Pine Script® v6 Documentation

2 Language

TradingView Pine Script Programming Language

Table of Contents

Redirect

Pine Script® v6 Documentation

Execution model

Execution model

Introduction

Pine Script® relies on an event-driven, sequential execution model to control how a script's compiled source code runs in charts, alerts, Deep Backtesting mode, and the Pine Screener.

In contrast to the traditional execution model of most programming languages, Pine's runtime system executes a script *repeatedly* on the sequence of *historical bars* and *realtime ticks* in the dataset on which it runs, performing *separate* calculations for *each bar* as it progresses. After each execution on a closed bar, the necessary data from that execution becomes part of an internal time series, and the script can use that data in its calculations on subsequent bars.

This combination of sequential executions and storage enables programmers to use minimal code to write scripts with dynamic calculations that advance across a dataset bar by bar.

The execution model and time series structure closely connect to the type system — together, they define how a script behaves as it runs on a dataset. Although it's possible to write simple scripts without understanding these foundational topics, learning about them and their nuances is key to becoming proficient in Pine Script.

This page explains the execution model in two parts: The basics and The details. The first part provides quick, actionable information about the model for beginners. The second part offers an *advanced*, in-depth breakdown of the model's workings and unique behaviors. To make the most of the information on this page, we recommend that newcomers to Pine Script start with The basics, learn about other topics in this manual, and then come back to this page for the advanced details.

The basics

The following sections outline core principles of the execution model for beginners. If you are new to Pine Script, start here.

Bar-by-bar execution

The dataset for a symbol on a given timeframe, as shown on a chart, consists of a sequence of bars representing a *time series*. Each bar in the sequence represents the price and volume for a specific time period. The first (leftmost) bar on a chart corresponds to the *earliest* period, and the last (rightmost) bar corresponds to the *most recent* period.

Much of the power of Pine Script stems from its ability to process this time series data efficiently. When a user runs a script, its code does *not* execute just once; it executes from start to end on *each bar* in the symbol's dataset individually, progressing from the first available bar to the most recent bar. Each separate script execution performs calculations or generates outputs (e.g., plots) for a *specific bar* using the data available on that bar.

A script can retrieve price, volume, and other essential data for each bar on which it executes by using the built-in variables that hold bar information, such as open, high, low, close, and volume. These variables automatically *update* before each new execution to store the values for the *current bar*.

For example, the simple script below uses the plot() function to display the series of close values (i.e., the closing price of each bar) on the chart:

```
//@version=6 indicator("Bar-by-bar execution demo", overlay = true,
behind_chart = false) // Plot the `close` series on the chart. // This call
defines the plotted point for the current bar on each execution.
plot(close, "Close price", chart.fg_color, 5)
```

When a user first adds this script to their chart, its code executes *once* for *every bar* in the available dataset. As the script runs on the data, two primary steps occur on each bar:

1. The close variable automatically updates to hold the current bar’s latest price.
2. The plot() function call plots the updated close value at the current bar’s position.

When the script finishes its run from the first bar to the most recent bar, the result is a simple *line plot* showing the progression of closing prices across the chart’s history:

image

Note that the above script evaluates the plot() function call once for every bar on the chart, not just once in total. On each separate execution, the call defines the plotted point for the current bar: the chart’s first bar during the first execution, the second bar during the next, and so on.

This pattern illustrates a key principle of Pine’s execution model: on each successive execution, a script *re-evaluates* function calls and other expressions within its required *scopes* to perform separate calculations for the current bar.

Repeated code evaluation also applies to variable declarations. By default, a script does not declare a variable only once throughout its runtime; the script *re-declares* that variable and assigns an initial value based on the current bar’s data during *each* new evaluation of its scope.

Let’s look at a simple example. The following script declares an x variable of the “int” type with an initial value of 0. Then, it increases the variable’s value by 10 with the addition assignment operator (+=). The script calls plot() to display the value of x on each bar in a separate pane:

image

```
//@version=6 indicator("Repeated declarations demo") //@variable A
user-defined variable. The script declares this variable and initializes it
to 0 on *every* execution. int x = 0 // Increase the value of `x` by 10
on every bar. x += 10 // Plot the value of `x`. // Because `x` begins at
0 on every execution, and the script adds 10 to that value, the plotted
value is always 10. plot(x, "`x` value", color.blue, 3)
```

As shown above, the script plots a value of 10 on every bar, because the x variable *does not* carry over from bar to bar; the script declares the variable *repeatedly*. On each bar, the script re-declares x with an initial value of 0, then adds 10 to that value, resulting in a final value of 10 for every plotted point.

Programmers can change the behavior of a variable, enabling it to *persist* and preserve updates to its value *across bars*, by including the var keyword in its declaration, as described in the Declaration modes section of the Variable declarations page.

Below, we modify the previous script by adding var to the x declaration. Now, the script declares and initializes x only *once* — on the *first bar* — and that variable persists across *all* bars that follow. The script now plots a line that *increases* by 10 on each bar, because x preserves the result from each addition across the chart’s history. The value changes from 0 to 10 on the first bar, then to 20 on the second, and so on:

image

```
//@version=6 indicator("Persistent declarations demo") //@variable A
*persistent* variable. The script initializes this variable only on the
*first execution*. // The variable preserves all changes to its value on
each closed bar. var int x = 0 // Increase the value of `x` by 10 on
every bar. x += 10 // Plot the `x` series on the chart. // Because the
script declares `x` using `var` and then increments its value, the value
never resets to 0. // The plotted value is 10 on the first bar, 20 on the
next, and so on. plot(x, "`x` series", color.blue, 3)
```

Storing and using data from previous bars

As a script runs on a dataset, the states of its variables, function calls, and other expressions are automatically *committed (saved)* to an internal *time series* on each bar, creating historical trails of previous bar values that the script can access during its calculations on the current bar. The script can use these previous values by doing either of the following:

- Using the [] history-referencing operator. The number in the square brackets represents how many *bars back* from the current bar the script looks to retrieve a past value. For instance, `close[1]` retrieves the close value from *one bar before* the current bar, and `close[100]` retrieves the value from *100 bars back*.
- Calling the built-in functions that calculate on past values internally, such as `ta.*()` functions. For instance, `ta.change(close, 10)` calculates the difference between the current value of close and its value from 10 bars back.

The example below uses both of the above techniques to perform calculations based on data from previous bars. The script calculates a series of bar-by-bar price returns and plots the result as color-coded columns. It declares two global variables on each bar: `priceReturn` for the calculated returns, and `returnColor` for the plot’s color. The `priceReturn` value is the result of dividing the current one-bar change in closing prices (`ta.change(close, 1)`) by the previous bar’s closing price (`close[1]`). The `returnColor` value is color.teal if the current value of `priceReturn` is higher than the value from the previous bar (`priceReturn[1]`), and color.maroon otherwise:

image

```
//@version=6 indicator("Storing and using data from previous bars
demo") //@variable The one-bar price return, based on the current and
*previous* bars' `close` values. // This variable's final value on each bar
automatically becomes part of the internal time series. float
priceReturn = ta.change(close, 1) / close[1] //@variable Is `color.teal` if
the `priceReturn` value is above the value on the previous bar;
```

`color.maroon` otherwise. color returnColor = priceReturn > priceReturn[1] ? color.teal : color.maroon // Plot the current `priceReturn` value as a column, colored using the value of `returnColor`. plot(priceReturn, "Price return", returnColor, 1, plot.style_columns)

Note that:

- This script does *not* plot a column on bar 0 (the *first* bar). The `priceReturn` value is na on that bar, because there is *no previous bar* available for the script to reference at that point.

Realtime bars

When a script first runs on a chart, all *closed* bars in the accessed dataset are *historical bars*. These bars represent data for elapsed time periods where the final price and volume are *confirmed*. All indicators execute **once** per historical bar.

When the rightmost bar on the chart is *open*, it is a *realtime bar*. Unlike a historical bar, whose values are final, a realtime bar *updates* its values as new price or volume data becomes available. After the bar closes, it becomes an *elapsed realtime bar*, which is then no longer subject to change as the script runs.

Because the final values for a realtime bar are *unknown* until the bar closes, an indicator executes differently on that bar than it does on historical bars. The script executes not once, but **repeatedly** on the realtime bar — once for each new *update (tick)* — to *recalculate* its results using the latest data.

Before each recalculation on the realtime bar, the data for a script’s variables, expressions, and outputs on that bar is *cleared*, or *reset*. We refer to this process as *rollback*. The purpose of rollback is to revert the script to the same confirmed state it had when the realtime bar opened. This process ensures that the script’s calculations for the bar operate only on the latest available data, without relying on *temporary data* from the bar’s *previous ticks*.

Let’s look at rollback and recalculation in action. The following script uses ta.stoch() to calculate the Stochastic oscillator based on the close, high, and low values over a specified number of bars, then plots the result in a separate pane. It also calls bgcolor() to highlight the background on each realtime bar — where barstate.isrealtime is `true` — for visual reference:

```
//@version=6 indicator("Recalculation on realtime bars demo") //
@variable The number of bars in the Stochastic calculation. Users can
change this value in the "Settings/Inputs" tab. int lengthInput =
input.int(10, "Length", 1) //@variable The Stochastic oscillator, based
on the `close`, `high`, and `low` values over `lengthInput` bars. float
stochastic = ta.stoch(close, high, low, lengthInput) // Plot the
`stochastic` value for each bar. plot(stochastic, "Stochastic %K",
color.teal, 3) // Highlight the background of each realtime bar.
bgcolor(barstate.isrealtime ? color.new(color.purple, 80) : na, title =
"Realtime background highlight")
```

When we add the script to our chart, it executes once per bar in the chart’s history, from the leftmost bar to the rightmost bar. However, the rightmost bar on our chart is still *open*. Therefore, it is a *realtime bar*, not a historical bar. After the script reaches that bar, it begins executing once for *every new update* to the bar’s data. Each new script execution calculates on the latest available prices and *replaces* the bar’s previous result.

For instance, in the initial image below, the oscillator’s value 10 seconds into the open realtime bar (the one with the purple background) is 32.08:

image

Every time the bar updates, rollback *resets* the script’s data for that bar, and the script *recalculates* its result using the latest high, low, and close values. Here, halfway through the realtime bar’s period, the oscillator’s plot now shows a value of 16.71:

image

Recalculation continues for each successive update to the bar. Then, the script reaches the bar's closing tick, where the prices become *confirmed*. On that tick, the script calculates the oscillator's final value of 19.35. Afterward, another realtime bar opens, and the pattern of rollback and recalculation continues on that bar:

image

Note that:

- Only the values for a realtime bar's *final tick* become part of the internal time series. The values from ticks *before* the bar's close are **not** saved.
- The input.int() function returns a value of the "input int" *qualified type*. Values qualified as "input" are established *before* the first script execution, and they remain consistent throughout the script's runtime. If the user changes the "Length" input to a new value, the script *restarts* to perform new calculations across the dataset using that value. See the Inputs page and the Qualifiers section of the Type system page to learn more about script inputs and the "input" qualifier.
- If the script restarts, all the realtime bars from the previous script run become *historical bars* in the new run. Therefore, after restarting, the script executes only **once** on each of those bars and does *not* highlight their background.

The details

The following sections provide in-depth details about Pine's execution model, including the mechanics of executions on historical bars and realtime bars, which events trigger script executions, and how the runtime system maintains data across executions in a time series format.

Executions on historical bars

When a script loads on the chart or in another location after an execution-triggering event, its compiled source code executes on *every* accessible bar in the current dataset in order, starting with the first bar.

While the script loads, the runtime system performs the following steps for *each bar* that it accesses:

1. It updates the built-in variables that hold bar information. For instance, the system sets the open, high, low, and close variables to hold the OHLC price values of the bar *before* each execution.
2. It executes the script's compiled code from start to end using the data available as of the current bar.
3. After the execution ends, the system commits (saves) all necessary data for the current bar to the time series. The script can then access that data from historical buffers during its executions on subsequent bars by using the history-referencing operator or the built-in functions that reference past bars internally.

These steps repeat for every successive bar up to the most recent bar. After the runtime system completes this process across the dataset, the script's committed *outputs* — such as plots, drawings, Pine Logs, and Strategy Tester results — become available to the user.

All the closed bars on which the script executes while loading are *historical*, because they represent data points that were confirmed before the event that triggered the loading process. By default, all scripts execute **once** for each historical bar.

Let's examine a simple indicator to understand how script executions work on historical bars.

The script below calculates the 20-bar moving average of close values and plots the result on the chart. The color of the plot depends on whether the average is above or below the value on the previous bar. The script also increments an `executionNum` variable to count code executions, then plots the result alongside bar_index for comparison. Additionally, it highlights the background of historical bars in orange for visual reference:

```
//@version=6 indicator("Executions on historical bars demo") //
@variable The average of the latest 20 `close` values. float sma =
ta.sma(close, 20) //@variable Is `color.green` if the `sma` value is
above the value on the previous bar; `color.red` otherwise. color
```

plotColor = sma > sma[1] ? color.green : color.red //@variable Tracks the current execution number, where 0 represents the first execution. varip int executionNum = -1 // Add 1 to the `executionNum` value. executionNum += 1 // Display the `sma` as a line plot on the main chart pane, colored by the `plotColor`. plot(sma, "SMA", plotColor, 3, force_overlay = true) // Display the `executionNum` and `bar_index` series in a separate pane. plot(executionNum, "Execution number", color.purple, 5) plot(bar_index, "Bar index", color.aqua, 2) // Highlight the chart's background in translucent orange when `barstate.ishistory` is `true`. bgcolor(barstate.ishistory ? color.new(color.orange, 70) : na, title = "Historical highlight", force_overlay = true)

The statements and expressions in this source code might appear static at first glance. However, they have *dynamic* behavior across bars because the system executes the script *repeatedly* — once for each successive data point. Below, we inspect the code step by step to explain how the script works during its historical executions.

The indicator() call at the top of the code is a declaration statement that defines the script’s type and properties once, at *compile time*. This statement does not execute as the script runs on the dataset:

```
indicator("Executions on historical bars demo")
```

Before each script execution on a bar, the runtime system updates the built-in bar_index and close variables required in the calculations. The bar_index value is the bar’s global *time series index*, where 0 represents the first bar, 1 represents the second, and so on. The close variable holds the bar’s *latest price*. For historical bars, its value is the *final price* at the bar’s closing time.

Each time that the script executes, it declares and initializes a global `sma` variable of the “float” type. This variable declaration happens on *every* execution because the code line does not specify a declaration mode. The variable’s assigned value is the result of a ta.sma() function call. The call returns the average of the latest 20 close values as of the current bar, or na if fewer than 20 bars are available. After the execution ends, the system commits the new value of `sma` to the time series:

```
//@variable The average of the latest 20 `close` values. float sma = ta.sma(close, 20)
```

Note that:

- The `//@variable` comment above the `sma` declaration is an annotation that *documents* the variable in the code. The Pine Editor displays the comment in a pop-up window when the user hovers the mouse pointer over the variable.

During each execution, the script also initializes a `plotColor` variable of the "color" type. The script uses a ternary operation that compares the current `sma` value to `sma[1]` — the *last committed value* for `sma` as of the *previous bar* — to determine the `plotColor` variable's assigned value. If the current `sma` value is higher than the last committed value, the `plotColor` value is color.green. Otherwise, it is color.red:

```
//@variable Is `color.green` if the `sma` value is above the value on the previous bar; `color.red` otherwise. color plotColor = sma > sma[1] ? color.green : color.red
```

In contrast to the variables above, the script *does not* initialize the `executionNum` variable on every execution. Instead, initialization happens only *once* — on the *first* bar — because the variable declaration is in the *global scope* and uses the varip keyword. Once initialized, the variable *persists* across all subsequent bars and the ticks within those bars. Only the reassignment or compound assignment operators can change its value:

```
//@variable Tracks the current execution number, where 0 represents the first execution. varip int executionNum = -1
```

The code following the `executionNum` declaration uses the addition assignment operator (+=) to increase the variable's value by one on each new execution. Starting from -1, the value increases to 0 on the first execution after initialization, then 1 on the second, and so on:

```
// Add 1 to the `executionNum` value. executionNum += 1
```

The script evaluates the plot() and bgcolor() calls on every execution. Each plot() call creates a new point on a line plot at the bar's location on the time axis. The bgcolor() call creates a background color for the bar based on a ternary expression. The background is translucent orange if barstate.ishistory is true. Otherwise, it is na (no color):

// Display the `sma` as a line plot on the main chart pane, colored by the `plotColor`. plot(sma, "SMA", plotColor, 3, force_overlay = true) // Display the `executionNum` and `bar_index` series in a separate pane. plot(executionNum, "Execution number", color.purple, 5) plot(bar_index, "Bar index", color.aqua, 2) // Highlight the chart's background in translucent orange when `barstate.ishistory` is `true`. bgcolor(barstate.ishistory ? color.new(color.orange, 70) : na, title = "Historical highlight", force_overlay = true)

Note that:

- The plot() and bgcolor() calls that include force_overlay = true display their visuals on the main chart pane. The other plot() calls output visuals in a separate pane, because the indicator() call does not include overlay = true.

After the system executes the script on all available data points and finishes loading, the script’s outputs then become visible on the chart:

image

Note that:

- When the script first loads, *all* bars, including the latest one, have an orange background because they initially represent *historical* data. However, the latest bar on our chart is still open, meaning it is a *realtime bar*. After a new tick arrives from the realtime data feed, the bar’s values update, and the script executes *again* on that bar. The orange background for the bar then *disappears* because the system sets the value of barstate.ishistory to false.
- The executionNum and bar_index values are identical on historical bars because the script executes *once per bar* on that part of the dataset. However, they begin to differ on the realtime bar. On that bar, the script executes after *every new update* to recalculate its results, and the executionNum value increases each time. See the Executions on realtime bars section to learn more.

- An alternative, more robust method to track code executions is to use the Pine Profiler. The profiler analyzes the total runtime and execution count of every significant part of the source code. To learn more about this feature, see the Profiling and optimization page.

It’s important to note that, unlike indicators, strategies can execute *more than once* per historical bar, depending on the specified calculation behavior. If the strategy() declaration statement includes `calc_on_order_fills = true`, or if the user selects the “After order is filled” checkbox in the “Settings/Properties” tab, the runtime system executes the script on *each available tick* where the broker emulator fills an order, or once per bar when there is no order to fill.

Let’s look at a simple example. The following strategy changes the direction of its simulated position on each execution. If there is an open short position or no position, the strategy places a market order to close all short trades and enter a long trade. If a long position is open, the strategy places a market order to close it and open a short trade.

