Pine Script™ v5 User Manual

TradingView

Dec 08, 2023

CONTENTS

CHAPTER

ONE

WELCOME TO PINE SCRIPT™ V5

Pine Script™ is TradingView's programming language. It allows traders to create their own trading tools and run them on our servers. We designed Pine Script™ as a lightweight, yet powerful, language for developing indicators and strategies that you can then backtest. Most of TradingView's built-in indicators are written in Pine Script™, and our thriving community of Pine Script™ programmers has published more than 100,000 Community Scripts.

It's our explicit goal to keep Pine Script™ accessible and easy to understand for the broadest possible audience. Pine Script™ is cloud-based and therefore different from client-side programming languages. While we likely won't develop Pine Script™ into a full-fledged language, we do constantly improve it and are always happy to consider requests for new features.

Because each script uses computational resources in the cloud, we must impose limits in order to share these resources fairly among our users. We strive to set as few limits as possible, but will of course have to implement as many as needed for the platform to run smoothly. Limitations apply to the amount of data requested from additional symbols, execution time, memory usage and script size.

CHAPTER

TWO

PINE SCRIPT™ PRIMER

2.1 First steps

2.1.1 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!

2.1.2 Using scripts

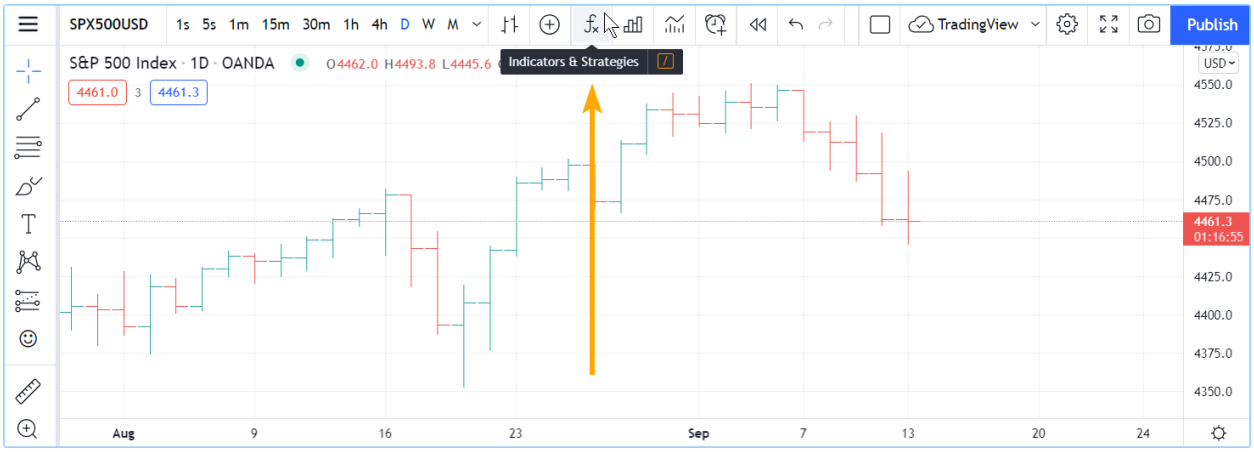
If you are interested in using technical indicators or strategies on TradingView, you can first start exploring the thousands of indicators already available on our platform. You can access existing indicators on the platform in two different ways:

- 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

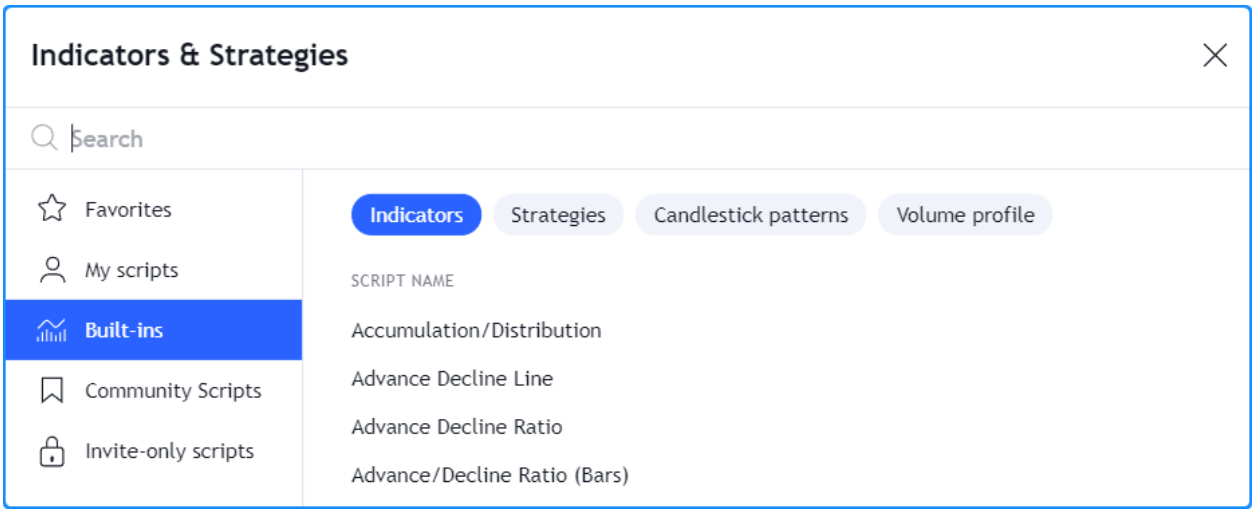
To explore and load scripts from you chart, use the "Indicators & Strategies" button:



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 scipts you have written and saved in the Pine 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.

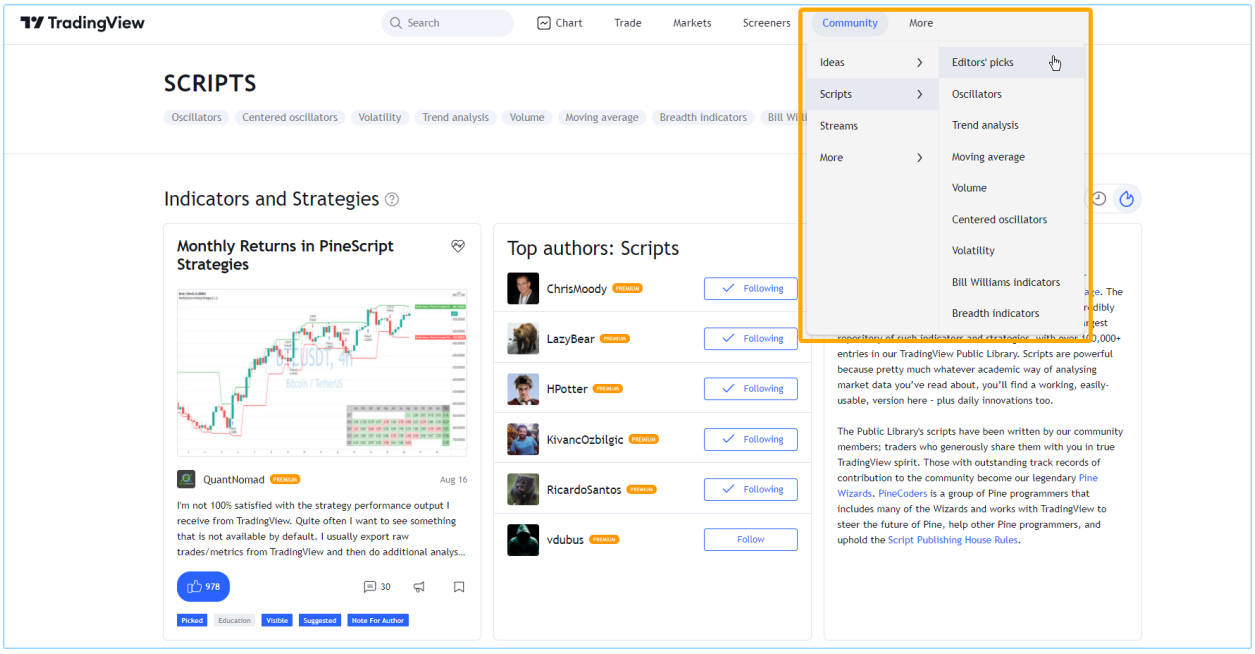
Here, the section containing the TradingView built-ins is selected:



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:



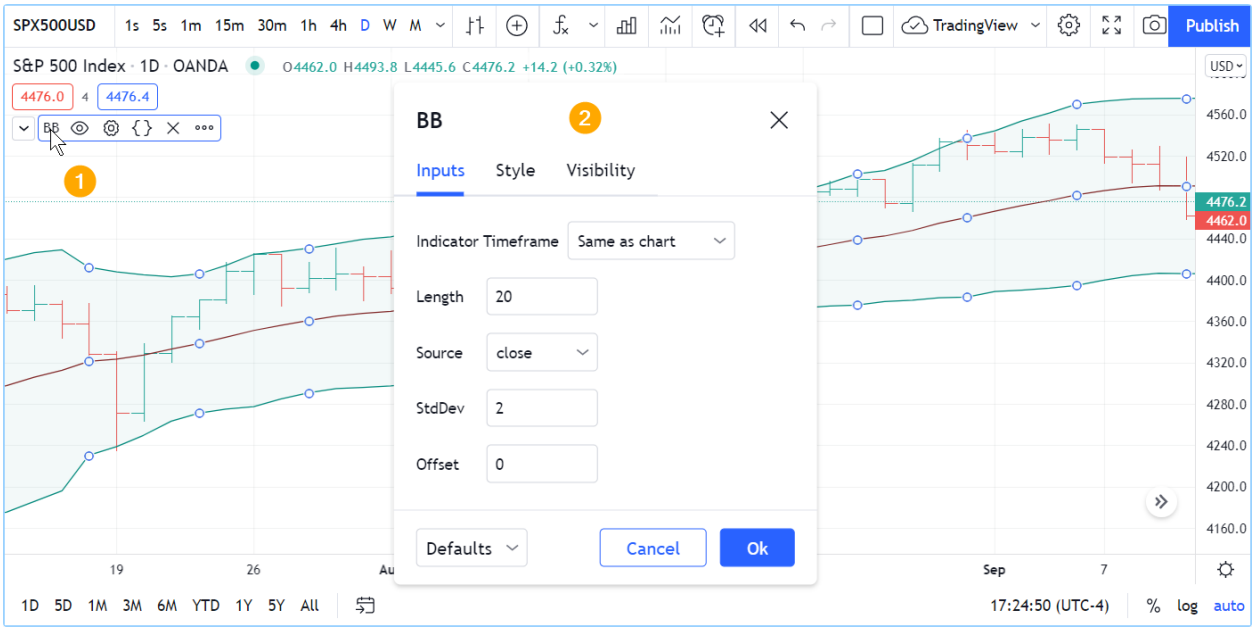
You can also search for scripts using the homepage's "Search" field, and filter scripts using different criteria. The Help Center has a page explaining the different types of scripts that are available.

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.

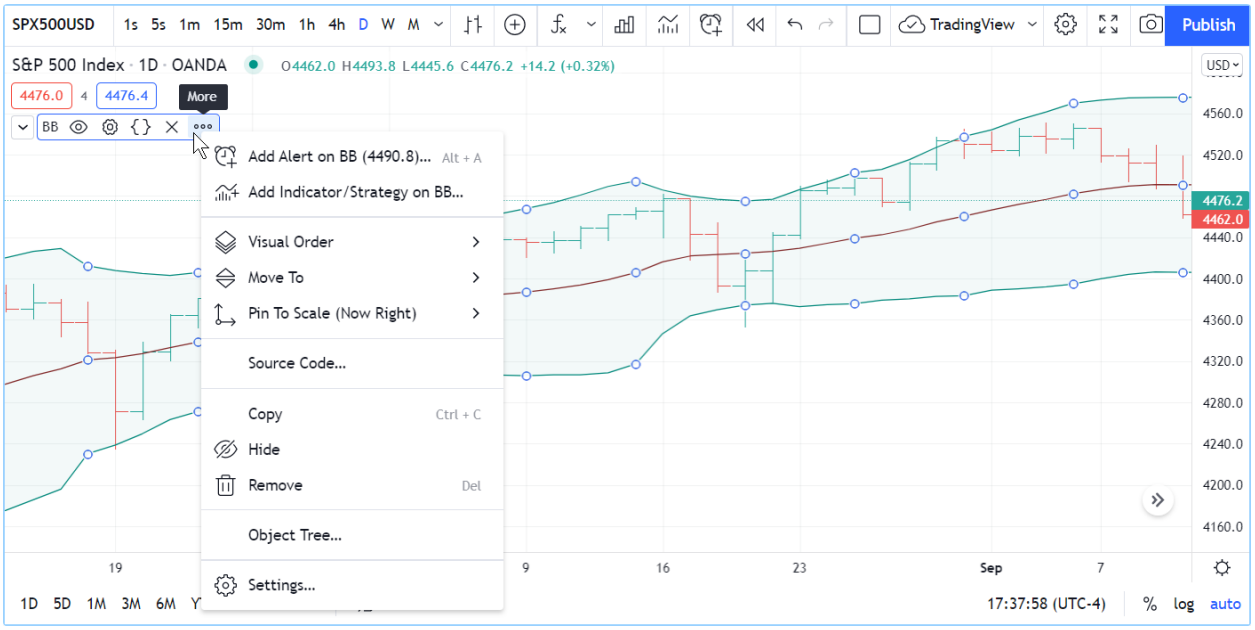
Changing script settings

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):



2.1.3 Reading scripts

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

- The TradingView built-in indicators
- Scripts selected as Editors' Picks
- Scripts by the authors the PineCoders account follows
- Many scripts by authors with high reputation and open-source publications.

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

To see the code of TradingView built-ins, load the indicator on your chart, then hover over its name and select the "Source code" curly braces icon (if you don't see it, it's because the indicator's source is unavailable). When you click on the icon, the Pine Editor will open and from there, you can see the script's code. If you want to play with it, you will need to 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 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 Editor (accessible from the "Pine Editor" tab at the bottom of the chart) by using the "Open/New default built-in script…" menu selection.

2.1.4 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*.

2.2 First indicator

2.2.1 The Pine Editor

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

- It highlights your code following Pine Script™ syntax.
- It pops up syntax reminders 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 Editor" tab at the bottom of your TradingView chart. This will open up the Editor's pane.

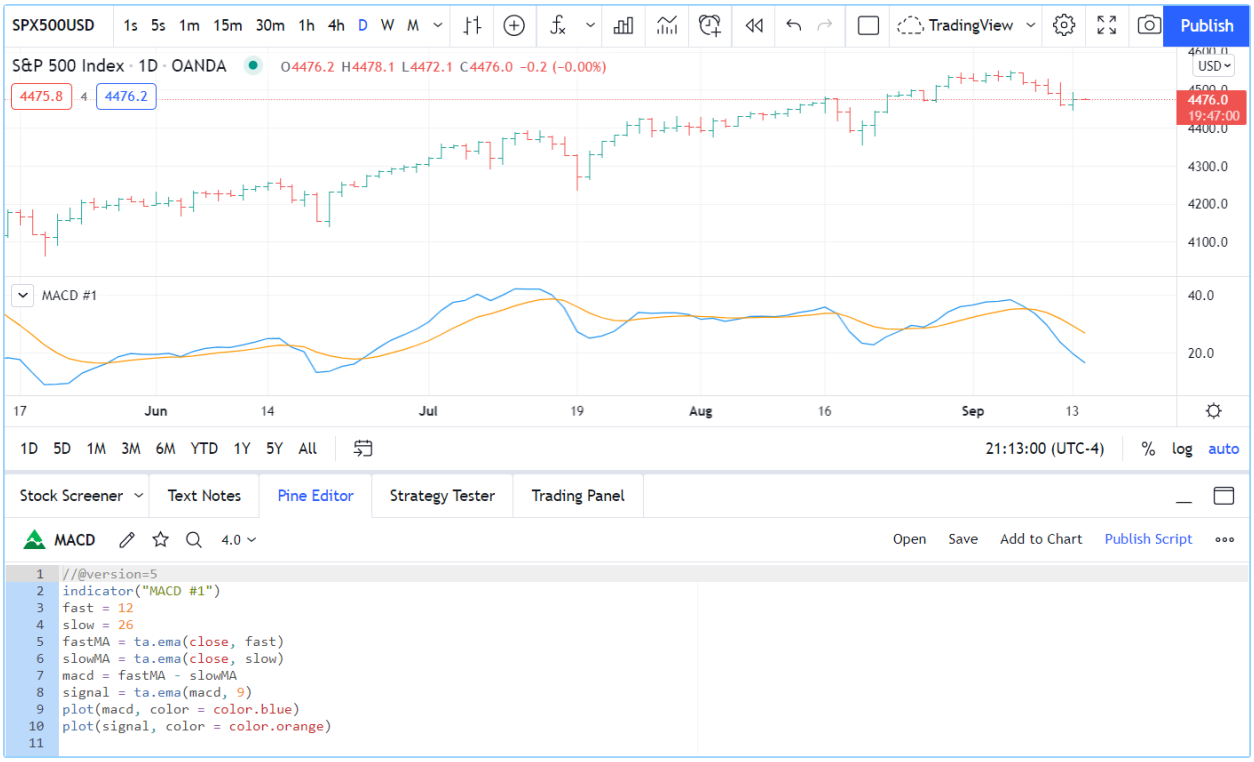
2.2.2 First version

We will now create our first working Pine script, an implementation of the MACD indicator in Pine Script™:

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

2.2.3 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:

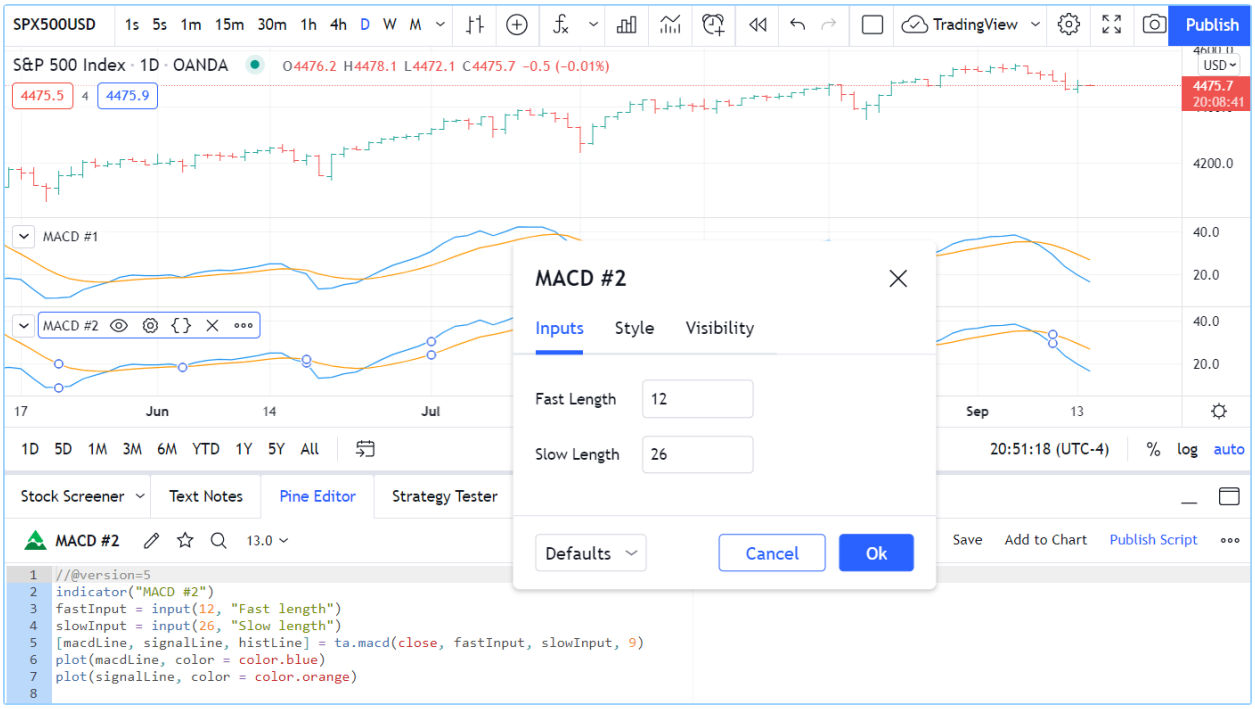
- Added inputs so we can change the lengths for the MAs
- We now use the ta.macd() built-in to calculate our MACD, which saves us three line and makes our code easier to read.

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

- Start by bringing up the "Open" dropdown menu at the top right of the Editor and choose "New 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.

2.2.4 Next

We now recommend you go to our *Next Steps* page.

2.3 Next steps

After your *first steps* and your *first indicator*, let us explore a bit more of the Pine Script™ landscape by sharing some pointers to guide you in your journey to learn Pine Script™.

2.3.1 "indicators" vs "strategies"

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

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

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

2.3.2 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.

2.3.3 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.

2.3.4 Publishing scripts

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

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

2.3.5 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 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.

2.3.6 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!

CHAPTER

THREE

LANGUAGE

3.1 Execution model

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

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

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

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

When the script first runs on a chart, all bars in a dataset are *historical bars*, except the rightmost one if a trading session is active. When trading is active on the rightmost bar, it is called the *realtime bar*. The realtime bar updates when a price

or volume change is detected. When the realtime bar closes, it becomes an *elapsed realtime bar* and a new realtime bar opens.

3.1.1 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.

3.1.2 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**.

3.1.3 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*.

3.1.4 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.

3.1.5 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
```

(continues on next page)

(continued from previous page)

```
    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.

3.2 Time series

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

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

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

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

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

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

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

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

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

When you grasp how time series can be efficiently handled using Pine Script™'s syntax and its *execution model*, you can define complex calculations using little code.

3.3 Script structure

A Pine script follows this general structure:

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

3.3.1 Version

A *compiler annotation* in the following form tells the compiler which of the versions of Pine Script™ the script is written in:

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

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

Notable changes to the current version of Pine Script™ are documented in the *Release notes*.

3.3.2 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*.

3.3.3 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 reassignement
- 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)
```

3.3.4 Comments

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

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

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

3.3.5 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)
```

3.3.6 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

```

(continues on next page)

(continued from previous page)

```
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)
```

3.4 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
```

3.5 Operators

3.5.1 Introduction

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

- Arithmetic operators
- Comparison operators
- Logical operators
- The ?: ternary operator
- The [] history-referencing operator

Other operators are used to assign values to variables:

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

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

This script will produce a compilation error:

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

The compiler will complain: *Cannot call 'ta.ema' with argument 'length'='adjustedLength'. An argument of 'series int' type was used but a 'simple int' is expected;.* This is happening because `lenInput` is an "input int" but `factor` is a "series int" (it can only be determined by looking at the value of year on each bar). The `adjustedLength` variable is

thus assigned a "series int" value. Our problem is that the Reference Manual entry for ta.ema() tells us that its `length` parameter requires a "simple" value, which is a weaker qualifier than "series", so a "series int" value is not allowed.

The solution to our conundrum requires:

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

3.5.2 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:

3.5.3 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`
```

3.5.4 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

3.5.5 `?:` 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.

3.5.6 `[ ]` history-referencing operator

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

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

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

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

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

When the market for the chart's symbol is open and the script is executing on the chart's last bar, the *realtime bar*, close returns the value of the current price. It will only contain the actual closing price of the realtime bar the last time the script is executed on that bar, when it closes.

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

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

These are all valid uses of the [] operator:

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

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

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

3.5.7 Operator precedence

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

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

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

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

3.5.8 `=` assignement operator

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

These are all valid variable declarations:

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

See the *Variable declarations* page for more information on how to declare variables.

3.5.9 `:=` reassignement operator

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

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

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

Note that:

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

The output of our script looks like this:

Note that:

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

See the *Variable reassignment* section for more information on how to reassign values to variables.

3.6 Variable declarations

3.6.1 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)
```

3.6.2 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.:

```
1 //@version=5
2 indicator("", "", true)
3 sensitivityInput = input.int(2, "Sensitivity", minval = 1, tooltip = "Higher values␣
  ↪make color changes less sensitive.")
4 ma = ta.sma(close, 20)
```

(continues on next page)

(continued from previous page)

```
maUp = ta.rising(ma, sensitivityInput)
maDn = ta.falling(ma, sensitivityInput)

// On first bar only, initialize color to gray
var maColor = color.gray
if maUp
    // MA has risen for two bars in a row; make it lime.
    maColor := color.lime
else if maDn
    // MA has fallen for two bars in a row; make it fuchsia.
    maColor := color.fuchsia

plot(ma, "MA", maColor, 2)
```

Note that:

- We initialize `maColor` on the first bar only, so it preserves its value across bars.
- On every bar, the if statement checks if the MA has been rising or falling for the user-specified number of bars (the default is 2). When that happens, the value of `maColor` must be reassigned a new value from within the if local blocks. To do this, we use the := reassignment operator.
- If we did not use the := reassignment operator, the effect would be to initialize a new `maColor` local variable which would have the same name as that of the global scope, but actually be a very confusing independent entity that would persist only for the length of the local block, and then disappear without a trace.

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

3.6.3 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)
```

(continues on next page)

(continued from previous page)

```
[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. Suppoose we want to extend the last bar's close line to the right of the right chart. We could write:

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

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

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

Note that:

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

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

`varip`

Understanding the behavior of variables using the varip declaration mode requires prior knowledge of Pine Script™'s *execution model* and *bar states*.

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

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

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

Let's look at the following code, which does not use varip:

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

plot(updateNo, style = plot.style_circles)
```

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

If we now use varip to declare the `updateNo` variable, the script behaves very differently:

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

plot(updateNo, style = plot.style_circles)
```

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

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

3.7 Conditional structures

3.7.1 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.

3.7.2 `if` structure

`if` used for its side effects

An if structure used for its side effects has the following syntax:

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

where:

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

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

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

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

When no <expression> has evaluated to true and no `else` clause exists, na is returned.

Using if structures for their side effects can be useful to manage the order flow in strategies, for example. While the same functionality can often be achieved using the `when` parameter in `strategy.*()` calls, code using if structures is easier to read:

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

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

```
//@version=5
indicator("", "", true)
var ourLabel = label.new(bar_index, na, na, color = color(na), textcolor = color.
↪orange)
if barstate.islast
    label.set_xy(ourLabel, bar_index + 2, hl2[1])
    label.set_text(ourLabel, str.tostring(bar_index + 1, "# bars in chart"))
```

Note that:

- We initialize the `ourLabel` variable on the script's first bar only, as we use the var declaration mode. The value used to initialize the variable is provided by the label.new() function call, which returns a label ID pointing to the label it creates. We use that call to set the label's properties because once set, they will persist until we change them.
- What happens next is that on each successive bar the Pine Script™ runtime will skip the initialization of `ourLabel`, and the if structure's condition (barstate.islast) is evaluated. It returns `false` on all bars until the last one, so the script does nothing on most historical bars after bar zero.
- On the last bar, barstate.islast becomes true and the structure's local block executes, modifying on each chart update the properties of our label, which displays the number of bars in the dataset.
- We want to display the label's text without a background, so we make the label's background na in the label.new() function call, and we use `hl2[1]` for the label's *y* position because we don't want it to move all the time. By using the average of the **previous** bar's high and low values, the label doesn't move until the moment when the next realtime bar opens.
- We use `bar_index + 2` in our label.set_xy() call to offset the label to the right by two bars.

