(r, "RSI", color.black)
plot(hiChannel, "High Channel")

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

```



Note that:

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

Debugging from inside functions

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

```

//@version=5
indicator("Debugging from inside functions", "", true)
hlca() =>
    var float avg = na
    hlca = math.avg(high, low, close, nz(avg, close))
    avg := ta.sma(hlca, 20)

h = hlca()
plot(h)

```

We need to inspect the value of `hlca` in the function's local scope as the function calculates, bar to

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

```
//@version=5
indicator("Debugging from inside functions", "", true)
hlca() =>
    var float avg = na
    instantVal = math.avg(high, low, close, nz(avg, close))
    avg := ta.sma(instantVal, 20)
    // Return two values instead of one.
    [avg, instantVal]

[h, instantVal] = hlca()
plot(h, "h")
plot(instantVal, "instantVal", color.black)
```



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

```
//@version=5
indicator("Debugging from inside functions", "", true)
// Create an array containing only one float element.
instantValGlobal = array.new_float(1)
hlca() =>
    var float avg = na
    instantVal = math.avg(high, low, close, nz(avg, close))
    // Set the array's only element to the current value of `instantVal`.
    array.set(instantValGlobal, 0, instantVal)
    avg := ta.sma(instantVal, 20)

h = hlca()
plot(h, "h")
// Retrieve the value of the array's only element which was set from inside the
function.
plot(array.get(instantValGlobal, 0), "instantValGlobal", color.black)
```

Debugging from inside `for` loops

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

```
//@version=5
indicator("Debugging from inside `for` loops")
lookbackInput = input.int(20, minval = 0)

float trBalance = 0
for i = 1 to lookbackInput
    trBalance := trBalance + math.sign(ta.tr - ta.tr[i])

hline(0)
plot(trBalance)
```

Extracting a single value

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

```
//@version=5
indicator("Debugging from inside `for` loops", max_lines_count = 500,
max_labels_count = 500)
lookbackInput = input.int(20, minval = 0)

float val = na
float trBalance = 0
for i = 1 to lookbackInput
    trBalance := trBalance + math.sign(ta.tr - ta.tr[i])
    if i == lookbackInput
        val := ta.tr[i]
hline(0)
plot(trBalance)
plot(val, "val", color.black)
```

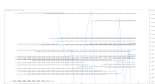


Using lines and labels

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

```
//@version=5
indicator("Debugging from inside `for` loops", max_lines_count = 500,
max_labels_count = 500)
lookbackInput = input.int(20, minval = 0)

float trBalance = 0
for i = 1 to lookbackInput
    trBalance := trBalance + math.sign(ta.tr - ta.tr[i])
    line.new(bar_index[1], ta.tr[i], bar_index, ta.tr[i], color = color.black)
    label.new(bar_index, ta.tr[i], str.tostring(i) + "•" +
str.tostring(ta.tr[i]), style = label.style_none, size = size.small)
hline(0)
plot(trBalance)
```



Note that:

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

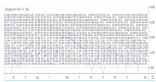
Extracting multiple values

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

```
//@version=5
indicator("Debugging from inside `for` loops", max_lines_count = 500,
max_labels_count = 500)
lookbackInput = input.int(20, minval = 0)

string = ""
float trBalance = 0
for i = 1 to lookbackInput
    trBalance := trBalance + math.sign(ta.tr - ta.tr[i])
    string := string + str.tostring(i, "00") + "." + str.tostring(ta.tr[i]) +
"\n"

label.new(bar_index, 0, string, style = label.style_none, size = size.small,
textalign = text.align_left)
hline(0)
plot(trBalance)
```



Note that:

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

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

```
for i = 1 to i_lookBack
    lowerRangeBalance := lowerRangeBalance + math.sign(ta.tr - ta.tr[i])
    if i % 2 == 0
        string := string + str.tostring(i, "00") + "." + str.tostring(ta.tr[i])
+ "\n"
```

Tips

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

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

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

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

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

```
; —— This is AHK code, not Pine Script®. ——
^+f:: SendInput plotchar(^v, "^v", "", location.top, size = size.tiny){Return}
^+p:: SendInput print(txt) => var lbl = label.new(bar_index, na, txt,
xloc.bar_index, yloc.price, color(na), label.style_none, color.gray, size.large,
text.align_left), label.set_xy(lbl, bar_index, highest(10)[1]),
label.set_text(lbl, txt) `nprint(){Left}
```

The second line will type a debugging [plotchar\(\)](#) call including an expression or variable name previously copied to the clipboard when we use `ctrl + shift + f`. Copying the `variableName` variable name or the `close > open` conditional expression to the clipboard and hitting `ctrl + shift + f` will, respectively, yield:

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

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