As with the previous example, this script increments an `executionNum` variable declared with varip to count new executions, plots the result alongside bar_index for comparison, and highlights the background of historical bars in orange with bgcolor():

image

//@version=6 strategy("Default strategy behavior on historical bars demo") // Place a market order to close short trades and enter a long trade when there is a short position or no position. // Otherwise, if a long position is open, place a market order to close the long trades and enter a short trade. if strategy.position_size <= 0 strategy.entry("Long", strategy.long) else strategy.entry("Short", strategy.short) //@variable Tracks the current execution number, where 0 represents the first execution. varip int executionNum = -1 // Add 1 to the `executionNum` value. executionNum += 1 // Display the `executionNum` and `bar_index` series in a separate pane. plot(executionNum, "Execution number", color.purple, 5) plot(bar_index, "Bar index", color.aqua, 2) // Highlight the chart's background in translucent orange when `barstate.ishistory` is `true`. bgcolor(barstate.ishistory ? color.new(color.orange, 70) : na, title = "Historical highlight", force_overlay = true)

Note that:

- The strategy.entry() command creates entry orders. By default, a long entry using this command reverses an open short position, and a short entry reverses an open long position. See the Reversing positions section of the Strategies page to learn more.

The script above uses the default calculation behavior: it places a new order only at the close of each bar. The broker emulator fills the order at the next bar’s opening price, as the trade markers on the chart above indicate. The `executionNum` and bar_index plots show the same values because the script executes only once per bar.

If we include `calc_on_order_fills = true` in the strategy() declaration statement, the runtime system *re-executes* the script on a bar after each new order fill to update the calculations. Our script’s logic generates a new order on *every* execution, and the broker emulator considers historical bars to have *four ticks* for filling orders by default (the open, high, low, and close). Therefore, with this change, the script executes **four times** per historical bar instead of only once. As shown below, the strategy now shows four trade markers on each historical bar, and the `executionNum` value is four times that of the bar_index variable:

image

//@version=6 strategy("Calculation after order fill on historical bars demo", calc_on_order_fills = true) // Place a market order to close short trades and enter a long trade when there is a short position or no position. // Otherwise, if a long position is open, place a market order to close the long trades and enter a short trade. if strategy.position_size <= 0 strategy.entry("Long", strategy.long) else strategy.entry("Short", strategy.short) //@variable Tracks the current execution number, where 0 represents the first execution. varip int executionNum = -1 // Add 1 to the `executionNum` value. executionNum += 1 // Display the `executionNum` and `bar_index` series in a separate pane.

```
plot(executionNum, "Execution number", color.purple, 5)
plot(bar_index, "Bar index", color.aqua, 2) // Highlight the chart's
background in translucent orange when `barstate.ishistory` is `true`.
bgcolor(barstate.ishistory ? color.new(color.orange, 70) : na, title =
"Historical highlight", force_overlay = true)
```

Note that:

- This script can execute *more than four* times per bar if it uses Bar Magnifier mode, because this mode enables the broker emulator to fill orders on historical bars using intrabar prices from a *lower timeframe*.
- The script can execute numerous times on a *realtime* bar, depending on the updates from the data feed, because *each new update* to the bar is a valid tick for filling the strategy’s orders.
- An alternative way to confirm the script’s increased execution count is to select and clear the “After order is filled” checkbox in the “Settings/Properties” tab while profiling the code.

Executions on realtime bars

After a script running on the chart or in an alert executes across all historical bars in a dataset, the runtime system continues to execute the script on the current bar, if it is open, and on any new bars that form later. We refer to these bars as *realtime bars*, because they represent incoming data from a separate data feed that the script can access only *after* it finishes loading.

As explained in the previous section, historical bars represent confirmed data points. By contrast, a realtime bar represents an initially *unconfirmed* data point that evolves as new updates (ticks) arrive from the realtime data feed. With each new tick, the bar’s high, low, close, volume, and other values update to represent the latest data while the bar remains open. After the bar closes, it becomes an *elapsed realtime bar*, whose values no longer change. Then, a new realtime bar opens after another tick arrives, and that bar updates as new data becomes available.

As an indicator or library script runs on an open realtime bar, its compiled code executes once after **every new update** from the data feed. With each new execution, the script recalculates its results for that bar using the latest data. Consequently, the states of the script's variables, expressions, and objects can *change* with each new execution while the bar remains open. The system *commits* the script's data for the realtime bar only after the bar closes.

After each script execution that occurs *before* a bar's closing tick, the runtime engine executes a *rollback* process. Rollback *resets* applicable script data to the latest committed states in the time series. This process enables the script to recalculate the bar's results using only the latest available data — without the influence of *temporary* data from executions on the bar's previous ticks.

Below, we explain how recalculation and rollback affect a script's data and outputs, along with some notable exceptions to this process:

Reinitialize variables

The runtime system erases the states of any variables that the script initializes during its executions before a bar's close, excluding those declared using the varip keyword. When the script executes again after rollback, it *reinitializes* the variables with new values or references based on the latest available data.

Likewise, the system does not preserve the *temporary* states of built-in variables that hold values for the current bar. Before the new script execution, it sets the variables to use the bar's most recent data. For instance, the system updates close, high, and low with the latest, highest, and lowest prices reported since the bar's opening time.

Reset changes to `var` variables

Variables that use the var keyword in their declaration are initialized only *once* — during the *first* execution of their scopes on a *closed bar*. Variables that use the var keyword in their declaration remain initialized after the *first* time that their scopes execute on a bar's *closing tick*. Their assigned values or references *persist* across subsequent bars, changing only after reassignment or compound assignment operations.

Although these variables preserve data across successive bars, they **do not** preserve data across executions on the *ticks* of an open bar. Rollback reverts all variables declared with var before the current bar to the last committed states in the time series as of the previous bar.

For instance, if a variable declared with var has a value of 20 on the open bar and 19 on the previous bar, the variable's value reverts to 19 before the script executes on the next tick of the same bar. The temporary value of 20 does not persist.

Replace plotted outputs

The `plot*()`, bgcolor(), barcolor(), and fill() functions create visual outputs on *every bar*. These outputs are *temporary* on the open realtime bar. When the script executes again after rollback, the new outputs for the bar from calls to these functions *replace* the ones from the previous tick.

For example, when the expression `plot(close)` executes on the open bar, it displays the bar's latest close value as of the current execution. However, the plotted result is **temporary** until the bar closes. After rollback, the close variable updates, then the script calls plot() again on the next execution to replace the output from the previous tick and display the new value.

Remove and revert objects

User-defined types (UDTs) and special types such as collections and drawing types are *reference types*. They define structures from which scripts create *objects* — independent entities that store data elsewhere in memory. Variables of these types hold *references* that provide access to specific objects and their data; the variables do **not** store objects directly.

If a script creates objects on an open bar and does not assign their references to variables declared with the varip keyword, the rollback process *removes* those objects. During the next execution on the open bar, the script creates *new objects* if the updated logic allows it.

For example, if a script calls label.new() to create a label object on the open bar, the system *deletes* that object during rollback. On the next execution, the script evaluates label.new() again, creating a *new* label that replaces the output. The label created on the previous tick no longer exists.

Similarly, for objects of built-in or user-defined types with references assigned to var variables, the rollback process reverts any changes to those objects that occur on the open bar. The only exception is for UDTs with *fields* that include the varip keyword. See the Objects page for more information.

Exceptions

The runtime system does not revert *all* the data from script executions on an open bar. The following are notable exceptions to the rollback process:

- Variables or fields declared with the varip keyword **do not** revert to a previously committed state. They persist across *all* script executions after initialization, even those on the ticks of an open realtime bar.
- Logged messages in the Pine Logs pane do not disappear after rollback. The messages from any `log.*()` calls during executions on the ticks of realtime bars remain in the pane until the script reloads.
- The data from strategy orders placed or filled on the ticks within a bar is not subject to rollback. If a strategy script creates orders or the broker emulator fills orders on an open bar, the data from those events persists.
- Rollback does not erase logs for alerts from the "Alerts" menu. All messages from a script alert remain visible until the user restarts the alert.
- Runtime errors from the system or the runtime.error() function completely *stop* script executions. If an error occurs at any point while a script executes on an open bar, the system halts the script

and does not revert the error after new updates from the data feed.

Let’s inspect the behavior of a simple indicator on realtime bars. The following script calculates an RSI of close values using ta.rsi() and displays the result with a plot() call. To track the number of executions that occur *per bar*, the script increments an `executions` variable declared with varip and calculates its one-bar change using ta.change(). The script converts each bar’s execution count to a string with str.tostring(), then displays the result in a color-coded label at the bar’s high. The label is purple if the bar is open. Otherwise, it is gray. The script also highlights the background of realtime bars in orange using bgcolor():

```
//@version=6 indicator("Executions on realtime bars demo") //
@variable The 14-bar RSI of `close` prices. float rsi = ta.rsi(close, 14) //
@variable Tracks the number of script executions, where 1 represents
the first execution. varip int executions = 0 // Add 1 to the `executions`
value. executions += 1 //@variable Is `color.gray` if the bar is
confirmed (closed); `color.purple` otherwise. color labelColor =
barstate.isconfirmed ? color.gray : color.purple // Calculate the one-bar
change in `executions`, then convert the value to a string and display
the result in a label. // Each call to `label.new()` creates a *new* `label`
object. label.new( bar_index, high, str.tostring(ta.change(executions)),
color = labelColor, textcolor = color.white, size = 20, force_overlay =
true ) // Plot the `rsi` value with colors based on whether the value is
above 50 or not. plot(rsi, "RSI", rsi > 50 ? color.teal : color.maroon, 3) //
Highlight the chart's background in translucent orange when
`barstate.isrealtime` is `true`. bgcolor(barstate.isrealtime ?
color.new(color.orange, 70) : na, title = "Realtime highlight",
force_overlay = true)
```

When we first add the script to the chart, it does *not* add an orange background to any bar because it calculates only on data that exists at the script’s loading time. This data is *historical*. Each bar’s label shows a value of 1 because indicators always execute *once* per historical bar:

image

Notice the countdown timer and the *purple* label for the latest bar in the chart above. These both indicate that the bar is *open* and subject to changes. A new update from the data feed affects the bar’s values, triggering rollback and a new script execution to recalculate the results.

When rollback occurs, the runtime system reverts the internal data of the ta.rsi() call to its last committed state, erases the state of the `rsi` variable, and deletes the latest label object. However, the system does not revert the `executions` variable because it uses the varip keyword.

After rollback, the system updates the built-in close, high, and `barstate.*` variables using the current bar’s latest data, and the new execution begins. The script evaluates the ta.rsi() call using the new close price and reinitializes the `rsi` variable with the returned value. Then, it increases the `executions` value by one, evaluates ta.change() again, and creates a new label at the bar’s current high price to show the updated result. Lastly, it evaluates the plot() and bgcolor() calls to replace the bar’s plotted visuals. The last bar’s label remains purple because the bar is still open, but the background color is now *orange* because barstate.isrealtime is `true`:

image

As subsequent updates become available from the data feed, the pattern of rollback and re-execution continues, and the script’s outputs for the bar update with each new execution:

image

The last time that rollback and another execution occur on this bar is after the *closing tick*, when the bar becomes an *elapsed* realtime bar. After the final execution, the bar’s label is *gray* because barstate.isconfirmed is `true`. The runtime system then *commits* necessary data from this execution to the time series for calculations on future bars.

Then, a new realtime bar opens after another update from the data feed, and the execution pattern continues:

image

Note that:

- Although the previous bar is now confirmed, it still has an orange background corresponding to a *realtime* state because it closed **after** the script's loading time. When the script later reloads after an execution-triggering event, that bar becomes *historical*.

It's important to note that strategies often execute differently than indicators on realtime bars. By default, they execute only **once** per bar at each *closing tick* without undergoing rollback. However, users can modify a strategy's calculation behavior to allow rollback and re-execution on a bar before its closing tick.

If the strategy() statement includes `calc_on_every_tick = true`, or if the user selects the "On every tick" checkbox in the "Settings/Properties" tab, the script executes on a realtime bar after *each new update* from the data feed, similar to an indicator.

Additionally, if the strategy() statement includes `calc_on_order_fills = true` or the user selects "After order is filled" in the "Settings/Properties" tab, the script executes on *each tick* where the broker emulator fills an order. With this behavior, the system can execute the script multiple times on the open bar, but only on the ticks where an *order fill* occurs.

To summarize the general process for script executions on realtime bars:

- An indicator or library script executes on the *first available tick* in an open realtime bar, then *once per update* to recalculate the results for the bar using the latest data. A strategy script executes only on the bar's *closing tick* by default, but users can modify its calculation behavior to allow executions while the bar is open.
- Before each new script execution on an open bar, the runtime system executes a *rollback* process, which *reverts* all applicable variables, expressions, and objects to their *last committed states* as of the previous bar's close.
- After the script executes on an *elapsed* realtime bar's closing tick, the system *commits* necessary data from that execution to the time series for access on later bars. It does **not** commit the data

from executions on the bar's *unconfirmed* values from previous ticks.

Events that trigger script executions

Several events cause a script to load and execute across all the available bars in a dataset. The specific events that trigger the loading process depend on where the script runs.

For a script on the chart, the following events always cause the script to load and perform *new executions* on every bar:

- The user adds the script to the chart for the first time from the Pine Editor or the "Indicators, metrics, and strategies" menu.
- The user saves an update to the script while it is active on the chart.
- The chart is refreshed while the script is active.

Other events also trigger the loading process for a script on the chart. However, these events do not *always* cause new script executions on past bars. The results from running a script with a unique combination of settings are often temporarily *cached*. If cached data exists for a selected combination of settings, the system loads the script using that data. See the Caching section for more information.

Below are the additional events that cause a script to load on the chart, either by performing new executions across the dataset or by using available cached data:

- The user selects new values for the inputs or strategy properties in the script's "Settings" menu.
- The script uses the chart.left_visible_bar_time or chart.right_visible_bar_time variable, and the visible chart range changes.
- The script uses the chart.fg_color or chart.bg_color variable, and the user changes the chart's background color.
- The chart loads a new dataset with a different *timeframe* or *ticker identifier*. Several user actions affect a chart's ticker ID, such as selecting a symbol from the "Symbol Search" menu, changing the

chart type, toggling data modifications in the chart's settings, and activating Bar Replay mode.
- The user opens or closes the Pine Logs pane.
- The user activates or deactivates the Pine Profiler.

For scripts used in other locations, the following events trigger the loading process:

- The user creates a new script alert from the "Create Alert" dialog box.
- The user pauses and restarts an alert instance from the "Alerts" menu.
- The user clicks the "Generate report" button in the Strategy Tester while Deep Backtesting mode is enabled.
- The user clicks the "Scan" button in the Pine Screener to run the script on the datasets from a chosen watchlist.

After a script loads, either of the following causes new script executions on an *open bar*:

- One of the events above causes the script to load again and execute across the *entire dataset* up to the bar.
- The script runs on the chart or in an alert, and the bar updates after new data becomes available. The system performs *rollback* and re-executes the script on that bar using the latest data. The only exception is if the script is a strategy that does not allow recalculation on the new tick.

When a script completely reloads on the chart or in an alert after an applicable event, all the elapsed realtime bars from the script's previous run become historical bars in the new run, because they represent *confirmed* data points that the script accesses from *a different data feed* as it loads.

The bars in a symbol's dataset come from two distinct data feeds: the *historical* feed and the *realtime* feed. The historical feed reports only the *final* values for each bar, whereas the realtime feed includes the *temporary* values from all available ticks. When a realtime bar becomes historical after a script restarts, the values from the bar's previous ticks are no longer accessible; only the **final** price, volume, and other values remain. Therefore, if a script relies on temporary data from realtime bars in its calculations, it might behave differently after reloading.

For example, the following script calculates the one-bar arithmetic return of the close series and displays the result as a line plot. On each realtime bar, the script updates three variables declared with varip to track the first, highest, and lowest return values calculated during executions across the bar’s ticks, then calls plotcandle() to plot a candle showing the values. Additionally, it uses bgcolor() to highlight the background of realtime bars in orange:

```
//@version=6 indicator("Reloading a script demo", precision = 5) //
@variable The one-bar arithmetic return of the `close` series. float
priceReturn = ta.change(close, 1) / close[1] // Declare persistent
variables to track the first, highest, and lowest `priceReturn` values
across ticks in // each realtime bar. varip float o = na varip float h = na
varip float l = na if barstate.isrealtime // On the first tick in the realtime
bar, reassign `o`, `h`, and `l` to hold the value of `priceReturn`. if
barstate.isnew o := priceReturn h := priceReturn l := priceReturn //
Otherwise, reassign `h` and `l` to the bar's highest and lowest
`priceReturn` value as of the current tick. else h := math.max(h,
priceReturn) l := math.min(l, priceReturn) // Plot candles to display the
`o`, `h`, `l`, and `priceReturn` values for each realtime bar. // The
candles do not appear on historical bars, because `o`, `h`, and `l` are
`na` on those bars. plotcandle(o, h, l, priceReturn, "Return candles",
color.blue, chart.fg_color, bordercolor = chart.fg_color) // Dispaly the
`priceReturn` series as a purple line plot. plot(priceReturn, "Return
plot", color.purple, 3) // Highlight the background of all realtime bars in
orange. bgcolor(barstate.isrealtime ? color.new(color.orange, 70) : na,
title = "Realtime highlight")
```

After the script loads on the chart and executes on several realtime bars, all the elapsed realtime bars, as well as the open realtime bar, include plotted return candles and an orange background color:

image

After an applicable event, such as a chart refresh, the script *reloads* and executes across the dataset again. All the closed bars with a realtime state in the previous run become *historical* bars in the new run. The results thus change because our script relies on realtime data. As shown below, the script does not display candles or background colors for previous bars after we refresh the chart. Those outputs appear only for the latest bar, after new ticks become available, because that bar is now the **only** one with a realtime state:

image

Note that:

- The barstate.isnew variable has a value of `true` when a realtime bar opens, and `false` on all subsequent updates to the bar. If the script reloads midway through a realtime bar's progression, only the background color appears on that bar. The script does not show a candle on the first realtime bar in that case, because its `o`, `h`, and `l` variables hold na until the first time that barstate.isnew is `true`.

Caching

When a script runs on a chart for the *first time* using a *unique configuration*, the data from that run is often temporarily cached for reuse. The cached data is erased after the chart is refreshed or the user updates the script's source code.

In this context, the configuration refers to the combined state of all script, chart, and developer tool settings that can affect the script's executions. This combination includes:

- The values of inputs in the script's "Settings/Inputs" tab.
- The values of the strategy properties in the "Settings/Properties" tab.
- The values of the `chart.*` variables whose qualifiers are "input" (chart.left_visible_bar_time, chart.right_visible_bar_time, chart.fg_color, and chart.bg_color).
- The chart's timeframe.
- The chart's ticker identifier.
- Whether the Pine Logs pane is open or closed.

- Whether the Pine Profiler is active or not.

Each time that a script runs using a unique combination of settings, it executes from start to end on each bar in the dataset to perform new calculations. If possible, the script’s data from the run is then cached. If cached data is available on past bars for a selected combination of settings, the runtime system loads the script using that data.