`if` used to return a value

An if structure used to return one or more values has the following syntax:

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

where:

- Parts enclosed in square brackets (`[]`) can appear zero or one time, and those enclosed in curly braces (`{}`) can appear zero or more times.
- <declaration_mode> is the variable's *declaration mode*
- <type> is optional, as in almost all Pine Script™ variable declarations (see *types*)
- <identifier> is the variable's *name*
- <expression> can be a literal, a variable, an expression or a function call.
- <local_block> consists of zero or more statements followed by a return value, which can be a tuple of values. It must be indented by four spaces or a tab.
- The value assigned to the variable is the return value of the <local_block>, or na if no local block is executed.

This is an example:

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

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

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

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

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

Scripts can contain `if` structures with nested `if` and other conditional structures. For example:

```
if condition1
    if condition2
        if condition3
            expression
```

However, nesting these structures is not recommended from a performance perspective. When possible, it is typically more optimal to compose a single `if` statement with multiple logical operators rather than several nested `if` blocks:

```
if condition1 and condition2 and condition3
    expression
```

3.7.3 `switch` structure

The switch structure exists in two forms. One switches on the different values of a key expression:

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

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

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

where:

- Parts enclosed in square brackets (`[]`) can appear zero or one time, and those enclosed in curly braces (`{}`) can appear zero or more times.

- <declaration_mode> is the variable's *declaration mode*
- <type> is optional, as in almost all Pine Script™ variable declarations (see *types*)
- <identifier> is the variable's *name*
- <expression> can be a literal, a variable, an expression or a function call.
- <local_block> consists of zero or more statements followed by a return value, which can be a tuple of values. It must be indented by four spaces or a tab.
- The value assigned to the variable is the return value of the <local_block>, or na if no local block is executed.
- The `=> <local_block>` at the end allows you to specify a return value which acts as a default to be used when no other case in the structure is executed.

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

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

As with the if structure, if no local block is exectuted, na is returned.

`switch` with an expression

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

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

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

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

plot(ma)
```

Note that:

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

`switch` without an expression

This is an example of a switch structure wich does not use an exppression:

```
//@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)
```

3.7.4 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
```

(continues on next page)

(continued from previous page)

```
    close
else
    "open"
```

3.8 Loops

3.8.1 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.

3.8.2 `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
```

(continues on next page)

(continued from previous page)

```
        // Only deqeue 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 array<line> hiPivotLines = array.new_line(qtyOfPivotsInput)
var array<line> loPivotLines = array.new_line(qtyOfPivotsInput)

// Detect new pivots.
float hiPivot = ta.pivothigh(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)
```

3.8.3 `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:

```
1  //@version=5
2  indicator("`for` loop")
3  lookbackInput = input.int(50, "Lookback in bars", minval = 1, maxval = 4999)
4  higherBars = 0
5  lowerBars = 0
6  if barstate.islast
7      var label lbl = label.new(na, na, "", style = label.style_label_left)
8      // Initialize the loop counter to its start value.
9      i = 1
10     // Loop until the `i` counter's value is <= the `lookbackInput` value.
11     while i <= lookbackInput
12         if high[i] > high
13             higherBars += 1
14         else if high[i] < high
15             lowerBars += 1
16         // Counter must be managed "manually".
17         i += 1
18     label.set_xy(lbl, bar_index, high)
19     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
```

3.9 Type system

3.9.1 Introduction

The Pine Script™ type system determines the compatibility of a script's values with various functions and operations. While it's possible to write simple scripts without knowing anything about the type system, a reasonable understanding of it is necessary to achieve any degree of proficiency with the language, and an in-depth knowledge of its subtleties will allow you to harness its full potential.

Pine Script™ uses *types* to classify all values, and it uses *qualifiers* to determine whether values are constant, established on the first script iteration, or dynamic throughout a script's execution. This system applies to all Pine values, including those from literals, variables, expressions, function returns, and function arguments.

The type system closely intertwines with Pine's *execution model* and *time series* concepts. Understanding all three is essential for making the most of the power of Pine Script™.

Note: For the sake of brevity, we often use "type" to refer to a "qualified type".

3.9.2 Qualifiers

Pine Script™ *qualifiers* identify when a value is accessible in the script's execution:

- Values qualified as *const* are established at compile time (i.e., when saving the script in the Pine Editor or adding it to the chart).
- Values qualified as *input* are available at input time (i.e., when changing values in the script's "Settings/Inputs" tab).
- Values qualified as *simple* are established at bar zero (the first bar of the script's execution).
- Values qualified as *series* can change throughout the script's execution.

Pine Script™ bases the dominance of type qualifiers on the following hierarchy: **const < input < simple < series**, where "const" is the *weakest* qualifier and "series" is the *strongest*. The qualifier hierarchy translates into this rule: whenever a variable, function, or operation is compatible with a specific qualified type, values with *weaker* qualifiers are also allowed.

Scripts always qualify their expressions' returned values based on the dominant qualifier in their calculations. For example, evaluating an expression that involves "const" and "series" values will return a value qualified as "series". Furthermore, scripts cannot change a value's qualifier to one that's lower on the hierarchy. If a value acquires a stronger qualifier (e.g., a value initially inferred as "simple" becomes "series" later in the script's execution), that state is irreversible.

Note that only values qualified as "series" can change throughout the execution of a script, which include those from various built-ins, such as close and volume, as well as the results of any operations that involve "series" values. Values qualified as "const", "input", or "simple" are consistent throughout a script's execution.

const

Values qualified as "const" are established at *compile time*, before the script starts its execution. Compilation initially occurs when saving a script in the Pine Editor, which does not require it to run on a chart. Values with the "const" qualifier never change between script iterations, not even on the initial bar of its execution.

Scripts can qualify values as "const" by using a *literal* value or calculating values from expressions that only use literal values or other variables qualified as "const".

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 color*: `#FF55C6`, `#FF55C6ff`
- *literal string*: `"A text literal"`, `"Embedded single quotes 'text'"`, `'Embedded double quotes "text"'`

Users can explicitly define variables and parameters that only accept "const" values by including the `const` keyword in their declaration.

Our *Style guide* recommends using uppercase SNAKE_CASE to name "const" variables for readability. While it is not a requirement, one can also use the var keyword when declaring "const" variables so the script only initializes them on the *first bar* of the dataset. See *this section* of our User Manual for more information.

Below is an example that uses "const" values within indicator() and plot() functions, which both require a value of the "const string" qualified type as their `title` argument:

```
//@version=5

// The following global variables are all of the "const string" qualified type:

//@variable The title of the indicator.
INDICATOR_TITLE = "const demo"
//@variable The title of the first plot.
var PLOT1_TITLE = "High"
//@variable The title of the second plot.
const string PLOT2_TITLE = "Low"
//@variable The title of the third plot.
PLOT3_TITLE = "Midpoint between " + PLOT1_TITLE + " and " + PLOT2_TITLE

indicator(INDICATOR_TITLE, overlay = true)

plot(high, PLOT1_TITLE)
plot(low, PLOT2_TITLE)
plot(hl2, PLOT3_TITLE)
```

The following example will raise a compilation error since it uses syminfo.ticker, which returns a "simple" value because it depends on chart information that's only accessible once the script starts its execution:

```
//@version=5

//@variable The title in the `indicator()` call.
var NAME = "My indicator for " + syminfo.ticker

indicator(NAME, "", true) // Causes an error because `NAME` is qualified as a "simple string".
plot(close)
```

input

Values qualified as "input" are established after initialization via the `input.*()` functions. These functions produce values that users can modify within the "Inputs" tab of the script's settings. When one changes any of the values in this tab, the script re-executes from the beginning of the chart's history to ensure its input values are consistent throughout its execution.

Note: The input.source() function is an exception in the `input.*()` namespace, as it returns "series" qualified values rather than "input" since built-in variables such as open, close, etc., as well as the values from another script's plots, are qualified as "series".

The following script plots the value of a `sourceInput` from the `symbolInput` and `timeframeInput` context. The request.security() call is valid in this script since its `symbol` and `timeframe` parameters allow "simple string" arguments, meaning they can also accept "input string" values because the "input" qualifier is *lower* on the hierarchy:

```
//@version=5
indicator("input demo", overlay = true)

//@variable The symbol to request data from. Qualified as "input string".
symbolInput = input.symbol("AAPL", "Symbol")
//@variable The timeframe of the data request. Qualified as "input string".
timeframeInput = input.timeframe("D", "Timeframe")
//@variable The source of the calculation. Qualified as "series float".
sourceInput = input.source(close, "Source")

//@variable The `sourceInput` value from the requested context. Qualified as "series float".
requestedSource = request.security(symbolInput, timeframeInput, sourceInput)

plot(requestedSource)
```

simple

Values qualified as "simple" are available only once the script begins execution on the *first* chart bar of its history, and they remain consistent during the script's execution.

Users can explicitly define variables and parameters that accept "simple" values by including the `simple` keyword in their declaration.

Many built-in variables return "simple" qualified values because they depend on information that a script can only obtain once it starts its execution. Additionally, many built-in functions require "simple" arguments that do not change over time. Wherever a script allows "simple" values, it can also accept values qualified as "input" or "const".

This script highlights the background to warn users that they're using a non-standard chart type. It uses the value of chart.is_standard to calculate the `isNonStandard` variable, then uses that variable's value to calculate a `warningColor` that also references a "simple" value. The `color` parameter of bgcolor() allows a "series color" argument, meaning it can also accept a "simple color" value since "simple" is lower on the hierarchy:

```
//@version=5
indicator("simple demo", overlay = true)

//@variable Is `true` when the current chart is non-standard. Qualified as "simple bool".
isNonStandard = not chart.is_standard
//@variable Is orange when the the current chart is non-standard. Qualified as "simple color".
simple color warningColor = isNonStandard ? color.new(color.orange, 70) : na

// Colors the chart's background to warn that it's a non-standard chart type.
bgcolor(warningColor, title = "Non-standard chart color")
```

series

Values qualified as "series" provide the most flexibility in scripts since they can change on any bar, even multiple times on the same bar.

Users can explicitly define variables and parameters that accept "series" values by including the `series` keyword in their declaration.

Built-in variables such as open, high, low, close, volume, time, and bar_index, and the result from any expression using such built-ins, are qualified as "series". The result of any function or operation that returns a dynamic value will always be a "series", as will the results from using the history-referencing operator [] to access historical values. Wherever a script allows "series" values, it will also accept values with any other qualifier, as "series" is the *highest* qualifier on the hierarchy.

This script displays the highest and lowest value of a `sourceInput` over `lengthInput` bars. The values assigned to the `highest` and `lowest` variables are of the "series float" qualified type, as they can change throughout the script's execution:

```
//@version=5
indicator("series demo", overlay = true)

//@variable The source value to calculate on. Qualified as "series float".
series float sourceInput = input.source(close, "Source")
//@variable The number of bars in the calculation. Qualified as "input int".
lengthInput = input.int(20, "Length")

//@variable The highest `sourceInput` value over `lengthInput` bars. Qualified as "series float".
series float highest = ta.highest(sourceInput, lengthInput)
//@variable The lowest `sourceInput` value over `lengthInput` bars. Qualified as "series float".
lowest = ta.lowest(sourceInput, lengthInput)

plot(highest, "Highest source", color.green)
plot(lowest, "Lowest source", color.red)
```

3.9.3 Types

Pine Script™ *types* classify values and determine the functions and operations they're compatible with. They include:

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

Fundamental types refer to the underlying nature of a value, e.g., a value of 1 is of the "int" type, 1.0 is of the "float" type, "AAPL" is of the "string" type, etc. Special types and user-defined types utilize *IDs* that refer to objects of a specific class. For example, a value of the "label" type contains an ID that acts as a *pointer* referring to a "label" object. The "void" type refers to the output from a function or *method* that does not return a usable value.

Pine Script™ can automatically convert values from some types into others. The auto-casting rules are: **int → float → bool**. See the *Type casting* section of this page for more information.

In most cases, Pine Script™ can automatically determine a value's type. However, we can also use type keywords to *explicitly* specify types for readability and for code that requires explicit definitions (e.g., declaring a variable assigned to na). For example:

```
//@version=5
indicator("Types demo", overlay = true)

//@variable A value of the "const string" type for the `ma` plot's title.
string MA_TITLE = "MA"

//@variable A value of the "input int" type. Controls the length of the average.
int lengthInput = input.int(100, "Length", minval = 2)

//@variable A "series float" value representing the last `close` that crossed over the `ma`.
var float crossValue = na

//@variable A "series float" value representing the moving average of `close`.
float ma = ta.sma(close, lengthInput)
//@variable A "series bool" value that's `true` when the `close` crosses over the `ma`.
bool crossUp = ta.crossover(close, ma)
//@variable A "series color" value based on whether `close` is above or below its `ma`.
color maColor = close > ma ? color.lime : color.fuchsia

// Update the `crossValue`.
if crossUp
    crossValue := close

plot(ma, MA_TITLE, maColor)
plot(crossValue, "Cross value", style = plot.style_circles)
plotchar(crossUp, "Cross Up", "▲", location.belowbar, size = size.small)
```

int

Values of the "int" type represent integers, i.e., whole numbers without any fractional quantities.

Integer literals are numeric values written in *decimal* notation. For example:

```
1
-1
750
```

Built-in variables such as bar_index, time, timenow, dayofmonth, and strategy.wintrades all return values of the "int" type.

float

Values of the "float" type represent floating-point numbers, i.e., numbers that can contain whole and fractional quantities.

Floating-point literals are numeric values written with a `.` delimiter. They may also contain the symbol `e` or `E` (which means "10 raised to the power of X", where X is the number after the `e` or `E` symbol). For example:

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

The internal precision of "float" values in Pine Script™ is 1e-16.

Built-in variables such as close, hlcc4, volume, ta.vwap, and strategy.position_size all return values of the "float" type.

bool

Values of the "bool" type represent the truth value of a comparison or condition, which scripts can use in *conditional structures* and other expressions.

There are only two literals that represent boolean values:

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

When an expression of the "bool" type returns na, scripts treat its value as `false` when evaluating conditional statements and operators.

Built-in variables such as barstate.isfirst, chart.is_heikinashi, session.ismarket, and timeframe.isdaily all return values of the "bool" type.

color

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

- `RR`, `GG` and `BB` pairs respectively represent the values for the color's red, green and blue components.
- `AA` is an optional value for the color's opacity (or *alpha* component) where `00` is invisible and `FF` opaque. When the literal does not include an `AA` pair, the script treats it as fully opaque (the same as using `FF`).
- The hexadecimal letters in the literals can be uppercase or lowercase.

These are examples of "color" literals:

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

Pine Script™ also has *built-in color constants*, including color.green, color.red, color.orange, color.blue (the default color in `plot*()` functions and many of the default color-related properties in *drawing types*), etc.

When using built-in color constants, it is possible to add transparency information to them via the color.new() function.

Note that when specifying red, green or blue components in `color.*()` functions, we use "int" or "float" arguments with values between 0 and 255. When specifying transparency, we use a value between 0 and 100, where 0 means fully opaque and 100 means completely transparent. For example:

```
1  //@version=5
2  indicator("Shading the chart's background", overlay = true)
3
4  //@variable A "const color" value representing the base for each day's color.
5  color BASE_COLOR = color.rgb(0, 99, 165)
6
7  //@variable A "series int" value that modifies the transparency of the `BASE_COLOR`␣
   ↪in `color.new()`.
8  int transparency = 50 + int(40 * dayofweek / 7)
9
10 // Color the background using the modified `BASE_COLOR`.
11 bgcolor(color.new(BASE_COLOR, transparency))
```

See the User Manual's page on *colors* for more information on using colors in scripts.

string

Values of the "string" type represent sequences of letters, numbers, symbols, spaces, and other characters.

String literals in Pine are characters enclosed in single or double quotation marks. For example:

```
"This is a string literal using double quotes."
'This is a string literal using single quotes.'
```

Single and double quotation marks are functionally equivalent in Pine Script™. A "string" enclosed within double quotation marks can contain any number of single quotation marks and vice versa:

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

Scripts can *escape* the enclosing delimiter in a "string" using the backslash character (\). For example:

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

We can create "string" values containing the new line escape character (`\n`) for displaying multi-line text with `plot*()` and `log.*()` functions and objects of *drawing types*. For example:

```
"This\nString\nHas\nOne\nWord\nPer\nLine"
```

We can use the + operator to concatenate "string" values:

```
"This is a " + "concatenated string."
```

The built-ins in the `str.*()` namespace create "string" values using specialized operations. For instance, this script creates a *formatted string* to represent "float" price values and displays the result using a label:

```
//@version=5
indicator("Formatted string demo", overlay = true)

//@variable A "series string" value representing the bar's OHLC data.
string ohlcString = str.format("Open: {0}\nHigh: {1}\nLow: {2}\nClose: {3}", open,␣
↪high, low, close)

// Draw a label containing the `ohlcString`.
label.new(bar_index, high, ohlcString, textcolor = color.white)
```

See our User Manual's page on *Text and shapes* for more information about displaying "string" values from a script.

Built-in variables such as syminfo.tickerid, syminfo.currency, and timeframe.period return values of the "string" type.

plot and hline

Pine Script™'s plot() and hline() functions return IDs that respectively reference instances of the "plot" and "hline" types. These types display calculated values and horizontal levels on the chart, and one can assign their IDs to variables for use with the built-in fill() function.

For example, this script plots two EMAs on the chart and fills the space between them using the fill() function:

```
//@version=5
indicator("plot fill demo", overlay = true)

//@variable A "series float" value representing a 10-bar EMA of `close`.
float emaFast = ta.ema(close, 10)
//@variable A "series float" value representing a 20-bar EMA of `close`.
float emaSlow = ta.ema(close, 20)

//@variable The plot of the `emaFast` value.
emaFastPlot = plot(emaFast, "Fast EMA", color.orange, 3)
//@variable The plot of the `emaSlow` value.
emaSlowPlot = plot(emaSlow, "Slow EMA", color.gray, 3)

// Fill the space between the `emaFastPlot` and `emaSlowPlot`.
fill(emaFastPlot, emaSlowPlot, color.new(color.purple, 50), "EMA Fill")
```

It's important to note that unlike other special types, there is no `plot` or `hline` keyword in Pine to explicitly declare a variable's type as "plot" or "hline".

Users can control where their scripts' plots display via the variables in the `display.*` namespace. Additionally, one script can use the values from another script's plots as *external inputs* via the input.source() function (see our User Manual's section on *source inputs*).

Drawing types

Pine Script™ drawing types allow scripts to create custom drawings on charts. They include the following: line, linefill, box, polyline, label, and table.

Each type also has a namespace containing all the built-ins that create and manage drawing instances. For example, the following `*.new()` constructors create new objects of these types in a script: line.new(), linefill.new(), box.new(), polyline.new(), label.new(), and table.new().

Each of these functions returns an *ID* which is a reference that uniquely identifies a drawing object. IDs are always qualified as "series", meaning their qualified types are "series line", "series label", etc. Drawing IDs act like pointers, as each ID references a specific instance of a drawing in all the functions from that drawing's namespace. For instance, the ID of a line returned by a line.new() call is used later to refer to that specific object once it's time to delete it with line.delete().

Chart points

Chart points are special types that represent coordinates on the chart. Scripts use the information from chart.point objects to determine the chart locations of *lines*, *boxes*, *polylines*, and *labels*.

Objects of this type contain three *fields*: `time`, `index`, and `price`. Whether a drawing instance uses the `time` or `price` field from a chart.point as an x-coordinate depends on the drawing's `xloc` property.

We can use any of the following functions to create chart points in a script:

- chart.point.new() - Creates a new chart.point with a specified `time`, `index`, and `price`.
- chart.point.now() - Creates a new chart.point with a specified `price` y-coordinate. The `time` and `index` fields contain the time and bar_index of the bar the function executes on.
- chart.point_from_index() - Creates a new chart.point with an `index` x-coordinate and `price` y-coordinate. The `time` field of the resulting instance is na, meaning it will not work with drawing objects that use an `xloc` value of xloc.bar_time.
- chart.point.from_time() - Creates a new chart.point with a `time` x-coordinate and `price` y-coordinate. The `index` field of the resulting instance is na, meaning it will not work with drawing objects that use an `xloc` value of xloc.bar_index.
- chart.point.copy() - Creates a new chart.point containing the same `time`, `index`, and `price` information as the `id` in the function call.

This example draws lines connecting the previous bar's high to the current bar's low on each chart bar. It also displays labels at both points of each line. The line and labels get their information from the `firstPoint` and `secondPoint` variables, which reference chart points created using chart.point_from_index() and chart.point.now():

```
1  //@version=5
2  indicator("Chart points demo", overlay = true)
3
4  //@variable A new `chart.point` at the previous `bar_index` and `high`.
5  firstPoint = chart.point.from_index(bar_index - 1, high[1])
6  //@variable A new `chart.point` at the current bar's `low`.
7  secondPoint = chart.point.now(low)
8
9  // Draw a new line connecting coordinates from the `firstPoint` and `secondPoint`.
10 // This line uses the `index` fields from the points as x-coordinates.
11 line.new(firstPoint, secondPoint, color = color.purple, width = 3)
12 // Draw a label at the `firstPoint`. Uses the point's `index` field as its x-
   ↪coordinate.
13 label.new(
14     firstPoint, str.tostring(firstPoint.price), color = color.green,
```

(continues on next page)

(continued from previous page)

```
     style = label.style_label_down, textcolor = color.white
 )
// Draw a label at the `secondPoint`. Uses the point's `index` field as its x-
↪coordinate.
label.new(
     secondPoint, str.tostring(secondPoint.price), color = color.red,
     style = label.style_label_up, textcolor = color.white
 )
```

Collections

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

- `array<int>` defines an array containing "int" elements.
- `array<label>` defines an array containing "label" IDs.
- `array<UDT>` defines an array containing IDs referencing objects of a *user-defined type (UDT)*.
- `matrix<float>` defines a matrix containing "float" elements.
- `matrix<UDT>` defines a matrix containing IDs referencing objects of a *user-defined type (UDT)*.
- `map<string, float>` defines a map containing "string" keys and "float" values.
- `map<int, UDT>` defines a map containing "int" keys and IDs of *user-defined type (UDT)* instances as values.

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

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

Note that:

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

User-defined types

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

```
[export] type <UDT_identifier>
    <field_type> <field_name> [= <value>]
    ...
```

where:

- `export` is the keyword that a library script uses to export the user-defined type. To learn more about exporting UDTs, see our User Manual's *Libraries* page.

- <UDT_identifier> is the name of the user-defined type.
- <field_type> is the type of the field.
- <field_name> is the name of the field.
- <value> is an optional default value for the field, which the script will assign to it when creating new objects of that UDT. If one does not provide a value, the field's default is na. The same rules as those governing the default values of parameters in function signatures apply to the default values of fields. For example, a UDT's default values cannot use results from the history-referencing operator [] or expressions.

This example declares a pivotPoint UDT with an "int" pivotTime field and a "float" priceLevel field that will respectively hold time and price information about a calculated pivot:

```
//@type             A user-defined type containing pivot information.
//@field pivotTime  Contains time information about the pivot.
//@field priceLevel Contains price information about the pivot.
type pivotPoint
    int   pivotTime
    float priceLevel
```

User-defined types support *type recursion*, i.e., the fields of a UDT can reference objects of the same UDT. Here, we've added a nextPivot field to our previous pivotPoint type that references another pivotPoint instance:

```
//@type             A user-defined type containing pivot information.
//@field pivotTime  Contains time information about the pivot.
//@field priceLevel Contains price information about the pivot.
//@field nextPivot  A `pivotPoint` instance containing additional pivot information.
type pivotPoint
    int        pivotTime
    float      priceLevel
    pivotPoint nextPivot
```

Scripts can use two built-in methods to create and copy UDTs: new() and copy(). See our User Manual's page on *Objects* to learn more about working with UDTs.

void

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

Scripts cannot use "void" results in expressions or assign them to variables. No void keyword exists in Pine Script™ since one cannot declare a variable of the "void" type.

3.9.4 `na` value

There is a special value in Pine Script™ called na, which is an acronym for *not available*. We use na to represent an undefined value from a variable or expression. It is similar to null in Java and None in Python.

Scripts can automatically cast na values to almost any type. However, in some cases, the compiler cannot infer the type associated with an na value because more than one type-casting rule may apply. For example:

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

The above line of code causes a compilation error because the compiler cannot determine the nature of the `myVar` variable, i.e., whether the variable will reference numeric values for plotting, string values for setting text in a label, or other values for some other purpose later in the script's execution.

To resolve such errors, we must explicitly declare the type associated with the variable. Suppose the `myVar` variable will reference "float" values in subsequent script iterations. We can resolve the error by declaring the variable with the float keyword:

```
float myVar = na
```

or by explicitly casting the na value to the "float" type via the float() function:

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

To test if the value from a variable or expression is na, we call the na() function, which returns `true` if the value is undefined. For example:

```
//@variable Is 0 if the `myVar` is `na`, `close` otherwise.
float myClose = na(myVar) ? 0 : close
```

Do not use the == comparison operator to test for na values, as scripts cannot determine the equality of an undefined value:

```
//@variable Returns the `close` value. The script cannot compare the equality of `na`␣
↪values, as they're undefined.
float myClose = myVar == na ? 0 : close
```

Best coding practices often involve handling na values to prevent undefined values in calculations.

For example, this line of code checks if the close value on the current bar is greater than the previous bar's value:

```
//@variable Is `true` when the `close` exceeds the last bar's `close`, `false`␣
↪otherwise.
bool risingClose = close > close[1]
```

On the first chart bar, the value of `risingClose` is na since there is no past close value to reference.

We can ensure the expression also returns an actionable value on the first bar by replacing the undefined past value with a value from the current bar. This line of code uses the nz() function to replace the past bar's close with the current bar's open when the value is na:

```
//@variable Is `true` when the `close` exceeds the last bar's `close` (or the current␣
↪`open` if the value is `na`).
bool risingClose = close > nz(close[1], open)
```

Protecting scripts against na instances helps to prevent undefined values from propagating in a calculation's results. For example, this script declares an `allTimeHigh` variable on the first bar. It then uses the math.max() between the `allTimeHigh` and the bar's high to update the `allTimeHigh` throughout its execution:

```
1 //@version=5
2 indicator("na protection demo", overlay = true)
3
4 //@variable The result of calculating the all-time high price with an initial value␣
  ↪of `na`.
5 var float allTimeHigh = na
6
7 // Reassign the value of the `allTimeHigh`.
8 // Returns `na` on all bars because `math.max()` can't compare the `high` to an␣
```

(continues on next page)

(continued from previous page)

```
↪undefined value.
allTimeHigh := math.max(allTimeHigh, high)

plot(allTimeHigh) // Plots `na` on all bars.
```

This script plots a value of na on all bars, as we have not included any na protection in the code. To fix the behavior and plot the intended result (i.e., the all-time high of the chart's prices), we can use nz() to replace na values in the `allTimeHigh` series:

```
//@version=5
indicator("na protection demo", overlay = true)

//@variable The result of calculating the all-time high price with an initial value␣
↪of `na`.
var float allTimeHigh = na

// Reassign the value of the `allTimeHigh`.
// We've used `nz()` to prevent the initial `na` value from persisting throughout the␣
↪calculation.
allTimeHigh := math.max(nz(allTimeHigh), high)

plot(allTimeHigh)
```

3.9.5 Type templates

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

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

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

Users can construct type templates from:

- Fundamental types: *int*, *float*, *bool*, *color*, and *string*
- The following special types: *line*, *linefill*, *box*, *polyline*, *label*, *table*, and *chart.point*
- *User-defined types (UDTs)*

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>()
```

3.9.6 Type casting

Pine Script™ includes an automatic type-casting mechanism that *casts* (converts) **"int"** values to **"float"** when necessary. Variables or expressions requiring "float" values can also use "int" values because any integer can be represented as a floating point number with its fractional part equal to 0.