```
print(txt) => var lbl = label.new(bar_index, na, txt, xloc.bar_index,
yloc.price, color(na), label.style_none, color.gray, size.large,
text.align_left), label.set_xy(lbl, bar_index, ta.highest(10)[1]),
label.set_text(lbl, txt)
print()
```

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

Publishing scripts

- [Script visibility and access](#)
 - [When you publish a script](#)
 - [Visibility](#)
 - [Public](#)
 - [Private](#)
 - [Access](#)
 - [Open](#)
 - [Protected](#)
 - [Invite-only](#)
- [Preparing a publication](#)
- [Publishing a script](#)
- [Updating a publication](#)

Programmers who wish to share their Pine scripts with other traders can publish them.

Note

If you write scripts for your personal use, there is no need to publish them; you can save them in the Pine Script® Editor and use the “Add to Chart” button to add your script to your chart.

Script visibility and access

When you publish a script, you control its **visibility** and **access**:

- **Visibility** is controlled by choosing to publish **publicly** or **privately**. See [How do private ideas and scripts differ from public ones?](#) in the Help Center for more details. Publish publicly when you have written a script you think can be useful to TradingView users. Public scripts are subject to moderation. To avoid moderation, ensure your publication complies with our [House Rules](#) and [Script Publishing Rules](#). Publish privately when you don't want your script visible to all other users, but want to share it with a few friends.
- **Access** determines if users will see your source code, and how they will be able to use your script. There are three access types: *open*, *protected* (reserved to paid accounts) or *invite-only* (reserved to Premium accounts). See [What are the different types of published scripts?](#) in the Help Center for more details.

When you publish a script

- The publication's title is determined by the argument used for the `title` parameter in the script's `indicator()` or `strategy()` declaration statement. That title is also used when TradingView search for script names.
- The name of your script on the chart will be the argument used for the `shorttitle` parameter in the script's `indicator()` or `strategy()` declaration statement, or the `title` argument in `library()`.
- Your script must have a description explaining what your script does and how to use it.
- The chart you are using when you publish will become visible in your publication, including any other scripts or drawings on it. Remove unrelated scripts or drawings from your chart before publishing your script.
- Your script's code can later be updated. Each update can include *release notes* which will appear, dated, under your original description.
- Scripts can be liked, shared, commented on or reported by other users.
- Your published scripts appear under the "SCRIPTS" tab of your user profile.
- A *script widget* and a *script page* are created for your script. The script widget is your script's placeholder showing in script feeds on the platform. It contains your script's title, chart and the first few lines of your description. When users click on your script **widget**, the script's **page** opens. It contains all the information relating to your script.

Visibility

Public

When you publish a public script:

- Your script will be included in our [Community Scripts](#) where it becomes visible to the millions of TradingView users on all internationalized versions of the site.
- Your publication must comply with [House Rules](#) and [Script Publishing Rules](#).
- If your script is an invite-only script, you must comply with our [Vendor Requirements](#).
- It becomes accessible through the search functions for scripts.
- You will not be able to edit your original description or its title, nor change its public/private visibility, nor its access type (open-source, protected, invite-only).
- You will not be able to delete your publication.

Private

When you publish a private script:

- It will not be visible to other users unless you share its url with them.
- It is visible to you from your user profile's "SCRIPTS" tab.
- Private scripts are identifiable by the "X" and "lock" icons in the top-right of their widget. The "X" is used to delete it.
- It is not moderated, unless you sell access to it or make it available publicly, as it is then no longer "private".
- You can update its original description and title.
- You cannot link to or mentioned it from any public TradingView content (ideas, script descriptions, comments, chats, etc.).
- It is not accessible through the search functions for scripts.

Access

Public or private scripts can be published using one of three access types: open, protected or invite-only. The access type you can select from will vary with the type of account you hold.

Open

The Pine Script[®] code of scripts published **open** is visible to all users. Open-source scripts on TradingView use the Mozilla license by default, but you may choose any license you want. You can find information on licensing at [GitHub](#).

Protected

The code of **protected** scripts is hidden from view and no one but its author can access it. While the script's code is not accessible, protected scripts can be used freely by any user. Only Pro, Pro+ or Premium accounts may publish public protected scripts.

Invite-only

The **invite-only** access type protects both the script's code and its use. The publisher of an invite-only script must explicitly grant access to individual users. Invite-only scripts are mostly used by script vendors providing paid access to their scripts. Only Premium accounts can publish invite-only scripts, and they must comply with our [Vendor Requirements](#).

TradingView does not benefit from script sales. Transactions concerning invite-only scripts are strictly between users and vendors; they do not involve TradingView.

Public invite-only scripts are the only scripts for which vendors are allowed to ask for payment on TradingView.

On their invite-only script's page, authors will see a "Manage Access" button. The "Manage Access" window allows authors to control who has access to their script.



Preparing a publication

1. Even if you intend to publish publicly, it is always best to start with a private publication because you can use it to validate what your final publication will look like. You can edit the title, description, code or chart of private publications, and contrary to public scripts, you

can delete private scripts when you don't need them anymore, so they are the perfect way to practice before sharing a script publicly. You can read more about preparing script descriptions in the [How We Write and Format Script Descriptions](#) publication.

2. Prepare your chart. Load your script on the chart and remove other scripts or drawings that won't help users understand your script. Your script's plots should be easy to identify on the chart that will be published with it.
3. Load your code in the Pine Editor if it isn't already. In the Editor, click the "Publish Script"

button:

4. A popup appears to remind you that if you publish publicly, it's important that your publication comply with House Rules. Once you're through the popup, place your description in the field below the script's title. The default title proposed for your publication is the `title` field from your script's code. It is always best to use that title; it makes it easier for users to search for your script if it is public. Select the visibility of your publication. We want to publish a private publication, so we check the "Private Script"

checkbox at the bottom-right of the "Publish Script" window:

5. Select the access type you want for your script: Open, Protected or Invite-only. We have selected "Open" for open-source.

6. Select the appropriate categories for your script (at least one is mandatory) and enter

optional custom tags.

7. Click the "Publish Private Script" button in the lower-right of the window. When the publication is complete, your published script's page will appear. You are done! You can confirm the publication by going to your User Profile and viewing your "SCRIPTS" tab. From there, you will be able to open your script's page and edit your private publication by using the "Edit" button in the top-right of your script's page. Note that you can also update private publications, just like you can public ones. If you want to share your private publication with a friend, privately send her the url from your script's page. Remember you are not allowed to share links to private publications in public TradingView content.

[Publishing a script](#)

Whether you intend to publish privately or publicly, first follow the steps in the previous section. If you intend to publish privately, you will be done. If you intend to publish publicly and are satisfied with the preparatory process of validating your private publication, follow the same steps as above but do not check the "Private Script" checkbox and click the "Publish Public Script" button at the bottom-right of the "Publish Script" page.

When you publish a new public script, you have a 15-minute window to make changes to your description or delete the publication. After that you will no longer be able to change your publication's title, description, visibility or access type. If you make an error, send a message to the [PineCoders](#) moderator account; they moderate script publications and will help.

[Updating a publication](#)

You can update both public or private script publications. When you update a script, its code must be different than the previously published version's code. You can add release notes with your update. They will appear after your script's original description in the script's page.

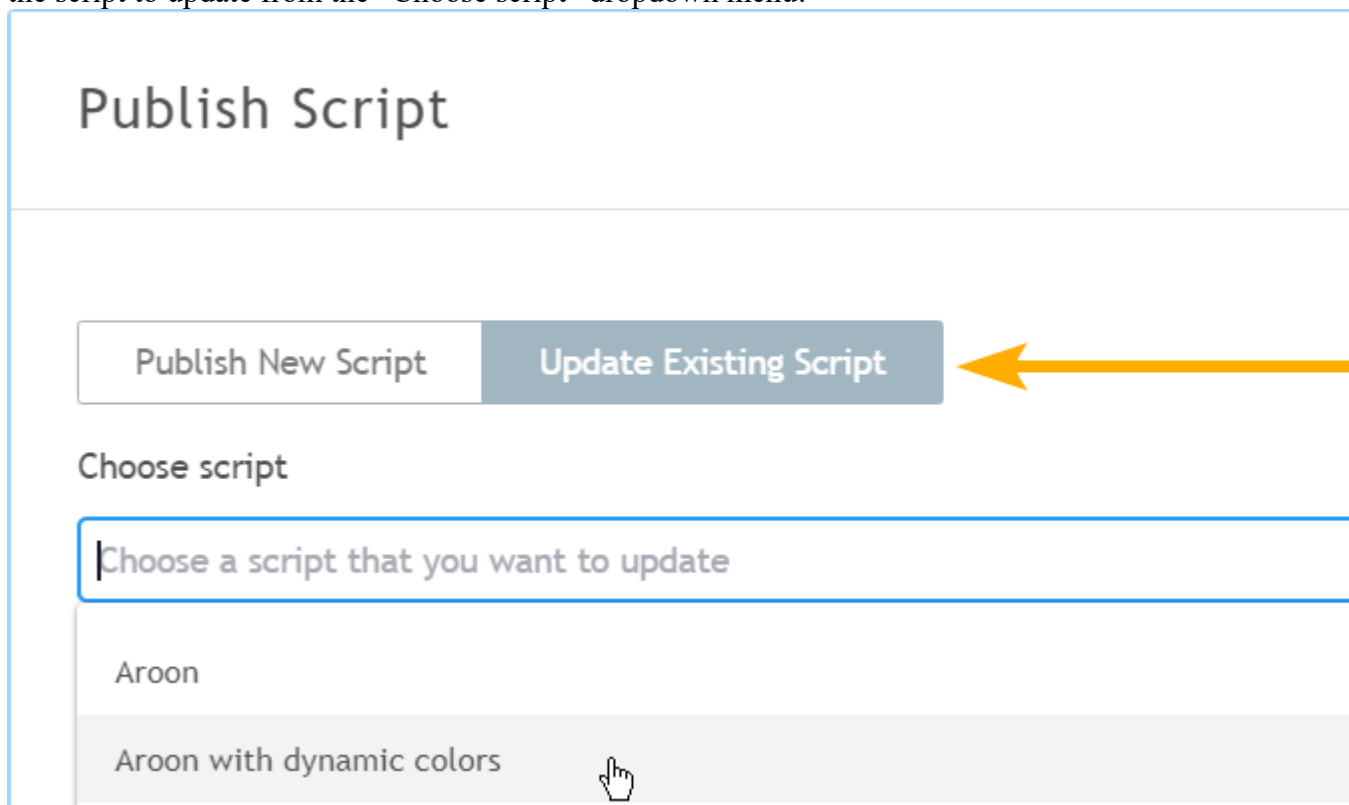
By default, the chart used when you update will replace the previous chart in your script's page.

You can choose not to update your script page's chart, however. Note that while you can update the chart displayed in the script's page, the chart from the script's widget will not update.

In the same way you can validate a public publication by first publishing a private script, you can also validate an update on a private publication before proceeding with it on your public one. The process of updating a published script is the same for public and private scripts.

If you intend to update both the code and chart of your published script, prepare your chart the same way you would for a new publication. In the following example, we will **not** be updating the publication's chart:

1. As you would for a new publication, load your script in the Editor and click the "Publish Script" button.
2. Once in the "Publish Script" window, select the "Update Existing Script" button. Then select the script to update from the "Choose script" dropdown menu:



3. Enter your release notes in the text field. The differences in your code are highlighted below your release notes.
4. We do not want to update the publication's chart, so we check the "Don't update the chart" checkbox:
5. Click the "Publish New Version" button. You're done.

Introduction

As is mentioned in our [Welcome](#) page:

Because each script uses computational resources in the cloud, we must impose limits in order to share these resources fairly among our users. We strive to set as few limits as possible, but will of course have to implement as many as needed for the platform to

run smoothly. Limitations apply to the amount of data requested from additional symbols, execution time, memory usage and script size.

If you develop complex scripts using Pine Script[®], sooner or later you will run into some of the limitations we impose. This section provides you with an overview of the limitations that you may encounter. There are currently no means for Pine Script[®] programmers to get data on the resources consumed by their scripts. We hope this will change in the future.

In the meantime, when you are considering large projects, it is safest to make a proof of concept in order to assess the probability of your script running into limitations later in your project.

Here are the limits imposed in the Pine Script[®] environment.

Time

Script compilation

Scripts must compile before they are executed on charts. Compilation occurs when you save a script from the editor or when you add a script to the chart. A two-minute limit is imposed on compilation time, which will depend on the size and complexity of your script, and whether or not a cached version of a previous compilation is available. When a compile exceeds the two-minute limit, a warning is issued. Heed that warning by shortening your script because after three consecutive warnings a one-hour ban on compilation attempts is enforced. The first thing to consider when optimizing code is to avoid repetitions by using functions to encapsulate oft-used segments, and call functions instead of repeating code.

Script execution

Once a script is compiled it can be executed. See the [Events triggering the execution of a script](#) for a list of the events triggering the execution of a script. The time allotted for the script to execute on all bars of a dataset varies with account types. The limit is 20 seconds for basic accounts, 40 for others.

Loop execution

The execution time for any loop on any single bar is limited to 500 milliseconds. The outer loop of embedded loops counts as one loop, so it will time out first. Keep in mind that even though a loop may execute under the 500 ms time limit on a given bar, the time it takes to execute on all the dataset's bars may nonetheless cause your script to exceed the total execution time limit. For example, the limit on total execution time will make it impossible for you script to execute a 400 ms loop on each bar of a 20,000-bar dataset because your script would then need 8000 seconds to execute.

Chart visuals

Plot limits

A maximum of 64 plot counts are allowed per script. The functions that generate plot counts are:

- [plot\(\)](#)
- [plotarrow\(\)](#)
- [plotbar\(\)](#)
- [plotcandle\(\)](#)

- [plotchar\(\)](#)
- [plotshape\(\)](#)
- [alertcondition\(\)](#)
- [bgcolor\(\)](#)
- [fill\(\)](#), but only if its `color` is of the [series](#) form.

The following functions do not generate plot counts:

- [hline\(\)](#)
- [line.new\(\)](#)
- [label.new\(\)](#)
- [table.new\(\)](#)
- [box.new\(\)](#)

One function call can generate up to seven plot counts, depending on the function and how it is called. When your script exceeds the maximum of 64 plot counts, the runtime error message will display the plot count generated by your script. Once you reach that point, you can determine how many plot counts a function call generates by commenting it out in a script. As long as your script still throws an error, you will be able to see how the actual plot count decreases after you have commented out a line.

The following example shows different function calls and the number of plot counts each one will generate:

```
//@version=5
indicator("Plot count example")

bool isUp = close > open
color isUpColor = isUp ? color.green : color.red
bool isDn = not isUp
color isDnColor = isDn ? color.red : color.green

// Uses one plot count each.
p1 = plot(close, color = color.white)
p2 = plot(open, color = na)

// Uses two plot counts for the `close` and `color` series.
plot(close, color = isUpColor)

// Uses one plot count for the `close` series.
plotarrow(close, colorup = color.green, colordown = color.red)

// Uses two plot counts for the `close` and `colorup` series.
plotarrow(close, colorup = isUpColor)

// Uses three plot counts for the `close`, `colorup`, and the `colordown`
series.
plotarrow(close - open, colorup = isUpColor, colordown = isDnColor)

// Uses four plot counts for the `open`, `high`, `low`, and `close` series.
plotbar(open, high, low, close, color = color.white)

// Uses five plot counts for the `open`, `high`, `low`, `close`, and `color`
series.
plotbar(open, high, low, close, color = isUpColor)

// Uses four plot counts for the `open`, `high`, `low`, and `close` series.
plotcandle(open, high, low, close, color = color.white, wickcolor = color.white,
bordercolor = color.purple)

// Uses five plot counts for the `open`, `high`, `low`, `close`, and `color`
```

```

series.
plotcandle(open, high, low, close, color = isUpColor, wickcolor = color.white,
bordercolor = color.purple)

// Uses six plot counts for the `open`, `high`, `low`, `close`, `color`, and
`wickcolor` series.
plotcandle(open, high, low, close, color = isUpColor, wickcolor = isUpColor ,
bordercolor = color.purple)

// Uses seven plot counts for the `open`, `high`, `low`, `close`, `color`,
`wickcolor`, and `bordercolor` series.
plotcandle(open, high, low, close, color = isUpColor, wickcolor = isUpColor ,
bordercolor = isUp ? color.lime : color.maroon)

// Uses one plot count for the `close` series.
plotchar(close, color = color.white, text = "|", textcolor = color.white)

// Uses two plot counts for the `close` and `color` series.
plotchar(close, color = isUpColor, text = "-", textcolor = color.white)

// Uses three plot counts for the `close`, `color`, and `textcolor` series.
plotchar(close, color = isUpColor, text = "0", textcolor = isUp ? color.yellow :
color.white)

// Uses one plot count for the `close` series.
plotshape(close, color = color.white, textcolor = color.white)

// Uses two plot counts for the `close` and `color` series.
plotshape(close, color = isUpColor, textcolor = color.white)

// Uses three plot counts for the `close`, `color`, and `textcolor` series.
plotshape(close, color = isUpColor, textcolor = isUp ? color.yellow :
color.white)

// Uses one plot count.
alertcondition(close > open, "close > open", "Up bar alert")

// Uses one plot count.
bgcolor(isUp ? color.yellow : color.white)

// Uses one plot count for the `color` series.
fill(p1, p2, color = isUpColor)

```

This example generates a plot count of 56. If we were to add two more instances of the last call to [plotcandle\(\)](#), the script would throw an error stating that the script now uses 70 plot counts, as each additional call to [plotcandle\(\)](#) generates seven plot counts, and $56 + (7 * 2)$ is 70.

Line, box, and label limits

Contrary to plots which can cover the entire dataset, by default, only the last 50 lines drawn by a script are visible on charts. The same goes for boxes and labels. You can increase the quantity of drawing objects preserved on charts up to a maximum of 500 by using the `max_lines_count`, `max_boxes_count` or `max_labels_count` parameters in the [indicator\(\)](#) or [strategy\(\)](#) declaration statements.

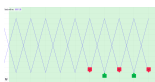
In this example we set the maximum quantity of last labels shown on the chart to 100:

```

//@version=5
indicator("Label limits example", max_labels_count = 100, overlay = true)
label.new(bar_index, high, str.tostring(high, format.mintick))

```

It's important to note that when you set any of the attributes of a drawing object to na, it still counts as a drawing on the chart and thus contributes to a script's drawing totals. To demonstrate this, the following script draws a "Buy" and "Sell" label on each bar with x values determined by the longCondition and shortCondition variables. The "Buy" label's x value is na when the bar index is even, and the "Sell" label's x value is na when the bar index is odd. Although the max_labels_count is 10 in this example, we can see that the script displays fewer than ten labels on the chart since the ones with na values also count toward the total:



```
//@version=5

// Approximate maximum number of label drawings
MAX_LABELS = 10

indicator("labels with na", overlay = false, max_labels_count = MAX_LABELS)

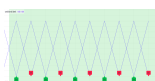
// Add background color for the last MAX_LABELS bars.
bgcolor(bar_index > last_bar_index - MAX_LABELS ? color.new(color.green, 80) :
na)

longCondition = bar_index % 2 != 0
shortCondition = bar_index % 2 == 0

// Add "Buy" and "Sell" labels on each new bar.
label.new(longCondition ? bar_index : na, 0, text = "Buy", color =
color.new(color.green, 0), style = label.style_label_up)
label.new(shortCondition ? bar_index : na, 0, text = "Sell", color =
color.new(color.red, 0), style = label.style_label_down)

plot(longCondition ? 1 : 0)
plot(shortCondition ? 1 : 0)
```

If we want the script to display the desired number of labels, we need to eliminate the ones with na x values so that they don't add to the script's label count. This example conditionally draws the "Buy" and "Sell" labels rather than always drawing them and setting their attributes to na on alternating bars:



```
//@version=5

// Approximate maximum number of label drawings
MAX_LABELS = 10

indicator("conditional labels", overlay = false, max_labels_count = MAX_LABELS)

// Add background color for the last MAX_LABELS bars.
bgcolor(bar_index > last_bar_index - MAX_LABELS ? color.new(color.green, 80) :
na)

longCondition = bar_index % 2 != 0
shortCondition = bar_index % 2 == 0

// Add a "Buy" label when `longCondition` is true.
if longCondition
    label.new(bar_index, 0, text = "Buy", color = color.new(color.green, 0),
style = label.style_label_up)
// Add a "Sell" label when `shortCondition` is true.
```

```

if shortCondition
    label.new(bar_index, 0, text = "Sell", color = color.new(color.red, 0),
style = label.style_label_down)

plot(longCondition ? 1 : 0)
plot(shortCondition ? 1 : 0)

```

Table limits

A maximum of nine tables can be displayed by a script, one for each of the possible locations: [position.bottom_center](#), [position.bottom_left](#), [position.bottom_right](#), [position.middle_center](#), [position.middle_left](#), [position.middle_right](#), [position.top_center](#), [position.top_left](#), or [position.top_right](#). If you place two tables in the same position, only the most recently added table will be visible.

`request.*()` calls

Number of calls

A script cannot make more than 40 calls to `request.*()` functions. All instances of calls to these functions are counted, even if they are included in code blocks or functions that are never actually used in the script's logic. The functions counting towards this limit are: [request.security\(\)](#), [request.security_lower_tf\(\)](#), [request.quandl\(\)](#), [request.financial\(\)](#), [request.dividends\(\)](#), [request.earnings\(\)](#) and [request.splits\(\)](#).

Intrabars

When accessing lower timeframes, with [request.security\(\)](#) or [request.security_lower_tf\(\)](#), a maximum of 100,000 intrabars can be used in calculations.

The quantity of chart bars covered with 100,000 intrabars will vary with the ratio of the chart's timeframe to the lower timeframe used, and with the average number of intrabars contained in each chart bar. For example, when using a 1min lower timeframe, chart bars at the 60min timeframe of an active 24x7 market will usually contain 60 intrabars each. Because $100,000 / 60 = 1666.67$, the quantity of chart bars covered by the 100,000 intrabars will typically be 1666. On markets where 60min chart bars do not always contain 60 1min intrabars, more chart bars will be covered.

Tuple element limit

All the `request.*()` functions in one script taken together cannot return more than 127 tuple values. Below we have an example showing what can cause this error and how to work around it:

```

//@version=5
indicator("Tuple values error")

// CAUSES ERROR:
[v1, v2, v3,...] = request.security(syminfo.tickerid, "1D", [s1, s2, s3,...])

// Works fine:
type myType
    int v1
    int v2
    int v3
    ...

myObj = request.security(syminfo.tickerid, "1D", myType.new())

```


Note that:

- In this example, we have a [request.security\(\)](#) function with at least three values in our tuple, and we could either have more than 127 values in our tuple above or more than 127 values between multiple [request.security\(\)](#) functions to throw this error.
- We get around the error by simply creating a User-defined object that can hold the same values without throwing an error.
- Using the `myType.new()` function is functionally the same as listing the same values in our `[s1, s2, s3, ...]` tuple.

Script size and memory

Compiled tokens

Before a script is executed, it is compiled into a tokenized Intermediate Language (IL). Using an IL allows Pine Script[®] to accommodate longer scripts by applying various optimizations before it is executed. The compiled form of indicators and strategies is limited to 68,000 tokens; libraries have a limit of 1 million tokens. There is no way to inspect the number of tokens created during compilation; you will only know your script exceeds the limit when the compiler reaches it.

Replacing code repetitions with function calls and using libraries to offload some of the workload are the most efficient ways to decrease the number of tokens your compiled script will generate.

The size of variable names and comments do not affect the number of compiled tokens.

Local blocks

Local blocks are segments of indented code used in function definitions or in [if](#), [switch](#), [for](#) or [while](#) structures, which allow for one or more local blocks.

Scripts are limited to 500 local blocks.

Variables

A maximum of 1000 variables are allowed per scope. Pine scripts always contain one global scope, and can contain zero or more local scopes. Local scopes are created by indented code such as can be found in functions or [if](#), [switch](#), [for](#) or [while](#) structures, which allow for one or more local blocks. Each local block counts as one local scope.

The branches of a conditional expression using a [?:](#) ternary operator do not count as local blocks.

Collections

Pine Script[®] collections ([arrays](#), [matrices](#), and [maps](#)) can have a maximum of 100,000 elements. Each key-value pair in a map contains two elements, meaning [maps](#) can contain a maximum of 50,000 key-value pairs.

Other limitations

Maximum bars back

References to past values using the `[]` history-referencing operator are dependent on the size of the

historical buffer maintained by the Pine Script® runtime, which is limited to a maximum of 5000 bars. [This Help Center page](#) discusses the historical buffer and how to change its size using either the `max_bars_back` parameter or the `max_bars_back()` function.

Maximum bars forward

When positioning drawings using `xloc.bar_index`, it is possible to use bar index values greater than that of the current bar as x coordinates. A maximum of 500 bars in the future can be referenced.

This example shows how we use the *maxval* parameter in our `input.int()` function call to cap the user-defined number of bars forward we draw a projection line so that it never exceeds the limit:

```
//@version=5
indicator("Max bars forward example", overlay = true)