This behavior enables users to change a script’s inputs, alter the chart, and toggle developer tools without losing information — including bar states — from previous script runs using different settings. Additionally, caching helps reduce loading times and resource requirements when switching between settings or adding multiple instances of the same script to the chart.

To understand this behavior, let’s revisit the script from the previous section. The script has different behaviors on historical and realtime bars. In the version below, we’ve added a `lengthInput` variable that holds the value from an input.int() call. The script uses this variable to define the length of the ta.change() calculation and the offset of the history-referencing operator:

//@version=6 indicator("Caching demo", precision = 5) //@variable The bar span of the `priceReturn` calculation. int lengthInput = input.int(5, "Length", 1) //@variable The arithmetic return of the `close` series across `lengthInput` bars. float priceReturn = ta.change(close, lengthInput) / close[lengthInput] // Declare persistent variables to track the first, highest, and lowest `priceReturn` values across ticks in // each realtime bar. varip float o = na varip float h = na varip float l = na if barstate.isrealtime // On the first tick in the realtime bar, reassign `o`, `h`, and `l` to hold the value of `priceReturn`. if barstate.isnew o := priceReturn h := priceReturn l := priceReturn // Otherwise, reassign `h` and `l` to the bar's highest and lowest `priceReturn` value as of the current tick. else h := math.max(h, priceReturn) l := math.min(l, priceReturn) // Plot candles to display the `o`, `h`, `l`, and `priceReturn` values for each realtime bar. // The candles do not appear on historical bars, because `o`, `h`, and `l` are `na` on those bars. plotcandle(o, h, l, priceReturn, "Return candles", color.blue, chart.fg_color, bordercolor = chart.fg_color) // Dispaly the `priceReturn` series as a purple line plot. plot(priceReturn, "Return plot", color.purple, 3) // Highlight the background of all realtime bars in orange. bgcolor(barstate.isrealtime ? color.new(color.orange, 70) : na, title = "Realtime highlight")

After we add the script to our 1m chart and let it run for a few minutes with a “Length” input value of 5, the script plots candles and highlights the background for the latest few bars, because barstate.isrealtime is true on those bars:

image

Let’s change the “Length” input to a new value, causing the script to reload and execute across the dataset again. Here, we changed the value from 5 to 10 and let the script execute on some new ticks. The script no longer displays candles and background colors for the same bars after restarting, because it now accesses the data for those formerly realtime bars from the *historical* data feed:

image

As shown above, the realtime bar information from the first run is *not available* when we change the script’s input to a new value. However, the data from that previous run still exists in memory. If we revert the “Length” input’s value to 5, the candle plot and background colors start on the same bar as the first run:

image

If we add a second instance of the script to the chart, using the same settings, the runtime system loads the new instance using the cached data instead of executing it entirely from scratch. As such, its outputs are *identical* to those from the first script instance, even though we added it to the chart a few bars later:

image

Similarly, cached data usually remains available even if we remove the script from our chart and add it again.

Time series

A symbol’s dataset is a form of *time series* — a sequence of collected values indexed by time. Each bar represents a distinct data point, anchored to a specific time, that contains price and volume data for a particular period. This data format thus shows how a symbol’s values progress across time in successive periodic steps.

Pine Script’s internal time series structure follows a similar format. After executing a script on a closed bar’s confirmed values, the runtime system *commits (saves)* the results of the script’s statements and expressions to internal time series for later use. Each bar with committed data has an assigned *index* in the series, where 0 represents the first bar, 1 represents the second, and so on. Scripts can retrieve this index with the bar_index variable.

Scripts can access the data committed to the time series on past bars by using the [] history-referencing operator. The value between the operator’s square brackets specifies the position of the referenced bar in the time series as a *relative offset* behind the current bar. For variables and expressions in the global scope, an offset value of 1 refers to the previous bar at `bar_index - 1` (one bar back), a value of 2 refers to the bar at `bar_index - 2` (two bars back), and so on. An offset of 0 always refers to the *current bar*.

For example, consider the open variable, which holds the opening price of the current bar on which the script executes. Before each script execution on a new bar, the runtime system commits the open value from the last execution on the previous bar. Then, it updates the variable to hold the current bar’s opening price. To access the committed open value for the previous bar, we can use the expression `open[1]`. To access the committed value from 10 bars back, we use `open[10]`.

The script below performs three history-referencing operations to retrieve the current bar’s open value, the value from one bar back, and the value from a user-specified number of bars back. Then, it plots the retrieved values on the chart for comparison:

image

```
//@version=6 indicator("History referencing demo", overlay = true,
behind_chart = false) //@variable The number of bars back from which
to retrieve the `open` price for `pastOpen`. int offsetInput =
input.int(10, "Bar offset", 0) //@variable The current bar's opening
price. `open[0]` is equivalent to using `open` without the `[]` operator.
float currOpen = open[0] //@variable The last committed `open` value.
Represents the previous bar's value, or `na` if no previous bar exists.
float prevOpen = open[1] //@variable The `open` value committed
`offsetInput` bars back, or `na` if no bar exists at that offset. float
pastOpen = open[offsetInput] // Plot `currOpen`, `prevOpen`, and
`pastOpen` for comparison. plot(currOpen, "Current `open`",
color.blue, 2) plot(prevOpen, "Previous bar `open`", color.purple, 3)
plot(pastOpen, "Past `open` from custom offset", color.orange, 4)
```

Note that:

- The expression `open[0]` is equivalent to using open without the history-referencing operator, because an offset of 0 refers to the current bar.
- At the beginning of the chart’s dataset, the expressions `open[1]` and `open[offsetInput]` return na because they refer to previous bars that are unavailable.
- Each history-referencing expression also leaves a trail of values in the time series. Therefore, it is possible to retrieve past states of the expression using another history-referencing operation, e.g., `(open[offsetInput])[1]`.
- Internally, the system maintains a *limited amount* of time series data for variables and expressions in fixed-length *historical buffers*. These buffers define the *maximum offsets* allowed for history-referencing operations. See the next section, Historical buffers, to learn more.

Another way that scripts use committed values from a time series is by calling the built-in functions that reference history internally, such as those in the `ta.*` namespace. For example, the expression `ta.highest(high, 20)` calculates the highest value from the high series over a 20-bar window. It compares the series’ current value to the committed values from the previous 19 bars to determine the result. The script below executes this call on each bar and plots the resulting series on the chart. Additionally, the script colors the background of the last 20 bars on the chart to highlight the bars used in the latest execution’s ta.highest() call:

image

```
//@version=6 indicator("History referencing in functions demo", overlay = true, behind_chart = false) //@variable The highest value from the `high` series across the 20 most recent bars. // The `ta.highest()` call compares the current `high` to the last 19 committed values. float highest = ta.highest(high, 20) // Plot the `highest` series on the chart. plot(highest, "20-bar high", color.purple, 3) // Color the background of the last 20 bars, i.e., the bars used by the latest execution's `ta.highest()` call. bgcolor(color.new(color.blue, 70), show_last = 20, title = "Last 20 bar highlight")
```

Note that:

- The first 19 bars of the chart have a plotted value of na, because the ta.highest() function call requires the high values from the current bar and 19 previous bars to calculate the result.
- All function calls and expressions that do not return “void” leave historical trails in the time series, just like variables. Therefore, scripts can use an expression such as `ta.highest(high, 20)[10]` to retrieve the 20-bar high from 10 bars back.
- The ta.highest() function and other functions that access past values from a time series must execute in the **global scope** for consistent calculations. Time series storage for variables and expressions in local scopes works *differently* than that for global values. See the Time series in scopes section for more information.

Historical buffers

To promote efficiency and help ensure computing resources remain available for all users, the Pine Script runtime system uses fixed-length *historical buffers* to maintain a *limited amount* of time series data for all variables and expressions. These historical buffers define the *maximum* number of committed data points that a script can access on any bar via the history-referencing operator or the built-in functions that reference past bars internally.

For most series, the underlying historical buffer can contain data from up to **5000** past bars. The only exception is for some built-in series such as open, close, and time, whose buffers can store data for *more* than 5000 bars.

Although these buffers can contain thousands of data points at their maximum size, a script might not *require* that much past data for its calculations on any bar. Therefore, the runtime system automatically optimizes the size of each series' historical buffer based on the historical references that the script performs as it loads on the dataset. Each resulting buffer contains *only* the amount of past data required by the script's calculations and *not more*.

For instance, if the maximum number of bars back for which a script references the value of a variable on historical bars is 500, the system maintains a historical buffer that includes only the latest 500 committed values of that variable. The buffer does not store 5000 committed values, because the script *does not* require all that extra data. This behavior thus helps to minimize a script's resource requirements while preserving the integrity of its calculations.

To determine the sufficient buffer size for each variable and expression in a script, the runtime system performs the following process during the script's loading time:

1. It analyzes all the historical references that occur while executing the script on the dataset's first **244 bars**, then sets the initial size of each buffer to the minimum size that accommodates those references.
2. While executing the script on subsequent bars, it checks if the script attempts to access data from previous bars that are beyond the limits of the defined buffers. If the script's historical references exceed the buffer limits on any bar, the system restarts the loading process and tries a larger buffer size.

3. In the rare case that a historical buffer’s size remains insufficient after several calculation attempts, the system stops the script and raises a runtime error.

It’s crucial to emphasize that the runtime system defines the sizes of all historical buffers only while executing a script on *historical bars*. It does **not** adjust any historical buffers during executions on *new bars* from the realtime data feed. If a script references past data from beyond a historical buffer’s limits while executing on a realtime bar, it causes a runtime error.

For example, the script below retrieves a past value from the close series using the history-referencing operator with an offset of 100 bars back on historical bars and 150 bars back on realtime bars. Because the script references data from 100 bars back during all historical executions, the system sets the close buffer’s size to include only 100 past values. Consequently, an error occurs when the script executes on the open realtime bar, because a historical offset of 150 is *beyond* the buffer’s limit:

```
//@version=6 indicator("Max bars back error demo", overlay = true) //
@variable The historical offset for retrieving past values from the
`close` series. // If the bar is historical, the offset is 100. Otherwise, the
offset is 150. int offset = barstate.ishistory ? 100 : 150 // @variable The
value of `close` from `offset` bars back. // This code causes a *runtime
error* on a realtime bar. During all code executions on historical bars, //
the script requires only the latest 100 past values of `close`, so the
system sets the buffer size to // include only the past 100 values. The
offset of 150 is thus *out of bounds*. float pastClose = close[offset] //
Plot the `pastClose` series. plot(pastClose, "Past `close`",
chart.fg_color, 3) // Highlight the background of all realtime bars in
orange. bgcolor(barstate.isrealtime ? color.new(color.orange, 70) : na,
title = "Realtime highlight")
```

For cases like these, programmers can *manually* set the size of a historical buffer to ensure it contains a sufficient amount of data by doing any of the following:

- Modify the script to reference the maximum required number of bars back with the [] operator during its execution on the *first bar*.
- Call the max_bars_back() function to explicitly set the historical buffer size for a *specific* series.
- Include a `max_bars_back` argument in the indicator() or strategy() declaration statement to set the initial size of *all* historical buffers.

Below, we modified the script by including the expression `max_bars_back(close, 150)`, which sets the size of the close buffer to include 150 past values. With the appropriate buffer size manually defined, the script’s history-referencing operation no longer causes an error on realtime bars:

image

//@version=6 indicator("Manual buffer sizing demo", overlay = true) // @variable The historical offset for retrieving past values from the `close` series. // If the bar is historical, the offset is 100. Otherwise, the offset is 150. int offset = barstate.ishistory ? 100 : 150 // Set the size of the `close` historical buffer to include 150 past values, ensuring the script has exactly // the amount of history that it requires on realtime bars. max_bars_back(close, 150) // @variable The value of `close` from `offset` bars back. // This code no longer causes an error when it executes on a realtime bar, because the historical // buffer has an appropriate size defined in advance. float pastClose = close[offset] // Plot the `pastClose` series. plot(pastClose, "Past `close`", chart.fg_color, 3) // Highlight the background of all realtime bars in orange. bgcolor(barstate.isrealtime ? color.new(color.orange, 70) : na, title = "Realtime highlight")

Time series in scopes

The *scope* of a variable or expression refers to the part of the script where it is defined and accessible in the code. Every script has one *global* scope and zero or more *local* scopes.

All variables and expressions in a script that are outside user-defined functions or methods, conditional structures, loops, and user-defined type or enum type declarations belong to the *global scope*. The script evaluates variables and expressions from this scope *once* for *every execution* across bars and ticks in the dataset.

All functions, methods, conditional structures, and loops create their own *local scopes*. The variables and expressions defined within a local scope belong exclusively to that scope. In contrast to the global scope, a script does *not* always evaluate a local scope once per execution; the script might evaluate the scope *zero*, *one*, or *several* times per execution, depending on its logic.

For the runtime system to commit data from a variable or expression and queue that data into a historical buffer on any bar, a script must *evaluate* the scope of that variable or expression once when it executes on the bar’s *closing tick*. If the script does not evaluate the scope, the runtime system *cannot* update the historical buffer for the variable or expression. Similarly, if the script evaluates the scope repeatedly within a loop, the historical buffer cannot store series data for *each* separate iteration, because each entry in the time series corresponds to a single bar.

Therefore, time series behave differently in global and local scopes: the historical buffers for global variables and expressions *always* contain committed data for *consecutive* past bars, whereas the buffers for local variables often contain an **inconsistent** history of committed data.

When a script references the history of a global variable using an expression such as `myVariable[1]`, the historical offset of 1 always refers to the confirmed `myVariable` value from the *previous bar*. In contrast, when using such an expression with a local variable, the offset of 1 refers to the most recent bar where the script executed the scope. It **does not** represent a specific number of bars back. Therefore, referencing the history of a local variable can cause *unintended results*.

The following example demonstrates how the historical buffers for a user-defined function’s local scope behave when a script does not call the function on *every* bar. The script below contains a custom `upDownColor()` function, which compares the current value of its `source` parameter to the last committed value (`source[1]`). The function returns color.blue if the current `source` value is higher than the previous value. Otherwise, it returns color.orange.

The script uses this function *conditionally*, inside a ternary operation, to determine the color of a plot that shows the remainder from dividing bar_index by a specified value. If the `remainder` variable's value is nonzero, the operation calls `upDownColor(remainder)` to calculate the color (blue or orange). If the value is 0, the operation does *not* use the call and instead returns color.gray. The `remainder` value *increases* on each bar, except for when it returns to 0 — causing the gray color. Therefore, a user might expect the plot's color to be only blue or gray on every bar. However, the color changes to *orange* on each bar after the one where the color is gray, even though the `remainder` value on that bar is *higher* than the value on the previous bar:

image

```
//@version=6 indicator("Local historical references demo") //@function
Returns `color.blue` if `source` is above its last committed value;
`color.orange` otherwise. // For consistent results, this function should
execute on *every bar*, because it uses the // history-referencing
operator on the `source` series. // // Even if the argument supplied to
`source` comes from a global variable, the `source` parameter
remains // part of the function's *local scope*. The system maintains a
*separate historical buffer* for the `source` // series in each function
call instance. The buffer contains only the committed `source` values
from the bars // where the function call occurs. If the call does not
occur on a bar, the buffer for `source` contains // **no data** for that
bar. upDownColor(float source) => source > source[1] ? color.blue :
color.orange //@variable The value by which to divide the `bar_index`
value. int divisorInput = input.int(5, "Divisor", 1) //@variable The
remainder of dividing `bar_index` by `divisorInput`. float remainder =
bar_index % divisorInput //@variable Is `color.orange` if `remainder`
equals 0, and the result of `upDownColor(remainder)` otherwise. // The
`upDownColor()` call does not execute on every bar. Therefore, it does
*not* always compare the // `remainder` value from one bar back to
calculate the color. Instead, the function compares the current //
```

`remainder` to the value from the last bar where `remainder` is nonzero. color plotColor = remainder == 0 ? color.gray : upDownColor(remainder) // Plot the `remainder` series and color it using `plotColor`. The plot is orange after each bar where `remainder == 0`, // because the `upDownColor()` function call does not have data for that bar to use in its logic. plot(remainder, "Remainder", plotColor, 5)

The script behaves this way because `upDownColor()` uses the history-referencing operator on the `source` series, which is *local* to the function's scope, and the script does not call the function on *every* execution. When the value of `remainder` is zero, the *first* expression in the ternary condition evaluates to `true`, and therefore the second branch of the ternary expression, which contains the function call, does *not* execute.

The compiler issues the following warning about the function directly in the Pine Editor:

```
The function `upDownColor()` should be called on each calculation f
```

The runtime system maintains a separate historical buffer for the local `source` series, but it cannot update that buffer unless the script *calls* the function. On each bar where `remainder` is 0, the call does not occur, and the system has no new value to commit to the time series. Therefore, the `source` buffer does *not* contain the value of `remainder`, or even an na value, for that bar. On the bar that follows, the local expression `source[1]` refers to the `source` value from the last bar where the `upDownColor()` call occurs — *two bars back* — and not the value of `remainder` from the previous bar. Because the value from two bars back is *higher* than the current value, the returned color is color.orange instead of color.blue.

We can fix this script's behavior by following the instructions in the compiler warning. Below, we modified the script by moving the `upDownColor()` call *outside* the ternary expression, enabling the script to execute it on *every bar*. The historical buffer for the function's `source` series now contains `remainder` values from *consecutive* bars.

With this change, an orange color does not appear because the function consistently compares values from *one* bar back:

image

//@version=6 indicator("Consistent historical references demo") // @function Returns `color.blue` if `source` is above its last committed value; `color.orange` otherwise. // For consistent results, this function should execute on *every bar*, because it uses the // history-referencing operator on the `source` series. upDownColor(float source) => source > source[1] ? color.blue : color.orange //@variable The value by which to divide the `bar_index` value. int divisorInput = input.int(5, "Divisor", 1) //@variable The remainder of dividing `bar_index` by `divisorInput`. float remainder = bar_index % divisorInput //@variable Is `color.blue` if `remainder` is above its previous value, and `color.orange` otherwise. color secondColor = upDownColor(remainder) //@variable `color.orange` if `remainder` equals 0, and `secondColor` otherwise. // This ternary operation no longer causes a warning. The scope of the `upDownColor()` call executes on // every bar, meaning its historical buffer consistently includes data for consecutive past bars. color plotColor = remainder == 0 ? color.gray : secondColor // Plot the `remainder` series and color it using `plotColor`. The plot is now blue or gray, but never orange. plot(remainder, "Remainder", plotColor, 5)

This behavior also applies to all built-in functions that reference past values internally, such as those in the `ta.*` namespace. For example, the ta.sma() function uses the current value of a `source` series and `length - 1` past values from that series to calculate a moving average. If a script calls this function only on *some* bars instead of on *every* bar, the historical buffer for `source` does not contain values for consecutive past bars. Therefore, such a call can cause unintended results, because the call calculates the returned average using an inconsistent history of values from previous bars.