For the sake of backward compatibility, Pine Script™ also automatically casts **"int"** and **"float"** values to **"bool"** when necessary. When passing numeric values to the parameters of functions and operations that expect "bool" types, Pine auto-casts them to "bool". However, we do not recommend relying on this behavior. Most scripts that automatically cast numeric values to the "bool" type will produce a *compiler warning*. One can avoid the compiler warning and promote code readability by using the bool() function, which explicitly casts a numeric value to the "bool" type.

When casting an "int" or "float" to "bool", a value of 0 converts to `false` and any other numeric value always converts to `true`.

This code below demonstrates deprecated auto-casting behavior in Pine. It creates a `randomValue` variable with a "series float" value on every bar, which it passes to the `condition` parameter in an if structure and the `series` parameter in a plotchar() function call. Since both parameters accept "bool" values, the script automatically casts the `randomValue` to "bool" when evaluating them:

```
//@version=5
indicator("Auto-casting demo", overlay = true)

//@variable A random rounded value between -1 and 1.
float randomValue = math.round(math.random(-1, 1))
//@variable The color of the chart background.
color bgColor = na

// This raises a compiler warning since `randomValue` is a "float", but `if` expects a "bool".
if randomValue
    bgColor := color.new(color.blue, 60)
// This does not raise a warning, as the `bool()` function explicitly casts the `randomValue` to "bool".
if bool(randomValue)
    bgColor := color.new(color.blue, 60)

// Display unicode characters on the chart based on the `randomValue`.
// Whenever `math.random()` returns 0, no character will appear on the chart because 0 converts to `false`.
plotchar(randomValue)
// We recommend explicitly casting the number with the `bool()` function to make the type transformation more obvious.
plotchar(bool(randomValue))

// Highlight the background with the `bgColor`.
bgcolor(bgColor)
```

It's sometimes necessary to cast one type to another when auto-casting rules do not suffice. For such cases, the following type-casting functions are available: int(), float(), bool(), color(), string(), line(), linefill(), label(), box(), and table().

The example below shows a code that tries to use a "const float" value as the `length` argument in the ta.sma() function call. The script will fail to compile, as it cannot automatically convert the "float" value to the required "int" type:

```
//@version=5
indicator("Explicit casting demo", overlay = true)

//@variable The length of the SMA calculation. Qualified as "const float".
```

(continues on next page)

(continued from previous page)

```
float LENGTH = 10.0

float sma = ta.sma(close, LENGTH) // Compilation error. The `length` parameter
↪requires an "int" value.

plot(sma)
```

The code raises the following error: *"Cannot call 'ta.sma' with argument 'length'='LENGTH'. An argument of 'const float' type was used but a 'series int' is expected."*

The compiler is telling us that the code is using a "float" value where an "int" is required. There is no auto-casting rule to cast a "float" to an "int", so we must do the job ourselves. In this version of the code, we've used the int() function to explicitly convert our "float" `LENGTH` value to the "int" type within the ta.sma() call:

```
//@version=5
indicator("explicit casting demo")

//@variable The length of the SMA calculation. Qualified as "const float".
float LENGTH = 10.0

float sma = ta.sma(close, int(LENGTH)) // Compiles successfully since we've converted
↪the `LENGTH` to "int".

plot(sma)
```

Explicit type casting is also handy when declaring variables assigned to na, as explained in the *previous section*.

For example, once could explicitly declare a variable with a value of na as a "label" type in either of the following, equivalent ways:

```
// Explicitly specify that the variable references "label" objects:
label myLabel = na

// Explicitly cast the `na` value to the "label" type:
myLabel = label(na)
```

3.9.7 Tuples

A *tuple* is a comma-separated set of expressions enclosed in brackets. When a function, *method*, or other local block returns more than one value, scripts return those values in the form of a tuple.

For example, the following *user-defined function* returns the sum and product of two "float" values:

```
//@function Calculates the sum and product of two values.
calcSumAndProduct(float a, float b) =>
    //@variable The sum of `a` and `b`.
    float sum = a + b
    //@variable The product of `a` and `b`.
    float product = a * b
    // Return a tuple containing the `sum` and `product`.
    [sum, product]
```

When we call this function later in the script, we use a *tuple declaration* to declare multiple variables corresponding to the values returned by the function call:

```
// Declare a tuple containing the sum and product of the `high` and `low`,␣
↪respectively.
[hlSum, hlProduct] = calcSumAndProduct(high, low)
```

Keep in mind that unlike declaring single variables, we cannot explicitly define the types the tuple's variables (`hlSum` and `hlProduct` in this case), will contain. The compiler automatically infers the types associated with the variables in a tuple.

In the above example, the resulting tuple contains values of the same type ("float"). However, it's important to note that tuples can contain values of *multiple types*. For example, the `chartInfo()` function below returns a tuple containing "int", "float", "bool", "color", and "string" values:

```
//@function Returns information about the current chart.
chartInfo() =>
    //@variable The first visible bar's UNIX time value.
    int firstVisibleTime = chart.left_visible_bar_time
    //@variable The `close` value at the `firstVisibleTime`.
    float firstVisibleClose = ta.valuewhen(ta.cross(time, firstVisibleTime), close, 0)
    //@variable Is `true` when using a standard chart type, `false` otherwise.
    bool isStandard = chart.is_standard
    //@variable The foreground color of the chart.
    color fgColor = chart.fg_color
    //@variable The ticker ID of the current chart.
    string symbol = syminfo.tickerid
    // Return a tuple containing the values.
    [firstVisibleTime, firstVisibleClose, isStandard, fgColor, symbol]
```

Tuples are especially handy for requesting multiple values in one request.security() call.

For instance, this `roundedOHLC()` function returns a tuple containing OHLC values rounded to the nearest prices that are divisible by the symbol's minimum tick value. We call this function as the `expression` argument in request.security() to request a tuple containing daily OHLC values:

```
//@function Returns a tuple of OHLC values, rounded to the nearest tick.
roundedOHLC() =>
    [math.round_to_mintick(open), math.round_to_mintick(high), math.round_to_
↪mintick(low), math.round_to_mintick(close)]

[op, hi, lo, cl] = request.security(syminfo.tickerid, "D", roundedOHLC())
```

We can also achieve the same result by directly passing a tuple of rounded values as the `expression` in the request.security() call:

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

Local blocks of *conditional structures*, including if and switch statements, can return tuples. For example:

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

and:

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

However, ternaries cannot contain tuples, as the return values in a ternary statement are not considered local blocks:

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

Note that all items within a tuple returned from a function are qualified as "simple" or "series", depending on its contents. If a tuple contains a "series" value, all other elements within the tuple will also adopt the "series" qualifier. For example:

```
//@version=5
indicator("Qualified types in tuples demo")

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

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

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

plot(r)
```

3.10 Built-ins

3.10.1 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`.

3.10.2 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.

3.10.3 Built-in functions

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

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

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

- Functions used as a declaration statement defining one of three types of Pine scripts, and its properties. Each script must begin with a call to one of these functions: indicator(), strategy() or library().
- Plotting or coloring functions: bgcolor(), plotbar(), plotcandle(), plotchar(), plotshape(), fill().
- Strategy functions placing orders, in the `strategy` namespace: strategy.cancel(), strategy.close(), strategy.entry(), strategy.exit(), strategy.order(), etc.
- Strategy functions returning information on indivdual past trades, in the `strategy` namespace: strategy.closedtrades.entry_bar_index(), strategy.closedtrades.entry_price(), strategy.closedtrades.entry_time(), strategy.closedtrades.exit_bar_index(), strategy.closedtrades.max_drawdown(), strategy.closedtrades.max_runup(), strategy.closedtrades.profit(), etc.
- Functions to generate alert events: alert() and alertcondition().

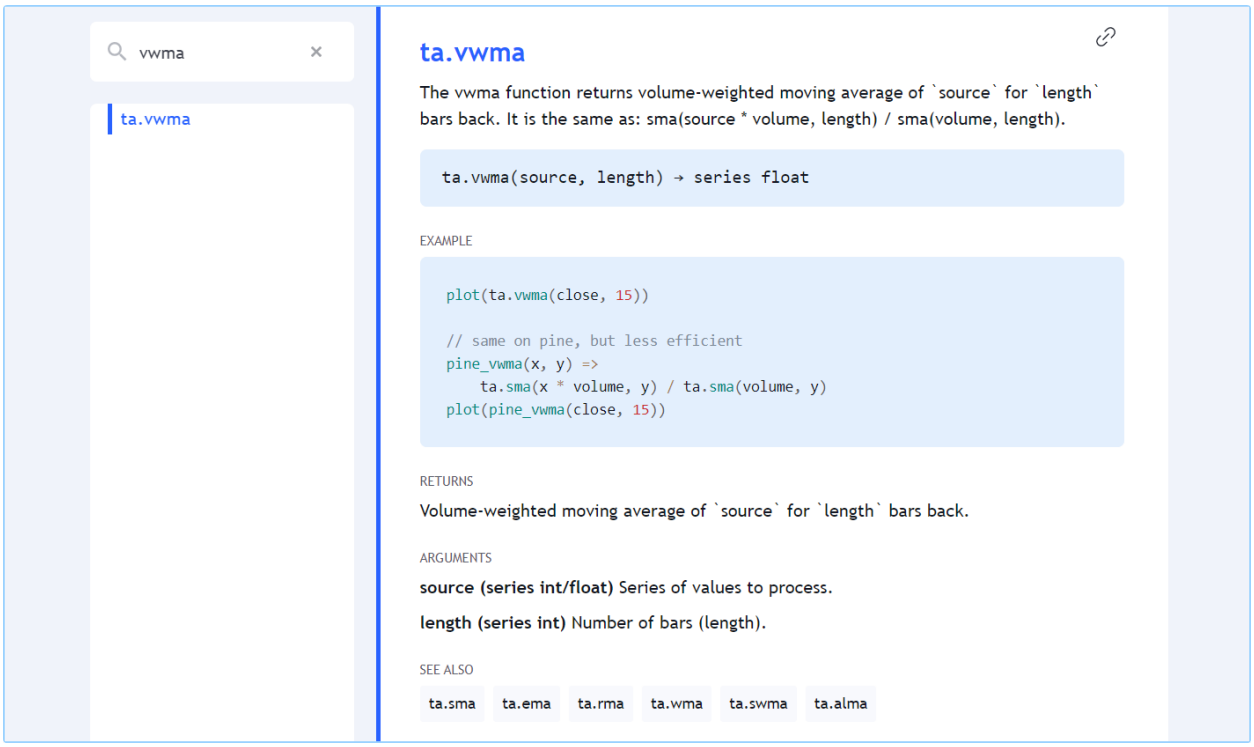
Other functions return a result, but we don't always use it, e.g.: hline(), plot(), array.pop(), label.new(), etc.

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

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

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

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



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

- What the function does.
- Its signature (or definition):

```
ta.vwma(source, length) → series float
```

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

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

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

Note that:

- We use the built-in variable close as the argument for the `source` parameter.
- We use `20` as the argument for the `length` parameter.

- If placed in the global scope (i.e., starting in a line's first position), it will be executed by the Pine Script™ runtime on each bar of the chart.

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

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

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

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

Mixing things up this way is not allowed:

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

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

To learn how to do this, one needs to understand Pine Script™'s *type system*. The Reference Manual entry for each built-in function includes an "ARGUMENTS" section which lists the qualified type required for the argument supplied to each of the function's parameters.

3.11 User-defined functions

3.11.1 Introduction

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

You can write your functions in two ways:

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

Functions can be located in two places:

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

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

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

3.11.2 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.

3.11.3 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))`.

3.11.4 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.

3.11.5 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)
```

3.11.6 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().

TradingView

Advanced

Pine Script™

3.12 Objects

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.

3.12.1 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.

3.12.2 Creating objects

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

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

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

Note that:

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

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

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

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

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

Or the equivalent:

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

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

Object placeholders can also be created by declaring na object names using the following:

```
pivotPoint foundPoint = na
```

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

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

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

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

Take note of this line from the above example:

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

This could also be written using the following:

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

When 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)
```

3.12.3 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
```

3.12.4 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:


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

(continues on next page)

(continued from previous page)

```
int legsInput = input(10)

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

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

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

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

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

3.12.5 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
```

(continues on next page)

(continued from previous page)

```
pivot2.x := 2000
// Both plot the value 2000.
plot(pivot1.x)
plot(pivot2.x)
```

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

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

```
//@version=5
indicator("")
type pivotPoint
    int x
    float y
pivot1 = pivotPoint.new()
pivot1.x := 1000
pivot2 = pivotPoint.copy(pivot1)
pivot2.x := 2000
// Plots 1000 and 2000.
plot(pivot1.x)
plot(pivot2.x)
```

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

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

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

type InfoLabel
    string info
    label  lbl

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

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

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

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

In this example, we have defined a `deepCopy()` method that instantiates a new `InfoLabel` object with its `lbl` field

pointing to a copy of the original's field. Changes to the `deep` copy's `lbl` field will not affect the `parent` object, as it points to a separate instance:

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

type InfoLabel
    string info
    label  lbl

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

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

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

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

3.12.6 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.

3.13 Methods

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.

3.13.1 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*.

3.13.2 Built-in methods

Pine Script™ includes built-in methods for all *special types*, including array, matrix, map, line, linefill, box, polyline, label, and table. These methods provide users with a more concise way to call specialized routines for these types within their scripts.

When using these special types, the expressions

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

and

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

are equivalent. For example, rather than using

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

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

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

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

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

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


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

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

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

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

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

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

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

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

float sourceInput  = input.source(close, "Source")
int   samplesInput = input.int(20, "Samples")
```

(continues on next page)

(continued from previous page)

```
int   n            = input.int(10, "Bars")
float multiplier   = input.float(2.0, "StdDev")

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

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

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

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

Note that:

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

3.13.3 User-defined methods

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

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

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

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

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

```

(continues on next page)

(continued from previous page)

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

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

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

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

Note that:

- Just as with user-defined functions, we use the `@function` *compiler annotation* to document method descriptions.

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

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

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

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

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

```
// @function         Computes Bollinger Band values from an array of data.
// @param srcArray   (array<float>) The array where the queue is maintained.
// @param multiplier (float) Standard deviaiton multiplier.
// @param calcuate   (bool) The method will only calculate new values when this is true.
```

(continues on next page)

(continued from previous page)

```
// @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

```

(continues on next page)

(continued from previous page)

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

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

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

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

3.13.4 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"

```

(continues on next page)

(continued from previous page)

```
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.

3.13.5 Advanced example

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

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

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

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

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

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

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

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

Note that:

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

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

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

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

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

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

Note that:

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

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


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

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

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

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

(continues on next page)

(continued from previous page)

```
            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.
```

(continues on next page)

(continued from previous page)

```
array<float> distArray = sourceArray.maintainQueue(sourceInput).featureScale().eCDF(n)
// Draw label.
makeLabel(sourceArray, distArray)
```

TradingView

Advanced

Pine Script™

3.14 Arrays

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.

3.14.1 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 qualified as "series". 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 "series" values.

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.

3.14.2 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:

```
1  //@version=5
2  indicator("Using `var`")
3  //@variable An array that expands its size by 1 on each bar.
4  var a = array.new<float>(0)
5  array.push(a, close)
6
7  if barstate.islast
8      //@variable A string containing the size of `a` and the current `bar_index` value.
9      string labelText = "Array size: " + str.tostring(a.size()) + "\nbar_index: " +␣
   ↪str.tostring(bar_index)
10     // Display the `labelText`.
11     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.

3.14.3 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:

```
1  //@version=5
2  indicator("Using `var`")
3  FILL_COLOR = color.green
4  var array<color> fillColors = array.from(
5       color.new(FILL_COLOR, 70),
6       color.new(FILL_COLOR, 75),
7       color.new(FILL_COLOR, 80),
8       color.new(FILL_COLOR, 85),
9       color.new(FILL_COLOR, 90)
10  )
11 // Cycle background through the array's colors.
12 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.

3.14.4 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)
```

(continues on next page)

(continued from previous page)

```
    labelText += str.tostring(array.get(a, i)) + "\n"
    i += 1

label.new(bar_index, high, text = labelText)
```

3.14.5 Scope

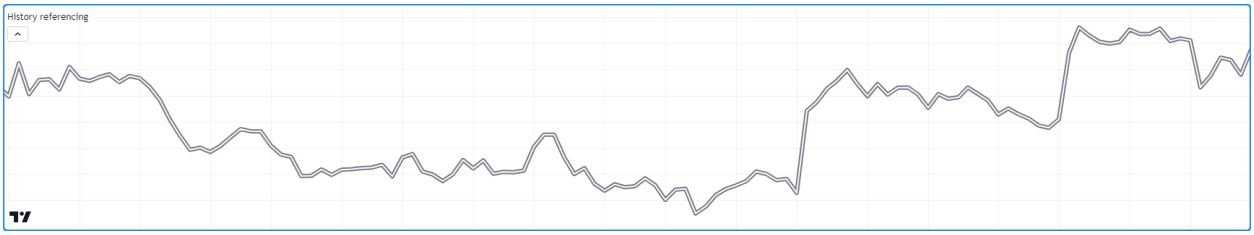
Users can declare arrays within the global scope of a script, as well as the local scopes of *functions*, *methods*, and *conditional structures*. Unlike some of the other built-in types, namely *fundamental* types, scripts can modify globally-assigned arrays from within local scopes, allowing users to implement global variables that any function in the script can directly interact with. We use the functionality here to calculate progressively lower or higher price levels:


```
//@version=5
indicator("Bands", "", true)
//@variable The distance ratio between plotted price levels.
factorInput = 1 + (input.float(-2., "Step %") / 100)
//@variable A single-value array holding the lowest `ohlc4` value within a 50 bar␣
↪window from 10 bars back.
level = array.new<float>(1, ta.lowest(ohlc4, 50)[10])

nextLevel(val) =>
    newLevel = level.get(0) * val
    // Write new level to the global `level` array so we can use it as the base in␣
↪the next function call.
    level.set(0, newLevel)
    newLevel

plot(nextLevel(1))
plot(nextLevel(factorInput))
plot(nextLevel(factorInput))
plot(nextLevel(factorInput))
```

3.14.6 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)
```

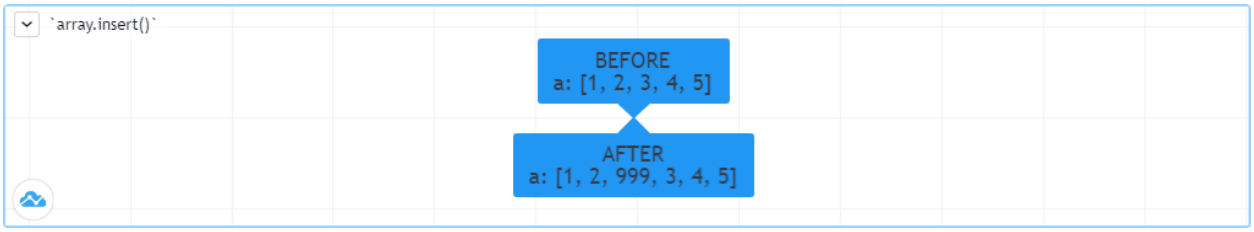
3.14.7 Inserting and removing array elements

Inserting

The following three functions can insert new elements into an array.

array.unshift() inserts a new element at the beginning of an array (index 0) and increases the index values of any existing elements by one.

array.insert() inserts a new element at the specified `index` and increases the index of existing elements at or after the `index` by one.



```
//@version=5
indicator("`array.insert()`")
a = array.new<float>(5, 0)
for i = 0 to 4
    array.set(a, i, i + 1)
if barstate.islast
    label.new(bar_index, 0, "BEFORE\na: " + str.tostring(a), size = size.large)
    array.insert(a, 2, 999)
    label.new(bar_index, 0, "AFTER\na: " + str.tostring(a), style = label.style_label_
↪up, size = size.large)
```

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

Removing

These four functions remove elements from an array. The first three also return the value of the removed element.

array.remove() removes the element at the specified `index` and returns that element's value.

array.shift() removes the first element from an array and returns its value.

array.pop() removes the last element of an array and returns its value.

array.clear() removes all elements from an array. Note that clearing an array won't delete any objects its elements referenced. See the example below that illustrates how this works:

```
//@version=5
indicator("`array.clear()` example", overlay = true)

// Create a label array and add a label to the array on each new bar.
var a = array.new<label>()
label lbl = label.new(bar_index, high, "Text", color = color.red)
array.push(a, lbl)

var table t = table.new(position.top_right, 1, 1)
// Clear the array on the last bar. This doesn't remove the labels from the chart.
if barstate.islast
    array.clear(a)
    table.cell(t, 0, 0, "Array elements count: " + str.tostring(array.size(a)),␣
↪bgcolor = color.yellow)
```

Using an array as a stack

Stacks are LIFO (last in, first out) constructions. They behave somewhat like a vertical pile of books to which books can only be added or removed one at a time, always from the top. Pine Script™ arrays can be used as a stack, in which case we use the array.push() and array.pop() functions to add and remove elements at the end of the array.

`array.push(prices, close)` will add a new element to the end of the `prices` array, increasing the array's size by one.

`array.pop(prices)` will remove the end element from the `prices` array, return its value and decrease the array's size by one.

See how the functions are used here to track successive lows in rallies:


```
//@version=5
indicator("Lows from new highs", "", true)
var lows = array.new<float>(0)
flushLows = false

// Remove last element from the stack when `_cond` is true.
array_pop(id, cond) => cond and array.size(id) > 0 ? array.pop(id) : float(na)

if ta.rising(high, 1)
    // Rising highs; push a new low on the stack.
    lows.push(low)
    // Force the return type of this `if` block to be the same as that of the next␣
↪block.
    bool(na)
else if lows.size() >= 4 or low < array.min(lows)
    // We have at least 4 lows or price has breached the lowest low;
    // sort lows and set flag indicating we will plot and flush the levels.
    array.sort(lows, order.ascending)
    flushLows := true

// If needed, plot and flush lows.
lowLevel = array_pop(lows, flushLows)
plot(lowLevel, "Low 1", low > lowLevel ? color.silver : color.purple, 2, plot.style_
↪linebr)
lowLevel := array_pop(lows, flushLows)
plot(lowLevel, "Low 2", low > lowLevel ? color.silver : color.purple, 3, plot.style_
↪linebr)
lowLevel := array_pop(lows, flushLows)
plot(lowLevel, "Low 3", low > lowLevel ? color.silver : color.purple, 4, plot.style_
↪linebr)
lowLevel := array_pop(lows, flushLows)
plot(lowLevel, "Low 4", low > lowLevel ? color.silver : color.purple, 5, plot.style_
↪linebr)

if flushLows
    // Clear remaining levels after the last 4 have been plotted.
    lows.clear()
```

Using an array as a queue

Queues are FIFO (first in, first out) constructions. They behave somewhat like cars arriving at a red light. New cars are queued at the end of the line, and the first car to leave will be the first one that arrived to the red light.

In the following code example, we let users decide through the script's inputs how many labels they want to have on their chart. We use that quantity to determine the size of the array of labels we then create, initializing the array's elements to `na`.

When a new pivot is detected, we create a label for it, saving the label's ID in the `pLabel` variable. We then queue the ID of that label by using array.push() to append the new label's ID to the end of the array, making our array size one greater than the maximum number of labels to keep on the chart.

Lastly, we de-queue the oldest label by removing the array's first element using array.shift() and deleting the label referenced by that array element's value. As we have now de-queued an element from our queue, the array contains `pivotCountInput` elements once again. Note that on the dataset's first bars we will be deleting `na` label IDs until the maximum number of labels has been created, but this does not cause runtime errors. Let's look at our code:


```
//@version=5
MAX_LABELS = 100
indicator("Show Last n High Pivots", "", true, max_labels_count = MAX_LABELS)

pivotCountInput = input.int(5, "How many pivots to show", minval = 0, maxval = MAX_
↪LABELS)
pivotLegsInput  = input.int(3, "Pivot legs", minval = 1, maxval = 5)

// Create an array containing the user-selected max count of label IDs.
var labelIds = array.new<label>(pivotCountInput)

pHi = ta.pivothigh(pivotLegsInput, pivotLegsInput)
if not na(pHi)
    // New pivot found; plot its label `i_pivotLegs` bars back.
    pLabel = label.new(bar_index[pivotLegsInput], pHi, str.tostring(pHi, format.
↪mintick), textcolor = color.white)
    // Queue the new label's ID by appending it to the end of the array.
    array.push(labelIds, pLabel)
    // De-queue the oldest label ID from the queue and delete the corresponding label.
    label.delete(array.shift(labelIds))
```

3.14.8 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().

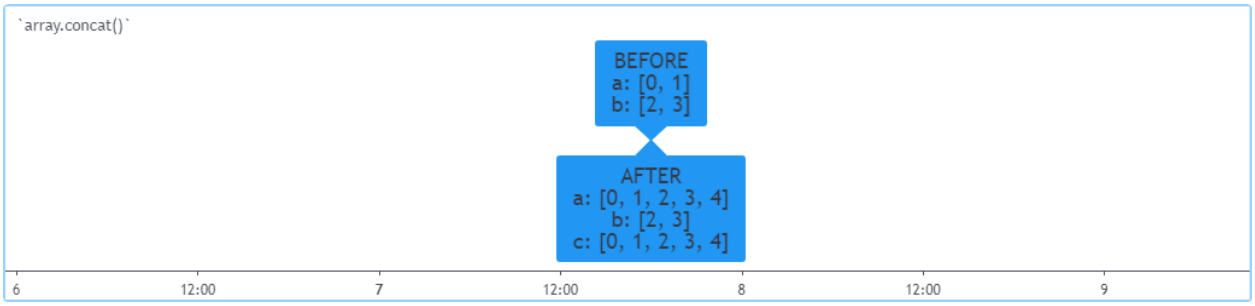
Note that contrary to the usual mathematical functions in Pine Script™, those used on arrays do not return `na` when some of the values they calculate on have `na` values. There are a few exceptions to this rule:

- 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.