// This function draws a `line` using bar index x-coordinates.
drawLine(bar1, y1, bar2, y2) =>
    // Only execute this code on the last bar.
    if barstate.islast
        // Create the line only the first time this function is executed on the
last bar.
        var line lin = line.new(bar1, y1, bar2, y2, xloc.bar_index)
        // Change the line's properties on all script executions on the last
bar.
        line.set_xy1(lin, bar1, y1)
        line.set_xy2(lin, bar2, y2)

// Input determining how many bars forward we draw the `line`.
int forwardBarsInput = input.int(10, "Forward Bars to Display", minval = 1,
maxval = 500)

// Calculate the line's left and right points.
int leftBar = bar_index[2]
float leftY = high[2]
int rightBar = leftBar + forwardBarsInput
float rightY = leftY + (ta.change(high)[1] * forwardBarsInput)

// This function call is executed on all bars, but it only draws the `line` on
the last bar.
drawLine(leftBar, leftY, rightBar, rightY)
```

Chart bars

The number of bars appearing on charts is dependent on the amount of historical data available for the chart's symbol and timeframe, and on the type of account you hold. When the required historical date is available, the minimum number of chart bars is:

- 20,000 bars for the Premium plan.
- 10,000 bars for Pro and Pro+ plans.
- 5000 bars for other plans.

Trade orders in backtesting

A maximum of 9000 orders can be placed when backtesting strategies. When using Deep Backtesting, the limit is 200,000.



FAQ

- [Get real OHLC price on a Heikin Ashi chart](#)
- [Get non-standard OHLC values on a standard chart](#)
- [Plot arrows on the chart](#)
- [Plot a dynamic horizontal line](#)
- [Plot a vertical line on condition](#)
- [Access the previous value](#)
- [Get a 5-days high](#)
- [Count bars in a dataset](#)
- [Enumerate bars in a day](#)
- [Find the highest and lowest values for the entire dataset](#)
- [Query the last non-na value](#)

Get real OHLC price on a Heikin Ashi chart

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

```
//@version=5
indicator("Visible OHLC", overlay=true)
c = close
plot(c)
```

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

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

```
//@version=5
indicator("Real OHLC", overlay = true)
t = ticker.new(syminfo.prefix, syminfo.ticker)
realC = request.security(t, timeframe.period, close)
plot(realC)
```

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

Get non-standard OHLC values on a standard chart

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

```
//@version=5
strategy("BarUpDn Strategy", overlay = true, default_qty_type =
strategy.percent_of_equity, default_qty_value = 10)
maxIdLossPcntInput = input.float(1, "Max Intraday Loss(%)")
strategy.risk.max_intraday_loss(maxIdLossPcntInput, strategy.percent_of_equity)
needTrade() => close > open and open > close[1] ? 1 : close < open and open <
close[1] ? -1 : 0
```

```

trade = request.security(ticker.heikinashi(syminfo.tickerid), timeframe.period,
needTrade())
if trade == 1
    strategy.entry("BarUp", strategy.long)
if trade == -1
    strategy.entry("BarDn", strategy.short)

```

Plot arrows on the chart

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

```

//@version=5
indicator('Ex 1', overlay = true)
condition = close >= open
plotshape(condition, color = color.lime, style = shape.arrowup, text = "Buy")
plotshape(not condition, color = color.red, style = shape.arrowdown, text =
"Sell")

```



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

```

//@version=5
indicator('buy/sell arrows', overlay = true)
condition = close >= open
plotchar(not condition, char='↓', color = color.lime, text = "Buy")
plotchar(condition, char='↑', location = location.belowbar, color = color.red,
text = "Sell")

```



Plot a dynamic horizontal line

There is the function `hline` in Pine Script[®], but it is limited to only plot a constant value. Here is a simple script with a workaround to plot a changing hline:

```

//@version=5
indicator("Horizontal line", overlay = true)
plot(close[10], trackprice = true, offset = -9999)
// `trackprice = true` plots horizontal line on close[10]
// `offset = -9999` hides the plot
plot(close, color = #FFFFFF) // forces display

```

Plot a vertical line on condition

```

//@version=5
indicator("Vertical line", overlay = true, scale = scale.none)
// scale.none means do not resize the chart to fit this plot
// if the bar being evaluated is the last bar on the chart (the most recent bar),
then cond is true
cond = barstate.islast
// when cond is true, plot a histogram with a line with height value of
100,000,000,000,000,000,000.00
// (10 to the power of 20)
// when cond is false, plot no numeric value (nothing is plotted)
// use the style of histogram, a vertical bar

```

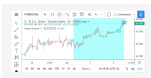
```
plot(cond ? 10e20 : na, style = plot.style_histogram)
```

Access the previous value

```
//@version=5
//...
s = 0.0
s := nz(s[1]) // Accessing previous values
if (condition)
    s := s + 1
```

Get a 5-days high

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



```
//@version=5
indicator("High of last 5 days", overlay = true)

// Milliseconds in 5 days: millisecs * secs * mins * hours * days
MS_IN_5DAYS = 1000 * 60 * 60 * 24 * 5

// The range check begins 5 days from the current time.
leftBorder = timenow - time < MS_IN_5DAYS
// The range ends on the last bar of the chart.
rightBorder = barstate.islast

// ——— Keep track of highest `high` during the range.
// Intialize `maxHi` with `var` on bar zero only.
// This way, its value is preserved, bar to bar.
var float maxHi = na
if leftBorder
    if not leftBorder[1]
        // Range's first bar.
        maxHi := high
    else if not rightBorder
        // On other bars in the range, track highest `high`.
        maxHi := math.max(maxHi, high)

// Plot level of the highest `high` on the last bar.
plotchar(rightBorder ? maxHi : na, "Level", "-", location.absolute, size =
size.normal)
// When in range, color the background.
bgcolor(leftBorder and not rightBorder ? color.new(color.aqua, 70) : na)
```

Count bars in a dataset

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

```
//@version=5
indicator("Bar Count", overlay = true, scale = scale.none)
plot(bar_index + 1, style = plot.style_histogram)
```

Enumerate bars in a day

```
//@version=5
indicator("My Script", overlay = true, scale = scale.none)

isNewDay() =>
  d = dayofweek
  na(d[1]) or d != d[1]

plot(ta.barssince(isNewDay()), style = plot.style_cross)
```

Find the highest and lowest values for the entire dataset

```
//@version=5
indicator("", "", true)

allTimeHi(source) =>
  var atHi = source
  atHi := math.max(atHi, source)

allTimeLo(source) =>
  var atLo = source
  atLo := math.min(atLo, source)

plot(allTimeHi(close), "ATH", color.green)
plot(allTimeLo(close), "ATL", color.red)
```

Query the last non-na value

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

```
//@version=5
indicator("")
series = close >= open ? close : na
vw = fixnan(series)
plot(series, style = plot.style_linebr, color = color.red) // series has na
values
plot(vw) // all na values are replaced with the last non-empty value
```

Error messages

- [The if statement is too long](#)
- [Script requesting too many securities](#)
- [Script could not be translated from: null](#)
- [line 2: no viable alternative at character '\\$'](#)
- [Mismatched input <...> expecting <??>](#)
- [Loop is too long \(> 500 ms\)](#)
- [Script has too many local variables](#)
- [Pine Script® cannot determine the referencing length of a series. Try using max_bars_back in the indicator or strategy function](#)

The if statement is too long

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

```
//@version=5
indicator("My script")

var e = 0
if barstate.islast
    a = 1
    b = 2
    c = 3
    d = 4
    e := a + b + c + d

plot(e)
```

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

```
//@version=5
indicator("My script")

var e = 0
doSomeWork() =>
    a = 1
    b = 2
    c = 3
    d = 4

    result = a + b + c + d

if barstate.islast
    e := doSomeWork()

plot(e)
```

Script requesting too many securities

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

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