The script below demonstrates how the results of the ta.sma() function can vary with the scope in which the function call occurs. The script declares three global variables to hold calculated SMA values: `controlSMA`, `localSMA`, and `globalSMA`. The script initializes `controlSMA` using the result of a ta.sma() function call, and it initializes the other two variables with na. Within the if structure, the script updates the value of `globalSMA`, and it updates `localSMA` using the result of another ta.sma() call with the same arguments as the first call.

As shown below, the `controlSMA` and `globalSMA` have the same value. Both use the result of the *global* ta.sma() call, which executes on *every bar*. The internal historical buffer for `source` in that call thus includes committed values for consecutive past bars. In contrast, the `localSMA` value differs, because the ta.sma() call for that variable does *not* execute on every bar. The buffer for that call’s local `source` series contains only the values from bars with an *even* bar_index value:

image

//@version=6 indicator("`ta.*()` functions in scopes demo", overlay = true, behind_chart = false) //@variable Is `true` if the `bar_index` is divisible by 2, and `false` otherwise. bool condition = bar_index % 2 == 0 //@variable The 20-bar moving average of `close` values. // This `ta.sma()` call executes in the global scope, so the script evaluates it on every bar. float controlSMA = ta.sma(close, 20) // Declare two additional variables to modify later within the `if` structure's scope. float globalSMA = na float localSMA = na if condition // Assign the `controlSMA` value to `globalSMA`. This code does not cause a warning. globalSMA := controlSMA // Call `ta.sma()` with the same arguments as before within this block and assign the result to `localSMA`. // The function call causes a warning, because it does not execute in the global scope. // The historical buffers for this `ta.sma()` call contain data only for the bars where `condition` is `true`, // thus leading to a *different* result. localSMA := ta.sma(close, 20) // Plot `controlSMA`, `globalSMA`, and `localSMA` for comparison. plot(controlSMA, "Control SMA", color.blue, 2) plot(globalSMA, "Global SMA", color.purple, 3, style = plot.style_circles) plot(localSMA, "Local SMA", color.gray, 3, style = plot.style_circles)

To summarize the behavior of time series in a script’s scopes:

- A script evaluates its global scope once on *every execution*. After each script execution on a bar’s closing tick, the system commits the data for variables and expressions in the global scope and updates their historical buffers. The resulting buffers thus include data for consecutive past bars, ensuring consistent results for operations and functions that rely on past data.

- A script evaluates local scopes *zero*, *one*, or *several* times per execution. The runtime system **cannot** maintain consistent bar-by-bar historical buffers for scopes that a script does *not* evaluate on every bar, or for scopes that the script evaluates *more than once* on a bar’s closing tick. Therefore, using the [] operator on local variables and expressions, or not calling functions that access past data once on each closing tick, can cause **unintended results**.

Next

Type system

Pine Script® v6 Documentation

User Manual / Language / Type system

Type system

Introduction

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

Pine Script uses types to classify all values, and it uses qualifiers to determine whether values and references are constant, established on the first script execution, or dynamic across executions. This system applies to all Pine values and references, including literals, variables, expressions, function returns, and function arguments.

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

Qualifiers

Pine Script *qualifiers* identify when values are accessible to a script:

- Values and references qualified as const are established at compile time (i.e., when saving the script in the Pine Editor or adding it to the chart).

- Values qualified as input are established at input time (i.e., when confirming values based on user input, primarily from the “Settings/Inputs” tab).
- Values qualified as simple are established at bar zero (i.e., the first script execution).
- Values qualified as series can change throughout the script’s executions.

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

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

It’s important to note that “series” values are the **only** ones that can change across script executions, including those from various built-ins, such as close and volume, as well as the results of expressions involving “series” values. All values qualified as “const”, “input”, or “simple” remain consistent across all script executions.

const

Values or references qualified as “const” are established at *compile time*, before the script starts its executions. Compilation initially occurs when saving a script in the Pine Editor, which does not require it to run on a chart. Values or references with the “const” qualifier *never change* between script executions, not even on the first execution.

All *literal* values and the results returned by expressions involving only values qualified as “const” automatically adopt the “const” qualifier.

These are some examples of literal values:

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

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

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

//@version=6 // The following global variables are all of the "const string" qualified type: //@variable The title of the indicator. INDICATOR_TITLE = "const demo" //@variable The title of the first plot. var PLOT1_TITLE = "High" //@variable The title of the second plot. const string PLOT2_TITLE = "Low" //@variable The title of the third plot. PLOT3_TITLE = "Midpoint between " + PLOT1_TITLE + " and " + PLOT2_TITLE indicator(INDICATOR_TITLE, overlay = true) plot(high, PLOT1_TITLE) plot(low, PLOT2_TITLE) plot(hl2, PLOT3_TITLE)

The following example will raise a compilation error since it uses syminfo.ticker, which returns a “simple” value because it depends on chart information that’s only accessible after the script’s first execution:

//@version=6 //@variable The title in the `indicator()` call. var NAME = "My indicator for " + syminfo.ticker indicator(NAME, "", true) // Causes an error because `NAME` is qualified as a "simple string". plot(close)

The const keyword allows the declaration of variables and parameters with constant *value assignments*. Declaring a variable with this keyword instructs the script to forbid using *reassignment* and *compound assignment* operations on it. For example, this script declares the `myVar` variable with the keyword, then attempts to assign a new “float” value to the variable with the addition assignment operator (+=), resulting in a compilation error:

```
//@version=6 indicator("Cannot reassign const demo") //@variable A
"float" variable declared as `const`, preventing reassignment. const
float myVar = 0.0 myVar += 1.0 // Causes an error. Reassignment and
compound assignments are not allowed on `const` variables.
plot(myVar)
```

It’s crucial to note that declaring a variable with the const keyword forces it to maintain a constant reference to the value returned by a specific expression, but that *does not* necessarily define the nature of the assigned value. For example, a script can declare a const variable that maintains a constant reference to an expression returning the *ID* of a *special type*. Although the script cannot *reassign* the variable, the assigned ID is a “series” value:

```
//@version=6 indicator("Constant reference to 'series' ID demo") //
@variable A `label` variable declared as `const`, preventing
reassignment. // Although the reference is constant, the ID of the
`label` is a "series" value. const label myVar = label.new(bar_index,
close)
```

input

Most values qualified as “input” are established after initialization via the `input.*()` functions. These functions produce values that users can modify within the “Inputs” tab of the script’s settings. When one changes any of the values in this tab, the script *restarts* from the beginning of the chart’s history to ensure its inputs are consistent throughout its executions. Some of Pine’s built-in variables, such as chart.bg_color also use the “input” qualifier, even though `input.*()` functions do not return them, since the script receives their values at *input time*.

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 “series string” arguments by default, meaning they can also accept “input string” values because the “input” qualifier is *lower* on the hierarchy:

```
//@version=6 indicator("input demo", overlay = true) //@variable The symbol to request data from. Qualified as "input string". symbolInput = input.symbol("AAPL", "Symbol") //@variable The timeframe of the data request. Qualified as "input string". timeframeInput = input.timeframe("D", "Timeframe") //@variable The source of the calculation. Qualified as "series float". sourceInput = input.source(close, "Source") //@variable The `sourceInput` value from the requested context. Qualified as "series float". requestedSource = request.security(symbolInput, timeframeInput, sourceInput) plot(requestedSource)
```

simple

Values qualified as “simple” are available on the first script execution, and they remain consistent across subsequent executions.

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

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

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

```
//@version=6 indicator("simple demo", overlay = true) //@variable Is
`true` when the current chart is non-standard. Qualified as "simple
bool". isNonStandard = not chart.is_standard //@variable Is orange
when the the current chart is non-standard. Qualified as "simple color".
simple color warningColor = isNonStandard ? color.new(color.orange,
70) : na // Colors the chart's background to warn that it's a non-
standard chart type. bgcolor(warningColor, title = "Non-standard chart
color")
```

series

Values qualified as “series” provide the most flexibility in scripts since they can change across executions.

Users can explicitly define variables and parameters that accept “series” values by including the `series` keyword in their declarations.

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

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

```
//@version=6 indicator("series demo", overlay = true) //@variable The
source value to calculate on. Qualified as "series float". series float
sourceInput = input.source(close, "Source") //@variable The number of
bars in the calculation. Qualified as "input int". lengthInput =
input.int(20, "Length") //@variable The highest `sourceInput` value
over `lengthInput` bars. Qualified as "series float". series float highest
= ta.highest(sourceInput, lengthInput) //@variable The lowest
`sourceInput` value over `lengthInput` bars. Qualified as "series float".
```

```
lowest = ta.lowest(sourceInput, lengthInput) plot(highest, "Highest
source", color.green) plot(lowest, "Lowest source", color.red)
```

Types

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

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

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

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

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

```
//@version=6 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

A `bool` variable can never be na, and any conditional structure that can return na will return `false` instead. For example, an if condition returns `bool` values, when the condition is not met and the `else` block is not specified, it will returns `false`.

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

color

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

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

These are examples of “color” literals:

#000000 // black color #FF0000 // red color #00FF00 // green color #0000FF // blue color #FFFFFF // white color #808080 // gray color #3ff7a0 // some custom color #FF000080 // 50% transparent red color

#FF0000ff // same as #FF0000, fully opaque red color #FF000000 // completely transparent red color

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

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

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

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

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

string

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

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

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

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

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

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

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

We can create “string” values containing the new line escape character (`\n`) for displaying multiline 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=6 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)
```

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

Refer to the Strings page to learn more about Pine strings. For information about displaying text from “string” values, see the Text and shapes and Debugging pages.

plot and hline

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

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

```
//@version=6 indicator("plot fill demo", overlay = true) //@variable A "series float" value representing a 10-bar EMA of `close`. float emaFast = ta.ema(close, 10) //@variable A "series float" value representing a 20-bar EMA of `close`. float emaSlow = ta.ema(close, 20) //@variable The plot of the `emaFast` value. emaFastPlot = plot(emaFast, "Fast EMA", color.orange, 3) //@variable The plot of the `emaSlow` value. emaSlowPlot = plot(emaSlow, "Slow EMA", color.gray, 3) // Fill the space between the `emaFastPlot` and `emaSlowPlot`. fill(emaFastPlot, emaSlowPlot, color.new(color.purple, 50), "EMA Fill")
```

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

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

Drawing types

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

Each type also has a namespace containing all the built-ins that create and manage drawing instances. For example, the following `*.new()`

constructors create new objects of these types in a script: line.new(), linefill.new(), box.new(), polyline.new(), label.new(), and table.new().

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

Chart points

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

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

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

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

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

```
//@version=6 indicator("Chart points demo", overlay = true) //
@variable A new `chart.point` at the previous `bar_index` and `high`.
firstPoint = chart.point.from_index(bar_index - 1, high[1]) //@variable A
new `chart.point` at the current bar's `low`. secondPoint =
chart.point.now(low) // Draw a new line connecting coordinates from
the `firstPoint` and `secondPoint`. // This line uses the `index` fields
from the points as x-coordinates. line.new(firstPoint, secondPoint, color
= color.purple, width = 3) // Draw a label at the `firstPoint`. Uses the
point's `index` field as its x-coordinate. label.new( firstPoint,
str.tostring(firstPoint.price), color = color.green, style =
label.style_label_down, textcolor = color.white ) // Draw a label at the
`secondPoint`. Uses the point's `index` field as its x-coordinate.
label.new( secondPoint, str.tostring(secondPoint.price), color =
color.red, style = label.style_label_up, textcolor = color.white )
```

Collections

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

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

- `map<int, UDT>` defines a map containing “int” keys and IDs of user-defined type (UDT) instances as values.

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

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

Note that:

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

User-defined types

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

The syntax to declare a user-defined type is:

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

where:

- export is the keyword that a library script uses to export the user-defined type. To learn more about exporting UDTs, see our User Manual’s Libraries page.
- `<UDT_identifier>` is the name of the user-defined type.
- `<field_type>` is the type of the field.
- `<field_name>` is the name of the field.
- `<value>` is an optional default value for the field, which the script will assign to it when creating new objects of that UDT. If one does

not provide a value, the field’s default is na. The same rules as those governing the default values of parameters in function signatures apply to the default values of fields. For example, a UDT’s default values cannot use results from the history-referencing operator [] or expressions.

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

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

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

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

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

Enum types

The enum keyword allows the creation of an *enum*, otherwise known as an *enumeration*, *enumerated type*, or *enum type*. An enum is a unique type construct containing distinct, named fields representing *members* (i.e., possible values) of the type. Enums allow programmers to control the values accepted by variables, conditional expressions, and collections, and they facilitate convenient dropdown input creation with the input.enum() function.

The syntax to declare an enum is as follows:

```
[export ]enum <enumName>
    <field_1>[ = <title_1>]
    <field_2>[ = <title_2>]
    ...
    <field_N>[ = <title_N>]
```

where:

- export is the optional keyword allowing a library to export the enum for use in other scripts. See this section to learn more about exporting enum types.
- <enumName> is the name of the enum type. Scripts can use the enum's name as the type keyword in variable declarations and type templates.
- `<field_*>` is the name of an enum field, representing a named member (value) of the `enumName` type. Each field must have a unique name that does not match the name or title of any other field in the enum. To retrieve an enum member, reference its field name using dot notation syntax (i.e., `enumName.fieldName`).
- `<title_*>` is a "const string" title assigned to a field. If one does not specify a title, the field's title is the "string" representation of its name. The input.enum() function displays field titles within its dropdown in the script's "Settings/Inputs" tab. Users can also retrieve a field's title with the str.tostring() function. As with field names, each field's title must not match the name or title of any other field in the enum.

This example declares an `maChoice` enum. Each field within this declaration represents a distinct member of the `maChoice` enum type:

//@enum An enumeration of named values for moving average selection. //@field sma Selects a Simple Moving Average. //@field ema Selects an Exponential Moving Average. //@field wma Selects a Weighted Moving Average. //@field hma Selects a Hull Moving Average. enum maChoice sma = "Simple Moving Average" ema = "Exponential

Moving Average" wma = "Weighted Moving Average" hma = "Hull Moving Average"

Note that:

- All the enum’s possible values are available upon the *first* script execution and do not change across subsequent executions. Hence, they automatically adopt the simple qualifier.

The script below uses the `maChoice` enum within an input.enum() call to create a *dropdown* input in the “Settings/Inputs” tab that displays all the field titles. The `maInput` value represents the member of the enum that corresponds to the user-selected title. The script uses the selected member within a switch structure to determine the built-in moving average it calculates:

//@version=6 indicator("Enum types demo", overlay = true) //@enum An enumeration of named values for moving average selection. //@field sma Selects a Simple Moving Average. //@field ema Selects an Exponential Moving Average. //@field wma Selects a Weighted Moving Average. //@field hma Selects a Hull Moving Average. enum maChoice sma = "Simple Moving Average" ema = "Exponential Moving Average" wma = "Weighted Moving Average" hma = "Hull Moving Average" // @variable The `maChoice` member representing a selected moving average name. maChoice maInput = input.enum(maChoice.sma, "Moving average type") //@variable The length of the moving average. int lengthInput = input.int(20, "Length", 1, 4999) //@variable The moving average selected by the `maInput`. float selectedMA = switch maInput maChoice.sma => ta.sma(close, lengthInput) maChoice.ema => ta.ema(close, lengthInput) maChoice.wma => ta.wma(close, lengthInput) maChoice.hma => ta.hma(close, lengthInput) // Plot the `selectedMA`. plot(selectedMA, "Selected moving average", color.teal, 3)

See the Enums page and the Enum input section of the Inputs page to learn more about using enums and enum inputs.

void

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

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

na value

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

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

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

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

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

```
float myVar = na
```

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

myVar = float(na)

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

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

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

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

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

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

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

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

//@version=6 indicator("na protection demo", overlay = true) // @variable The result of calculating the all-time high price with an initial value of `na`. var float allTimeHigh = na // Reassign the value of the `allTimeHigh`. // Returns `na` on all bars because `math.max()` can't compare the `high` to an undefined value. allTimeHigh := math.max(allTimeHigh, high) plot(allTimeHigh) // Plots `na` on all bars.

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

```
//@version=6 indicator("na protection demo", overlay = true) //
@variable The result of calculating the all-time high price with an initial
value of `na`. var float allTimeHigh = na // Reassign the value of the
`allTimeHigh`. // We've used `nz()` to prevent the initial `na` value
from persisting throughout the calculation. allTimeHigh :=
math.max(nz(allTimeHigh), high) plot(allTimeHigh)
```

Type templates

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

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

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

Users can construct type templates from:

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

Note that:

- Maps can use any of these types as *values*, but they can only accept fundamental types or enum types as *keys*.

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

```
//@version=6 indicator("Type templates demo") //@variable A variable
initially assigned to `na` that accepts arrays of "int" values. array<int>
intArray = na //@variable An empty matrix that holds "float" values.
floatMatrix = matrix.new<float>() //@variable An empty map that
holds "string" keys and "color" values. stringColorMap =
map.new<string, color>()
```

Type casting

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

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=6 indicator("Explicit casting demo", overlay = true) //
@variable The length of the SMA calculation. Qualified as "const float".
float LENGTH = 10.0 float sma = ta.sma(close, LENGTH) // Compilation
error. The `length` parameter requires an "int" value. plot(sma)
```

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

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

```
//@version=6 indicator("explicit casting demo") //@variable The length
of the SMA calculation. Qualified as "const float". float LENGTH = 10.0
float sma = ta.sma(close, int(LENGTH)) // Compiles successfully since
we've converted the `LENGTH` to "int". plot(sma)
```

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

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

```
// Explicitly specify that the variable references "label" objects: label
myLabel = na // Explicitly cast the `na` value to the "label" type:
myLabel = label(na)
```

Tuples

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

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

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

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

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

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

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

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

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

For instance, this `roundedOHLC()` function returns a tuple containing OHLC values rounded to the nearest prices that are divisible by the symbol’s minimum tick size (syminfo.mintick). 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=6 indicator("Qualified types in tuples demo") getParameters(float source, simple int length) => // The second item in the tuple becomes a "series" if `source` is a "series" // because all returned tuple elements adopt the strongest qualifier. [source, length] // Although the `length` argument is a "const int", the `len` variable is a "series" because `src` is a "series". [src, len] = getParameters(close, 20) // Causes a compilation error because `ta.ema()` requires a "simple int" `length` argument. plot(ta.ema(src, len))
```

Previous

Next

Script structure

Script structure

A Pine script follows this general structure:

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

Version

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

//@version=6

- The version number is a number from 1 to 6.
- The compiler annotation is not mandatory. When omitted, version 1 is assumed. It is strongly recommended to always use the latest version of the language.
- While it is synctactically correct to place the version compiler annotation anywhere in the script, it is much more useful to readers when it appears at the top of the script.

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

Declaration statement

All Pine scripts must contain one declaration statement, which is a call to one of these functions:

- indicator()
- strategy()
- library()

The declaration statement:

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

Each script type has distinct basic requirements. Scripts that do not meet these criteria cause a compilation error:

- Indicators must call at least one function that creates a script output, such as plot(), plotshape(), barcolor(), line.new(), log.info(), alert(), etc.
- Strategies must call at least one order placement command or other output function.
- Libraries must export at least one user-defined function, method, type, or enum.