3.14.9 Manipulating arrays

Concatenation

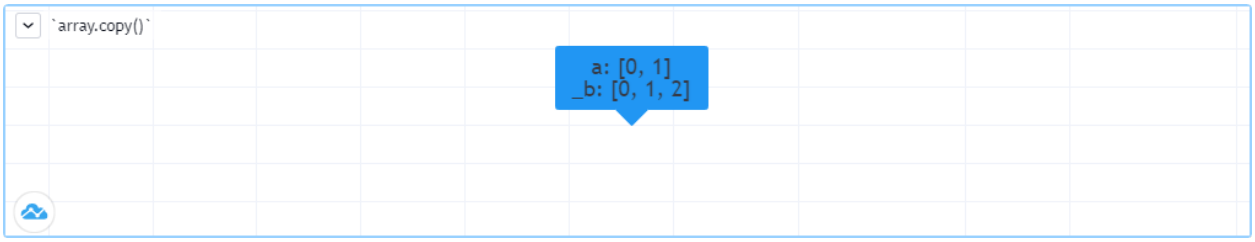
Two arrays can be merged—or concatenated—using array.concat(). When arrays are concatenated, the second array is appended to the end of the first, so the first array is modified while the second one remains intact. The function returns the array ID of the first array:




```
//@version=5
indicator("`array.concat()`")
a = array.new<float>(0)
b = array.new<float>(0)
array.push(a, 0)
array.push(a, 1)
array.push(b, 2)
array.push(b, 3)
if barstate.islast
    label.new(bar_index, 0, "BEFORE\na: " + str.tostring(a) + "\nb: " + str.
↪tostring(b), size = size.large)
    c = array.concat(a, b)
    array.push(c, 4)
    label.new(bar_index, 0, "AFTER\na: " + str.tostring(a) + "\nb: " + str.
↪tostring(b) + "\nc: " + str.tostring(c), style = label.style_label_up, size = size.
↪large)
```

Copying

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




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

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

Joining

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

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

Sorting

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


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

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

Reversing

Use array.reverse() to reverse an array:

```
//@version=5
indicator("`array.reverse()`")
a = array.new<float>(0)
array.push(a, 0)
array.push(a, 1)
array.push(a, 2)
```

(continues on next page)

(continued from previous page)

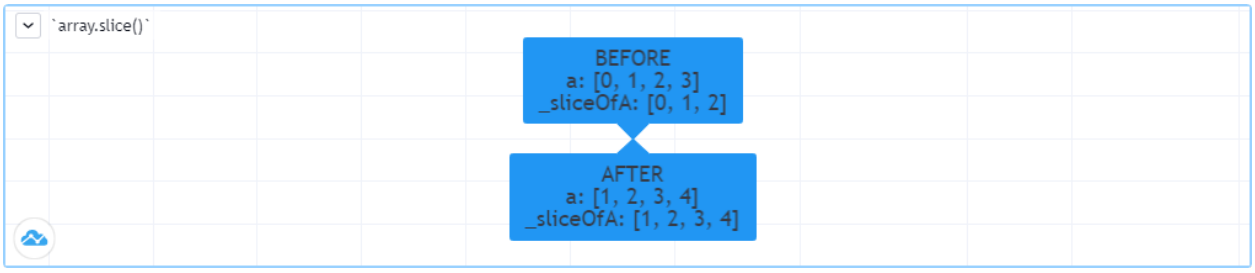
```
if barstate.islast
    array.reverse(a)
    label.new(bar_index, 0, "a: " + str.tostring(a))
```

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)
```

3.14.10 Searching arrays

We can test if a value is part of an array with the array.includes() function, which returns true if the element is found. We can find the first occurrence of a value in an array by using the array.indexof() function. The first occurence is the one with the lowest index. We can also find the last occurrence of a value with array.lastindexof():

```
//@version=5
indicator("Searching in arrays")
valueInput = input.int(1)
a = array.new<float>(0)
array.push(a, 0)
array.push(a, 1)
array.push(a, 2)
array.push(a, 1)
if barstate.islast
    valueFound      = array.includes(a, valueInput)
    firstIndexFound = array.indexof(a, valueInput)
    lastIndexFound  = array.lastindexof(a, valueInput)
    label.new(bar_index, 0, "a: " + str.tostring(a) +
      "\nFirst " + str.tostring(valueInput) + (firstIndexFound != -1 ? " value was␣
↪found at index: " + str.tostring(firstIndexFound) : " value was not found.") +
      "\nLast " + str.tostring(valueInput)  + (lastIndexFound  != -1 ? " value was␣
↪found at index: " + str.tostring(lastIndexFound) : " value was not found."))
```

We can also perform a binary search on an array but note that performing a binary search on an array means that the array will first need to be sorted in ascending order only. The array.binary_search() function will return the value's index if it was found or -1 if it wasn't. If we want to always return an existing index from the array even if our chosen value wasn't found, then we can use one of the other binary search functions available. The array.binary_search_leftmost() function, which returns an index if the value was found or the first index to the left where the value would be found. The array.binary_search_rightmost() function is almost identical and returns an index if the value was found or the first index to the right where the value would be found.

3.14.11 Error handling

Malformed `array.*()` call syntax in Pine scripts will cause the usual **compiler** error messages to appear in Pine Editor's console, at the bottom of the window, when you save a script. Refer to the Pine Script™ v5 Reference Manual when in doubt regarding the exact syntax of function calls.

Scripts using arrays can also throw **runtime** errors, which appear as an exclamation mark next to the indicator's name on the chart. We discuss those runtime errors in this section.

Index xx is out of bounds. Array size is yy

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

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

```
//@version=5
indicator("Out of bounds index")
```

(continues on next page)

(continued from previous page)

```
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 that a valid array ID pointing to an existing array. Note that an array created with no elements in it, as you do when you use `a = array.new_int(0)`, has a valid ID nonetheless. This code will throw the error we are discussing:

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

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

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

or:

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

Array is too large. Maximum size is 100000

This error will appear if your code attempts to declare an array with a size greater than 100,000. It will also occur if, while dynamically appending elements to an array, a new element would increase the array's size past the maximum.

Cannot create an array with a negative size

We haven't found any use for arrays of negative size yet, but if you ever do, we may allow them :)

Cannot use shift() if array is empty.

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

Cannot use pop() if array is empty.

This error will occur if array.pop() is called to remove the last element of an empty array.

Index 'from' should be less than index 'to'

When two indices are used in functions such as array.slice(), the first index must always be smaller than the second one.

Slice is out of bounds of the parent array

This message occurs whenever the parent array's size is modified in such a way that it makes the shallow copy created by a slice point outside the boundaries of the parent array. This code will reproduce it because after creating a slice from index 3 to 4 (the last two elements of our five-element parent array), we remove the parent's first element, making its size four and its last index 3. From that moment on, the shallow copy which is still poiting to the "window" at the parent array's indices 3 to 4, is pointing out of the parent array's boundaries:

```
//@version=5
indicator("Slice out of bounds")
a = array.new<float>(5, 0)
b = array.slice(a, 3, 5)
array.remove(a, 0)
c = array.indexof(b, 2)
plot(c)
```

3.15 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.

3.15.1 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.

3.15.2 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.

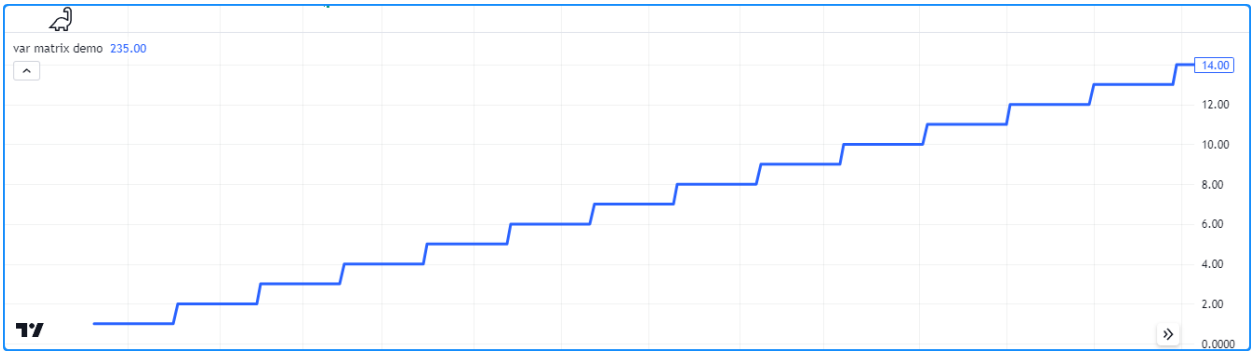
Here, we declare a `myMatrix` variable referencing a new `matrix<float>` instance with two rows, two columns, and an `initial_value` of 0. The variable gets its type information from the new object in this case, so it doesn't require an explicit type declaration:

```
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:




```
1  //@version=5
2  indicator("var matrix demo")
3
4  //@variable A 1x2 rectangular matrix declared only at `bar_index == 0`, i.e., the␣
   ↪first bar.
5  var m = matrix.new<int>(1, 2, 0)
6
7  //@variable Is `true` on every 20th bar.
8  bool update = bar_index % 20 == 0
9
10 if update
11     int currentValue = m.get(0, 0) // Get the current value of the first row and␣
   ↪column.
12     m.set(0, 0, currentValue + 1)  // Set the first row and column element value to␣
   ↪`currentValue + 1`.
13
14 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

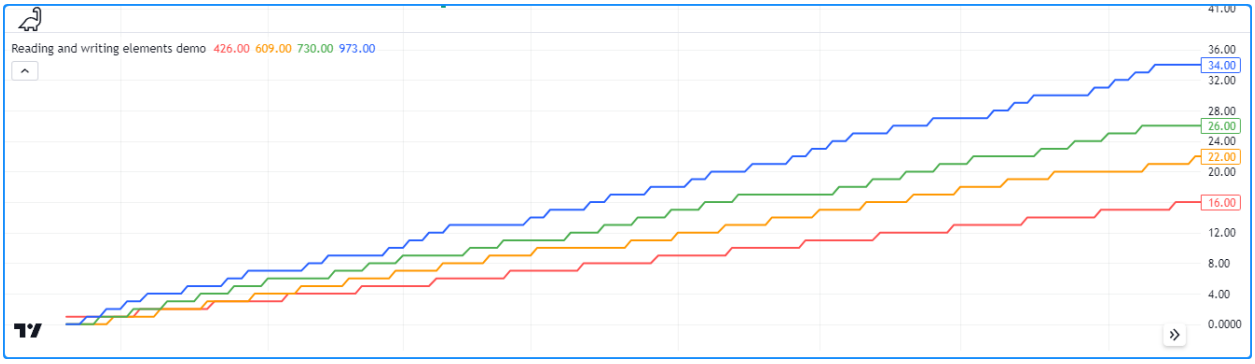
can only hold int, float, bool, color, or string types or *user-defined types* that exclusively contain within their fields these types or collections (*arrays*, matrices, or *maps*) of these types.

3.15.3 Reading and writing matrix elements

`matrix.get()` and `matrix.set()`

To retrieve the value from a matrix at a specified `row` and `column` index, use matrix.get(). This function locates the specified matrix element and returns its value. Similarly, to overwrite a specific element's value, use matrix.set() to assign the element at the specified `row` and `column` to a new `value`.

The example below defines a square matrix `m` with two rows and columns and an `initial_value` of 0 for all elements on the first bar. The script adds 1 to each element's value on different bars using the m.get() and m.set() methods. It updates the first row's first value once every 11 bars, the first row's second value once every seven bars, the second row's first value once every five bars, and the second row's second value once every three bars. The script plots each element's value on the chart:




```
//@version=5
indicator("Reading and writing elements demo")

//@variable A 2x2 square matrix of `float` values.
var m = matrix.new<float>(2, 2, 0.0)

switch
    bar_index % 11 == 0 => m.set(0, 0, m.get(0, 0) + 1.0) // Adds 1 to the value at row 0, column 0 every 11th bar.
    bar_index % 7  == 0 => m.set(0, 1, m.get(0, 1) + 1.0) // Adds 1 to the value at row 0, column 1 every 7th bar.
    bar_index % 5  == 0 => m.set(1, 0, m.get(1, 0) + 1.0) // Adds 1 to the value at row 1, column 0 every 5th bar.
    bar_index % 3  == 0 => m.set(1, 1, m.get(1, 1) + 1.0) // Adds 1 to the value at row 1, column 1 every 3rd bar.

plot(m.get(0, 0), "Row 0, Column 0 Value", color.red, 2)
plot(m.get(0, 1), "Row 0, Column 1 Value", color.orange, 2)
plot(m.get(1, 0), "Row 1, Column 0 Value", color.green, 2)
plot(m.get(1, 1), "Row 1, Column 1 Value", color.blue, 2)
```

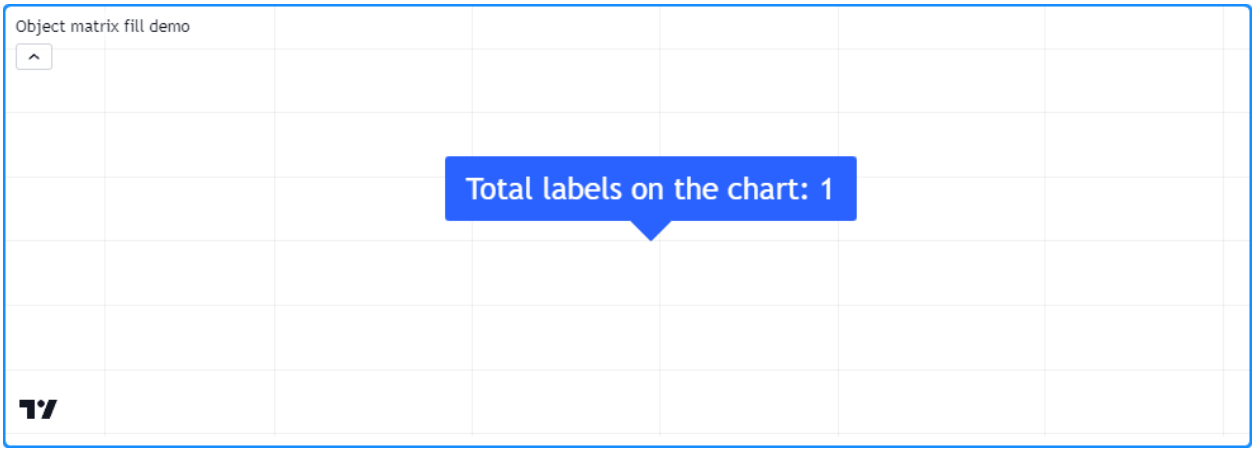
`matrix.fill()`

To overwrite all matrix elements with a specific value, use matrix.fill(). This function points all items in the entire matrix or within the `from_row/column` and `to_row/column` index range to the `value` specified in the call. For example, this snippet declares a 4x4 square matrix, then fills its elements with a random value:

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

Note when using matrix.fill() with matrices containing special types (line, linefill, box, polyline, label, table, or chart.point) or *UDTs*, all replaced elements will point to the same object passed in the function call.

This script declares a matrix with four rows and columns of label references, which it fills with a new label object on the first bar. On each bar, the script sets the `x` attribute of the label referenced at row 0, column 0 to bar_index, and the `text` attribute of the one referenced at row 3, column 3 to the number of labels on the chart. Although the matrix can reference 16 (4x4) labels, each element points to the *same* instance, resulting in only one label on the chart that updates its `x` and `text` attributes on each bar:



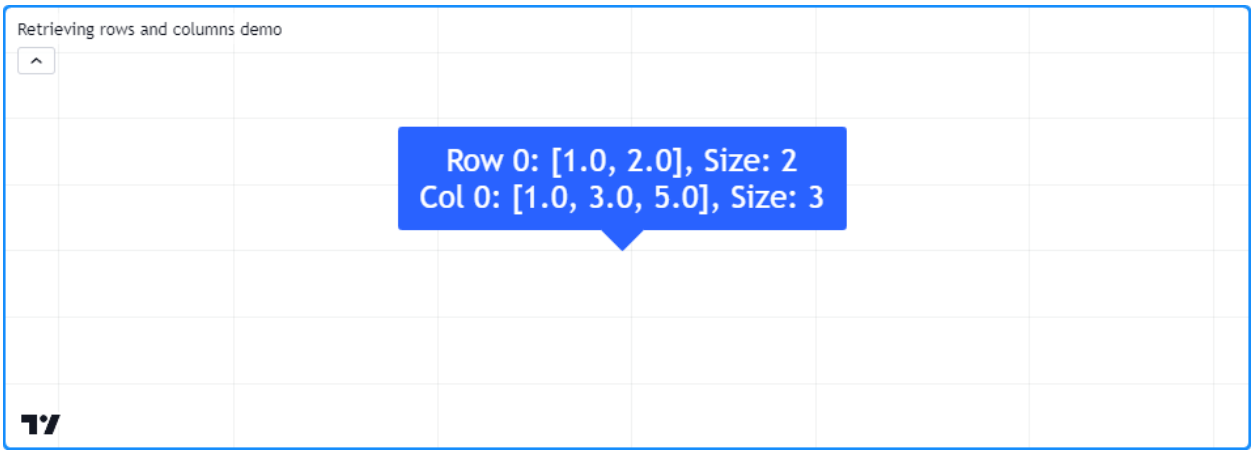

```
//@version=5
indicator("Object matrix fill demo")

//@variable A 4x4 label matrix.
var matrix<label> m = matrix.new<label>(4, 4)

// Fill `m` with a new label object on the first bar.
if bar_index == 0
    m.fill(label.new(0, 0, textcolor = color.white, size = size.huge))

//@variable The number of label objects on the chart.
int numLabels = label.all.size()

// Set the `x` of the label from the first row and column to `bar_index`.
m.get(0, 0).set_x(bar_index)
// Set the `text` of the label at the last row and column to the number of labels.
m.get(3, 3).set_text(str.format("Total labels on the chart: {0}", numLabels))
```

3.15.4 Rows and columns

Retrieving

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

The script below populates a 3x2 `m` matrix with the values 1 - 6 on the first chart bar. It calls the m.row() and m.col() methods to access the first row and column arrays from the matrix and displays them on the chart in a label along with the array sizes:




```
//@version=5
indicator("Retrieving rows and columns demo")

//@variable A 3x2 rectangular matrix.
var matrix<float> m = matrix.new<float>(3, 2)

if bar_index == 0
    m.set(0, 0, 1.0) // Set row 0, column 0 value to 1.
    m.set(0, 1, 2.0) // Set row 0, column 1 value to 2.
    m.set(1, 0, 3.0) // Set row 1, column 0 value to 3.
    m.set(1, 1, 4.0) // Set row 1, column 1 value to 4.
    m.set(2, 0, 5.0) // Set row 1, column 0 value to 5.
    m.set(2, 1, 6.0) // Set row 1, column 1 value to 6.

//@variable The first row of the matrix.
array<float> row0 = m.row(0)
//@variable The first column of the matrix.
array<float> column0 = m.col(0)

//@variable Displays the first row and column of the matrix and their sizes in a label.
var label debugLabel = label.new(0, 0, color = color.blue, textcolor = color.white, size = size.huge)
debugLabel.set_x(bar_index)
debugLabel.set_text(str.format("Row 0: {0}, Size: {1}\nCol 0: {2}, Size: {3}", row0, m.columns(), column0, m.rows()))
```

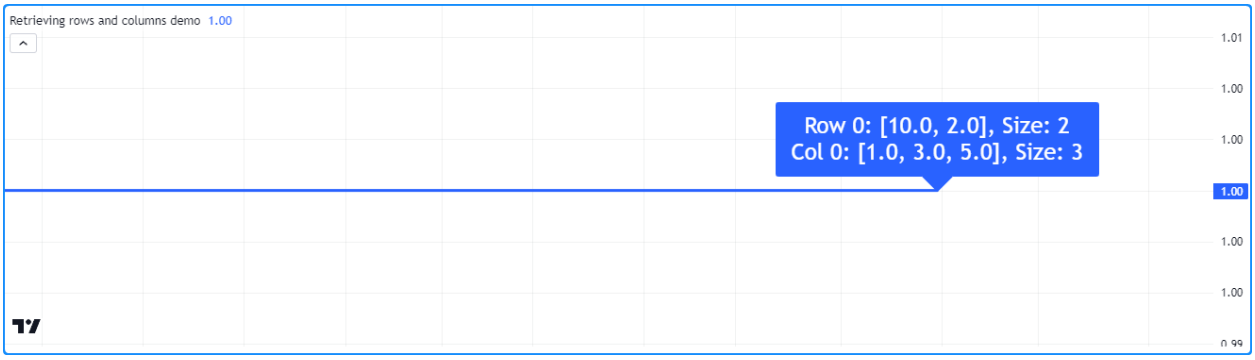
Note that:

- To get the sizes of the arrays displayed in the label, we used the rows() and columns() methods rather than

array.size() to demonstrate that the size of the `row0` array equals the number of columns and the size of the `column0` array equals the number of rows.

matrix.row() and matrix.col() copy the references in a row/column to a new array. Modifications to the *arrays* returned by these functions do not directly affect the elements or the shape of a matrix.

Here, we've modified the previous script to set the first element of `row0` to 10 via the array.set() method before displaying the label. This script also plots the value from row 0, column 0. As we see, the label shows that the first element of the `row0` array is 10. However, the plot shows that the corresponding matrix element still has a value of 1:




```
//@version=5
indicator("Retrieving rows and columns demo")

//@variable A 3x2 rectangular matrix.
var matrix<float> m = matrix.new<float>(3, 2)

if bar_index == 0
    m.set(0, 0, 1.0) // Set row 0, column 0 value to 1.
    m.set(0, 1, 2.0) // Set row 0, column 1 value to 2.
    m.set(1, 0, 3.0) // Set row 1, column 0 value to 3.
    m.set(1, 1, 4.0) // Set row 1, column 1 value to 4.
    m.set(2, 0, 5.0) // Set row 1, column 0 value to 5.
    m.set(2, 1, 6.0) // Set row 1, column 1 value to 6.

//@variable The first row of the matrix.
array<float> row0 = m.row(0)
//@variable The first column of the matrix.
array<float> column0 = m.col(0)

// Set the first `row` element to 10.
row0.set(0, 10)

//@variable Displays the first row and column of the matrix and their sizes in a label.
var label debugLabel = label.new(0, m.get(0, 0), color = color.blue, textcolor = color.white, size = size.huge)
debugLabel.set_x(bar_index)
debugLabel.set_text(str.format("Row 0: {0}, Size: {1}\nCol 0: {2}, Size: {3}", row0, m.columns(), column0, m.rows()))

// Plot the first element of `m`.
plot(m.get(0, 0), linewidth = 3)
```

Although changes to an array returned by matrix.row() or matrix.col() do not directly affect a parent matrix, it's important to note the resulting array from a matrix containing *UDTs* or special types, including line, linefill, box, polyline, label, table, or chart.point, behaves as a *shallow copy* of a row/column, i.e., the elements within an array returned from these

functions point to the same objects as the corresponding matrix elements.

This script contains a custom `myUDT` type containing a `value` field with an initial value of 0. It declares a 1x1 `m` matrix to hold a single `myUDT` instance on the first bar, then calls `m.row(0)` to copy the first row of the matrix as an array. On every chart bar, the script adds 1 to the `value` field of the first `row` array element. In this case, the `value` field of the matrix element increases on every bar as well since both elements reference the same object:

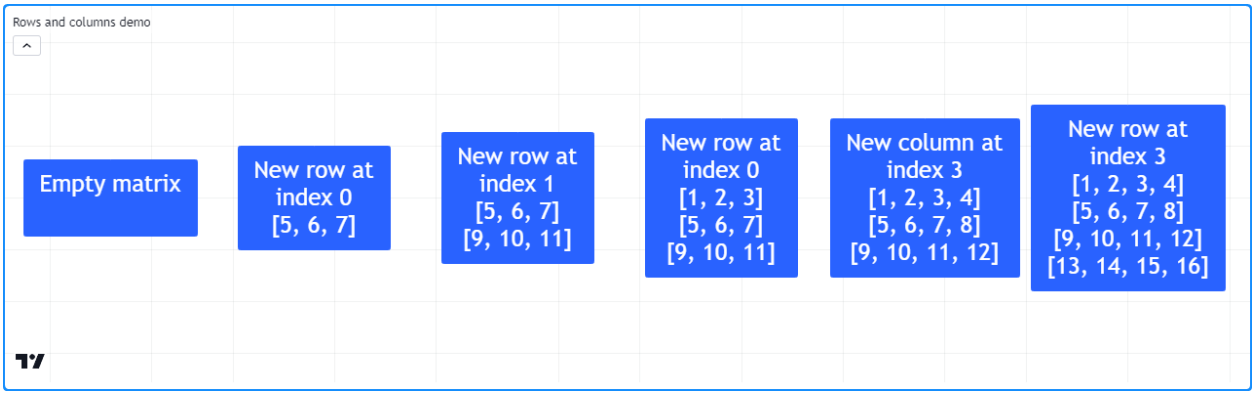
```
//@version=5
indicator("Row with reference types demo")

//@type A custom type that holds a float value.
type myUDT
    float value = 0.0

//@variable A 1x1 matrix of `myUDT` type.
var matrix<myUDT> m = matrix.new<myUDT>(1, 1, myUDT.new())
//@variable A shallow copy of the first row of `m`.
array<myUDT> row = m.row(0)
//@variable The first element of the `row`.
myUDT firstElement = row.get(0)

firstElement.value += 1.0 // Add 1 to the `value` field of `firstElement`. Also affects the element in the matrix.

plot(m.get(0, 0).value, linewidth = 3) // Plot the `value` of the `myUDT` object from the first row and column of `m`.
```

Inserting

Scripts can add new rows and columns to a matrix via matrix.add_row() and matrix.add_col(). These functions insert the value references from an array into a matrix at the specified `row/column` index. If the `id` matrix is empty (has no rows or columns), the `array_id` in the call can be of any size. If a row/column exists at the specified index, the matrix increases the index value for the existing row/column and all after it by 1.

The script below declares an empty `m` matrix and inserts rows and columns using the m.add_row() and m.add_col() methods. It first inserts an array with three elements at row 0, turning `m` into a 1x3 matrix, then another at row 1, changing the shape to 2x3. After that, the script inserts another array at row 0, which changes the shape of `m` to 3x3 and shifts the index of all rows previously at index 0 and higher. It inserts another array at the last column index, changing the shape to 3x4. Finally, it adds an array with four values at the end row index.