```
//@version=5
indicator("Securities count")
a = request.security(syminfo.tickerid, '42', close) // (1) first unique
security call
b = request.security(syminfo.tickerid, '42', close) // same call as above, will
not produce new security call after optimizations

plot(a)
plot(a + 2)
```

```

plot(b)

sym(p) => // no security call on this line
    request.security(syminfo.tickerid, p, close)
plot(sym('D')) // (2) one indirect call to security
plot(sym('W')) // (3) another indirect call to security

request.security(syminfo.tickerid, timeframe.period, open) // result of this
line is never used, and will be optimized out

```

Script could not be translated from: null

```
study($)
```

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

```
// @version=2
study($)
```

line 2: no viable alternative at character '\$'

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

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

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

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

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

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

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

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

Loop is too long (> 500 ms)

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

```
//@version=5
indicator("Loop is too long", max_bars_back = 101)
```



```

s = 0
for i = 1 to 1e3 // to make it longer
  for j = 0 to 100
    if timestamp(2017, 02, 23, 00, 00) <= time[j] and time[j] <
timestamp(2017, 02, 23, 23, 59)
      s := s + 1
plot(s)

```

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

```

//@version=5
indicator("Loop is too long", max_bars_back = 101)
bar_back_at(t) =>
  i = 0
  step = 51
  for j = 1 to 100
    if i < 0
      i := 0
      break
    if step == 0
      break
    if time[i] >= t
      i := i + step
      i
    else
      i := i - step
      i
    step := step / 2
  step
  i

s = 0
for i = 1 to 1e3 // to make it longer
  s := s - bar_back_at(timestamp(2017, 02, 23, 23, 59)) +
    bar_back_at(timestamp(2017, 02, 23, 00, 00))
  s
plot(s)

```

Script has too many local variables

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

```

var1 = expr1
var2 = expr2
var3 = var1 + var2

```

can be converted into:

```

var3 = expr1 + expr2

```

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

The error appears in cases where Pine Script® wrongly autodetects the required maximum length of series used in a script. This happens when a script's flow of execution does not allow Pine Script® to inspect the use of series in branches of conditional statements (`if`, `iff` or `?`), and Pine Script® cannot automatically detect how far back the series is referenced. Here is an example of a script causing this problem:

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

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

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

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

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

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

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

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

```
//@version=5
indicator("My Script")
```

```
f(off) =>
  t = 0.0
  s = close
  max_bars_back(s, 301)
  if bar_index > 242
    t := s[off]
  t
plot(f(301))
```

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



To Pine Script[®] version 5

- [Introduction](#)
- [v4 to v5 converter](#)
- [Renamed functions and variables](#)
- [Renamed function parameters](#)
- [Removed an `rsi\(\)` overload](#)
- [Reserved keywords](#)
- [Removed `iff\(\)` and `offset\(\)`](#)
- [Split of `input\(\)` into several functions](#)
- [Some function parameters now require built-in arguments](#)
- [Deprecated the `transp` parameter](#)
- [Changed the default session days for `time\(\)` and `time_close\(\)`](#)
- [`strategy.exit\(\)` now must do something](#)
- [Common script conversion errors](#)
- [All variable, function, and parameter name changes](#)

Introduction

This guide documents the **changes** made to Pine Script[®] from v4 to v5. It will guide you in the adaptation of existing Pine scripts to Pine Script[®] v5. See our [Release notes](#) for a list of the **new** features in Pine Script[®] v5.

The most frequent adaptations required to convert older scripts to v5 are:

- Changing [study\(\)](#) for [indicator\(\)](#) (the function's signature has not changed).
- Renaming built-in function calls to include their new namespace (e.g., [highest\(\)](#) in v4 becomes [ta.highest\(\)](#) in v5).
- Restructuring inputs to use the more specialized `input.*()` functions.
- Eliminating uses of the deprecated `transp` parameter by using [color.new\(\)](#) to simultaneously define color and transparency for use with the `color` parameter.

- If you used the `resolution` and `resolution_gaps` parameters in v4's [study\(\)](#), they will require changing to `timeframe` and `timeframe_gaps` in v5's [indicator\(\)](#).

[v4 to v5 converter](#)

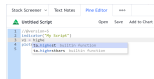
The Pine Script® Editor includes a utility to automatically convert v4 scripts to v5. To access it, open a script with `//@version=4` in it and select the “Convert to v5” option in the “More” menu identified by three dots at the top-right of the Editor’s pane:



Not all scripts can be automatically converted from v4 to v5. If you want to convert the script manually or if your indicator returns a compilation error after conversion, use the following sections to determine how to complete the conversion. A list of some errors you can encounter during the automatic conversion and how to fix them can be found in the [Common script conversion errors](#) section of this guide.

[Renamed functions and variables](#)

For clarity and consistency, many built-in functions and variables were renamed in v5. The inclusion of v4 function names in a new namespace is the cause of most changes. For example, the [sma\(\)](#) function in v4 is moved to the `ta.` namespace in v5: [ta.sma\(\)](#). Remembering the new namespaces is not necessary; if you type the older name of a function without its namespace in the Editor and press the ‘Auto-complete’ hotkey (Ctrl + Space, or Cmd + Space on MacOS), a popup showing matching suggestions appears:



Not counting functions moved to new namespaces, only two functions have been renamed:

- `study()` is now [indicator\(\)](#).
- `tickerid()` is now [ticker.new\(\)](#).

The full list of renamed functions and variables can be found in the [All variable, function, and parameter name changes](#) section of this guide.

[Renamed function parameters](#)

The parameter names of some built-in functions were changed to improve the nomenclature. This has no bearing on most scripts, but if you used these parameter names when calling functions, they will require adaptation. For example, we have standardized all mentions:

```
// Valid in v4. Not valid in v5.
timev4 = time(resolution = "1D")
// Valid in v5.
timev5 = time(timeframe = "1D")
// Valid in v4 and v5.
timeBoth = time("1D")
```

The full list of renamed function parameters can be found in the [All variable, function, and parameter name changes](#) section of this guide.

Removed an `rsi()` overload

In v4, the [rsi\(\)](#) function had two different overloads:

- `rsi(series float, simple int)` for the normal RSI calculation, and
- `rsi(series float, series float)` for an overload used in the MFI indicator, which did a calculation equivalent to $100.0 - (100.0 / (1.0 + \text{arg1} / \text{arg2}))$.

This caused a single built-in function to behave in two very different ways, and it was difficult to distinguish which one applied because it depended on the type of the second argument. As a result, a number of indicators misused the function and were displaying incorrect results. To avoid this, the second overload was removed in v5.

The [ta.rsi\(\)](#) function in v5 only accepts a “simple int” argument for its `length` parameter. If your v4 code used the now deprecated overload of the function with a `float` second argument, you can replace the whole `rsi()` call with the following formula, which is equivalent:

```
100.0 - (100.0 / (1.0 + arg1 / arg2))
```

Note that when your v4 code used a “series int” value as the second argument to [rsi\(\)](#), it was automatically cast to “series float” and the second overload of the function was used. While this was syntactically correct, it most probably did **not** yield the result you expected. In v5, [ta.rsi\(\)](#) requires a “simple int” for the argument to `length`, which precludes dynamic (or “series”) lengths. The reason for this is that RSI calculations use the [ta.rma\(\)](#) moving average, which is similar to [ta.ema\(\)](#) in that it relies on a length-dependent recursive process using the values of previous bars. This makes it impossible to achieve correct results with a “series” length that could vary bar to bar.

If your v4 code used a length that was “const int”, “input int” or “simple int”, no changes are required.

Reserved keywords

A number of words are reserved and cannot be used for variable or function names. They are: `catch`, `class`, `do`, `ellipse`, `in`, `is`, `polygon`, `range`, `return`, `struct`, `text`, `throw`, `try`. If your v4 indicator uses any of these, rename your variable or function for the script to work in v5.

Removed `iff()` and `offset()`

The [iff\(\)](#) and [offset\(\)](#) functions have been removed. Code using the [iff\(\)](#) function can be rewritten using the ternary operator:

```
// iff(<condition>, <return_when_true>, <return_when_false>)  
// Valid in v4, not valid in v5  
barColorIff = iff(close >= open, color.green, color.red)  
// <condition> ? <return_when_true> : <return_when_false>  
// Valid in v4 and v5  
barColorTernary = close >= open ? color.green : color.red
```

Note that the ternary operator is evaluated “lazily”; only the required value is calculated (depending on the condition’s evaluation to `true` or `false`). This is different from [iff\(\)](#), which always evaluated both values but returned only the relevant one.

Some functions require evaluation on every bar to correctly calculate, so you will need to make special provisions for these by pre-evaluating them before the ternary:

```
// `iff()` in v4: `highest()` and `lowest()` are calculated on every bar
v1 = iff(close > open, highest(10), lowest(10))
plot(v1)
// In v5: forced evaluation on every bar prior to the ternary statement.
h1 = ta.highest(10)
l1 = ta.lowest(10)
v1 = close > open ? h1 : l1
plot(v1)
```

The [offset\(\)](#) function was deprecated because the more readable `[]` operator is equivalent:

```
// Valid in v4. Not valid in v5.
prevClosev4 = offset(close, 1)
// Valid in v4 and v5.
prevClosev5 = close[1]
```

[Split of `input\(\)` into several functions](#)

The v4 [input\(\)](#) function was becoming crowded with a plethora of overloads and parameters. We split its functionality into different functions to clear that space and provide a more robust structure to accommodate the additions planned for inputs. Each new function uses the name of the `input.*` type of the v4 `input()` call it replaces. E.g., there is now a specialized [input.float\(\)](#) function replacing the v4 `input(1.0, type = input.float)` call. Note that you can still use `input(1.0)` in v5, but because only [input.float\(\)](#) allows for parameters such as `minval`, `maxval`, etc., it is more powerful. Also note that [input.int\(\)](#) is the only specialized input function that does not use its equivalent v4 `input.integer` name. The `input.*` constants have been removed because they were used as arguments for the `type` parameter, which was deprecated.

To convert, for example, a v4 script using an input of type `input.symbol`, the [input.symbol\(\)](#) function must be used in v5:

```
// Valid in v4. Not valid in v5.
aaplTicker = input("AAPL", type = input.symbol)
// Valid in v5
aaplTicker = input.symbol("AAPL")
```

The [input\(\)](#) function persists in v5, but in a simpler form, with less parameters. It has the advantage of automatically detecting input types “bool/color/int/float/string/source” from the argument used for `defval`:

```
// Valid in v4 and v5.
// While "AAPL" is a valid symbol, it is only a string here because
`input.symbol()` is not used.
tickerString = input("AAPL", title = "Ticker string")
```

[Some function parameters now require built-in arguments](#)

In v4, built-in constants such as `plot.style_area` used as arguments when calling Pine Script[®] functions corresponded to pre-defined values of a specific type. For example, the value of `barmerge.lookahead_on` was `true`, so you could use `true` instead of the named constant when supplying an argument to the `lookahead` parameter in a [security\(\)](#) function call. We found this to be a common source of confusion, which caused unsuspecting programmers to produce code yielding unintended results.

In v5, the use of correct built-in named constants as arguments to function parameters requiring them is mandatory:

```
// Not valid in v5: `true` is used as an argument for `lookahead`.
request.security(syminfo.tickerid, "1D", close, lookahead = true)
// Valid in v5: uses a named constant instead of `true`.
request.security(syminfo.tickerid, "1D", close, lookahead =
barmerge.lookahead_on)

// Would compile in v4 because `plot.style_columns` was equal to 5.
// Won't compile in v5.
a = 2 * plot.style_columns
plot(a)
```

To convert your script from v4 to v5, make sure you use the correct named built-in constants as function arguments.

Deprecated the `transp` parameter

The `transp=` parameter used in the signature of many v4 plotting functions was deprecated because it interfered with RGB functionality. Transparency must now be specified along with the color as an argument to parameters such as `color`, `textcolor`, etc. The [color.new\(\)](#) or [color.rgb\(\)](#) functions will be needed in those cases to join a color and its transparency.

Note that in v4, the [bgcolor\(\)](#) and [fill\(\)](#) functions had an optional `transp` parameter that used a default value of 90. This meant that the code below could display Bollinger Bands with a semi-transparent fill between two bands and a semi-transparent background color where bands cross price, even though no argument is used for the `transp` parameter in its [bgcolor\(\)](#) and [fill\(\)](#) calls:

```
//@version=4
study("Bollinger Bands", overlay = true)
[middle, upper, lower] = bb(close, 5, 4)
plot(middle, color=color.blue)
p1PlotID = plot(upper, color=color.green)
p2PlotID = plot(lower, color=color.green)
crossUp = crossover(high, upper)
crossDn = crossunder(low, lower)
// Both `fill()` and `bgcolor()` have a default `transp` of 90
fill(p1PlotID, p2PlotID, color = color.green)
bgcolor(crossUp ? color.green : crossDn ? color.red : na)
```

In v5 we need to explicitly mention the 90 transparency with the color, yielding:

```
//@version=5
indicator("Bollinger Bands", overlay = true)
[middle, upper, lower] = ta.bb(close, 5, 4)
plot(middle, color=color.blue)
p1PlotID = plot(upper, color=color.green)
p2PlotID = plot(lower, color=color.green)
crossUp = ta.crossover(high, upper)
crossDn = ta.crossunder(low, lower)
var TRANSP = 90
// We use `color.new()` to explicitly pass transparency to both functions
fill(p1PlotID, p2PlotID, color = color.new(color.green, TRANSP))
bgcolor(crossUp ? color.new(color.green, TRANSP) : crossDn ?
color.new(color.red, TRANSP) : na)
```

Changed the default session days for `time()` and `time_close()`

The default set of days for session strings used in the [time\(\)](#) and [time_close\(\)](#) functions, and

returned by [input.session\(\)](#), has changed from "23456" (Monday to Friday) to "1234567" (Sunday to Saturday):

```
// On symbols that are traded during weekends, this will behave differently in
v4 and v5.
t0 = time("1D", "1000-1200")
// v5 equivalent of the behavior of `t0` in v4.
t1 = time("1D", "1000-1200:23456")
// v5 equivalent of the behavior of `t0` in v5.
t2 = time("1D", "1000-1200:1234567")
```

This change in behavior should not have much impact on scripts running on conventional markets that are closed during weekends. If it is important for you to ensure your session definitions preserve their v4 behavior in v5 code, add ":23456" to your session strings. See this manual's page on [Sessions](#) for more information.

[`strategy.exit\(\)` now must do something](#)

Gone are the days when the [strategy.exit\(\)](#) function was allowed to loiter. Now it must actually have an effect on the strategy by using at least one of the following parameters: profit, limit, loss, stop, or one of the following pairs: trail_offset combined with either trail_price or trail_points. When uses of [strategy.exit\(\)](#) not meeting these criteria trigger an error while converting a strategy to v5, you can safely eliminate these lines, as they didn't do anything in your code anyway.

[Common script conversion errors](#)

Invalid argument 'style'/'linestyle' in 'plot'/'hline' call

To make this work, you need to change the "int" arguments used for the style and linestyle arguments in [plot\(\)](#) and [hline\(\)](#) for built-in constants:

```
// Will cause an error during conversion
plotStyle = input(1)
hlineStyle = input(1)
plot(close, style = plotStyle)
hline(100, linestyle = hlineStyle)

// Will work in v5
//@version=5
indicator("")
plotStyleInput = input.string("Line", options = ["Line", "Stepline",
"Histogram", "Cross", "Area", "Columns", "Circles"])
hlineStyleInput = input.string("Solid", options = ["Solid", "Dashed", "Dotted"])

plotStyle = plotStyleInput == "Line" ? plot.style_line :
    plotStyleInput == "Stepline" ? plot.style_stepline :
    plotStyleInput == "Histogram" ? plot.style_histogram :
    plotStyleInput == "Cross" ? plot.style_cross :
    plotStyleInput == "Area" ? plot.style_area :
    plotStyleInput == "Columns" ? plot.style_columns :
    plot.style_circles

hlineStyle = hlineStyleInput == "Solid" ? hline.style_solid :
    hlineStyleInput == "Dashed" ? hline.style_dashed :
    hline.style_dotted

plot(close, style = plotStyle)
```



```
hline(100, linestyle = hlineStyle)
```

See the [Some function parameters now require built-in arguments](#) section of this guide for more information.

Undeclared identifier ‘input.%input_name%’

To fix this issue, remove the `input.*` constants from your code:

```
// Will cause an error during conversion
_integer = input.integer
_bool = input.bool
i1 = input(1, "Integer", _integer)
i2 = input(true, "Boolean", _bool)

// Will work in v5
i1 = input.int(1, "Integer")
i2 = input.bool(true, "Boolean")
```

See the User Manual’s page on [Inputs](#), and the [Some function parameters now require built-in arguments](#) section of this guide for more information.

Invalid argument ‘when’ in ‘strategy.close’ call

This is caused by a confusion between [strategy.entry\(\)](#) and [strategy.close\(\)](#).

The second parameter of [strategy.close\(\)](#) is `when`, which expects a “bool” argument. In v4, it was allowed to use `strategy.long` an argument because it was a “bool”. With v5, however, named built-in constants must be used as arguments, so `strategy.long` is no longer allowed as an argument to the `when` parameter.

The `strategy.close("Short", strategy.long)` call in this code is equivalent to `strategy.close("Short")`, which is what must be used in v5:

```
// Will cause an error during conversion
if (longCondition)
    strategy.close("Short", strategy.long)
    strategy.entry("Long", strategy.long)

// Will work in v5:
if (longCondition)
    strategy.close("Short")
    strategy.entry("Long", strategy.long)
```

See the [Some function parameters now require built-in arguments](#) section of this guide for more information.

Cannot call ‘input.int’ with argument ‘minval’=’%value%’. An argument of ‘literal float’ type was used but a ‘const int’ is expected

In v4, it was possible to pass a “float” argument to `minval` when an “int” value was being input. This is no longer possible in v5; “int” values are required for “int” inputs:

```
// Works in v4, will break on conversion because minval is a 'float' value
int_input = input(1, "Integer", input.integer, minval = 1.0)

// Works in v5
int_input = input.int(1, "Integer", minval = 1)
```

See the User Manual's page on [Inputs](#), and the [Some function parameters now require built-in arguments](#) section of this guide for more information.

All variable, function, and parameter name changes

Removed functions and variables

v4	v5
<code>input.bool input</code>	Replaced by <code>input.bool()</code>
<code>input.color input</code>	Replaced by <code>input.color()</code>
<code>input.float input</code>	Replaced by <code>input.float()</code>
<code>input.integer input</code>	Replaced by <code>input.int()</code>
<code>input.resolution input</code>	Replaced by <code>input.timeframe()</code>
<code>input.session input</code>	Replaced by <code>input.session()</code>
<code>input.source input</code>	Replaced by <code>input.source()</code>
<code>input.string input</code>	Replaced by <code>input.string()</code>
<code>input.symbol input</code>	Replaced by <code>input.symbol()</code>
<code>input.time input</code>	Replaced by <code>input.time()</code>
<code>iff()</code>	Use the <code>?:</code> operator instead
<code>offset()</code>	Use the <code>[]</code> operator instead

Renamed functions and parameters

No namespace change

v4	v5
<code>study(<...>, resolution, resolution_gaps, <...>)</code>	<code>indicator(<...>, timeframe, timeframe_gaps, <...>)</code>
<code>strategy.entry(long)</code>	<code>strategy.entry(direction)</code>

strategy.order(long)	strategy.order(direction)
time(resolution)	time(timeframe)
time_close(resolution)	time_close(timeframe)
nz(x, y)	nz(source, replacement)

“ta” namespace for technical analysis functions and variables

v4	v5
Indicator functions and variables	
accdist	ta.accdist
alma()	ta.alma()
atr()	ta.atr()
bb()	ta.bb()
bbw()	ta bbw()
cci()	ta.cci()
cmo()	ta.cmo()
cog()	ta.cog()
dmi()	ta.dmi()
ema()	ta.ema()
hma()	ta.hma()
iii	ta.iii
kc()	ta.kc()
kcw()	ta.kcw()
linreg()	ta.linreg()
macd()	ta.macd()
mfi()	ta.mfi()
mom()	ta.mom()
nvi	ta.nvi
obv	ta.obv
pvi	ta.pvi
pvt	ta.pvt
rma()	ta.rma()
roc()	ta.roc()
rsi(x, y)	ta.rsi(source, length)
sar()	ta.sar()

sma()	ta.sma()
stoch()	ta.stoch()
supertrend()	ta.supertrend()
swma(x)	ta.swma(source)
tr	ta.tr
tr()	ta.tr()
tsi()	ta.tsi()
vwap	ta.vwap
vwap(x)	ta.vwap(source)
vwma()	ta.vwma()
wad	ta.wad
wma()	ta.wma()
wpr()	ta.wpr()
wvad	ta.wvad
Supporting functions	
barsince()	ta.barsince()
change()	ta.change()
correlation(source_a, source_b, length)	ta.correlation(source1, source2, length)
cross(x, y)	ta.cross(source1, source2)
crossover(x, y)	ta.crossover(source1, source2)
crossunder(x, y)	ta.crossunder(source1, source2)
cum(x)	ta.cum(source)
dev()	ta.dev()
falling()	ta.falling()
highest()	ta.highest()
highestbars()	ta.highestbars()
lowest()	ta.lowest()
lowestbars()	ta.lowestbars()
median()	ta.median()
mode()	ta.mode()
percentile_linear_interpolation()	ta.percentile_linear_interpolation()

percentile_nearest_rank()	ta.percentile_nearest_rank()
percentrank()	ta.percentrank()
pivohigh()	ta.pivohigh()
pivotlow()	ta.pivotlow()
range()	ta.range()
rising()	ta.rising()
stdev()	ta.stdev()
valuewhen()	ta.valuewhen()
variance()	ta.variance()

“math” namespace for math-related functions and variables

v4	v5
abs(x)	math.abs(number)
acos(x)	math.acos(number)
asin(x)	math.asin(number)
atan(x)	math.atan(number)
avg()	math.avg()
ceil(x)	math.ceil(number)
cos(x)	math.cos(angle)
exp(x)	math.exp(number)
floor(x)	math.floor(number)
log(x)	math.log(number)
log10(x)	math.log10(number)
max()	math.max()
min()	math.min()
pow()	math.pow()
random()	math.random()
round(x, precision)	math.round(number, precision)