Code

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

- variable declaration

- variable reassignment
- function declaration
- built-in function call, user-defined function call or a library function call
- if, for, while, switch, type, or enum *structure*.

Statements can be arranged in multiple ways:

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

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

//@version=6 indicator("My Script") plot(close)

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

//@version=6 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 strategy by selecting “New blank strategy” instead:

```
//@version=6 strategy("My Strategy", overlay=true, margin_long=100, margin_short=100) longCondition = ta.crossover(ta.sma(close, 14), ta.sma(close, 28)) if (longCondition) strategy.entry("My Long Entry Id", strategy.long) shortCondition = ta.crossunder(ta.sma(close, 14), ta.sma(close, 28)) if (shortCondition) strategy.entry("My Short Entry Id", strategy.short)
```

Comments

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

```
//@version=6 indicator("") // This line is a comment a = close // This is also a comment plot(a)
```

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

Line wrapping

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

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

may be wrapped as:

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

A long plot() call may be wrapped as:

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

Expressions inside *local* code blocks can also use line wrapping. However, because the code in a local block requires indentation by four spaces or a tab, we recommend using a *larger* indentation that is not a multiple of four spaces for the wrapped lines inside the block. For example:

```
upDown(float s) => var int ud = 0 bool isEqual = s == s[1] bool isGrowing = s > s[1] ud := isEqual ? 0 : isGrowing ? (ud <= 0 ? 1 : ud + 1) : (ud >= 0 ? -1 : ud - 1)
```

Wrapped lines can also include comments:

```
//@version=6 indicator("") c = open > close ? color.red : high > high[1] ? color.lime : // A comment low < low[1] ? color.blue : color.black bgcolor(c)
```

Compiler annotations

Compiler annotations are comments that issue special instructions for a script:

- `//@version=` specifies the PineScript version that the compiler will use. The number in this annotation should not be confused with the script’s version number, which updates on every saved change to the code.
- `//@description` sets a custom description for scripts that use the library() declaration statement.
- `//@function`, `//@param` and `//@returns` add custom descriptions for a user-defined function or method, its parameters, and its result when placed above the function declaration.
- `//@type` adds a custom description for a user-defined type (UDT) when placed above the type declaration.
- `//@enum` adds a custom description for an enum types when placed above the enum declaration.
- `//@field` adds a custom description for the field of a user-defined type (UDT) or an enum types when placed above the type or enum declaration.
- `//@variable` adds a custom description for a variable when placed above its declaration.
- `//@strategy_alert_message` provides a default message for strategy scripts to pre-fill the “Message” field in the alert creation dialog.

The Pine Editor also features two specialized annotations, `//#region` and `//#endregion`, that create *collapsible* code regions. Clicking the dropdown arrow next to a `//#region` line collapses all the code between that line and the nearest `//#endregion` annotation below it.

This example draws a triangle using three interactively selected points on the chart. The script illustrates how one can use compiler and Editor annotations to document code and make it easier to navigate:

image

//@version=6 indicator("Triangle", "", true) //#region ———————————————————————— Constants and inputs int TIME_DEFAULT = 0 float PRICE_DEFAULT = 0.0 x1Input = input.time(TIME_DEFAULT, "Point 1", inline = "1", confirm = true) y1Input = input.price(PRICE_DEFAULT, "", inline = "1", tooltip = "Pick point 1", confirm = true) x2Input = input.time(TIME_DEFAULT, "Point 2", inline = "2", confirm = true) y2Input = input.price(PRICE_DEFAULT, "", inline = "2", tooltip = "Pick point 2", confirm = true) x3Input = input.time(TIME_DEFAULT, "Point 3", inline = "3", confirm = true) y3Input = input.price(PRICE_DEFAULT, "", inline = "3", tooltip = "Pick point 3", confirm = true) //#endregion //#region ———————————————————————— Types and functions // @type Used to represent the coordinates and color to draw a triangle. // @field time1 Time of first point. // @field time2 Time of second point. // @field time3 Time of third point. // @field price1 Price of first point. // @field price2 Price of second point. // @field price3 Price of third point. // @field lineColor Color to be used to draw the triangle lines. type Triangle int time1 int time2 int time3 float price1 float price2 float price3 color lineColor //@function Draws a triangle using the coordinates of the `t` object. //@param t (Triangle) Object representing the triangle to be drawn. //@returns The ID of the last line drawn. drawTriangle(Triangle t) => line.new(t.time1, t.price1, t.time2, t.price2, xloc = xloc.bar_time, color = t.lineColor) line.new(t.time2, t.price2, t.time3, t.price3, xloc = xloc.bar_time, color = t.lineColor) line.new(t.time1, t.price1, t.time3, t.price3, xloc = xloc.bar_time, color = t.lineColor) //#endregion //#region ———————————————————————— Calculations // Draw the triangle only once on the last historical bar. if barstate.islastconfirmedhistory //@variable Used to hold the Triangle object to be drawn. Triangle triangle = Triangle.new() triangle.time1 := x1Input triangle.time2 := x2Input triangle.time3 := x3Input triangle.price1 := y1Input triangle.price2 := y2Input triangle.price3 := y3Input triangle.lineColor := color.purple drawTriangle(triangle) // #endregion

Previous

Next

Identifiers

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

Previous

Next

Variable declarations

Pine Script® v6 Documentation

User Manual / Language / Variable declarations

Variable declarations

Introduction

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

```
[<declaration_mode>] [<type>] <identifier> = <expression> | <struct
```

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 a valid *type keyword* with an optional *qualifier prefix*. Specifying a variable’s type is optional in most cases. See the Type system page to learn more.
- <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. Note that the last declaration requires four lines of code because it uses the returned value from an if statement:

BULL_COLOR = color.lime i = 1 len = input(20, "Length") float f = 10.5 closeRoundedToTick = math.round_to_mintick(close) sma = ta.sma(close, 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

The formal syntax of a variable declaration is:

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

<declaration_mode>
    var | varip

<type>
    int | float | bool | color | string | line | linefill | label |
```

Initialization with na

In most cases, an explicit type declaration is redundant because type is automatically inferred from the value on the right of the = at compile time, so the decision to use them is often a matter of preference. For example:

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

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

Tuple declarations

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

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

Using an underscore (_) as an identifier

When declaring a variable, it is possible to use a single underscore (_) as its identifier. A value assigned to such a variable cannot be accessed. You can assign any number of values to a _ identifier anywhere in the script, even if the current scope already has such an assignment.

This is particularly useful when a tuple returns unneeded values. Let’s write another Bollinger Bands script. Here, we only need the bands themselves, without the center line:

```
//@version=6 indicator("Underscore demo") // We do not need the
middle Bollinger Bands value, and do not use it. // To make this clear,
we assign it to the `_` identifier. [_, bbUpper, bbLower] = ta.bb(close, 5,
4) // We can continue to use `_` in the same code without causing
compilation errors: [bbMiddleLong, _, _] = ta.bb(close, 20, 2)
plot(bbUpper)
```

Variable reassignment

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

```
//@version=6 indicator("", "", true) sensitivityInput = input.int(2,
"Sensitivity", minval = 1, tooltip = "Higher values make color changes
less sensitive.") ma = ta.sma(close, 20) maUp = ta.rising(ma,
sensitivityInput) maDn = ta.falling(ma, sensitivityInput) // On first bar
only, initialize color to gray var maColor = color.gray if maUp // MA has
risen for two bars in a row; make it lime. maColor := color.lime else if
maDn // MA has fallen for two bars in a row; make it fuchsia.
maColor := color.fuchsia plot(ma, "MA", maColor, 2)
```

Note that:

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

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

Declaration modes

Understanding the impact that declaration modes have on the behavior of variables requires prior knowledge of Pine Script’s execution model.

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

- “On each bar”, when none is specified
- var
- varip

On each bar

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

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

var

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

```
//@version=6 indicator("Green Bars Count") var count = 0 isGreen =
close >= open if isGreen count := count + 1 plot(count)
```

image

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

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

```
//@version=6 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=6 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=6 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=6 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.

Previous

Next

Operators

Operators

Introduction

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

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

Other operators are used to assign values to variables:

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

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

This script will produce a compilation error:

```
//@version=6 indicator("") lenInput = input.int(14, "Length") factor = year > 2020 ? 3 : 1 adjustedLength = lenInput * factor ma = ta.ema(close, adjustedLength) // Compilation error! plot(ma)
```

The compiler will complain: *Cannot call ‘ta.ema’ with argument ‘length’=‘adjustedLength’. An argument of ‘series int’ type was used but a ‘simple int’ is expected;*. This is happening because `lenInput` is an “input int” but `factor` is a “series int” (it can only be determined by looking at the value of [year](#) on each bar). The `adjustedLength` variable is thus assigned a “series int” value. Our problem is that the Reference Manual entry for [ta.ema()](#) tells us that its `length` parameter requires a “simple” value, which is a weaker qualifier than “series”, so a “series int” value is not allowed.

The solution to our conundrum requires:

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

Arithmetic operators

There are five arithmetic operators in Pine Script®:

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

The arithmetic operators above are all *binary*, meaning they need two *operands* — or values — to work on, as in the example operation `1 + 2`. The `+` and `-` can also be *unary* operators, which means they work on one operand, as in the example values `-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.

Note that when using the division operator with “int” operands, if the two “int” values are not evenly divisible, the result of the division is always a number with a fractional value, e.g., 5/2 = 2.5. To discard the fractional remainder, wrap the division with the int() function, or round the result using math.round(), math.floor(), or math.ceil().

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

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

```
//@version=6 indicator("Modulo function") modulo(series int a, series int b) => a - b * math.floor(nz(a/b)) plot(modulo(-1, 100))
```

Comparison operators

There are six comparison operators in Pine Script:

Operator	Meaning
<	Less Than
<=	Less Than or Equal To
!=	Not Equal
==	Equal
>	Greater Than
>=	Greater Than or Equal To

Comparison operations are binary, and return a result of type “bool”, i.e., true or false. The == equal and != not equal operators can work with operands of any fundamental type, such as colors and strings, while the other comparison operators are only applicable to numerical values. Therefore, "a" != "b" is a valid comparison, but "a" > "b" is invalid.

Examples:

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

Logical operators

There are three logical operators in Pine Script:

Operator	Meaning
`not`	Negation
and	Logical Conjunction
`or`	Logical Disjunction

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

and operator truth table:

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

`or` operator truth table:

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

`?:` ternary operator

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

condition ? valueWhenConditionIsTrue : valueWhenConditionIsFalse

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

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

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

The example is calculated from left to right:

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

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

[] history-referencing operator

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

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

Because series grow dynamically, as the script calculates on successive bars, a constant historical offset refers 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 handle these special cases using the na() and nz() functions, na values can introduce invalid results in your script's calculations that can affect calculations all the way to the realtime bar.

These are all valid uses of the [] operator:

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

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

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

Operator precedence

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

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

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

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

= assignment operator

The = operator assigns an initial value or reference to a declared variable. It means *this is a new variable, and it starts with this value*.

These are all valid variable declarations:

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

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

:= reassignment operator

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

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

```
//@version=6 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 ta.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 ta.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:

image

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.

Previous

Next

Conditional structures

Conditional structures

Introduction

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

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

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

Some Pine Script built-in functions are **not** callable from within the local blocks of conditional structures, including barcolor(), bgcolor(), plot(), plotshape(), plotchar(), plotarrow(), plotcandle(), plotbar(), hline(), fill(), alertcondition(), indicator(), strategy(), and library().

This restriction does not entail their functionality cannot be controlled by conditions evaluated by your script — only that it cannot be done by including them in conditional structures. Note that while `input*.()` function calls are allowed in local blocks, their functionality is the same as if they were in the script's *global scope*.

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

if structure

if used for its side effects

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

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

where:

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

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

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

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

When no <expression> has evaluated to true and no `else` clause exists, na is returned. The only exception to this is if the structure returns "bool" values — in that case, false is returned instead.

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=6 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. If other local blocks return “bool” values, false will be returned instead.

This is an example:

//@version=6 indicator("", "", true) string barState = if barstate.islastconfirmedhistory "islastconfirmedhistory" else if barstate.isnew "isnew" else if barstate.isrealtime "isrealtime" else "other" f_print(_text) => var table _t = table.new(position.middle_right, 1, 1) table.cell(_t, 0, 0, _text, bgcolor = color.yellow) f_print(barState)

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

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

x = if close > open close

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

if condition1 if condition2 if condition3 expression

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

if condition1 and condition2 and condition3 expression

switch structure

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

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

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

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

where:

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

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

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

As with the if structure, if no local block is exectuted, the expression returns either false (when other local blocks return a "bool" value) or na (in all other cases).

`switch` with an expression

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

```
//@version=6 indicator("Switch using an expression", "", true) string
maType = input.string("EMA", "MA type", options = ["EMA", "SMA",
"RMA", "WMA"]) int maLength = input.int(10, "MA length", minval = 2)
float ma = switch maType "EMA" => ta.ema(close, maLength) "SMA"
=> ta.sma(close, maLength) "RMA" => ta.rma(close, maLength)
"WMA" => ta.wma(close, maLength) => runtime.error("No matching
MA type found.") float(na) plot(ma)
```

Note that:

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

switch without an expression

This is an example of a switch structure which does not use an expression:

```
//@version=6 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=6 strategy("Switch without an expression", "", true)
switch // Compiler warning! Will not calculate correctly!
ta.crossover( ta.sma(close, 14), ta.sma(close, 28)) =>
strategy.entry("Long ID", strategy.long) ta.crossunder(ta.sma(close,
14), ta.sma(close, 28)) => strategy.entry("Short ID", strategy.short)
```

Matching local block type requirement

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

This code compiles fine because close and open are both of the `float` type:

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

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

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

Previous

Next

Loops

Pine Script® v6 Documentation

User Manual / Language / Loops

Loops

Introduction

Loops are structures that repeatedly execute a block of statements based on specified criteria. They allow scripts to perform repetitive tasks without requiring duplicated lines of code. Pine Script® features three distinct loop types: for, while, and for…in.

Every loop structure in Pine Script consists of two main parts: a *loop header* and a *loop body*. The loop header determines the criteria under which the loop executes. The loop body is the indented block of code (local block) that the script executes on each loop cycle (*iteration*) as long as the header's conditions remain valid. See the Common characteristics section to learn more.

Understanding when and how to use loops is essential for making the most of the power of Pine Script. Inefficient or unnecessary usage of loops can lead to suboptimal runtime performance. However, effectively using loops when necessary enables scripts to perform a wide range of calculations that would otherwise be impractical or impossible without them.

When loops are unnecessary

Pine's execution model and time series structure make loops *unnecessary* in many situations.

When a user adds a Pine script to a chart, it runs within the equivalent of a *large loop*, executing its code once on *every* historical bar and realtime tick in the available data. Scripts can access the values from the executions on previous bars with the history-referencing operator, and calculated values can *persist* across executions when assigned to variables declared with the var or varip keywords. These capabilities enable scripts to utilize bar-by-bar calculations to accomplish various tasks instead of relying on explicit loops.

In addition, several built-ins, such as those in the `ta.*` namespace, are internally optimized to eliminate the need to use loops for various calculations.

Let’s consider a simple example demonstrating unnecessary loop usage in Pine Script. To calculate the average close over a specified number of bars, newcomers to Pine may write a code like the following, which uses a for loop to calculate the sum of historical values over `lengthInput` bars and divides the result by the `lengthInput`:

image

```
//@version=6 indicator("Unnecessary loops demo", overlay = true) //
@variable The number of bars in the calculation window. int
lengthInput = input.int(defval = 20, title = "Length") //@variable The
sum of `close` values over `lengthInput` bars. float closeSum = 0 //
Loop over the most recent `lengthInput` bars, adding each bar's
`close` to the `closeSum`. for i = 0 to lengthInput - 1 closeSum +=
close[i] //@variable The average `close` value over `lengthInput` bars.
float avgClose = closeSum / lengthInput // Plot the `avgClose`.
plot(avgClose, "Average close", color.orange, 2)
```

Using a for loop is an **unnecessary**, inefficient way to accomplish tasks like this in Pine. There are several ways to utilize the execution model and the available built-ins to eliminate this loop. Below, we replaced these calculations with a simple call to the ta.sma() function. This code is shorter, and it achieves the same result much more efficiently:

image

```
//@version=6 indicator("Unnecessary loops corrected demo", overlay =
true) //@variable The number of bars in the calculation window. int
lengthInput = input.int(defval = 20, title = "Length") //@variable The
average `close` value over `lengthInput` bars. float avgClose =
```

ta.sma(close, lengthInput) // Plot the `avgClose`. plot(avgClose, "Average close", color.blue, 2)

Note that:

- Users can see the substantial difference in efficiency between these two example scripts by analyzing their performance with the Pine Profiler.

When loops are necessary

Although Pine's execution model, time series, and available built-ins often eliminate the need for loops in many cases, not all iterative tasks have loop-free alternatives. Loops *are necessary* for several types of tasks, including:

- Iterating through or manipulating collections (arrays, matrices, and maps)
- Performing calculations that one **cannot** accomplish with loop-free expressions or the available built-ins
- Looking back through history to analyze past bars with a reference value only available on the *current bar*

For example, a loop is *necessary* to identify which past bars' high values are above the current bar's high because the current value is **not** obtainable during a script's executions on previous bars. The script can only access the current bar's value while it executes on that bar, and it must *look back* through the historical series during that execution to compare the previous values.

The script below uses a for loop to compare the high values of `lengthInput` previous bars with the last historical bar's high. Within the loop, it calls label.new() to draw a circular label above each past bar that has a high value exceeding that of the last historical bar:

image

//@version=6 indicator("Necessary loop demo", overlay = true, max_labels_count = 500) //@variable The number of previous `high` values to compare to the last historical bar's `high`. int lengthInput =

input.int(20, "Length", 1, 500) if barstate.islastconfirmedhistory // Draw a horizontal line segment at the last historical bar's `high` to visualize the level. line.new(bar_index - lengthInput, high, bar_index, high, color = color.gray, style = line.style_dashed, width = 2) // Create a `for` loop that counts from 1 to `lengthInput`. for i = 1 to lengthInput // Draw a circular `label` above the bar from `i` bars ago if that bar's `high` is above the current `high`. if high[i] > high label.new(bar_index - i, na, "", yloc = yloc.abovebar, color = color.purple, style = label.style_circle, size = size.tiny) // Highlight the last historical bar. barcolor(barstate.islastconfirmedhistory ? color.orange : na, title = "Last historical bar highlight")

Note that:

- Each *iteration* of the for loop retrieves a previous bar's high with the history-referencing operator [], using the loop's *counter* (`i`) as the historical offset. The label.new() call also uses the counter to determine each label's x-coordinate.
- The indicator declaration statement includes `max_labels_count = 500`, meaning the script can show up to 500 labels on the chart.
- The script calls barcolor() to highlight the last historical chart bar, and it draws a horizontal line at that bar's high for visual reference.

Common characteristics

The for, while, and for...in loop statements all have similarities in their structure, syntax, and general behavior. Before we explore each specific loop type, let's familiarize ourselves with these characteristics.

Structure and syntax

In any loop statement, programmers define the criteria under which a script remains in a loop and performs *iterations*, where an iteration refers to *one execution* of the code within the loop's local block (*body*). These criteria are part of the *loop header*. A script evaluates the header's criteria *before* each iteration, only allowing new iterations to occur while they remain valid. When the header’s criteria are no longer valid, the script *exits* the loop and skips over its body.

The specific header syntax varies with each loop statement (for, while, or for...in) because each uses *distinct* criteria to control its iterations. Effective use of loops entails choosing the structure with control criteria best suited for a script’s required tasks. See the `for` loops, `while` loops, and `for...in` loops sections below for more information on each loop statement and its control criteria.

All loop statements in Pine Script follow the same general syntax:

```
[variables = | :=] loop_header
    statements | continue | break
    return_expression
```

Where:

- `loop_header` represents the loop structure’s header statement, which defines the criteria that control the loop’s iterations.
- `statements` represents the code statements and expressions within the loop’s body, i.e., the *indented* block of code beneath the loop header. All code within the body belongs to the loop’s local scope.
- `continue` and `break` are loop-specific *keywords* that control the flow of a loop’s iterations. The `continue` keyword instructs the script to *skip* the remainder of the current loop iteration and *continue* to the next iteration. The `break` keyword prompts the script to *stop* the current iteration and *exit* the loop entirely. See this section below for more information.
- `return_expression` refers to the *last* code line or block within the loop’s body. The loop returns the results from this code after the final iteration. If the loop skips parts of some iterations or stops prematurely due to a `continue` or `break` statement, the returned values or references are those of the latest iteration that evaluated this code. To use the loop’s returned results, assign them to a variable or tuple.
- `variables` represents an optional variable or tuple to hold the values or references from the last evaluation of the

return_expression. The script can assign the loop’s returned results to variables only if the results are not void. If the loop’s conditions prevent iteration, or if no iterations evaluate the return_expression, the variables’ assigned values and references are na.

Scope

All code lines that a script executes within a loop must have an indentation of *four spaces* or a *tab* relative to the loop’s header. The indented lines following the header define the loop’s *body*. This code represents a *local block*, meaning that all the definitions within the body are accessible only during the loop’s execution. In other words, the code within the loop’s body is part of its *local scope*.

Scripts can modify and reassign most variables from *outer* scopes inside a loop. However, any variables declared within the loop’s body strictly belong to that loop’s local scope. A script **cannot** access a loop’s declared variables *outside* its local block.

Note that:

- Variables declared within a loop’s *header* are also part of the local scope. For instance, a script cannot use the *counter variable* in a for loop anywhere but within the loop’s local block.

The body of any Pine loop statement can include conditional structures and *nested* loop statements. When a loop includes nested structures, each structure within the body maintains a *distinct* local scope. For example, variables declared within an *outer* loop’s scope are accessible to an *inner* loop. However, any variables declared within the inner loop’s scope are **not** accessible to the outer loop.

The simple example below demonstrates how a loop’s local scope works. This script calls label.new() within a for loop on the last historical bar to draw labels above lengthInput past bars. The color of each label depends on the labelColor variable declared *within* the loop’s local block, and each label’s location depends on the loop counter (i):

image

```
//@version=6 indicator("Loop scope demo", overlay = true) //@variable
The number of bars in the calculation. int lengthInput = input.int(20,
"Lookback length", 1) if barstate.islastconfirmedhistory for i = 1 to
lengthInput //@variable Has a value of `color.blue` if `close[i]` is above
the current `close`, `color.orange` otherwise. // This variable is LOCAL
to the `for` loop's scope. color labelColor = close[i] > close ?
color.blue : color.orange // Display a colored `label` on the historical
`high` from `i` bars back, using `labelColor` to set the color.
label.new(bar_index - i, high[i], "", color = labelColor, size =
size.normal)
```

In the above code, the `i` and `labelColor` variables are only accessible to the for loop’s local scope. They are **not** usable within any outer scopes. Here, we added a label.new() call *after* the loop with `bar_index - i` as the x argument and `labelColor` as the `color` argument. This code causes a *compilation error* because neither `i` nor `labelColor` are valid variables in the outer scope:

```
//@version=6 indicator("Loop scope demo", overlay = true) //@variable
The number of bars in the calculation. int lengthInput = input.int(20,
"Lookback length", 1) if barstate.islastconfirmedhistory for i = 1 to
lengthInput //@variable Has a value of `color.blue` if `close[i]` is above
the current `close`, `color.orange` otherwise. // This variable is LOCAL
to the `for` loop's scope. color labelColor = close[i] > close ?
color.blue : color.orange // Display a colored `label` on the historical
`high` from `i` bars back, using `labelColor` to set the color.
label.new(bar_index - i, high[i], "", color = labelColor, size =
size.normal) // Call `label.new()` to using the `i` and `labelColor`
variables outside the loop's local scope. // This code causes a
compilation error because these variables are not accessible in this
location. label.new( bar_index - i, low, "Scope test", textcolor =
color.white, color = labelColor, style = label.style_label_up )
```

Keywords and return expressions

Every loop in Pine Script implicitly *returns* values, references, or void. A loop’s returned results come from the *latest* execution of the *last* expression or nested structure within its body as of the final iteration.

The results are usable only if they are not of the void type. Loops return na results for values or references when no iterations occur. Scripts can add a variable or tuple assignment to a loop statement to hold the returned results for use in additional calculations outside the loop's local scope.

The values or references that a loop returns usually come from evaluating the last written expression or nested code block on the *final* iteration. However, a loop's body can include `continue` and `break` keywords to control the flow of iterations beyond the criteria the loop header specifies, which can also affect the returned results. Programmers often include these keywords within conditional structures to control how iterations behave when certain conditions occur.

The `continue` keyword instructs a script to *skip* the remaining statements and expressions in the current loop iteration, re-evaluate the loop header's criteria, and proceed to the *next* iteration. The script *exits* the loop if the header's criteria do not allow another iteration.

The `break` keyword instructs a script to *stop* the loop entirely and immediately *exit* at that point without allowing any subsequent iterations. After breaking the loop, the script skips any remaining code within the loop's body and *does not* re-evaluate the header's criteria.

If a loop skips parts of iterations or stops prematurely due to a `continue` or `break` statement, it returns the values and references from the *last iteration* where the script *evaluated* the return expression. If the script did not evaluate the return expression across *any* of the loop's iterations, the loop returns na results for all non-void types.

The example below selectively displays numbers from an array within a label on the last historical bar. It uses a for…in loop to iterate through the array's elements and build a "string" to use as the displayed text. The loop's body contains an if statement that controls the flow of specific iterations. If the `number` in the current iteration is 8, the script immediately *exits* the loop using the `break` keyword. Otherwise, if the `number` is even, it *skips* the rest of the current iteration and moves to the next one using the `continue` keyword.

If neither of the if statement’s conditions occur, the script evaluates the *last expression* within the loop’s body (i.e., the return expression), which converts the current `number` to a “string” and concatenates the result with the `tempString` value. The loop returns the *last evaluated result* from this expression after termination. The script assigns the returned value to the `finalLabelText` variable and uses that variable as the `text` argument in the label.new() call:

image

```
//@version=6 indicator("Loop keywords and variable assignment
demo") //@variable An `array` of arbitrary "int" values to selectively
convert to "string" and display in a `label`. var array<int>
randomArray = array.from(1, 5, 2, -3, 14, 7, 9, 8, 15, 12) // Label
creation logic. if barstate.islastconfirmedhistory //@variable A "string"
containing representations of selected values from the `randomArray`.
string tempString = "" //@variable The final text to display in the
`label`. The `for..in` loop returns the result after it terminates. string
finalLabelText = for number in randomArray // Stop the current iteration
and exit the loop if the `number` from the `randomArray` is 8. if
number == 8 break // Skip the rest of the current iteration and proceed
to the next iteration if the `number` is even. else if number % 2 == 0
continue // Convert the `number` to a "string", append ", ", and
concatenate the result with the current `tempString`. // This code
represents the loop's return expression. tempString +=
str.tostring(number) + ", " // Display the value of the `finalLabelText`
within a `label` on the current bar. label.new(bar_index, 0,
finalLabelText, color = color.blue, textcolor = color.white, size =
size.huge)
```

Note that:

- The label displays only *odd* numbers from the array because the script does not reassign the `tempString` when the loop iteration’s `number` is even. However, it does not include the *last* odd number from the array (15) because the loop stops when `number == 8`, preventing iteration over the remaining `randomArray` elements.
- When the script exits the loop due to the `break` keyword, the loop’s return value becomes the last evaluated result from the

`tempString` reassignment expression. In this case, the last time that code executes is on the iteration where `number == 9`.

for loops

The for loop statement creates a *count-controlled* loop, which uses a *counter* variable to manage the iterative executions of its local code block. The counter starts at a predefined initial value, and the loop increments or decrements the counter by a fixed amount after each iteration. The loop stops its iterations after the counter reaches a specified final value.

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

```
[variables = | :=] for counter = from_num to to_num [by step_num]
    statements | continue | break
    return_expression
```

Where the following parts define the *loop header*:

- `counter` represents the counter variable, which can be any valid identifier. The loop increments or decrements this variable's value from the initial value (`from_num`) to the final value (`to_num`) by a fixed amount (`step_num`) after each iteration. The last possible iteration occurs when the variable's value reaches the `to_num` value.
- `from_num` is the `counter` variable's initial value on the first iteration.
- `to_num` is the *final* `counter` value for which the loop's header allows a new iteration. The loop adjusts the `counter` value by the `step_num` amount until it reaches or passes this value. If the script modifies the `to_num` during a loop iteration, the loop header uses the new value to control the allowed subsequent iterations.
- `step_num` is a positive value representing the amount by which the `counter` value increases or decreases until it reaches or passes the `to_num` value. If the `from_num` value is greater than the *initial* `to_num` value, the loop *subtracts* this amount from the `counter` value after each iteration. Otherwise, the loop *adds* this amount after each iteration. The default is 1.

Refer to the Common characteristics section above for detailed information about the `variables`, `statements`, `continue`, `break`, and `return_expression` parts of the loop's syntax.

This simple script demonstrates a for loop that draws several labels at future bar indices during its execution on the last historical chart bar. The loop's counter starts at 0, then increases by 1 until it reaches a value of 10, at which point the final iteration occurs:

image

```
//@version=6 indicator("Simple `for` loop demo") if barstate.islastconfirmedhistory // Define a `for` loop that iterates from `i == 0` to `i == 10` by 1 (11 total iterations). for i = 0 to 10 // Draw a new label `i` bars ahead of the current bar. label.new(bar_index + i, 0, str.tostring(i), textcolor = color.white, size = size.large)
```

Note that:

- The `i` variable represents the loop's *counter*. This variable is local to the loop's scope, meaning no *outer scopes* can access it. The code uses the variable within the loop's body to determine the location and text of each label drawing.
- Programmers often use `i`, `j`, and `k` as loop counter identifiers. However, *any* valid variable name is allowed. For example, this code behaves the same if we name the counter `offset` instead of `i`.
- The for loop structure *automatically* manages the counter variable. We do not need to define code in the loop's body to increment its value.

The direction in which a for loop adjusts its counter depends on the *initial* `from_num` and `to_num` values in the loop's header, and the direction does not change across iterations. The loop counts *upward* after each iteration when the `to_num` value is *above* the `from_num` value, as shown in the previous example. If the `to_num` value is *below* the `from_num` value, the loop counts *downward* instead.

The script below calculates and plots the volume-weighted moving average (VWMA) of open prices across a specified number of bars.

Then, it uses a downward-counting for loop to compare the last historical bar's value to the values from previous bars, starting with the oldest bar in the specified lookback window. On each loop iteration, the script retrieves a previous bar's vwmaOpen value, calculates the difference from the current bar's value, and displays the result in a label at the past bar's opening price:

image

```
//@version=6 indicator("`for` loop demo", "VWMA differences", true, max_labels_count = 500) //@variable Display color for indicator visuals. const color DISPLAY_COLOR = color.rgb(17, 127, 218) //@variable The number of bars in the `vwmaOpen` calculation. int maLengthInput = input.int(20, "VWMA length", 1) //@variable The number of past bars to look back through and compare to the current bar. int lookbackInput = input.int(15, "Lookback length", 1, 500) //@variable The volume-weighted moving average of `open` values over `maLengthInput` bars. float vwmaOpen = ta.vwma(open, maLengthInput) if barstate.islastconfirmedhistory // Define a `for` loop that counts *downward* from `i == lookbackInput` to `i == 1`. for i = lookbackInput to 1 //@variable The difference between the `vwmaOpen` from `i` bars ago and the current `vwmaOpen`. float vwmaDifference = vwmaOpen[i] - vwmaOpen //@variable A "string" representation of `vwmaDifference`, rounded to two fractional digits. string displayText = (vwmaDifference > 0 ? "+" : "") + str.tostring(vwmaDifference, "0.00") // Draw a label showing the `displayText` at the `open` of the bar from `i` bars back. label.new( bar_index - i, open[i], displayText, textcolor = color.white, color = DISPLAY_COLOR, style = label.style_label_lower_right, size = size.normal ) // Plot the `vwmaOpen` value. plot(vwmaOpen, "VWMA", color = DISPLAY_COLOR, linewidth = 2)
```

Note that:

- The script uses the loop's counter (i) to within the history-referencing operator to retrieve past values of the vwmaOpen series. It also uses the counter to determine the location of each label drawing.

- The loop in this example *decreases* the counter by one on each iteration because the final counter value in the loop's header (1) is less than the starting value (`lookbackInput`).

Programmers can use for loops to iterate through collections, such as arrays and matrices. The loop's counter can serve as an *index* for retrieving or modifying a collection's contents. For example, this code block uses array.get() inside a for loop to successively retrieve elements from an array:

int lastIndex = array.size(myArray) - 1 for i = 0 to lastIndex element = array.get(i)

Note that:

- Array *indexing* starts from 0, but the array.size() function *counts* array elements starting from 1. Therefore, we must subtract 1 from the array's size to get the maximum index value. This way, the loop counter avoids representing an out-of-bounds index on the last loop iteration.
- The for...in loop statement is often the *preferred* way to loop through collections. However, programmers may prefer a for loop for some tasks, such as looping through stepped index values, iterating over a collection's contents in reverse or a nonlinear order, and more. See the Looping through arrays and Looping through matrices sections to learn more about the best practices for looping through these collection types.

The script below executes ta.rsi() and ta.mom() calls to calculate the RSI and momentum of close prices over three different lengths (10, 20, and 50), then displays the results using a table on the last chart bar. It stores "string" values for the header title within arrays and the "float" values of the calculated indicators within a 2x3 matrix. The script uses a for loop to access the elements in the arrays and initialize the `displayTable` header cells. It then uses *nested* for loops to iterate over the *row* and *column* indices in the `taMatrix`, access elements, convert their values to strings, and populate the remaining table cells:

image

```
//@version=6 indicator("`for` loop with collections demo", "Table of TA
Indexes", overlay = true) // Calculate the RSI and momentum of `close`
values with constant lengths of 10, 20, and 50. float rsi10 =
ta.rsi(close, 10) float rsi20 = ta.rsi(close, 20) float rsi50 = ta.rsi(close,
50) float mom10 = ta.mom(close, 10) float mom20 = ta.mom(close,
20) float mom50 = ta.mom(close, 50) if barstate.islast //@variable A
`table` that displays indicator values in the top-right corner of the
chart. var table displayTable = table.new( position.top_right, columns =
5, rows = 4, border_color = color.black, border_width = 1 ) //@variable
An array containing the "string" titles to display within the side header
of each table row. array<string> sideHeaderTitles = array.from("TA
Index", "RSI", "Momentum") //@variable An array containing the
"string" titles to representing the length of each displayed indicator.
array<string> topHeaderTitles = array.from("10", "20", "50") //
@variable A matrix containing the values to display within the table.
matrix<float> taMatrix = matrix.new<float>() // Populate the
`taMatrix` with indicator values. The first row contains RSI data and the
second contains momentum. taMatrix.add_row(0, array.from(rsi10,
rsi20, rsi50, mom10, mom20, mom50)) taMatrix.reshape(2, 3) //
Initialize top header cells. displayTable.cell(1, 0, "Bars Length",
text_color = color.white, bgcolor = color.blue)
displayTable.merge_cells(1, 0, 3, 0) // Initialize additional header cells
within a `for` loop. for i = 0 to 2 displayTable.cell(0, i + 1,
sideHeaderTitles.get(i), text_color = color.white, bgcolor = color.blue)
displayTable.cell(i + 1, 1, topHeaderTitles.get(i), text_color =
color.white, bgcolor = color.purple) // Use nested `for` loops to iterate
through the row and column indices of the `taMatrix`. for i = 0 to
taMatrix.rows() - 1 for j = 0 to taMatrix.columns() - 1 //@variable The
value stored in the `taMatrix` at the `i` row and `j` column. float
elementValue = taMatrix.get(i, j) // Initialize a cell in the `displayTable`
at the `i + 2` row and `j + 1` column showing a "string" //
representation of the `elementValue`. displayTable.cell( column = j + 1,
row = i + 2, text = str.tostring(elementValue, "#.##"), text_color =
chart.fg_color )
```

Note that:

- Both arrays of header names (`sideHeaderTitles` and `topHeaderTitles`) contain the same number of elements, enabling the script to iterate through their contents simultaneously using a single for loop.
- The nested for loops iterate over *all* the index values in the `taMatrix`. The *outer* loop iterates over each *row* index, and the *inner* loop iterates over every *column* index on each outer loop iteration.
- The script creates and displays the table only on the last historical bar and all realtime bars because the historical states of tables are *never* visible. See the Reducing drawing updates of the Profiling and optimization page for more information.

It’s important to note that a for loop’s header *dynamically* evaluates the `to_num` value at the start of *every iteration*. If the `to_num` argument is a variable and the script changes its value during an iteration, the loop uses the *new value* to update its stopping condition. Likewise, the stopping condition can change across iterations when the `to_num` argument is an expression or function call that depends on data modified in the loop’s scope, such as a call to array.size() on a locally resized array or str.length() on an adjusted string. Therefore, scripts can use for loops to perform iterative tasks where the exact number of required iterations is *not predictable* in advance, similar to while loops.

For example, the following script uses a dynamic for loop to determine the historical offset of the most recent bar whose close differs from the current bar’s close by at least one standard deviation. The script declares a `barOffset` variable with an initial value of zero and uses that variable to define the loop counter’s `to_num` boundary. Within the loop’s scope, the script increments the `barOffset` by one if the referenced bar’s `close` is not far enough from the current bar’s value. Each time the `barOffset` value increases, the loop increases its final counter value, allowing an *extra iteration*. The script plots the `barOffset` and the corresponding bar’s close for visual reference:

image