The resulting matrix has four rows and columns and contains values 1-16 in ascending order. The script displays the rows of `m` after each row/column insertion with a user-defined `debugLabel()` function to visualize the process:



```
//@version=5
indicator("Rows and columns demo")

//@function Displays the rows of a matrix in a label with a note.
//@param    this The matrix to display.
//@param    barIndex The `bar_index` to display the label at.
//@param    bgColor The background color of the label.
//@param    textColor The color of the label's text.
//@param    note The text to display above the rows.
method debugLabel(
     matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
     color textColor = color.white, string note = ""
 ) =>
    labelText = note + "\n" + str.tostring(this)
    if barstate.ishistory
        label.new(
             barIndex, 0, labelText, color = bgColor, style = label.style_label_
↪center,
             textcolor = textColor, size = size.huge
         )

//Create an empty matrix.
var m = matrix.new<float>()

if bar_index == last_bar_index - 1
    debugLabel(m, bar_index - 30, note = "Empty matrix")

    // Insert an array at row 0. `m` will now have 1 row and 3 columns.
    m.add_row(0, array.from(5, 6, 7))
    debugLabel(m, bar_index - 20, note = "New row at\nindex 0")

    // Insert an array at row 1. `m` will now have 2 rows and 3 columns.
    m.add_row(1, array.from(9, 10, 11))
    debugLabel(m, bar_index - 10, note = "New row at\nindex 1")

    // Insert another array at row 0. `m` will now have 3 rows and 3 columns.
    // The values previously on row 0 will now be on row 1, and the values from row 1␣
↪will be on row 2.
    m.add_row(0, array.from(1, 2, 3))
    debugLabel(m, bar_index, note = "New row at\nindex 0")

    // Insert an array at column 3. `m` will now have 3 rows and 4 columns.
    m.add_col(3, array.from(4, 8, 12))
    debugLabel(m, bar_index + 10, note = "New column at\nindex 3")

    // Insert an array at row 3. `m` will now have 4 rows and 4 columns.
    m.add_row(3, array.from(13, 14, 15, 16))
    debugLabel(m, bar_index + 20, note = "New row at\nindex 3")
```

Note: Just as the row or column arrays *retrieved* from a matrix of line, linefill, box, polyline, label, table, chart.point, or *UDT* instances behave as shallow copies, the elements of matrices containing such types reference the same objects as the *arrays* inserted into them. Modifications to the element values in either object affect the other in such cases.

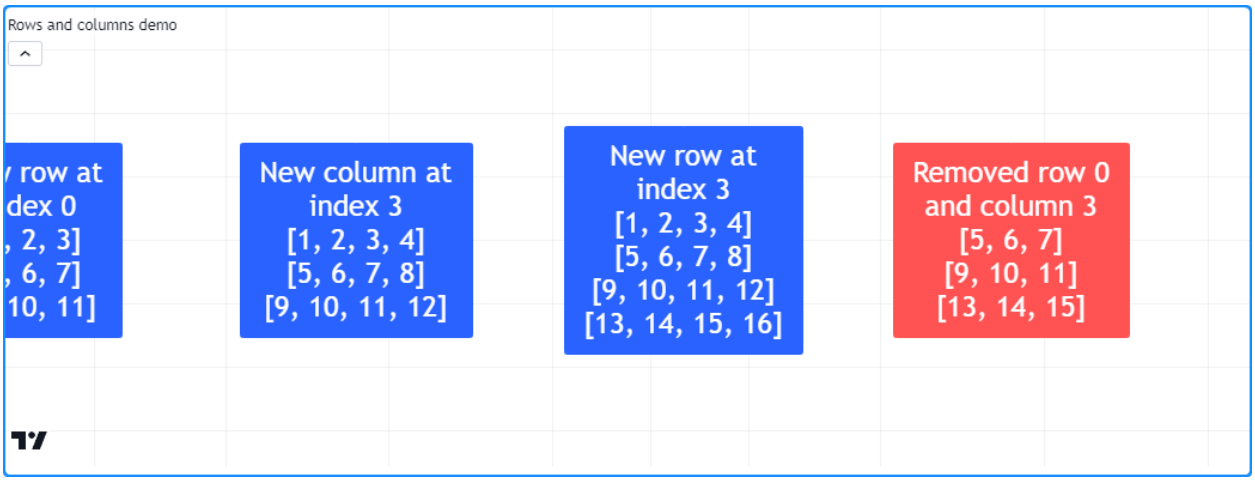
Removing

To remove a specific row or column from a matrix, use matrix.remove_row() and matrix.remove_col(). These functions remove the specified row/column and decrease the index values of all rows/columns after it by 1.

For this example, we've added these lines of code to our "Rows and columns demo" script from the *section above*:

```
// Removing example

    // Remove the first row and last column from the matrix. `m` will now have 3 rows␣
↪and 3 columns.
    m.remove_row(0)
    m.remove_col(3)
    debugLabel(m, bar_index + 30, color.red, note = "Removed row 0\nand column 3")
```

This code removes the first row and the last column of the `m` matrix using the m.remove_row() and m.remove_col() methods and displays the rows in a label at `bar_index + 30`. As we can see, `m` has a 3x3 shape after executing this block, and the index values for all existing rows are reduced by 1:



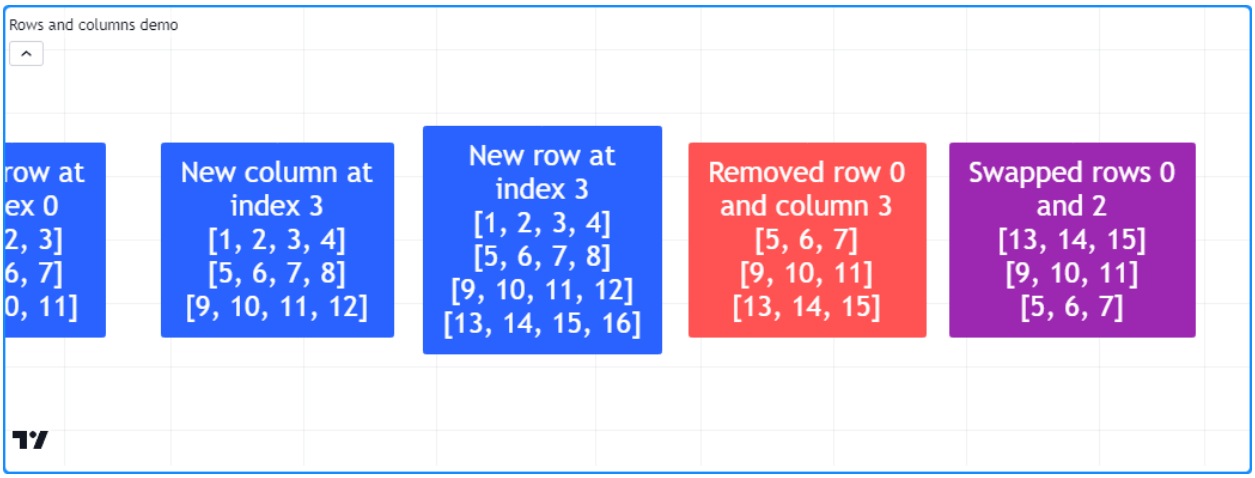
Swapping

To swap the rows and columns of a matrix without altering its dimensions, use matrix.swap_rows() and matrix.swap_columns(). These functions swap the locations of the elements at the `row1`/`column1` and `row2`/`column2` indices.

Let's add the following lines to the *previous example*, which swap the first and last rows of `m` and display the changes in a label at `bar_index + 40`:

```
// Swapping example

    // Swap the first and last row. `m` retains the same dimensions.
    m.swap_rows(0, 2)
    debugLabel(m, bar_index + 40, color.purple, note = "Swapped rows 0\nand 2")
```

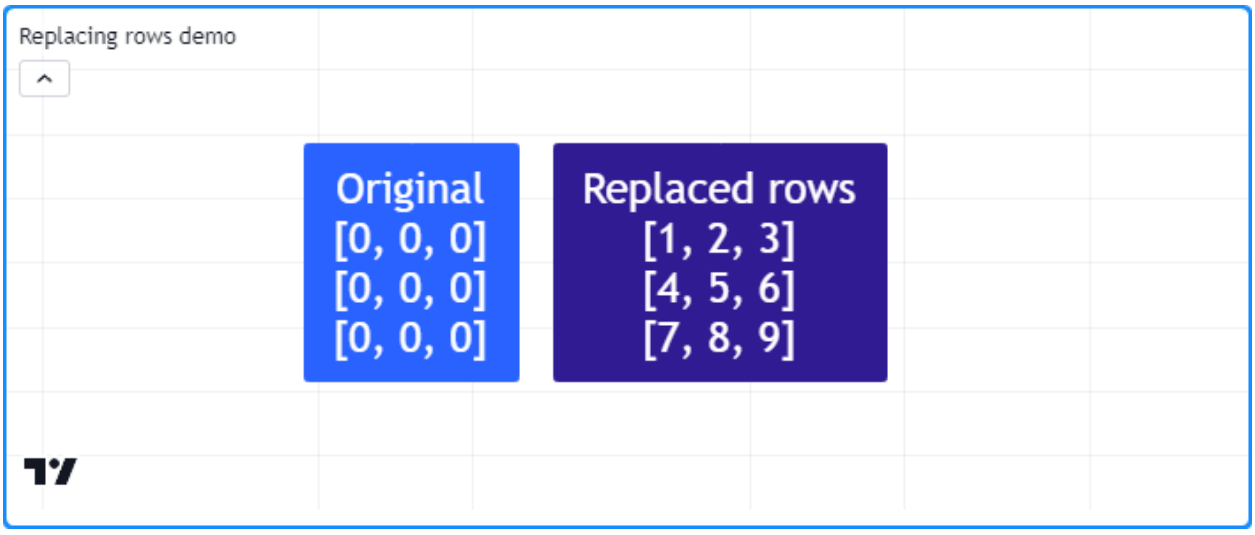
In the new label, we see the matrix has the same number of rows as before, and the first and last rows have traded places:



Replacing

It may be desirable in some cases to completely *replace* a row or column in a matrix. To do so, *insert* the new array at the desired `row/column` and *remove* the old elements previously at that index.

In the following code, we've defined a `replaceRow()` method that uses the add_row() method to insert the new `values` at the `row` index and uses the remove_row() method to remove the old row that moved to the `row + 1` index. This script uses the `replaceRow()` method to fill the rows of a 3x3 matrix with the numbers 1-9. It draws a label on the chart before and after replacing the rows using the custom `debugLabel()` method:




```
//@version=5
indicator("Replacing rows demo")

//@function Displays the rows of a matrix in a label with a note.
//@param    this The matrix to display.
//@param    barIndex The `bar_index` to display the label at.
//@param    bgColor The background color of the label.
//@param    textColor The color of the label's text.
//@param    note The text to display above the rows.
method debugLabel(
     matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
```

(continues on next page)

(continued from previous page)

```
     color textColor = color.white, string note = ""
 ) =>
    labelText = note + "\n" + str.tostring(this)
    if barstate.ishistory
        label.new(
             barIndex, 0, labelText, color = bgColor, style = label.style_label_
↪center,
             textcolor = textColor, size = size.huge
         )

//@function Replaces the `row` of `this` matrix with a new array of `values`.
//@param    row The row index to replace.
//@param    values The array of values to insert.
method replaceRow(matrix<float> this, int row, array<float> values) =>
    this.add_row(row, values) // Inserts a copy of the `values` array at the `row`.
    this.remove_row(row + 1)  // Removes the old elements previously at the `row`.

//@variable A 3x3 matrix.
var matrix<float> m = matrix.new<float>(3, 3, 0.0)

if bar_index == last_bar_index - 1
    m.debugLabel(note = "Original")
    // Replace each row of `m`.
    m.replaceRow(0, array.from(1.0, 2.0, 3.0))
    m.replaceRow(1, array.from(4.0, 5.0, 6.0))
    m.replaceRow(2, array.from(7.0, 8.0, 9.0))
    m.debugLabel(bar_index + 10, note = "Replaced rows")
```

3.15.5 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):

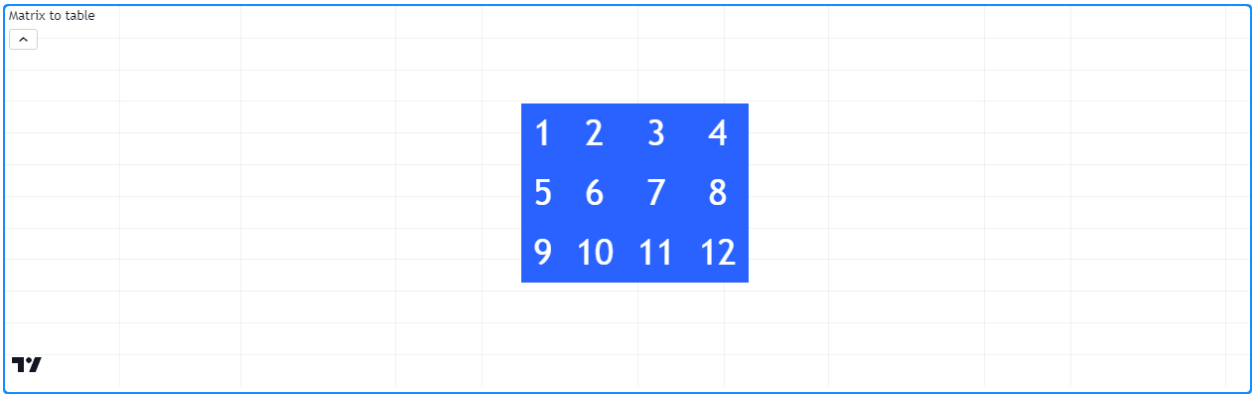
```
for row = 0 to m.rows() - 1
```

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>()

```

(continues on next page)

(continued from previous page)

```
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)

```

(continues on next page)

(continued from previous page)

```
//@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:

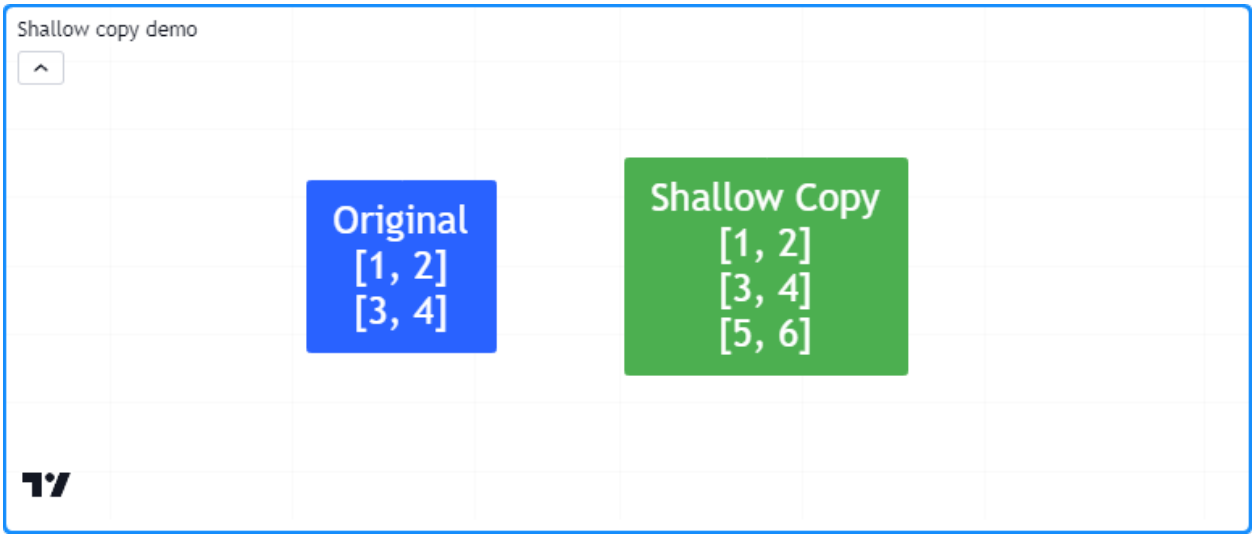
- `for...in` loops can also reference the index value of each row. For example, `for [i, row] in m` creates a tuple containing the `i` row index and the corresponding `row` array from the `m` matrix on each loop iteration.

3.15.6 Copying a matrix

Shallow copies

Pine scripts can copy matrices via matrix.copy(). This function returns a *shallow copy* of a matrix that does not affect the shape of the original matrix or its references.

For example, this script assigns a new matrix to the `myMatrix` variable and adds two columns. It creates a new `myCopy` matrix from `myMatrix` using the myMatrix.copy() method, then adds a new row. It displays the rows of both matrices in labels via the user-defined `debugLabel()` function:




```
//@version=5
indicator("Shallow copy demo")

//@function Displays the rows of a matrix in a label with a note.
//@param    this The matrix to display.
//@param    barIndex The `bar_index` to display the label at.
//@param    bgColor The background color of the label.
//@param    textColor The color of the label's text.
//@param    note The text to display above the rows.
method debugLabel(
     matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
     color textColor = color.white, string note = ""
 ) =>
    labelText = note + "\n" + str.tostring(this)
    if barstate.ishistory
        label.new(
             barIndex, 0, labelText, color = bgColor, style = label.style_label_
↪center,
             textcolor = textColor, size = size.huge
         )

//@variable A 2x2 `float` matrix.
matrix<float> myMatrix = matrix.new<float>()
myMatrix.add_col(0, array.from(1.0, 3.0))
myMatrix.add_col(1, array.from(2.0, 4.0))

//@variable A shallow copy of `myMatrix`.
matrix<float> myCopy = myMatrix.copy()
// Add a row to the last index of `myCopy`.
myCopy.add_row(myCopy.rows(), array.from(5.0, 6.0))

if bar_index == last_bar_index - 1
    // Display the rows of both matrices in separate labels.
    myMatrix.debugLabel(note = "Original")
    myCopy.debugLabel(bar_index + 10, color.green, note = "Shallow Copy")
```

It's important to note that the elements within shallow copies of a matrix point to the same values as the original matrix. When matrices contain special types (line, linefill, box, polyline, label, table, or chart.point) or *user-defined types*, the elements of a shallow copy reference the same objects as the original.

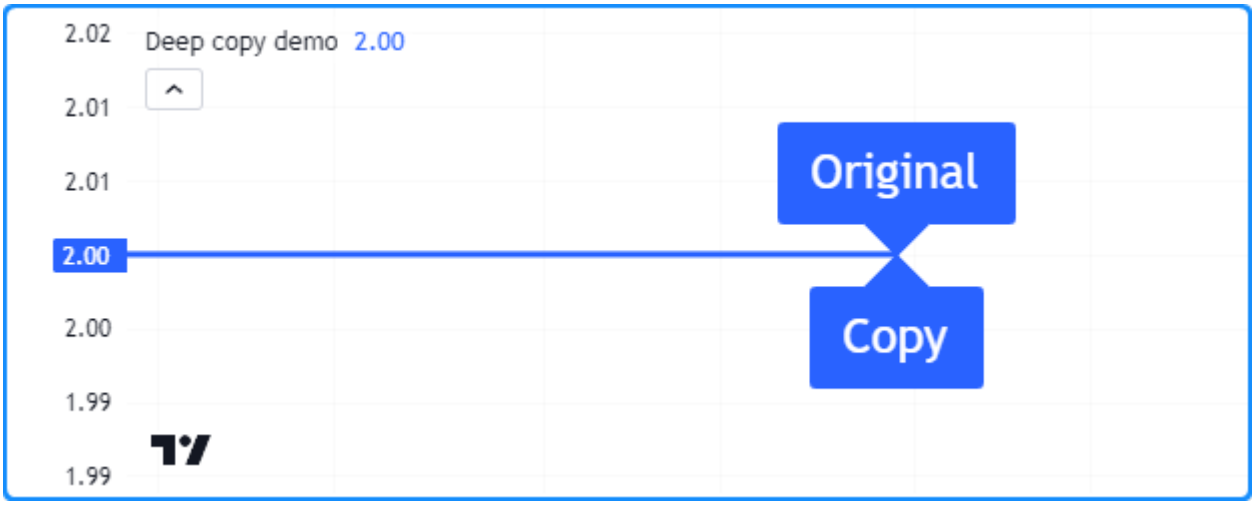
This script declares a `myMatrix` variable with a `newLabel` as the initial value. It then copies `myMatrix` to a `myCopy` variable via myMatrix.copy() and plots the number of labels. As we see below, there's only one label on the chart, as the element in `myCopy` references the same object as the element in `myMatrix`. Consequently, changes to the element values in `myCopy` affect the values in both matrices:


```
//@version=5
indicator("Shallow copy demo")

//@variable Initial value of the original matrix elements.
var label newLabel = label.new(
     bar_index, 1, "Original", color = color.blue, textcolor = color.white, size =␣
↪size.huge
 )

//@variable A 1x1 matrix containing a new `label` instance.
var matrix<label> myMatrix = matrix.new<label>(1, 1, newLabel)
//@variable A shallow copy of `myMatrix`.
var matrix<label> myCopy = myMatrix.copy()

//@variable The first label from the `myCopy` matrix.
label testLabel = myCopy.get(0, 0)

// Change the `text`, `style`, and `x` values of `testLabel`. Also affects the␣
↪`newLabel`.
testLabel.set_text("Copy")
testLabel.set_style(label.style_label_up)
testLabel.set_x(bar_index)

// Plot the total number of labels.
plot(label.all.size(), linewidth = 3)
```

Deep copies

One can produce a *deep copy* of a matrix (i.e., a matrix whose elements point to copies of the original values) by explicitly copying each object the matrix references.

Here, we've added a `deepCopy()` user-defined method to our previous script. The method creates a new matrix and uses *nested for loops* to assign all elements to copies of the originals. When the script calls this method instead of the built-in copy(), we see that there are now two labels on the chart, and any changes to the label from `myCopy` do not affect the one from `myMatrix`:




```
//@version=5
indicator("Deep copy demo")

//@function Returns a deep copy of a label matrix.
method deepCopy(matrix<label> this) =>
    //@variable A deep copy of `this` matrix.
    matrix<label> that = this.copy()
    for row = 0 to that.rows() - 1
        for column = 0 to that.columns() - 1
            // Assign the element at each `row` and `column` of `that` matrix to a copy of the retrieved label.
            that.set(row, column, that.get(row, column).copy())
    that

//@variable Initial value of the original matrix.
var label newLabel = label.new(
     bar_index, 2, "Original", color = color.blue, textcolor = color.white, size = size.huge
 )

//@variable A 1x1 matrix containing a new `label` instance.
var matrix<label> myMatrix = matrix.new<label>(1, 1, newLabel)
//@variable A deep copy of `myMatrix`.
var matrix<label> myCopy = myMatrix.deepCopy()

//@variable The first label from the `myCopy` matrix.
label testLabel = myCopy.get(0, 0)

// Change the `text`, `style`, and `x` values of `testLabel`. Does not affect the `newLabel`.
```

(continues on next page)

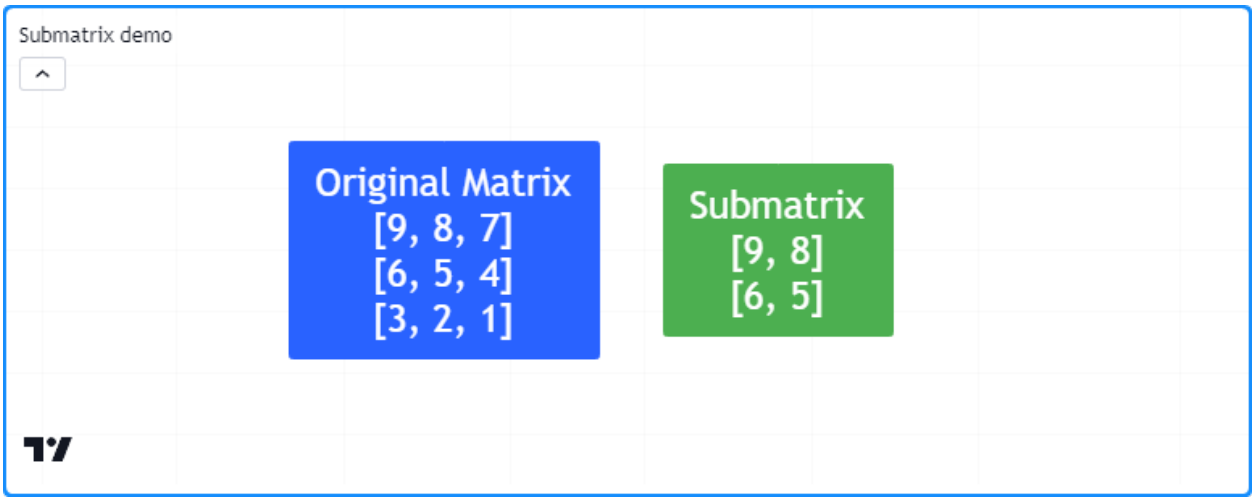
(continued from previous page)

```
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
         )
```

(continues on next page)

(continued from previous page)

```
20
21  //@variable A 3x3 matrix of values.
22  var m = matrix.new<float>()
23
24  if bar_index == last_bar_index - 1
25      // Add columns to `m`.
26      m.add_col(0, array.from(9, 6, 3))
27      m.add_col(1, array.from(8, 5, 2))
28      m.add_col(2, array.from(7, 4, 1))
29      // Display the rows of `m`.
30      m.debugLabel(note = "Original Matrix")
31
32      //@variable A 2x2 submatrix of `m` containing the first two rows and columns.
33      matrix<float> mSub = m.submatrix(from_row = 0, to_row = 2, from_column = 0, to_
    ↪column = 2)
34      // Display the rows of `mSub`
35      debugLabel(mSub, bar_index + 10, bgColor = color.green, note = "Submatrix")
```

3.15.7 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:


```
1  //@version=5
2  indicator("Scope and history demo", "Bar ratio comparison")
3
```

(continues on next page)

(continued from previous page)

```
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(
```

(continues on next page)

(continued from previous page)

```
    position.middle_center, globalMatrix.columns() + 1, globalMatrix.rows() + 1, bgcolor = color.navy
)

// Define value cells.
for [i, row] in globalAvg
    for [j, value] in row
        color textColor = value > 0.333 ? color.orange : color.gray
        infoTable.cell(j + 1, i + 1, str.tostring(value), text_color = textColor, text_size = size.huge)

// Define header cells.
infoTable.cell(0, 1, "Body ratio", text_color = color.white, text_size = size.huge)
infoTable.cell(0, 2, "Upper wick ratio", text_color = color.white, text_size = size.huge)
infoTable.cell(0, 3, "Lower wick ratio", text_color = color.white, text_size = size.huge)
infoTable.cell(1, 0, "Current average", text_color = color.white, text_size = size.huge)
infoTable.cell(2, 0, str.format("{0} bars ago", length), text_color = color.white, text_size = size.huge)
```

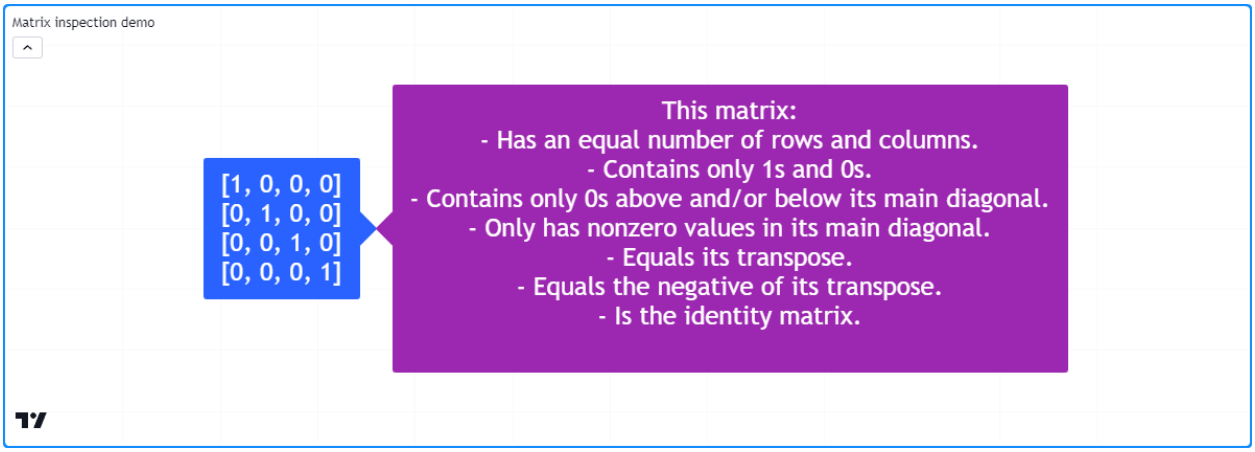
Note that:

- The `addData()` and `calcAvg()` functions have no parameters, as they directly interact with the `globalMatrix` and `length` variables declared in the outer scope.
- `calcAvg()` calculates the average by adding `previous` matrices using matrix.sum() and multiplying all elements by `1 / length` using matrix.mult(). We discuss these and other specialized functions in our *Matrix calculations* section below.

3.15.8 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,
```

(continues on next page)

(continued from previous page)

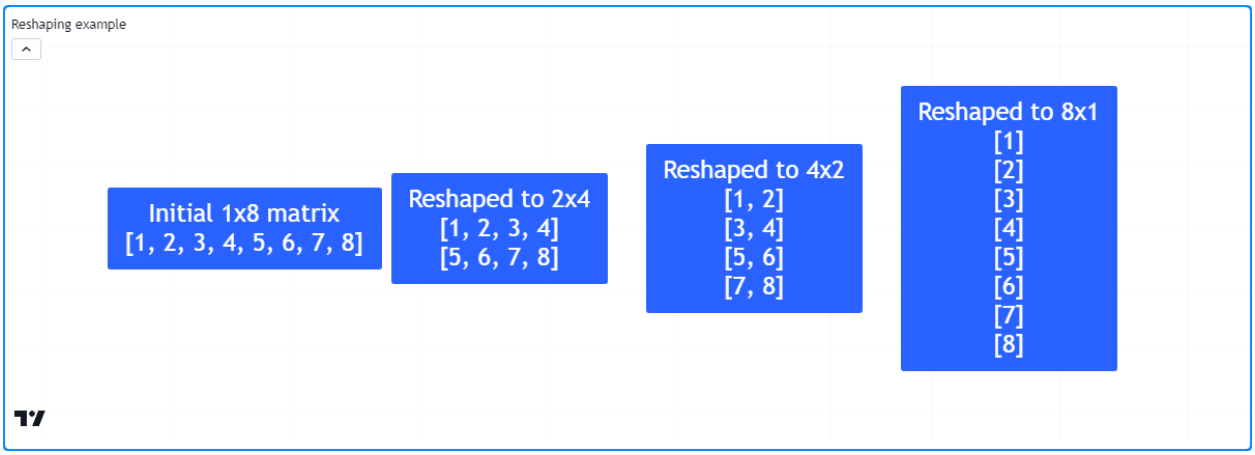
```
        textcolor = color.white, size = size.huge
    )
    // Display the result of `m.inspect()` in a purple label.
    label.new(
        bar_index, 0, m.inspect(), color = color.purple, style = label.style_label_
↪left,
        textcolor = color.white, size = size.huge
    )
```

3.15.9 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_
```

(continues on next page)

(continued from previous page)

```
↪center,
             textcolor = textColor, size = size.huge
         )

//@variable A matrix containing the values 1-8.
matrix<int> m = matrix.new<int>()

if bar_index == last_bar_index - 1
    // Add the initial vector of values.
    m.add_row(0, array.from(1, 2, 3, 4, 5, 6, 7, 8))
    m.debugLabel(note = "Initial 1x8 matrix")

    // Reshape. `m` now has 2 rows and 4 columns.
    m.reshape(2, 4)
    m.debugLabel(bar_index + 10, note = "Reshaped to 2x4")

    // Reshape. `m` now has 4 rows and 2 columns.
    m.reshape(4, 2)
    m.debugLabel(bar_index + 20, note = "Reshaped to 4x2")

    // Reshape. `m` now has 8 rows and 1 column.
    m.reshape(8, 1)
    m.debugLabel(bar_index + 30, note = "Reshaped to 8x1")
```

Note that:

- The order of elements in `m` does not change with each `m.reshape()` call.
- When reshaping a matrix, the product of the `rows` and `columns` arguments must equal the matrix.elements_count() value, as matrix.reshape() cannot change the number of elements in a matrix.

Reversing

One can reverse the order of all elements in a matrix using matrix.reverse(). This function moves the references of an m-by-n matrix `id` at the i-th row and j-th column to the m - 1 - i row and n - 1 - j column.

For example, this script creates a 3x3 matrix containing the values 1-9 in ascending order, then uses the reverse() method to reverse its contents. It displays the original and modified versions of the matrix in labels on the chart via `m.debugLabel()`:

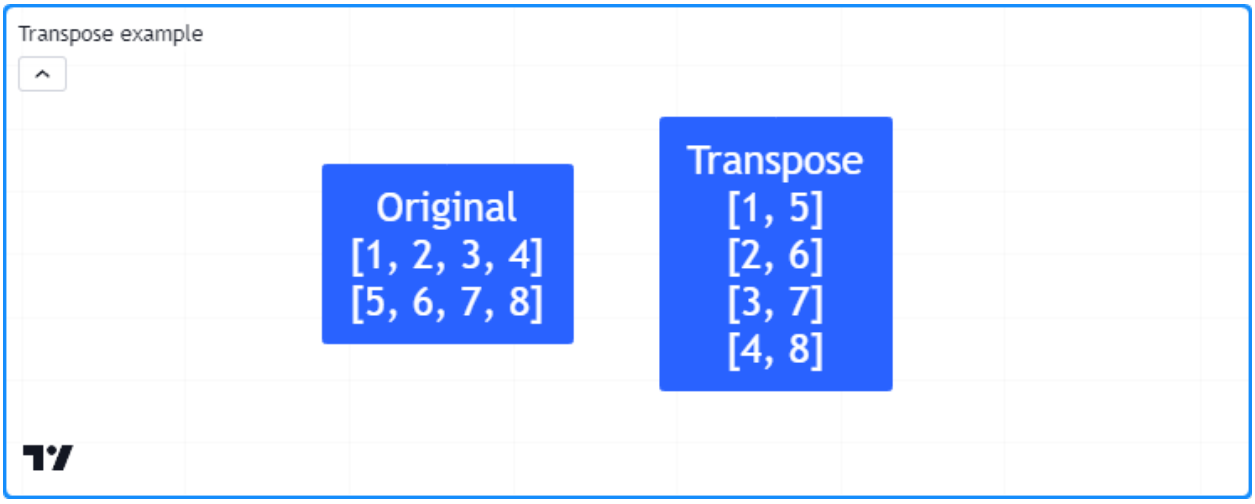
```
//@version=5
indicator("Reversing demo")

//@function Displays the rows of a matrix in a label with a note.
//@param    this The matrix to display.
//@param    barIndex The `bar_index` to display the label at.
//@param    bgColor The background color of the label.
//@param    textColor The color of the label's text.
//@param    note The text to display above the rows.
method debugLabel(
     matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
     color textColor = color.white, string note = ""
 ) =>
    labelText = note + "\n" + str.tostring(this)
    if barstate.ishistory
        label.new(
             barIndex, 0, labelText, color = bgColor, style = label.style_label_
↪center,
             textcolor = textColor, size = size.huge
         )

//@variable A 3x3 matrix.
matrix<float> m = matrix.new<float>()

// Add rows to `m`.
m.add_row(0, array.from(1, 2, 3))
m.add_row(1, array.from(4, 5, 6))
m.add_row(2, array.from(7, 8, 9))

if bar_index == last_bar_index - 1
    // Display the contents of `m`.
    m.debugLabel(note = "Original")
    // Reverse `m`, then display its contents.
    m.reverse()
    m.debugLabel(bar_index + 10, color.red, note = "Reversed")
```

Transposing

Transposing a matrix is a fundamental operation that flips all rows and columns in a matrix about its *main diagonal* (the diagonal vector of all values in which the row index equals the column index). This process produces a new matrix with reversed row and column dimensions, known as the *transpose*. Scripts can calculate the transpose of a matrix using matrix.transpose().

For any m-row, n-column matrix, the matrix returned from matrix.transpose() will have n rows and m columns. All elements in a matrix at the i-th row and j-th column correspond to the elements in its transpose at the j-th row and i-th column.

This example declares a 2x4 `m` matrix, calculates its transpose using the m.transpose() method, and displays both matrices on the chart using our custom `debugLabel()` method. As we can see below, the transposed matrix has a 4x2 shape, and the rows of the transpose match the columns of the original:

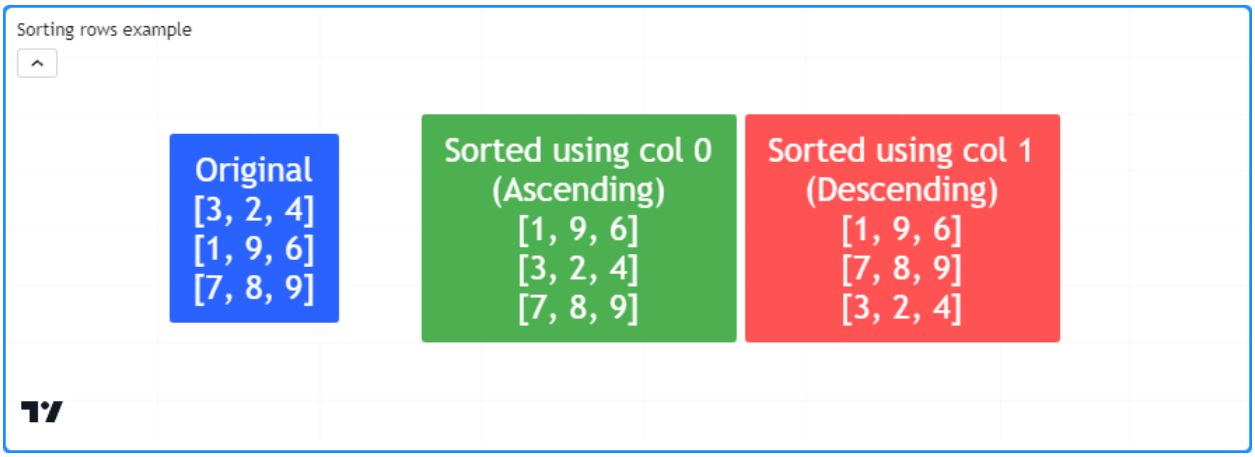



```
//@version=5
indicator("Transpose example")

//@function Displays the rows of a matrix in a label with a note.
//@param    this The matrix to display.
//@param    barIndex The `bar_index` to display the label at.
//@param    bgColor The background color of the label.
//@param    textColor The color of the label's text.
//@param    note The text to display above the rows.
method debugLabel(
     matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
     color textColor = color.white, string note = ""
 ) =>
    labelText = note + "\n" + str.tostring(this)
    if barstate.ishistory
        label.new(
             barIndex, 0, labelText, color = bgColor, style = label.style_label_
↪center,
             textcolor = textColor, size = size.huge
         )

//@variable A 2x4 matrix.
matrix<int> m = matrix.new<int>()

// Add columns to `m`.
m.add_col(0, array.from(1, 5))
m.add_col(1, array.from(2, 6))
m.add_col(2, array.from(3, 7))
m.add_col(3, array.from(4, 8))

//@variable The transpose of `m`. Has a 4x2 shape.
matrix<int> mt = m.transpose()

if bar_index == last_bar_index - 1
    m.debugLabel(note = "Original")
    mt.debugLabel(bar_index + 10, note = "Transpose")
```

Sorting

Scripts can sort the contents of a matrix via matrix.sort(). Unlike array.sort(), which sorts *elements*, this function organizes all *rows* in a matrix in a specified `order` (order.ascending by default) based on the values in a specified `column`.

This script declares a 3x3 `m` matrix, sorts the rows of the `m1` copy in ascending order based on the first column, then sorts the rows of the `m2` copy in descending order based on the second column. It displays the original matrix and sorted copies in labels using our `debugLabel()` method:




```
//@version=5
indicator("Sorting rows example")

//@function Displays the rows of a matrix in a label with a note.
//@param    this The matrix to display.
//@param    barIndex The `bar_index` to display the label at.
//@param    bgColor The background color of the label.
//@param    textColor The color of the label's text.
//@param    note The text to display above the rows.
method debugLabel(
     matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
     color textColor = color.white, string note = ""
 ) =>
    labelText = note + "\n" + str.tostring(this)
    if barstate.ishistory
        label.new(
             barIndex, 0, labelText, color = bgColor, style = label.style_label_
↪center,
             textcolor = textColor, size = size.huge
         )

//@variable A 3x3 matrix.
matrix<int> m = matrix.new<int>()

if bar_index == last_bar_index - 1
    // Add rows to `m`.
    m.add_row(0, array.from(3, 2, 4))
    m.add_row(1, array.from(1, 9, 6))
    m.add_row(2, array.from(7, 8, 9))
    m.debugLabel(note = "Original")

    // Copy `m` and sort rows in ascending order based on the first column (default).
    matrix<int> m1 = m.copy()
```

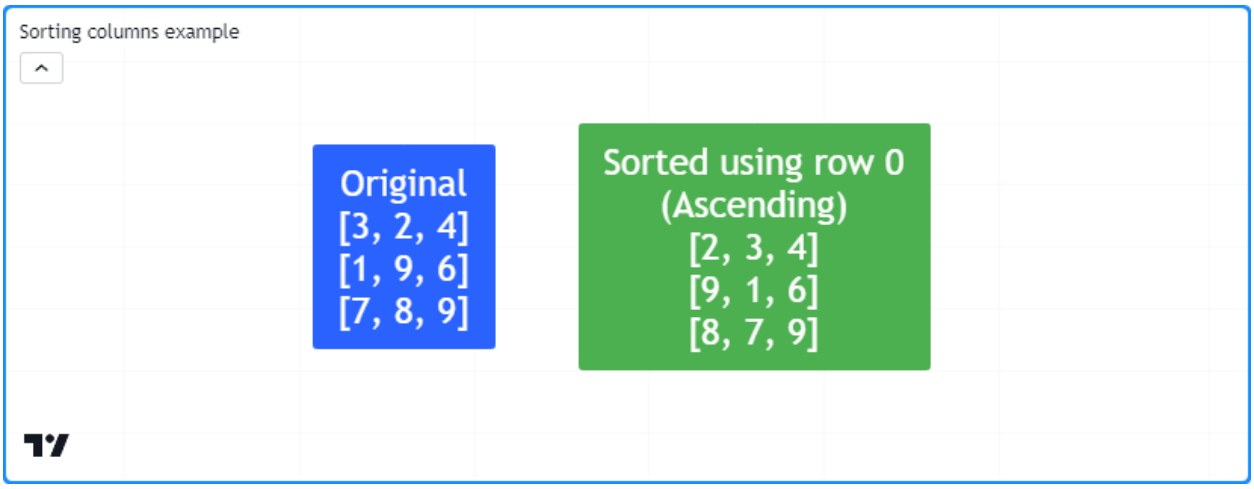
(continues on next page)

(continued from previous page)

```
    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`.
```

(continues on next page)

(continued from previous page)

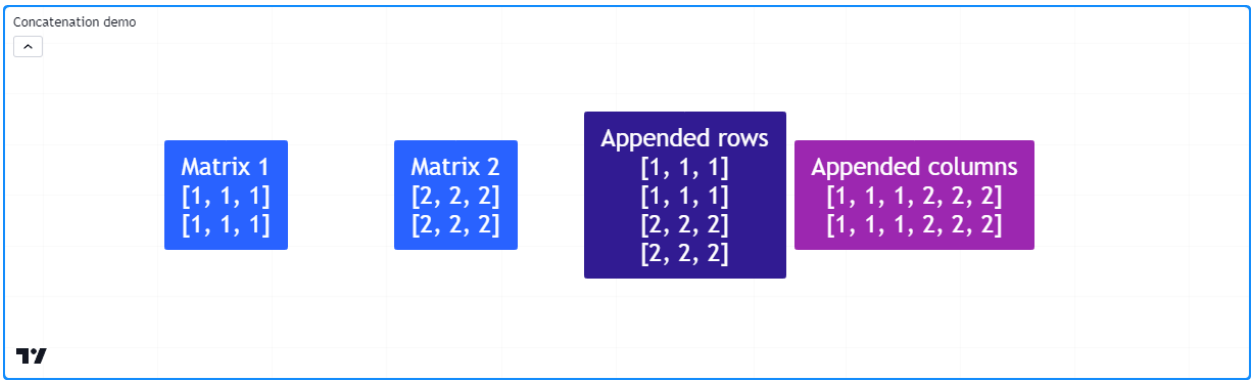
```
method sortColumns(matrix<int> this, int row = 0, bool ascending = true) =>
    //@variable The transpose of `this` matrix.
    matrix<int> thisT = this.transpose()
    //@variable Is `order.ascending` when `ascending` is `true`, `order.descending`␣
↪otherwise.
    order = ascending ? order.ascending : 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.
```

(continues on next page)

(continued from previous page)

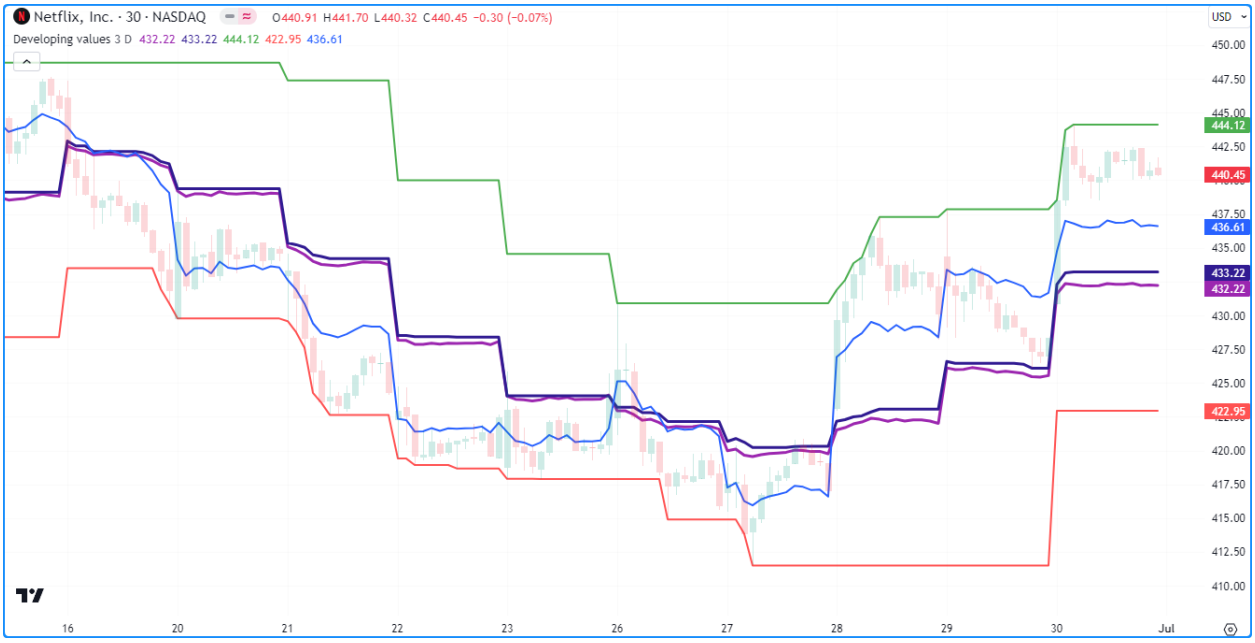
```
//@param    barIndex The `bar_index` to display the label at.
//@param    bgColor The background color of the label.
//@param    textColor The color of the label's text.
//@param    note The text to display above the rows.
method debugLabel(
     matrix<float> this, int barIndex = bar_index, color 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")
```

3.15.10 Matrix calculations

Element-wise calculations

Pine scripts can calculate the *average*, *minimum*, *maximum*, and *mode* of all elements within a matrix via matrix.avg(), matrix.min(), matrix.max(), and matrix.mode(). These functions operate the same as their `array.*` equivalents, allowing users to run element-wise calculations on a matrix, its *submatrices*, and its *rows and columns* using the same syntax. For example, the built-in `*.avg()` functions called on a 3x3 matrix with values 1-9 and an array with the same nine elements will both return a value of 5.

The script below uses `*.avg()`, `*.max()`, and `*.min()` methods to calculate developing averages and extremes of OHLC data in a period. It adds a new column of open, high, low, and close values to the end of the `ohlcData` matrix whenever `queueColumn` is `true`. When `false`, the script uses the get() and set() matrix methods to adjust the elements in the last column for developing HLC values in the current period. It uses the `ohlcData` matrix, a submatrix(), and row() and col() arrays to calculate the developing OHLC4 and HL2 averages over `length` periods, the maximum high and minimum low over `length` periods, and the current period's developing OHLC4 price:




```
//@version=5
indicator("Element-wise calculations example", "Developing values", overlay = true)

//@variable The number of data points in the averages.
int length = input.int(3, "Length", 1)
//@variable The timeframe of each reset period.
string timeframe = input.timeframe("D", "Reset Timeframe")

//@variable A 4x`length` matrix of OHLC values.
var matrix<float> ohlcData = matrix.new<float>(4, length)

//@variable Is `true` at the start of a new bar at the `timeframe`.
bool queueColumn = timeframe.change(timeframe)

if queueColumn
    // Add new values to the end column of `ohlcData`.
    ohlcData.add_col(length, array.from(open, high, low, close))
    // Remove the oldest column from `ohlcData`.
    ohlcData.remove_col(0)
else
    // Adjust the last element of column 1 for new highs.
    if high > ohlcData.get(1, length - 1)
        ohlcData.set(1, length - 1, high)
    // Adjust the last element of column 2 for new lows.
    if low < ohlcData.get(2, length - 1)
        ohlcData.set(2, length - 1, low)
    // Adjust the last element of column 3 for the new closing price.
    ohlcData.set(3, length - 1, close)

//@variable The `matrix.avg()` of all elements in `ohlcData`.
avgOHLC4 = ohlcData.avg()
//@variable The `matrix.avg()` of all elements in rows 1 and 2, i.e., the average of␣
↪all `high` and `low` values.
avgHL2   = ohlcData.submatrix(from_row = 1, to_row = 3).avg()
//@variable The `matrix.max()` of all values in `ohlcData`. Equivalent to `ohlcData.
```

(continues on next page)

(continued from previous page)

```
↪row(1).max()`.
maxHigh = ohlcData.max()
//@variable The `array.min()` of all `low` values in `ohlcData`. Equivalent to␣
↪`ohlcData.min()`.
minLow = ohlcData.row(2).min()
//@variable The `array.avg()` of the last column in `ohlcData`, i.e., the current␣
↪OHLC4.
ohlc4Value = ohlcData.col(length - 1).avg()

plot(avgOHLC4, "Average OHLC4", color.purple, 2)
plot(avgHL2, "Average HL2", color.navy, 2)
plot(maxHigh, "Max High", color.green)
plot(minLow, "Min Low", color.red)
plot(ohlc4Value, "Current OHLC4", color.blue)
```

Note that:

- In this example, we used array.*() and matrix.*() methods interchangeably to demonstrate their similarities in syntax and behavior.
- Users can calculate the matrix equivalent of array.sum() by multiplying the matrix.avg() by the matrix.elements_count().

Special calculations

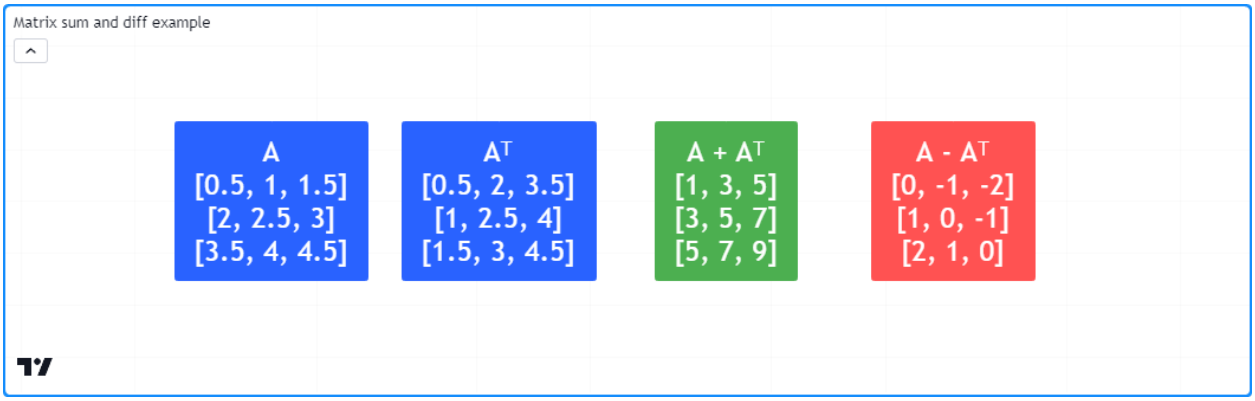
Pine Script™ features several built-in functions for performing essential matrix arithmetic and linear algebra operations, including matrix.sum(), matrix.diff(), matrix.mult(), matrix.pow(), matrix.det(), matrix.inv(), matrix.pinv(), matrix.rank(), matrix.trace(), matrix.eigenvalues(), matrix.eigenvectors(), and matrix.kron(). These functions are advanced features that facilitate a variety of matrix calculations and transformations.

Below, we explain a few fundamental functions with some basic examples.

`matrix.sum()` and `matrix.diff()`

Scripts can perform addition and subtraction of two matrices with the same shape or a matrix and a scalar value using the matrix.sum() and matrix.diff() functions. These functions use the values from the `id2` matrix or scalar to add to or subtract from the elements in `id1`.