```
//@version=6 indicator("`for` loop with dynamic `to_num` demo") //
@variable The length of the standard deviation. int lengthInput =
input.int(20, "Length", 1, 4999) //@variable The standard deviation of
`close` prices over `lengthInput` bars. float stdev = ta.stdev(close,
lengthInput) //@variable The minimum bars back where the past bar's
`close` differs from the current `close` by at least `stdev`. // Used as
the weight value in the weighted average. int barOffset = 0 // Define a
`for` loop that iterates from 0 to `barsBack`. for i = 0 to barOffset //
Add 1 for each bar where the distance from that bar's `close` to the
current bar's `close` is less than `stdev`. // Each time `barsBack`
increases, it changes the loop's `to_num` boundary, allowing another
iteration. barOffset += math.abs(close - close[i]) < stdev ? 1 : 0 //
@variable A gradient color for the `barOffset` plot. color offsetColor =
color.from_gradient(barOffset, 0, lengthInput, color.blue,
color.orange) // Plot the `barOffset` in a separate pane. plot(barOffset,
"Bar offset", offsetColor, 1, plot.style_columns) // Plot the historical
`close` price from `barOffset` bars back in the main chart pane.
plot(close[barOffset], "Historical bar's price", color.blue, 3,
force_overlay = true)
```

Note that:

- Changing the `to_num` value on an iteration does not affect the established *direction* in which the loop adjusts its counter variable. For instance, if the loop in this example changed `barOffset` to -1 on any iteration, it would stop immediately after that iteration ends without reducing the `i` value.
- The script uses `force_overlay = true` in the second plot() call to display the historical closing price on the main chart pane.

while loops

The while loop statement creates a *condition-controlled* loop, which uses a *conditional expression* to control the executions of its local block. The loop continues its iterations as long as the specified condition remains `true`.

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

```
[variables = | :=] while condition
    statements | continue | break
    return_expression
```

Where the `condition` in the loop's *header* can be a literal, variable, expression, or function call that returns a "bool" value.

Refer to the Common characteristics section above for detailed information about the `variables`, `statements`, `continue`, `break`, and `return_expression` parts of the loop's syntax.

A while loop's header evaluates its `condition` before each iteration. Consequently, when the script modifies the condition within an iteration, the loop's header reflects those changes on the *next* iteration.

Depending on the specified condition in the loop header, a while loop can behave similarly to a for loop, continuing iteration until a *counter* variable reaches a specified limit. For example, the following script uses a for loop and while loop to perform the same task. Both loops draw a label displaying their respective counter value on each iteration:

image

//@version=6 indicator("`while` loop with a counter condition demo") if barstate.islastconfirmedhistory // A `for` loop that creates blue labels displaying each `i` value. for i = 0 to 10 label.new(bar_index + i, 0, str.tostring(i), color = color.blue, textcolor = color.white, size = size.large, style = label.style_label_down) //@variable An "int" to use as a counter within a `while` loop. int j = 0 // A `while` loop that creates orange labels displaying each `j` value. while j <= 10 label.new(bar_index + j, 0, str.tostring(j), color = color.orange, textcolor = color.white, size = size.large, style = label.style_label_up) // Update the `j` counter within the local block. j += 1

Note that:

- When a while loop uses count-based logic, it must explicitly manage the user-specified counter within the local block. In contrast, a for loop increments its counter automatically.
- The script declares the variable the while loop uses as a counter *outside* the loop’s scope, meaning its value is usable in additional calculations after the loop terminates.
- If this code did not increment the `j` variable within the while loop’s body, the value would *never* reach 10, meaning the loop would run *indefinitely* until causing a runtime error.

Because a while loop’s execution depends on its condition remaining `true`, and the condition might not change on a specific iteration, the *precise* number of expected iterations might not be knowable *before* the loop begins. Therefore, while loops are often helpful in scenarios where the exact loop boundaries are *unknown*.

The script below tracks when the chart’s close crosses outside Keltner Channels with a user-specified length and channel width. When the price crosses outside the current bar’s channel, the script draws a box highlighting all the previous *consecutive* bars with close values within that price window. The script uses a while loop to analyze past bars’ prices and incrementally adjust the left side of each new box until the drawing covers all the latest consecutive bars in the current range:

image

```
//@version=6 indicator("`while` loop demo", "Price window boxes",
true) //@variable The length of the channel. int lengthInput =
input.int(20, "Channel length", 1, 4999) //@variable The width
multiplier of the channel. float widthInput = input.float(2.0, "Width
multiplier", 0) //@variable The `lengthInput`-bar EMA of `close` prices.
float ma = ta.ema(close, lengthInput) //@variable The `lengthInput`-
bar ATR, multiplied by the `widthInput`. float atr = ta.atr(lengthInput) *
widthInput //@variable The lower bound of the channel. float
channelLow = ma - atr //@variable The upper bound of the channel.
float channelHigh = ma + atr //@variable Is `true` when the `close`
price is outside the current channel range, `false` otherwise. bool
```

priceOutsideChannel = close < channelLow or close > channelHigh // Check if the `close` crossed outside the channel range, then analyze the past bars within the current range. if priceOutsideChannel and not priceOutsideChannel[1] //@variable A box that highlights consecutive past bars within the current channel's price window. box windowBox = box.new(bar_index, channelHigh, bar_index, channelLow, border_width = 2, bgcolor = color.new(color.gray, 85)) //@variable The lookback index for box adjustment. The `while` loop increments this value on each iteration. int i = 1 // Use a `while` loop to look backward through close` prices. The loop iterates as long as the past `close` // from `i` bars ago is between the current bar's `channelLow` and `channelHigh`. while close[i] >= channelLow and close[i] <= channelHigh // Adjust the left side of the box. windowBox.set_left(bar_index - i) // Add 1 to the `i` value to check the `close` from the next bar back on the next iteration. i += 1 // Plot the `channelLow` and `channelHigh` for visual reference. plot(channelLow, "Channel low") plot(channelHigh, "Channel high")

Note that:

- The left and right edges of boxes sit within the horizontal *center* of their respective bars, meaning that each drawing spans from the middle of the first consecutive bar to the middle of the last bar within each window.
- This script uses the `i` variable as a history-referencing index within the *conditional expression* the while loop checks on each iteration. The variable **does not** behave as a loop counter, as the iteration boundaries are **unknown**. The loop executes its local block repeatedly until the condition becomes `false`.

`for...in` loops

The for...in loop statement creates a *collection-controlled* loop, which uses the *contents* of a collection to control its iterations. This loop structure is often the preferred approach for looping through arrays, matrices, and maps.

A for...in loop traverses a collection *in order*, retrieving one of its stored items on each iteration. Therefore, the loop’s boundaries depend directly on the number of *items* (array *elements*, matrix *rows*, or map *key-value pairs*).

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

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

Where `item` is a *variable* that holds sequential values or references from the specified `collection_id`. The variable starts with the collection's *first item* and takes on successive items in order after each iteration. This form is convenient when a script must access values from an array or matrix iteratively but does not require the item's *index* in its calculations.

The *second form* has a slightly different syntax that includes a tuple in its *header*:

```
[variables = | :=] for [index, item] in collection_id
    statements | continue | break
    return_expression
```

Where `index` is a variable that contains the *index* or *key* of the retrieved `item`. This form is convenient when a task requires using a collection's items *and* their indices in iterative calculations. This form of the for...in loop is *required* when directly iterating through the contents of a map. See this section below for more information.

Refer to the Common characteristics section above for detailed information about the `variables`, `statements`, `continue`, `break`, and `return_expression` parts of the loop's syntax.

The iterative behavior of a for...in loop depends on the *type* of collection the header specifies as the `collection_id`:

- When using an array in the header, the loop performs *element-wise* iteration, meaning the retrieved `item` on each iteration is one of the array's *elements*.

- When using a matrix in the header, the loop performs *row-wise* iteration, which means that each `item` represents a *row array*.
- When using a map in the header, the loop performs *pair-wise* iteration, which retrieves a *key* and corresponding *value* on each iteration.

Looping through arrays

Pine scripts can iterate over the elements of arrays using any loop structure. However, the for...in loop is typically the most convenient because it automatically verifies the size of an array when controlling iterations. With other loop structures, programmers must carefully set the header's boundaries or conditions to *prevent* the loop from attempting to access an element at a *nonexistent* index.

For example, a for loop can access an array's elements using the counter variable as the lookup index in functions such as array.get(). However, programmers must ensure the counter always represents a *valid index* to prevent out-of-bounds errors. Additionally, if an array might be *empty*, programmers must set conditions to prevent the loop's execution entirely.

The code below shows a for loop whose counter boundaries depend on the number of elements in an array. If the array is empty, containing zero elements, the header's final counter value is na, which *prevents* iteration. Otherwise, the final value is *one less* than the array's size (i.e., the index of the last element):

```
for index = 0 to (array.size(myArray) == 0 ? na : array.size(myArray) - 1) element = array.get(myArray, index)
```

In contrast, a for...in loop automatically validates an array's size and *directly* accesses its elements, providing a more convenient solution than a traditional for loop. The line below achieves the *same effect* as the code above without requiring the programmer to define boundaries explicitly or use the array.get() function to access each element:

```
for element in myArray
```

The following example examines bars on a lower timeframe to gauge the strength of *intrabar* trends within each chart bar. The script uses a request.security_lower_tf() call to retrieve an array of intrabar hl2 prices from a calculated `lowerTimeframe`. Then, it uses a for...in loop to access each `price` within the `intrabarPrices` array and compare the value to the current close to calculate the bar’s `strength`. The script plots the `strength` as columns in a separate pane:

image

//@version=6 indicator("`for element in array` demo", "Intrabar strength") //@variable A valid timeframe closest to one-tenth of the current chart's timeframe, "1" if the timeframe is too small. var string lowerTimeframe = timeframe.from_seconds(math.max(int(timeframe.in_seconds() / 10), 60)) //@variable An array of intrabar `hl2` prices calculated from the `lowerTimeframe`. array<float> intrabarPrices = request.security_lower_tf("", lowerTimeframe, hl2) //@variable The excess trend strength of `intrabarPrices`. float strength = 0.0 // Loop directly through the `intrabarPrices` array. Each iteration's `price` represents an array element. for price in intrabarPrices // Subtract 1 from the `strength` if the retrieved `price` is above the current bar's `close` price. if price > close strength -= 1 // Add 1 to the `strength` if the retrieved `price` is below the current bar's `close` price. else if price < close strength += 1 //@variable Is `color.teal` when the `strength` is positive, `color.maroon` otherwise. color strengthColor = strength > 0 ? color.teal : color.maroon // Plot the `strength` as columns colored by the `strengthColor`. plot(strength, "Intrabar strength", strengthColor, 1, plot.style_columns)

The second form of the for...in loop is a convenient solution when a script’s calculations require accessing each element *and* corresponding index within an array:

for [index, element] in myArray

For example, suppose we want to display a *numerated* list of array elements within a label while excluding values at specific indices. We can use the second form of the for...in loop structure to accomplish this task. The simple script below declares a `stringArray` variable that references an array of predefined “string” values. On the last historical bar, the script uses a for...in loop to access each `index` and `element` in the `stringArray` to construct the `labelText`, which it uses in a label.new() call after the loop ends:

image

```
//@version=6 indicator("`for [index, item] in array` demo", "Array
numerated output") //@variable An array of "string" values to display
as a numerated list. var array<string> stringArray = array.from("First",
"Second", "Third", "Before Last", "Last") if
barstate.islastconfirmedhistory //@variable A "string" modified within a
loop to display within the `label`. string labelText = "Array values: \n" //
Loop through the `stringArray`, accessing each `index` and
corresponding `element`. for [index, element] in stringArray // Skip the
third `element` (at `index == 2`) in the `labelText`. Include an
"ELEMENT SKIPPED" message instead. if index == 2 labelText += "--
ELEMENT SKIPPED -- \n" continue labelText += str.tostring(index + 1) +
": " + element + "\n" // Display the `labelText` within a `label`.
label.new( bar_index, 0, labelText, textcolor = color.white, size =
size.huge, style = label.style_label_center, textalign = text.align_left )
```

Note that:

- This example adds 1 to the `index` in the str.tostring() call to start the numerated list with a value of `"1"`, because array indices always begins at 0.
- On the *third* loop iteration, when `index == 2`, the script adds an `"-- ELEMENT SKIPPED --"` message to the `labelText` instead of the retrieved `element` and uses the `continue` keyword to skip the remainder of the iteration. See this section above to learn more about loop keywords.

Let’s explore an advanced example demonstrating the utility of for...in loops. The following indicator draws a fixed number of horizontal lines at pivot high values calculated from a ta.pivothigh() call, and it analyzes the lines within a loop to determine which ones represent active (*uncrossed*) pivots.

Each time the script detects a new pivot high point, it creates a new line, *inserts* that line at the beginning of the `pivotLines` array, then removes the oldest element and deletes its ID. The script accesses each line within the array using a for...in loop, analyzing and modifying the properties of the line referenced on each iteration. When the current high crosses above the `pivotLine`, the script changes its style to signify that it is no longer an active level. Otherwise, it extends the line’s x2 coordinate and uses its price to calculate the average *active* pivot value. The script also plots each pivot high value and the average active pivot value on the chart:

image

//@version=6 indicator("`for...in` loop with arrays demo", "Active high pivots", true, max_lines_count = 500) //@variable The number of bars required on the left and right to confirm a pivot point. int pivotBarsInput = input.int(5, "Pivot leg length", 1) //@variable The number of recent pivot lines to analyze. Controls the size of the `pivotLines` array. int maxRecentLines = input.int(20, "Maximum recent lines", 1, 500) //@variable An array that acts as a queue holding the most recent pivot high lines. var array<line> pivotLines = array.new<line>(maxRecentLines) //@variable The pivot high price, or `na` if no pivot is found. float highPivotPrice = ta.pivothigh(pivotBarsInput, pivotBarsInput) if not na(highPivotPrice) // @variable The `chart.point` for the start of the line. Does not contain `time` information. firstPoint = chart.point.from_index(bar_index - pivotBarsInput, highPivotPrice) //@variable The `chart.point` for the end of each line. Does not contain `time` information. secondPoint = chart.point.from_index(bar_index, highPivotPrice) //@variable A horizontal line at the new pivot level. line hiPivotLine = line.new(firstPoint, secondPoint, width = 2, color = color.green) // Insert the `hiPivotLine` at the beginning of the `pivotLines` array. pivotLines.unshift(hiPivotLine) // Remove the oldest line from the array and delete its ID. line.delete(pivotLines.pop()) //@variable The sum of active pivot prices. float activePivotSum = 0.0 //@variable The number of active pivot high levels. int numActivePivots = 0 // Loop through the `pivotLines` array, directly accessing each `pivotLine` element. for pivotLine in pivotLines //@variable The `x2` coordinate of the `pivotline`. int lineEnd = pivotLine.get_x2() // Move to the next `pivotline` in the array if the current line is inactive. if pivotLine.get_x2() < bar_index - 1 continue //@variable The price value of the `pivotLine`. float pivotPrice = pivotLine.get_price(bar_index) // Change the style of the `pivotLine` and stop extending its display if the `high` is above the `pivotPrice`. if high > pivotPrice pivotLine.set_color(color.maroon) pivotLine.set_style(line.style_dotted) pivotLine.set_width(1) continue // Extend the `pivotLine` and add the `pivotPrice` to the `activePivotSum` when the loop allows a full iteration. pivotLine.set_x2(bar_index) activePivotSum += pivotPrice numActivePivots += 1 //@variable The average active pivot high value. float avgActivePivot = activePivotSum / numActivePivots // Plot crosses at the `highPivotPrice`, offset backward by the `pivotBarsInput`. plot(highPivotPrice, "High pivot marker", color.green, 3, plot.style_cross, offset = -pivotBarsInput) // Plot the `avgActivePivot` as a line with breaks. plot(avgActivePivot, "Avg. active pivot", color.orange, 3, plot.style_linebr)

Note that:

- The loop in this example executes on *every bar* because it has to compare active pivot line prices with the current high value, then use the remaining active prices to calculate the bar’s `avgActivePivot` value.
- Pine Script features several ways to calculate averages, many of which *do not* require a loop. However, a loop is necessary in this example because the script uses information only available on the **current bar** to determine which prices contribute toward the average.
- The *first* form of the for...in loop is the most convenient option in this example because we need direct access to the lines referenced within the `pivotLines` array, but we do not need the corresponding *index* values.

Looping through matrices

Pine scripts can iterate over the contents of a matrix in several different ways. Unlike arrays, matrices use *two* indices to reference their elements because they store data in a *rectangular* format. The first index refers to *rows*, and the second refers to *columns*. If a programmer opts to use for or while loops to iterate through matrices instead of using for…in, they must carefully define the loop boundaries or conditions to avoid out-of-bounds errors.

This code block shows a for loop that performs *row-wise* iteration, looping through each *row index* in a matrix and using the value in a matrix.row() call to retrieve a row array. If the matrix is empty, the loop statement uses a final loop counter value of na to *prevent* iteration. Otherwise, the final counter is the last row index, which is *one less* than the value returned by matrix.rows():

```
for rowIndex = 0 to (myMatrix.rows() == 0 ? na : myMatrix.rows() - 1)
rowArray = myMatrix.row(rowIndex)
```

Note that:

- If we replace the matrix.rows() and matrix.row() calls with matrix.columns() and matrix.col(), the loop performs *column-wise* iteration instead.

The for…in loop statement is the more convenient approach to loop over and access the rows of a matrix in order, as it automatically validates the number of rows and retrieves an array of the current row's elements on each iteration:

```
for rowArray in myMatrix
```

When a script's calculations require access to each row from a matrix and its corresponding *index*, programmers can use the second form of the for…in loop:

```
for [rowIndex, rowArray] in myMatrix
```

Note that:

- The for…in loop only performs **row-wise** iteration on matrices. To *emulate* column-wise iteration, programmers can use a for…in loop on a transposed copy.

The following example creates a custom string representing the rows of a matrix with extra information. When the script executes on the last historical bar, it creates a 3x3 matrix populated with values from math.random() calls. Using the first form of the for...in loop, the script iterates through each row in the matrix to create a “string” value representing the row’s contents, its average, and whether the average is above 0.5. Before the end of each iteration, the script concatenates the constructed string with the `labelText` value. After the loop ends, the script creates a label to display the `labelText` variable’s final value:

image