This script demonstrates a simple example of matrix addition and subtraction in Pine. It creates a 3x3 matrix, calculates its *transpose*, then calculates the matrix.sum() and matrix.diff() of the two matrices. This example displays the original matrix, its transpose, and the resulting sum and difference matrices in labels on the chart:



```
//@version=5
indicator("Matrix sum and diff example")

//@function Displays the rows of a matrix in a label with a note.
//@param    this The matrix to display.
//@param    barIndex The `bar_index` to display the label at.
//@param    bgColor The background color of the label.
//@param    textColor The color of the label's text.
//@param    note The text to display above the rows.
method debugLabel(
     matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
     color textColor = color.white, string note = ""
 ) =>
    labelText = note + "\n" + str.tostring(this)
    if barstate.ishistory
        label.new(
             barIndex, 0, labelText, color = bgColor, style = label.style_label_
↪center,
             textcolor = textColor, size = size.huge
         )

//@variable A 3x3 matrix.
m = matrix.new<float>()

// Add rows to `m`.
m.add_row(0, array.from(0.5, 1.0, 1.5))
m.add_row(1, array.from(2.0, 2.5, 3.0))
m.add_row(2, array.from(3.5, 4.0, 4.5))

if bar_index == last_bar_index - 1
    // Display `m`.
    m.debugLabel(note = "A")
    // Get and display the transpose of `m`.
    matrix<float> t = m.transpose()
    t.debugLabel(bar_index + 10, note = "A")
    // Calculate the sum of the two matrices. The resulting matrix is symmetric.
    matrix.sum(m, t).debugLabel(bar_index + 20, color.green, note = "A + A")
    // Calculate the difference between the two matrices. The resulting matrix is␣
↪antisymmetric.
    matrix.diff(m, t).debugLabel(bar_index + 30, color.red, note = "A - A")
```

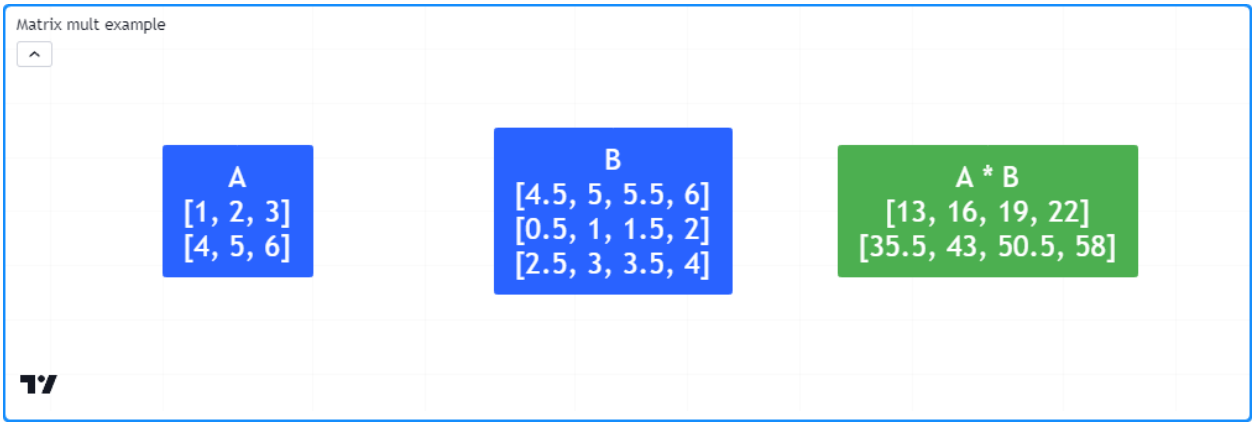
Note that:

- 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))

```

(continues on next page)

(continued from previous page)

```
// Add rows to `b`.
b.add_row(0, array.from(0.5, 1.0, 1.5, 2.0))
b.add_row(1, array.from(2.5, 3.0, 3.5, 4.0))
b.add_row(0, array.from(4.5, 5.0, 5.5, 6.0))

if bar_index == last_bar_index - 1
    //@variable The result of `a` * `b`.
    matrix<float> ab = a.mult(b)
    // Display `a`, `b`, and `ab` matrices.
    debugLabel(a, note = "A")
    debugLabel(b, bar_index + 10, note = "B")
    debugLabel(ab, bar_index + 20, color.green, note = "A * B")
```

Note that:

- In contrast to the multiplication of scalars, matrix multiplication is *non-commutative*, i.e., `matrix.mult(a, b)` does not necessarily produce the same result as `matrix.mult(b, a)`. In the context of our example, the latter will raise a runtime error because the number of columns in `b` doesn't equal the number of rows in `a`.

When multiplying a matrix and an array, this function treats the operation the same as multiplying `id1` by a single-column matrix, but it returns an array with the same number of elements as the number of rows in `id1`. When matrix.mult() passes a scalar as its `id2` value, the function returns a new matrix whose elements are the elements in `id1` multiplied by the `id2` value.

`matrix.det()`

A *determinant* is a scalar value associated with a square matrix that describes some of its characteristics, namely its invertibility. If a matrix has an inverse, its determinant is nonzero. Otherwise, the matrix is *singular* (non-invertible). Scripts can calculate the determinant of a matrix via matrix.det().

Programmers can use determinants to detect similarities between matrices, identify *full-rank* and *rank-deficient* matrices, and solve systems of linear equations, among other applications.

For example, this script utilizes determinants to solve a system of linear equations with a matching number of unknown values using Cramer's rule. The user-defined `solve()` function returns an array containing solutions for each unknown value in the system, where the n-th element of the array is the determinant of the coefficient matrix with the n-th column replaced by the column of constants divided by the determinant of the original coefficients.

In this script, we've defined the matrix `m` that holds coefficients and constants for these three equations:

```
3 * x0 + 4 * x1 - 1 * x2 = 8
5 * x0 - 2 * x1 + 1 * x2 = 4
2 * x0 - 2 * x1 + 1 * x2 = 1
```

The solution to this system is `(x0 = 1, x1 = 2, x2 = 3)`. The script calculates these values from `m` via `m.solve()` and plots them on the chart:


```
//@version=5
indicator("Determinants example", "Cramer's Rule")

//@function Solves a system of linear equations with a matching number of unknowns using Cramer's rule.
//@param    this An augmented matrix containing the coefficients for each unknown and the results of
//          the equations. For example, a row containing the values 2, -1, and 3 represents the equation
//          `2 * x0 + (-1) * x1 = 3`, where `x0` and `x1` are the unknown values in the system.
//@returns  An array containing solutions for each variable in the system.
solve(matrix<float> this) =>
    //@variable The coefficient matrix for the system of equations.
    matrix<float> coefficients = this.submatrix(from_column = 0, to_column = this.columns() - 1)
    //@variable The array of resulting constants for each equation.
    array<float> constants = this.col(this.columns() - 1)
    //@variable An array containing solutions for each unknown in the system.
    array<float> result = array.new<float>()

    //@variable The determinant value of the coefficient matrix.
    float baseDet = coefficients.det()
    matrix<float> modified = na
    for col = 0 to coefficients.columns() - 1
        modified := coefficients.copy()
        modified.add_col(col, constants)
        modified.remove_col(col + 1)

        // Calculate the solution for the column's unknown by dividing the determinant of `modified` by the `baseDet`.
        result.push(modified.det() / baseDet)

    result

//@variable A 3x4 matrix containing coefficients and results for a system of three equations.
m = matrix.new<float>()

// Add rows for the following equations:
// Equation 1: 3 * x0 + 4 * x1 - 1 * x2 = 8
// Equation 2: 5 * x0 - 2 * x1 + 1 * x2 = 4
// Equation 3: 2 * x0 - 2 * x1 + 1 * x2 = 1
m.add_row(0, array.from(3.0, 4.0, -1.0, 8.0))
m.add_row(1, array.from(5.0, -2.0, 1.0, 4.0))
m.add_row(2, array.from(2.0, -2.0, 1.0, 1.0))
```

(continues on next page)

(continued from previous page)

```
40
41 //@variable An array of solutions to the unknowns in the system of equations␣
   ↪represented by `m`.
42 solutions = solve(m)
43
44 plot(solutions.get(0), "x0", color.red, 3)   // Plots 1.
45 plot(solutions.get(1), "x1", color.green, 3) // Plots 2.
46 plot(solutions.get(2), "x2", color.blue, 3)  // Plots 3.
```

Note that:

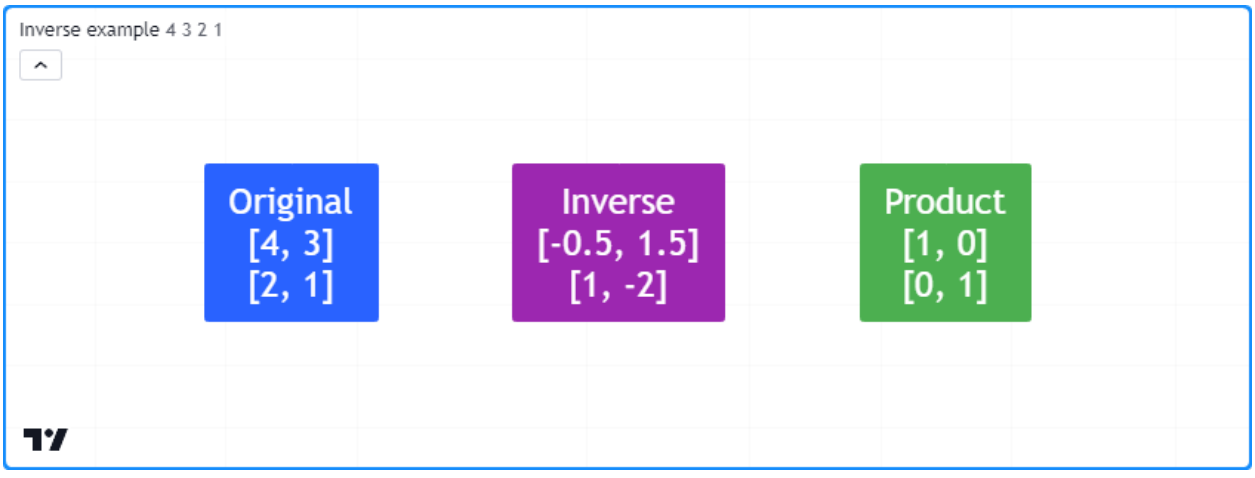
- Solving systems of equations is particularly useful for *regression analysis*, e.g., linear and polynomial regression.
- Cramer's rule works fine for small systems of equations. However, it's computationally inefficient on larger systems. Other methods, such as Gaussian elimination, are often preferred 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:




```
1 //@version=5
2 indicator("Inverse example")
3
4 // Element inputs for the 2x2 matrix.
5 float r0c0 = input.float(4.0, "Row 0, Col 0")
6 float r0c1 = input.float(3.0, "Row 0, Col 1")
7 float r1c0 = input.float(2.0, "Row 1, Col 0")
```

(continues on next page)

(continued from previous page)

```
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.

3.15.11 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.
```

(continues on next page)

(continued from previous page)

```
9
10 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:

```
1  //@version=5
2  indicator("na matrix methods demo")
3
4  //@variable A `matrix` variable assigned to `na`.
5  matrix<float> m = na
6
7  mCopy = m.copy() // Raises a runtime error. You can't copy a matrix that doesn't␣
   ↪exist.
8
9  if bar_index == last_bar_index - 1
10     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:

```
1  //@version=5
2  indicator("Matrix too large demo")
3
4  var matrix<float> m = matrix.new<float>()
5
6  if bar_index == 0
7      for i = 1 to 1000
8          // This raises an error because the script adds 101 elements on each␣
   ↪iteration.
9          // 1000 rows * 101 elements per row = 101000 total elements. This is too␣
   ↪large.
10         m.add_row(m.rows(), array.new<float>(101, i))
11
12 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)
```

(continues on next page)

(continued from previous page)

```
8
9  mSum = matrix.mult(m1, m2) // Raises an error. The number of columns in `m1` and rows␣
   ↪in `m2` aren't equal.
10
11 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:

```
1  //@version=5
2  indicator("Non-square demo")
3
4  //@variable A 3x5 matrix.
5  matrix<float> m = matrix.new<float>(3, 5, 1)
6
7  plot(m.det()) // Raises a runtime error. You can't calculate the determinant of a 3x5␣
   ↪matrix.
```

TradingView

Advanced

Pine Script™

3.16 Maps

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.

3.16.1 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.

3.16.2 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.

3.16.3 Reading and writing

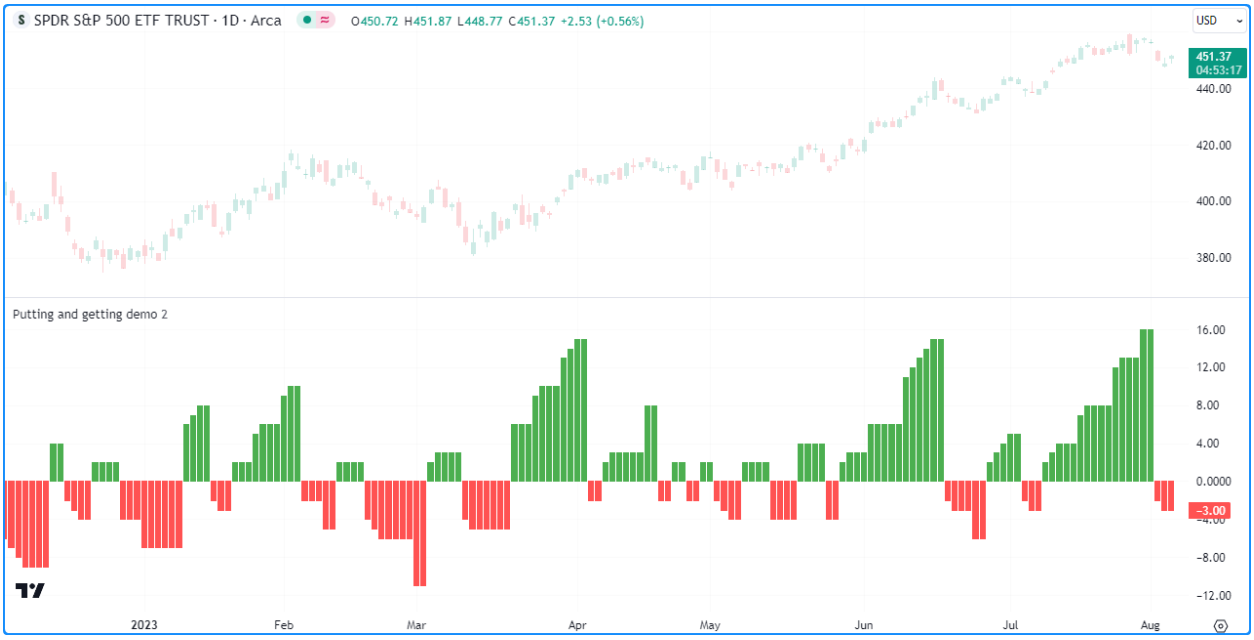
Putting and getting key-value pairs

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)
```

(continues on next page)

(continued from previous page)

```

// Put the "Difference" between current "Rising" and "Falling" values into the `data`␣
↪map.
data.put("Difference", data.get("Rising") - data.get("Falling"))

//@variable The difference between the last "Rising" and "Falling" `bar_index`.
int index = data.get("Difference")

//@variable Returns `color.green` when `index` is positive, `color.red` when negative,
↪ and `color.gray` otherwise.
color indexColor = index > 0 ? color.green : index < 0 ? color.red : color.gray

plot(index, color = indexColor, style = plot.style_columns)
```

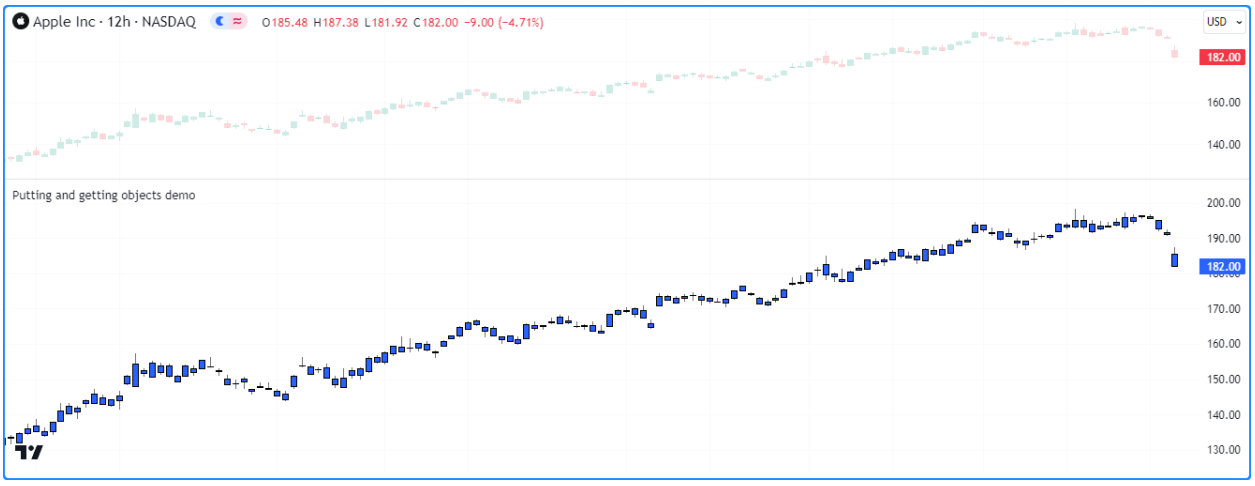
Note that:

- This script replaces the values associated with the “Rising”, “Falling”, and “Difference” keys on successive data.put() calls, as each of these keys is unique and can only appear once in the `data` map.
- Replacing the pairs in a map does not change the internal *insertion order* of its keys. We discuss this further in the *next section*.

Similar to working with other collections, when putting a value of a *special type* (line, linefill, box, polyline, label, table, or chart.point) or a *user-defined type* into a map, it’s important to note the inserted pair’s `value` points to that same object without copying it. Modifying the value referenced by a key-value pair will also affect the original object.

For example, this script contains a custom `ChartData` type with `o`, `h`, `l`, and `c` fields. On the first chart bar, the script declares a `myMap` variable and adds the pair `("A", myData)`, where `myData` is a `ChartData` instance with initial field values of `na`. It adds the pair `("B", myData)` to `myMap` and updates the object from this pair on every bar via the user-defined `update()` method.

Each change to the object with the “B” key affects the one referenced by the “A” key, as shown by the candle plot of the “A” object’s fields:




```
//@version=5
indicator("Putting and getting objects demo")

//@type A custom type to hold OHLC data.
type ChartData
    float o
    float h
```

(continues on next page)

(continued from previous page)

```
    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*.

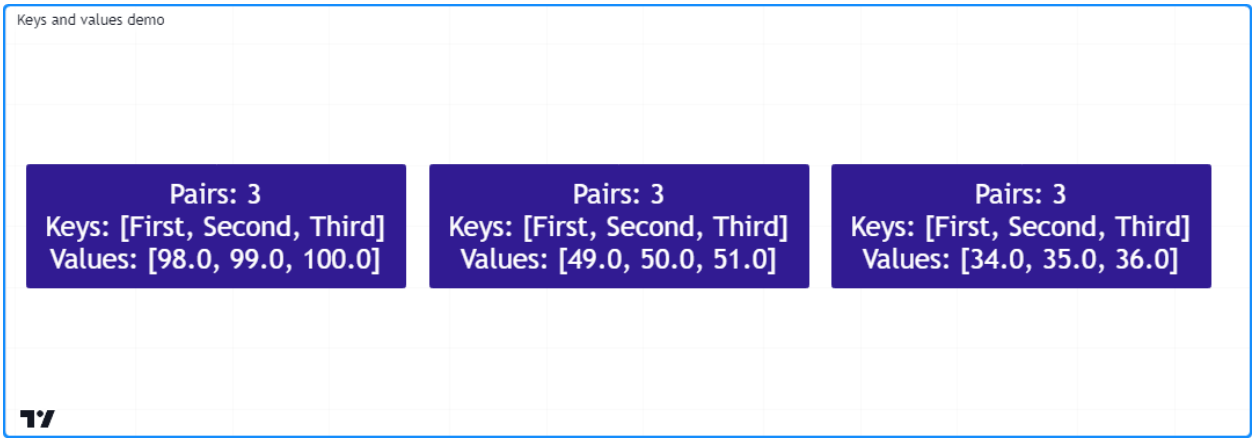
Inspecting keys and values

`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`:



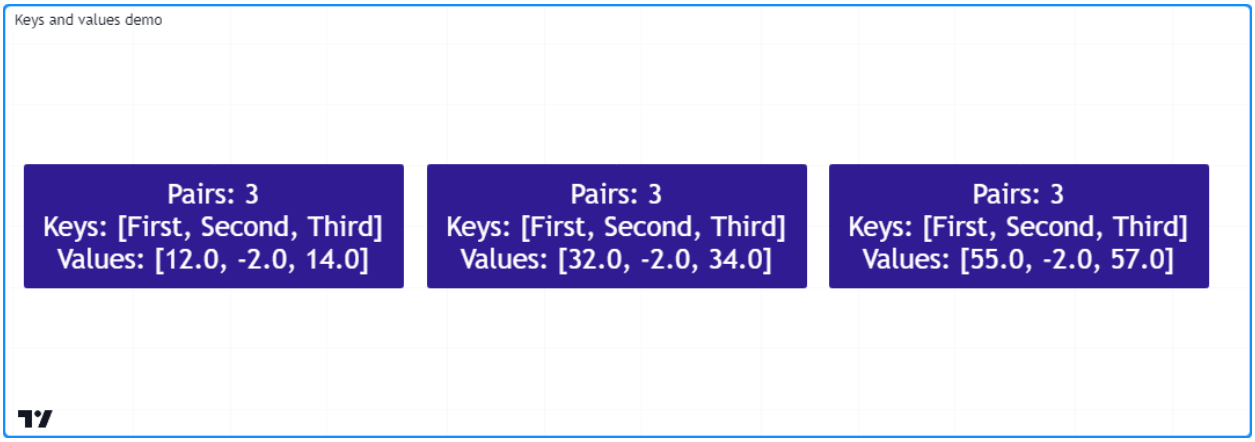

```
//@version=5
indicator("Keys and values demo")

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:



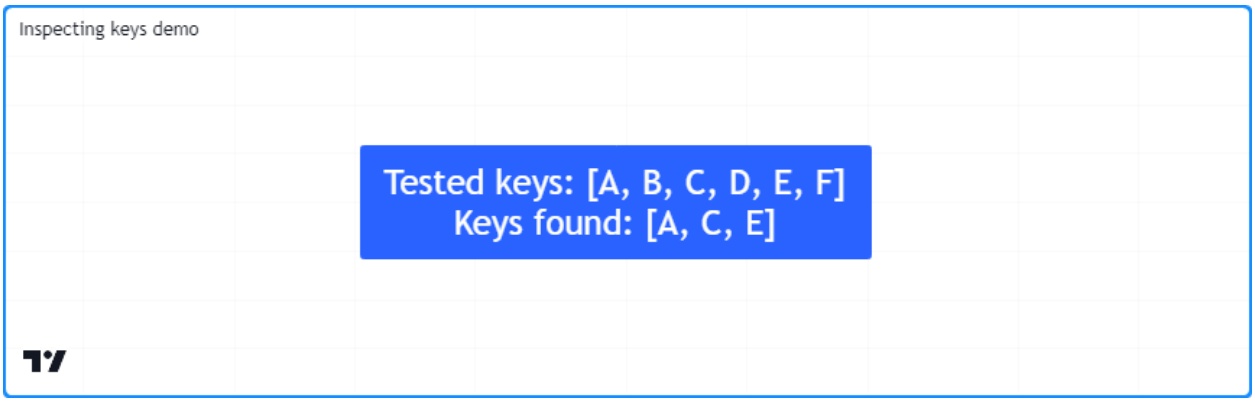

```
//@version=5
indicator("Keys and values demo")

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

`map.contains()`

To check if a specific `key` exists within a map `id`, use map.contains(). This function is a convenient alternative to calling array.includes() on the array returned from map.keys().

For example, this script checks if various keys exist within an `m` map, then displays the results in a label:



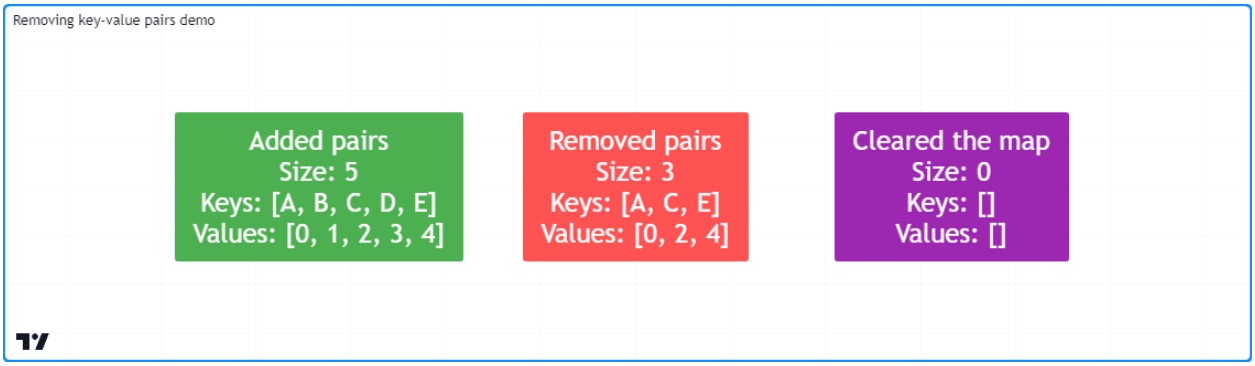

```
//@version=5
indicator("Inspecting keys demo")

//@variable A map containing `string` keys and `string` values.
m = map.new<string, string>()

// Put key-value pairs into the map.
m.put("A", "B")
m.put("C", "D")
m.put("E", "F")

//@variable An array of keys to check for in `m`.
array<string> testKeys = array.from("A", "B", "C", "D", "E", "F")

//@variable An array containing all elements from `testKeys` found in the keys of `m`.
array<string> mappedKeys = array.new<string>()

for key in testKeys
    // Add the `key` to `mappedKeys` if `m` contains it.
    if m.contains(key)
        mappedKeys.push(key)

//@variable A string representing the `testKeys` array and the elements found within␣
↪the keys of `m`.
string testText = str.format("Tested keys: {0}\nKeys found: {1}", testKeys,␣
↪mappedKeys)

if bar_index == last_bar_index - 1
    //@variable Displays the `testText` in a label at the `bar_index` before the last.
    label debugLabel = label.new(
         bar_index, 0, testText, style = label.style_label_center,
         textcolor = color.white, size = size.huge
     )
```

Removing key-value pairs

To remove a specific key-value pair from a map `id`, use map.remove(). This function removes the `key` and its associated value from the map while preserving the insertion order of other key-value pairs. It returns the removed value if the map *contained* the `key`. Otherwise, it returns na.

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

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




```
//@version=5
indicator("Removing key-value pairs demo")

//@function Returns a label to display the keys and values from a map.
method debugLabel(
     map<string, int> this, int barIndex = bar_index,
     color bgColor = color.blue, string note = ""
 ) =>
    //@variable A string representing the size, keys, and values in `this` map.
    string repr = str.format(
         "{0}\nSize: {1}\nKeys: {2}\nValues: {3}",
         note, this.size(), str.tostring(this.keys()), str.tostring(this.values())
     )
    label.new(
         barIndex, 0, repr, color = bgColor, style = label.style_label_center,
         textcolor = color.white, size = size.huge
     )

if bar_index == last_bar_index - 1
    //@variable A map containing `string` keys and `int` values.
    m = map.new<string, int>()

    // Put key-value pairs into `m`.
    for [i, key] in array.from("A", "B", "C", "D", "E")
        m.put(key, i)
    m.debugLabel(bar_index, color.green, "Added pairs")

    //@variable An array of keys to remove from `m`.
    array<string> removeKeys = array.from("B", "B", "D", "F", "a")

    // Remove each `key` in `removeKeys` from `m`.
    for key in removeKeys
        m.remove(key)
```

(continues on next page)

(continued from previous page)

```
    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:

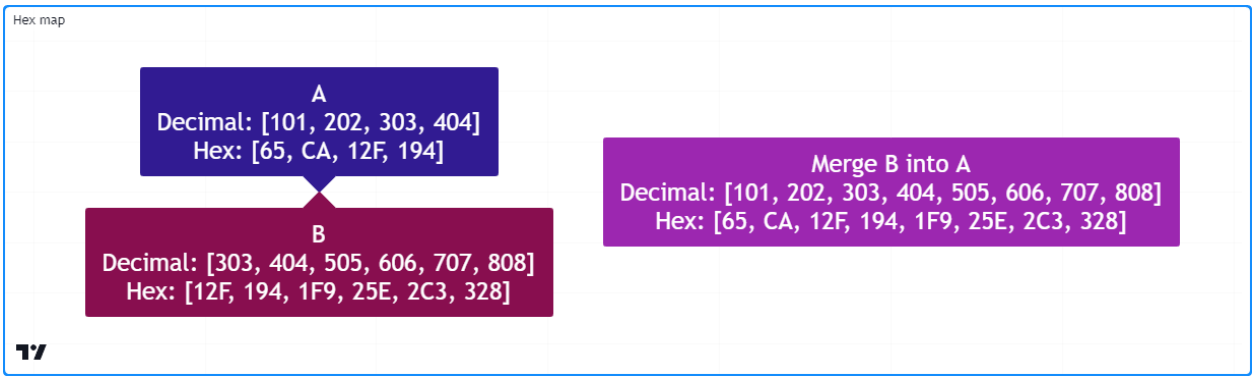
- Not all strings in the `removeKeys` array were present in the keys of `m`. Attempting to remove non-existent keys ("F", "a", and the second "B" in this example) has no effect on a map's contents.

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.
```

(continues on next page)

(continued from previous page)

```
        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")
```

3.16.4 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:

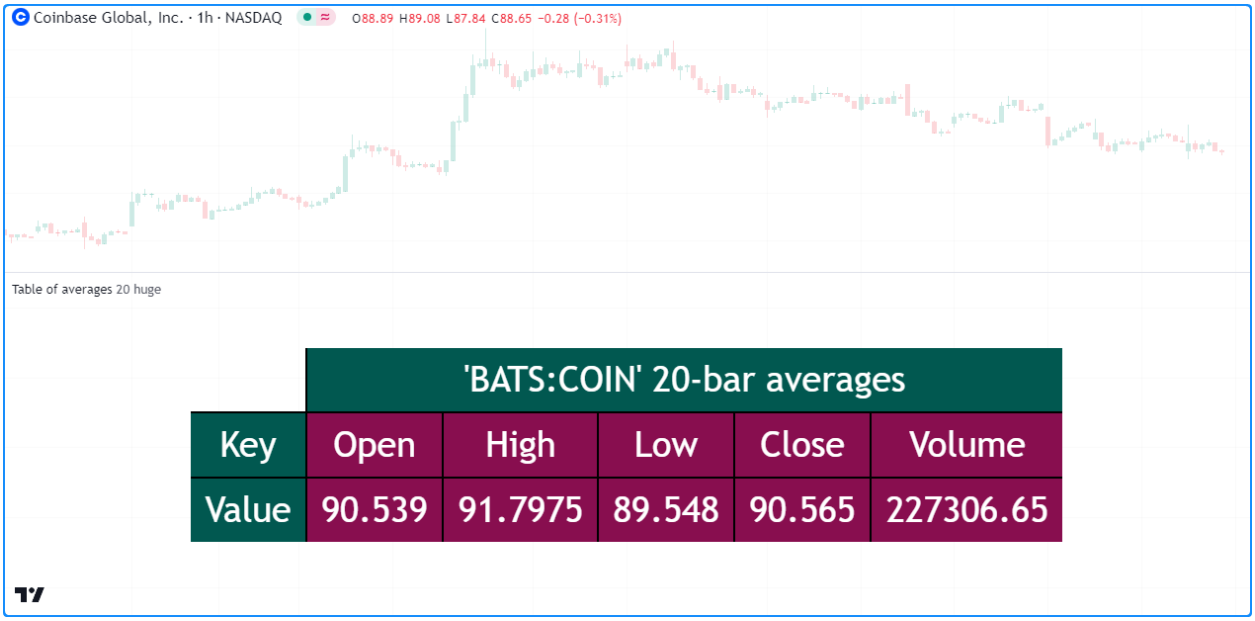
```
for key in thisMap.keys()
    value = thisMap.get(key)
```

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

```

(continues on next page)

(continued from previous page)

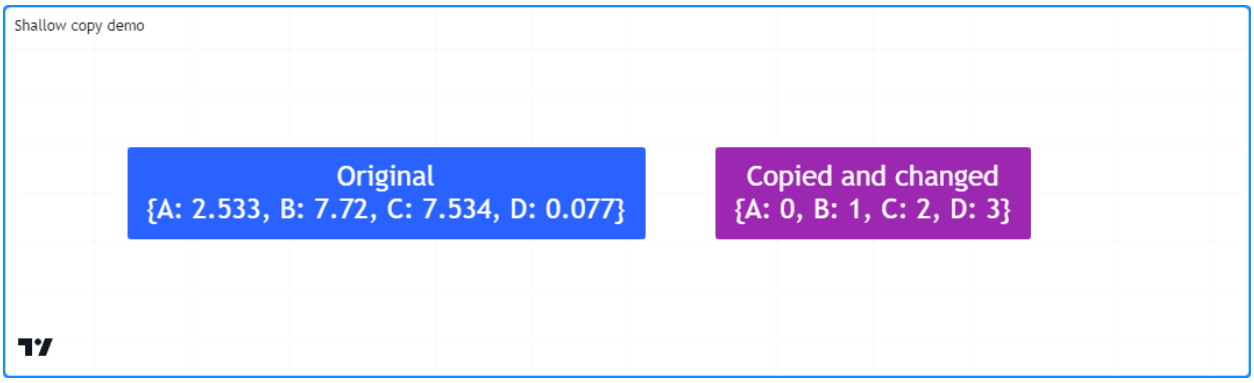
```
    //@variable A table that displays the key-value pairs of `this` map.
    table result = table.new(
         position, this.size() + 1, 3, border_width = 2, border_color = borderColor
     )
    // Initialize top and side header cells.
    result.cell(1, 0, header, bgcolor = headerColor, text_color = 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)
```

3.16.5 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`.
```

(continues on next page)

(continued from previous page)

```
    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")
```

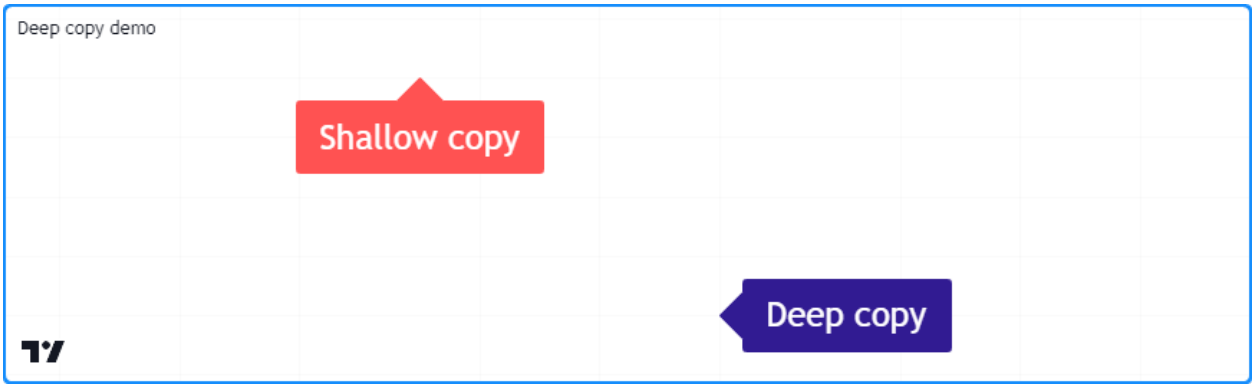
Deep copies

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, box, polyline, label, table, chart.point or a *UDT*) point to the same objects as the original. Modifying the objects referenced by a shallow copy will affect the instances referenced by the original map and vice versa.

To ensure changes to objects referenced by a copied map do not affect instances referenced in other locations, one can make a *deep copy* by creating a new map with key-value pairs containing copies of each value in the original map.

This example creates an `original` map of string keys and label values and puts a key-value pair into it. The script copies the map to a `shallow` variable via the built-in copy() method, then to a `deep` variable using a custom `deepCopy()` method.

As we see from the chart, changes to the label retrieved from the `shallow` copy also affect the instance referenced by the `original` map, but changes to the one from the `deep` copy do not:




```
//@version=5
indicator("Deep copy demo")

//@function Returns a deep copy of `this` map.
method deepCopy(map<string, label> this) =>
    //@variable A deep copy of `this` map.
    result = map.new<string, label>()
    // Add key-value pairs with copies of each `value` to the `result`.
    for [key, value] in this
        result.put(key, value.copy())
    result //Return the `result`.

//@variable A map containing `string` keys and `label` values.
var original = map.new<string, label>()

if bar_index == last_bar_index - 1
    // Put a new key-value pair into the `original` map.
    map.put(
```

(continues on next page)

(continued from previous page)

```
        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.

3.16.6 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)
```

3.16.7 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:

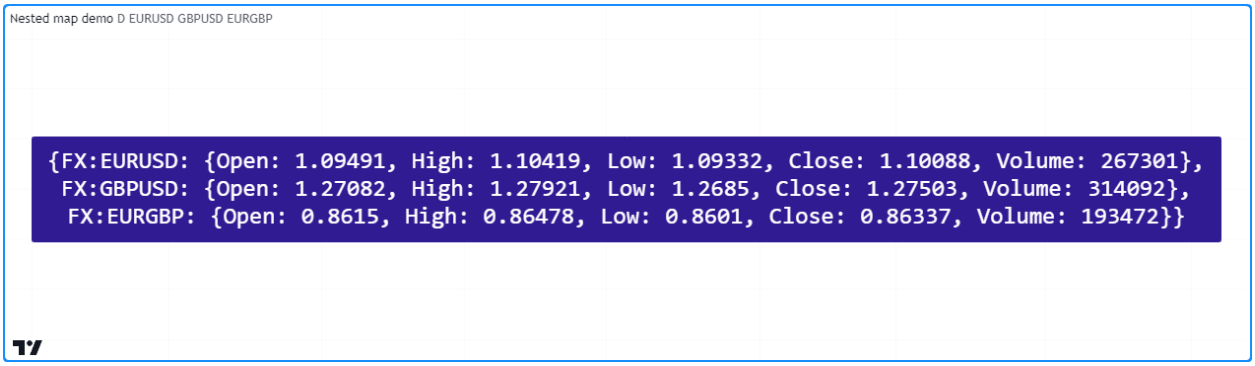
```
//@type A wrapper type for maps with `string` keys and `float` values.
type Wrapper
    map<string, float> data
```

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,
```

(continues on next page)

(continued from previous page)

```
         [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)
```

CHAPTER

FOUR

CONCEPTS

4.1 Alerts

4.1.1 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.

4.1.2 Script alerts

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

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

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

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

`alert()` function events

The alert() function has the following signature:

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

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

freq
An "input string" specifying the triggering frequency of the alert. Valid arguments are:

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

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

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

Using all `alert()` calls

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

```
1 //@version=5
2 indicator("All `alert()` calls")
3 r = ta.rsi(close, 20)
4
5 // Detect crosses.
6 xUp = ta.crossover( r, 50)
7 xDn = ta.crossunder(r, 50)
```

(continues on next page)

(continued from previous page)

```
// 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)
```

(continues on next page)

(continued from previous page)

```
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)

```

(continues on next page)

(continued from previous page)

```
// 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`.

4.1.3 `alertcondition()` events

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

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

The alertcondition() function has the following signature:

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

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

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

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

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

Using one condition

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

```
//@version=5
indicator("`alertcondition()` on single condition")
r = ta.rsi(close, 20)

xUp = ta.crossover( r, 50)
xDn = ta.crossunder(r, 50)

plot(r, "RSI")
hline(50)
plotchar(xUp, "Long",  "▲", location.bottom, color.lime, size = size.tiny)
plotchar(xDn, "Short", "▼", location.top,    color.red,  size = size.tiny)

alertcondition(xUp, "Long Alert",  "Go long")
alertcondition(xDn, "Short Alert", "Go short ")
```

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

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

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

Note that:

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

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

Using compound conditions

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

This script demonstrates one way to do it:

```
//@version=5
indicator("`alertcondition()` on multiple conditions")
detectLongsInput  = input.bool(true, "Detect Longs")
detectShortsInput = input.bool(true, "Detect Shorts")

r = ta.rsi(close, 20)
// Detect crosses.
xUp = ta.crossover( r, 50)
xDn = ta.crossunder(r, 50)
// Only generate entries when the trade's direction is allowed in inputs.
enterLong  = detectLongsInput  and xUp
enterShort = detectShortsInput and xDn

plot(r)
plotchar(enterLong,  "Go Long",  "▲", location.bottom, color.lime, size = size.tiny)
plotchar(enterShort, "Go Short", "▼", location.top,    color.red,  size = size.tiny)
hline(50)
// Trigger the alert when one of the conditions is met.
alertcondition(enterLong or enterShort, "Compound alert", "Entry")
```

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

Placeholders

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

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

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

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

{{open}}, {{high}}, {{low}}, {{close}}, {{volume}}
: Corresponding values of the bar on which the alert has been triggered.

{{plot_0}}, {{plot_1}}, [...], {{plot_19}}
: Value of the corresponding plot number. Plots are numbered from zero to 19 in order of appearance in the script,

so only one of the first 20 plots can be used. For example, the built-in "Volume" indicator has two output series: Volume and Volume MA, so you could use the following:

```
alertcondition(volume > 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:

```
1 //@version=5
2 indicator("")
3 r = ta.rsi(close, 14)
4 xUp = ta.crossover(r, 50)
5 plot(r, "RSI", display = display.none)
6 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.

4.1.4 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.

4.2 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 be generated using the color.new() function.

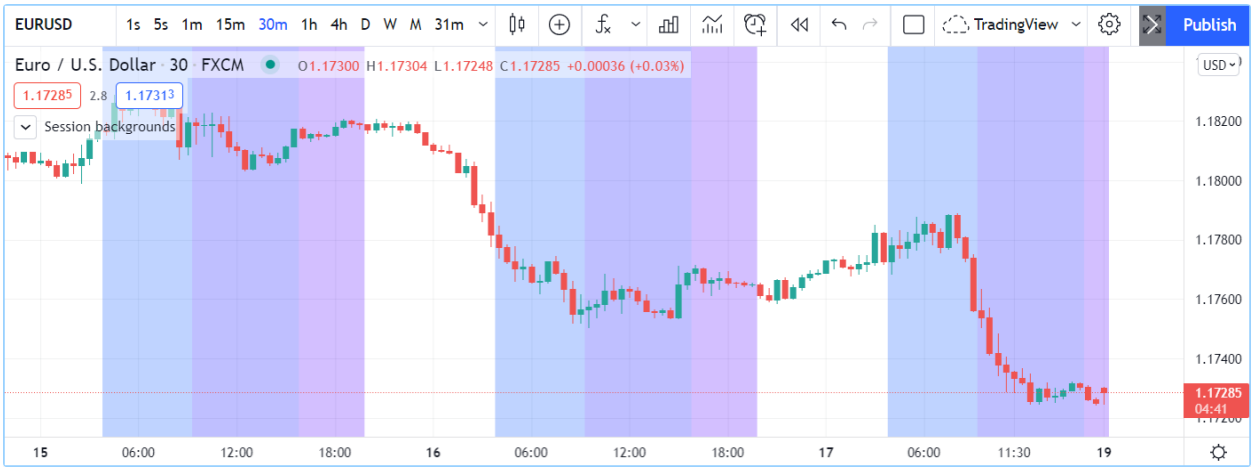
Here is a script that colors the background of trading sessions (try it on 30min EURUSD, for example):

```
//@version=5
indicator("Session backgrounds", overlay = true)

// Default color constants using tranparency of 25.
BLUE_COLOR   = #0050FF40
PURPLE_COLOR = #0000FF40
PINK_COLOR   = #5000FF40
NO_COLOR     = color(na)

// Allow user to change the colors.
preMarketColor  = input.color(BLUE_COLOR, "Pre-market")
regSessionColor = input.color(PURPLE_COLOR, "Pre-market")
postMarketColor = input.color(PINK_COLOR, "Pre-market")

// Function returns `true` when the bar's time is
timeInRange(tf, session) =>
    time(tf, session) != 0

// Function prints a message at the bottom-right of the chart.
f_print(_text) =>
    var table _t = table.new(position.bottom_right, 1, 1)
    table.cell(_t, 0, 0, _text, bgcolor = color.yellow)

var chartIs30MinOrLess = timeframe.isseconds or (timeframe.isintraday and timeframe.
↪multiplier <=30)
sessionColor = if chartIs30MinOrLess
    switch
        timeInRange(timeframe.period, "0400-0930") => preMarketColor
        timeInRange(timeframe.period, "0930-1600") => regSessionColor
        timeInRange(timeframe.period, "1600-2000") => postMarketColor
        => NO_COLOR
else
    f_print("No background is displayed.\nChart timeframe must be <= 30min.")
```

(continues on next page)

(continued from previous page)

```
    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.

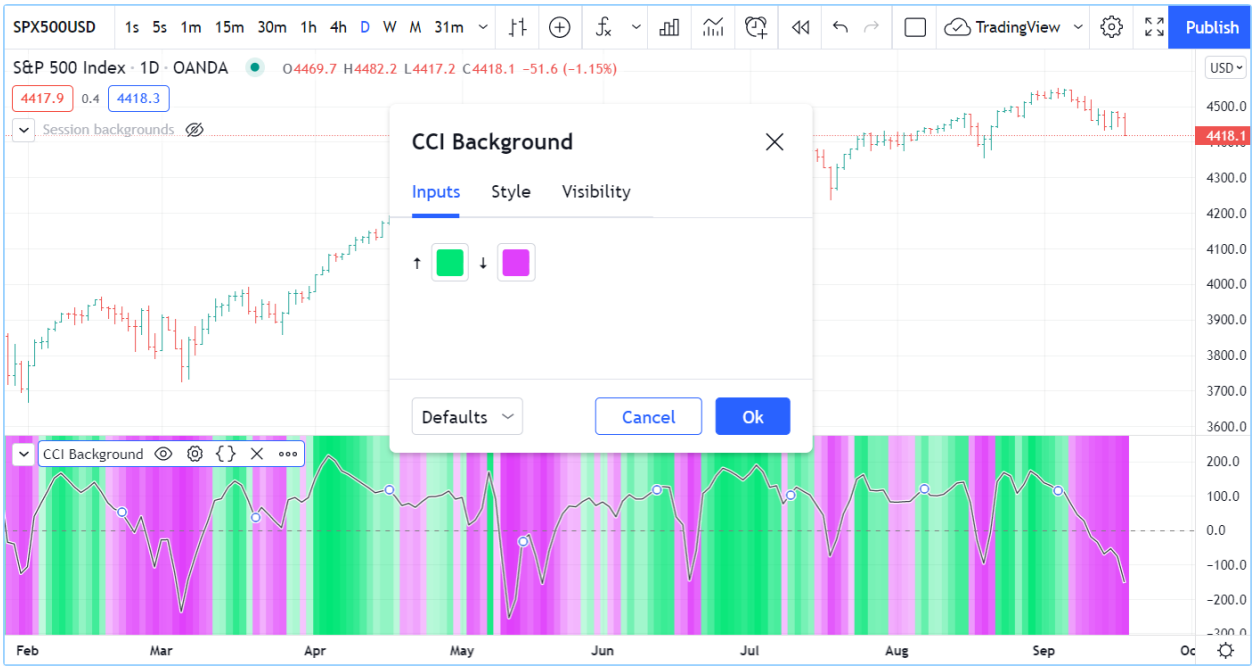
In our next example, we generate a gradient for the background of a CCI line:

```
//@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.
```

(continues on next page)

(continued from previous page)

```
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.

4.3 Bar coloring

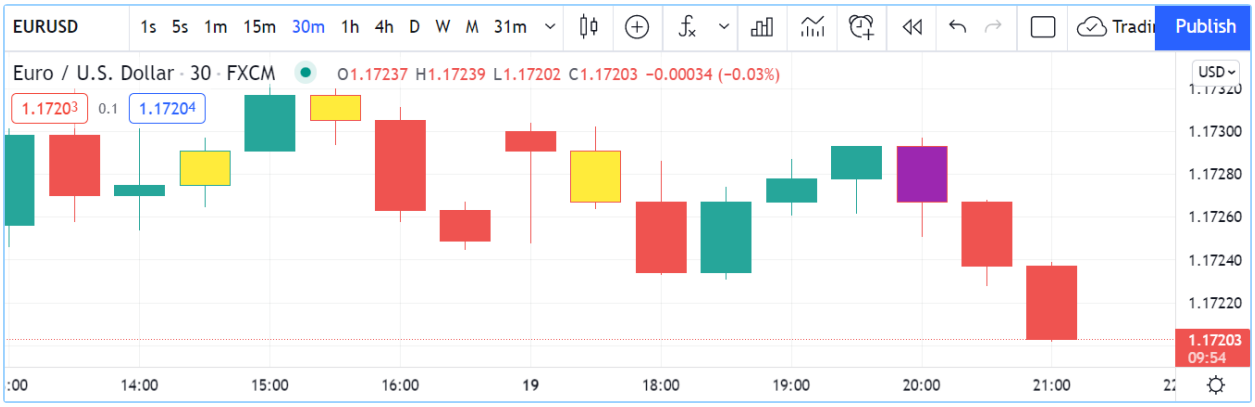
The barcolor() function lets you color chart bars. It is the only Pine Script™ function that allows a script running in a pane to affect the chart.

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:




```
1 //@version=5
2 indicator("barcolor example", overlay = true)
3 isUp = close > open
4 isDown = close <= open
5 isOutsideUp = high > high[1] and low < low[1] and isUp
6 isOutsideDown = high > high[1] and low < low[1] and isDown
7 isInside = high < high[1] and low > low[1]
8 barcolor(isInside ? color.yellow : isOutsideUp ? color.aqua : isOutsideDown ? color.
  ↪purple : na)
```

Note that:

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

4.4 Bar plotting

4.4.1 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.

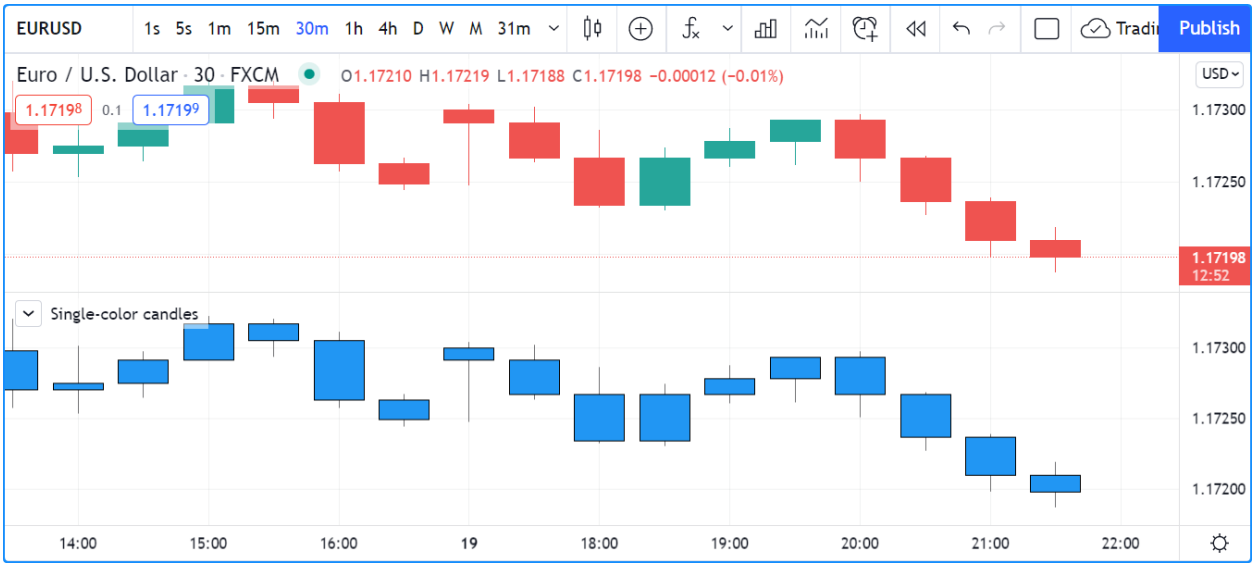
4.4.2 Plotting candles with `plotcandle()`

The signature of plotcandle() is:

```
plotcandle(open, high, low, close, title, color, wickcolor, editable, show_last,␣
↪bordercolor, display) → void
```

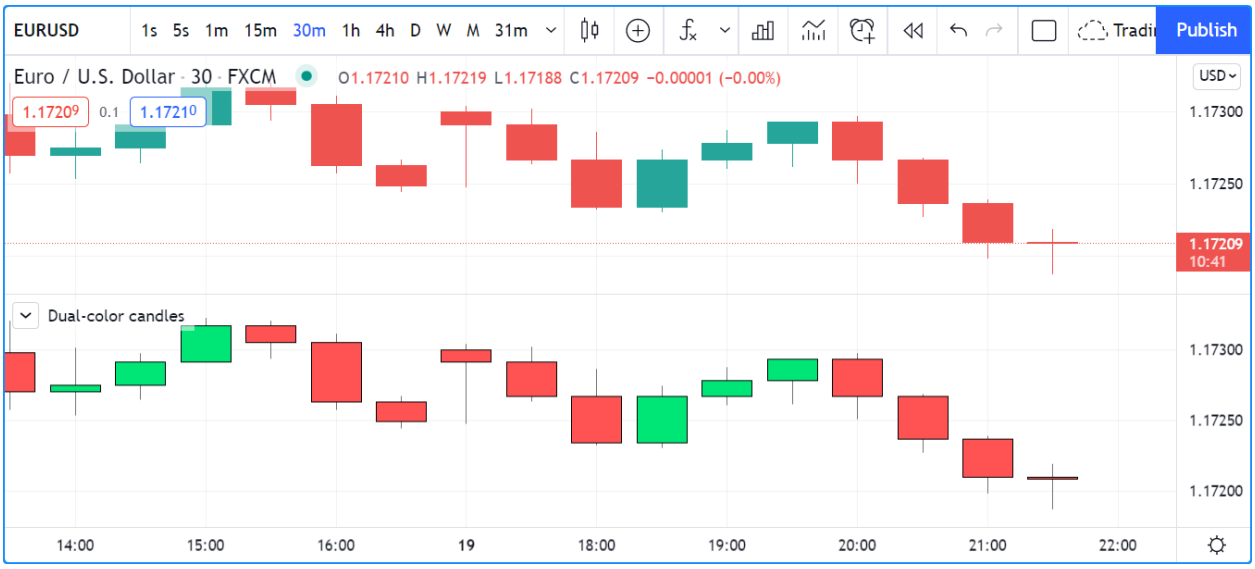
This plots simple candles, all in blue, using the habitual OHLC values, in a separate pane:

```
1 //@version=5
2 indicator("Single-color candles")
3 plotcandle(open, high, low, close)
```



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

```
1 //@version=5
2 indicator("Example 2")
3 paletteColor = close >= open ? color.lime : color.red
4 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)
```