```
//@version=6 indicator("`for row in matrix` demo", "Custom matrix label") //@variable Generates a random value between 0 and 1, rounded to 4 decimal places. rand() => math.round(math.random(), 4) if barstate.islastconfirmedhistory //@variable A matrix of randomized values to format and display in a `label`. matrix<float> randomMatrix = matrix.new<float>() // Add a row of 9 randomized values and reshape the matrix to 3x3. randomMatrix.add_row( 0, array.from(rand(), rand(), rand(), rand(), rand(), rand(), rand(), rand(), rand()) ) randomMatrix.reshape(3, 3) //@variable A custom "string" representation of `randomMatrix` information. Modified within a loop. string labelText = "Matrix rows: \n" // Loop through the rows in the `randomMatrix`. for row in randomMatrix //@variable The average element value within the `row`. float rowAvg = row.avg() //@variable An upward arrow when the `rowAvg` is above 0.5, a downward arrow otherwise. string directionChar = rowAvg > 0.5 ? "⬆" : "⬇" // Add a "string" representing the `row` array, its average, and the `directionChar` to the `labelText`. labelText += str.format("Row: {0} Avg: {1} {2}\n", row, rowAvg, directionChar) // Draw a `label` displaying the `labelText` on the current bar. label.new( bar_index, 0, labelText, color = color.purple, textcolor = color.white, size = size.huge, style = label.style_label_center, textalign = text.align_left )
```

Working with matrices often entails iteratively accessing their *elements*, not just their rows and columns, typically using *nested loops*. For example, this code block uses an outer for loop to iterate over row indices. The inner for loop iterates over column indices on *each* outer loop iteration and calls matrix.get() to access an element:

```
for rowIndex = 0 to (myMatrix.rows() == 0 ? na : myMatrix.rows() - 1) for columnIndex = 0 to myMatrix.columns() - 1 element = myMatrix.get(rowIndex, columnIndex)
```

Alternatively, a more convenient approach for this type of task is to use nested for...in loops. The outer for...in loop in this code block retrieves each row array in a matrix, and the inner for...in statement loops through that array:

```
for rowArray in myMatrix for element in rowArray
```

The script below creates a 3x2 matrix, then accesses and modifies its elements within nested for...in loops. Both loops use the second form of the for...in statement to retrieve index values and corresponding items. The outer loop accesses a row index and row array from the matrix. The inner loop accesses each index and respective element from that array.

Within the nested loop’s iterations, the script converts each `element` to a “string” and initializes a table cell at the `rowIndex` row and `colIndex` column. Then, it uses the loop header variables within matrix.set() to update the matrix element. After the outer loop terminates, the script displays a “string” representation of the *updated* matrix within a label:

image

```
//@version=6 indicator("Nested `for...in` loops on matrices demo") if barstate.islastconfirmedhistory //@variable A matrix containing numbers to display. matrix<float> displayNumbers = matrix.new<float>() // Populate the `displayNumbers` matrix and reshape to 3x2. displayNumbers.add_row(0, array.from(1, 2, 3, 4, 5, 6)) displayNumbers.reshape(3, 2) //@variable A table that displays the elements of the `displayNumbers` before modification. table displayTable = table.new( position = position.middle_center, columns = displayNumbers.columns(), rows = displayNumbers.rows(), bgcolor = color.purple, border_color = color.white, border_width = 2 ) // Loop through the `displayNumbers`, retrieving the `rowIndex` and the
```

```
current `row`. for [rowIndex, row] in displayNumbers // Loop through
the current `row` on each outer loop iteration to retrieve the `colIndex`
and `element`. for [colIndex, element] in row // Initialize a table cell at
the `rowIndex` row and `colIndex` column displaying the current
`element`. displayTable.cell(column = colIndex, row = rowIndex, text =
str.tostring(element), text_color = color.white, text_size = size.huge ) //
Update the `displayNumbers` value at the `rowIndex` and `colIndex`.
displayNumbers.set(rowIndex, colIndex,
math.round(math.exp(element), 3)) // Draw a `label` to display a
"string" representation of the updated `displayNumbers` matrix.
label.new( x = bar_index, y = 0, text = "Matrix now modified: \n" +
str.tostring(displayNumbers), color = color.orange, textcolor =
color.white, size = size.huge, style = label.style_label_up )
```

Looping through maps

The for...in loop statement is the primary, most convenient approach for iterating over the data within Pine Script maps.

Unlike arrays and matrices, maps are *unordered collections* that store data in *key-value pairs*. Rather than traversing an internal lookup index, a script references the *keys* from the pairs within a map to access its *values*. Therefore, when looping through a map, scripts must perform *pair-wise* iteration, which entails retrieving key-value pairs across iterations rather than indexed elements or rows.

Note that:

- Although maps are unordered collections, Pine Script internally tracks the *insertion order* of their key-value pairs.

One way to access the data from a map is to use the map.keys() function, which returns an array containing all the *keys* from the map, sorted in their insertion order. A script can use the for...in structure to loop through the array of keys and call map.get() to retrieve corresponding values:

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

However, the more convenient, *recommended* approach is to loop through a map directly *without* creating new arrays. To loop through a map directly, use the second form of the for...in loop statement. Using this loop with a map creates a tuple containing a *key* and respective *value* on each iteration. As when looping through a map.keys() array, this *direct* for...in loop iterates through a map's contents in their insertion order:

for [key, value] in myMap

Note that:

- The second form of the for...in loop is the **only** way to iterate *directly* through a map. A script cannot directly loop through this collection type without retrieving a key and value on each iteration.

Let's consider a simple example demonstrating how a for...in loop works on a map. When the script below executes on the last historical bar, it declares a `simpleMap` variable to reference a map of "string" keys and "float" values. The script uses map.put() to insert the keys from the newKeys array into the collection with corresponding values from math.random() calls. Then, it uses a for...in loop to iterate through the key-value pairs from the map and construct the `displayText` string. After the loop ends, the script uses a label to visualize the string:

image

//@version=6 indicator("Looping through map demo") if barstate.islastconfirmedhistory //@variable A map of "string" keys and "float" values to render within a `label`. map<string, float> simpleMap = map.new<string, float>() //@variable An array of "string" values representing the keys to put into the map. array<string> newKeys = array.from("A", "B", "C", "D", "E") // Put key-value pairs into the `simpleMap`. for key in newKeys simpleMap.put(key, math.random(1, 20)) //@variable A "string" representation of the `simpleMap` contents. Modified within a loop. string displayText = "simpleMap content: \n " // Loop through each key-value pair within the `simpleMap`. for [key, value] in simpleMap // Add a "string" representation of the pair to the `displayText`. displayText += key + ": " + str.tostring(value, "#.##") + "\n " // Draw a `label` showing the `displayText` on the current bar. label.new(x = bar_index, y = 0, text = displayText, color = color.green, textcolor = color.white, size = size.huge, textalign = text.align_left, style = label.style_label_center)

Note that:

- This script uses both forms of the for...in loop statement. The first loop iterates through the “string” elements of the newKeys array to put key-value pairs into the map referenced by `simpleMap`, and the second iterates directly through the map’s key-value pairs to construct the custom string.

Previous

Next

Built-ins

Pine Script® v6 Documentation

User Manual / Language / Built-ins

Built-ins

Introduction

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

On this page, we present an overview of some of Pine'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 v6 Reference Manual. It is called a "Reference Manual" because it is the definitive reference on the Pine Script language. It is an essential tool that will accompany you anytime you code in Pine, whether you are a beginner or an expert. If you are learning your first programming language, make the Reference Manual your friend. Ignoring it will make your programming experience with Pine Script difficult and frustrating — as it would with any other programming language.

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

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

- The ta.tr variable returns the "True Range" of the current bar. The ta.tr(true) function call also returns the "True Range", but when

the previous close value which is normally needed to calculate it is na, it calculates using `high - low` instead.
- The time variable gives the time at the open of the current bar. The time(timeframe) function returns the time of the bar's open from the `timeframe` specified, even if the chart's timeframe is different. The time(timeframe, session) function returns the time of the bar's open from the `timeframe` specified, but only if it is within the `session` time. The time(timeframe, session, timezone) function returns the time of the bar's open from the `timeframe` specified, but only if it is within the `session` time in the specified `timezone`.

Built-in variables

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

- Price- and volume-related variables: open, high, low, close, hl2, hlc3, ohlc4, and volume.
- Symbol-related information in the `syminfo` namespace: syminfo.basecurrency, syminfo.currency, syminfo.description, syminfo.main_tickerid, syminfo.mincontract, 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, timeframe.main_period, and timeframe.period.
- Bar states in the `barstate` namespace (see the Bar states page): barstate.isconfirmed, barstate.isfirst, barstate.ishistory, barstate.islast, barstate.islastconfirmedhistory, barstate.isnew, and barstate.isrealtime.
- Strategy-related information in the `strategy` namespace: strategy.equity, strategy.initial_capital, strategy.grossloss, strategy.grossprofit, strategy.wintrades, strategy.losstrades,

strategy.position_size, strategy.position_avg_price, strategy.wintrades, etc.

Built-in functions

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

- Math-related functions in the `math` namespace: math.abs(), math.log(), math.max(), math.random(), math.round_to_mintick(), etc.
- Technical indicators in the `ta` namespace: ta.sma(), ta.ema(), ta.macd(), ta.rsi(), ta.supertrend(), etc.
- Support functions often used to calculate technical indicators in the `ta` namespace: ta.barssince(), ta.crossover(), ta.highest(), etc.
- Functions to request data from other symbols or timeframes in the `request` namespace: request.dividends(), request.earnings(), request.financial(), request.quandl(), request.security(), request.splits().
- Functions to manipulate strings in the `str` namespace: str.format(), str.length(), str.tonumber(), str.tostring(), etc.
- Functions used to define the input values that script users can modify in the script's "Settings/Inputs" tab, in the `input` namespace: input(), input.color(), input.int(), input.session(), input.symbol(), etc.
- Functions used to manipulate colors in the `color` namespace: color.from_gradient(), color.rgb(), color.new(), 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 v6 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:

image

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.

Previous

Next

User-defined functions

Pine Script® v6 Documentation

User Manual / Language / User-defined functions

User-defined functions

Introduction

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

You can write your functions in two ways:

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

Functions can be located in two places:

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

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

- They cannot be embedded. All functions are defined in the script's global scope.

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

Single-line functions

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

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

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

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

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

Here is an example:

f(x, y) => x + y

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

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

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

Multi-line functions

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

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

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

where:

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

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

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

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

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

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

Scopes in the script

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

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

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

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

Functions that return multiple results

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

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

Special syntax is required for calling such functions:

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

Limitations

User-defined functions can contain calls to most built-in functions within their local scopes. However, calls to any of the following functions are **not** allowed in a function's scope: barcolor(), bgcolor(), plot(), plotshape(), plotchar(), plotarrow(), plotcandle(), plotbar(), hline(), fill(), alertcondition(), indicator(), strategy(), or library().

Previous

Next

Objects

Pine Script® v6 Documentation

User Manual / Language / Objects

ADVANCED

Objects

Introduction

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

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

Creating objects

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

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

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

Note that:

- We use the type keyword to declare the creation of a UDT.
- We name our new UDT `pivotPoint`.
- After the first line, we create a local block containing the type and name of each field.
- The x field will hold the x-coordinate of the pivot. It is declared as an “int” because it will hold either a timestamp or a bar index of “int” type.

- y is a “float” because it will hold the pivot’s price.
- `xloc` is a field that will specify the units of x: xloc.bar_index or xloc.bar_time. We set its default value to xloc.bar_time by using the = operator. When an object is created from that UDT, its `xloc` field will thus be set to that value.

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

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

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

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

Or the equivalent:

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

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

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

```
pivotPoint foundPoint = na
```

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

```
//@version=6 indicator("Pivot labels", overlay = true) int legsInput =
input(10) // Define the `pivotPoint` UDT. type pivotPoint int x float y
string xloc = xloc.bar_time // Detect high pivots. pivotHighPrice =
ta.pivothigh(legsInput, legsInput) if not na(pivotHighPrice) // A new high
pivot was found; display a label where it occurred `legsInput` bars
back. foundPoint = pivotPoint.new(time[legsInput], pivotHighPrice)
```

```
label.new( foundPoint.x, foundPoint.y, str.tostring(foundPoint.y, format.mintick), foundPoint.xloc, textcolor = color.white)
```

Take note of this line from the above example:

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

This could also be written using the following:

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

When using the var keyword while declaring a variable assigned to an object of a user-defined type, the keyword automatically applies to all the object’s fields:

```
//@version=6 indicator("Objects using `var` demo") //@type A custom type to hold index, price, and volume information. type BarInfo int index = bar_index float price = close float vol = volume //@variable A `BarInfo` instance whose fields persist through all iterations, starting from the first bar. var BarInfo firstBar = BarInfo.new() //@variable A `BarInfo` instance declared on every bar. BarInfo currentBar = BarInfo.new() // Plot the `index` fields of both instances to compare the difference. plot(firstBar.index) plot(currentBar.index)
```

It’s important to note that assigning an object to a variable that uses the varip keyword does *not* automatically allow the object’s fields to persist without rolling back on each *intrabar* update. One must apply the keyword to each desired field in the type declaration to achieve this behavior. For example:

```
//@version=6 indicator("Objects using `varip` fields demo") //@type A custom type that counts the bars and ticks in the script's execution. type Counter int bars = 0 varip int ticks = 0 //@variable A `Counter` object whose reference persists throughout all bars. var Counter counter = Counter.new() // Add 1 to the `bars` and `ticks` fields. The `ticks` field is not subject to rollback on unconfirmed bars. counter.bars += 1 counter.ticks += 1 // Plot both fields for comparison. plot(counter.bars, "Bar counter", color.blue, 3) plot(counter.ticks, "Tick counter", color.purple, 3)
```

Note that:

- We used the var keyword to specify that the `Counter` object assigned to the `counter` variable persists throughout the script’s execution.
- The `bars` field rolls back on realtime bars, whereas the `ticks` field does not since we included varip in its declaration.

Changing field values

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

This line of our previous example:

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

Could be written using the following:

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

Collecting objects

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

This example declares an empty array that will hold objects of a `pivotPoint` user-defined type:

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

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

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

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

image

//@version=6 indicator("Pivot Points High", overlay = true) int legsInput = input(10) // Define the `pivotPoint` UDT containing the time and price of pivots. type pivotPoint int openTime float level // Create an empty `pivotPoint` array. var pivotHighArray = array.new<pivotPoint>() // Detect new pivots (`na` is returned when no pivot is found). pivotHighPrice = ta.pivothigh(legsInput, legsInput) // Add a new `pivotPoint` object to the end of the array for each detected pivot. if not na(pivotHighPrice) // A new pivot is found; create a new object of `pivotPoint` type, setting its `openTime` and `level` fields. newPivot = pivotPoint.new(time[legsInput], pivotHighPrice) // Add the new pivot object to the array. array.push(pivotHighArray, newPivot) // On the last historical bar, draw pivot labels and connecting lines. if barstate.islastconfirmedhistory var pivotPoint previousPoint = na for eachPivot in pivotHighArray // Display a label at the pivot point. label.new(eachPivot.openTime, eachPivot.level, str.tostring(eachPivot.level, format.mintick), xloc.bar_time, textcolor = color.white) // Create a line between pivots. if not na(previousPoint) // Only create a line starting at the loop's second iteration because lines connect two pivots. line.new(previousPoint.openTime, previousPoint.level, eachPivot.openTime, eachPivot.level, xloc = xloc.bar_time) // Save the pivot for use in the next iteration. previousPoint := eachPivot

Copying objects

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

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

```
//@version=6 indicator("") type pivotPoint int x float y pivot1 =
pivotPoint.new() pivot1.x := 1000 pivot2 = pivot1 pivot2.x := 2000 //
Both plot the value 2000. plot(pivot1.x) plot(pivot2.x)
```

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

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

```
//@version=6 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=6 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=6 indicator("Deep Copy") type InfoLabel string info label lbl
method set(InfoLabel this, int x = na, int y = na, string info = na) => if
not na(x) this.lbl.set_x(x) if not na(y) this.lbl.set_y(y) if not na(info)
this.info := info this.lbl.set_text(this.info) method deepCopy(InfoLabel
this) => InfoLabel.new(this.info, this.lbl.copy()) var parent =
InfoLabel.new("", label.new(0, 0)) var deep = parent.deepCopy()
parent.set(bar_index, 0, "Parent") deep.set(bar_index, 1, "Deep Copy")
```

Shadowing

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

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

Previous

Next

Enums

Pine Script® v6 Documentation

User Manual / Language / Enums

ADVANCED

Enums

Introduction

Pine Script Enums, otherwise known as *enumerations*, *enumerated types*, or enum types, are unique data types with all possible values (*members*) explicitly defined by the programmer. They provide a human-readable, expressive way to declare distinct sets of *predefined values* that variables, conditional expressions, and collections can accept, allowing more strict control over the values used in a script's logic.

Declaring an enum

To declare an enum, use the enum keyword with the following syntax:

```
[export ]enum <enumName>
    <field_1>[ = <title_1>]
    <field_2>[ = <title_2>]
    ...
    <field_N>[ = <title_N>]
```

Each **field** in the enum represents a unique, *named member* (value) of the enum type. Users can specify optional "const string" **titles** for enum fields to add extra information about what their values represent. If the programmer does not specify a field's title, its title is the "string" representation of its name. Enum inputs display enum field titles within their dropdown menus in a script's "Settings/Inputs" tab. Scripts can also retrieve enum field titles using the str.tostring() function, allowing their use in additional calculations. See this section below for more information.

While the above syntax may look similar to the syntax for declaring user-defined types (UDTs), it’s crucial to understand that enum types and UDTs serve different purposes. Scripts use UDTs to create objects with “series” fields that can hold values of *any* specified type. In contrast, enums are distinct groups of “simple” fields representing the specific, *predefined values* of the same *unique* type that variables, expressions, and collections can accept.

For example, this code block declares a `Signal` enum with three fields: `buy`, `sell`, and `neutral`. Each field represents a distinct member (possible value) of the `Signal` enum type:

//@enum An enumeration of named values representing buy, sell, and neutral signal states. //@field buy Represents a "Buy signal" state. //@field sell Represents a "Sell signal" state. //@field neutral Represents a "neutral" state. enum Signal buy = "Buy signal" sell = "Sell signal" neutral

Note that:

- The `Signal` identifier represents the enum’s name, which signifies the *unique type* the fields belong to.
- We used the `//@enum` and `//@field` annotations to document the meaning of the enum and its fields.
- Unlike the buy and `sell` fields, the `neutral` field does not include a specified title. As such, its title is the “string” representation of its *name* (“neutral”).

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

enumName.fieldName

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

For instance, this line of code declares a `mySignal` variable whose value is the `neutral` member of the `Signal` enum. Any value assigned to this variable later must also be of the same enum type:

```
mySignal = Signal.neutral
```

Note that the above line does not require declaring the variable's *type* as `Signal` because the compiler can automatically infer that information from the assigned value. If we use na as the initial value instead, we must use `Signal` as the type keyword to specify that `mySignal` will accept a `Signal` member:

```
Signal mySignal = na
```

Using enums

Scripts can compare enum members with the == and != operators and use them in conditional structures, allowing the convenient creation of logical patterns with a reduced risk of unintended values or operations.

The following example declares an `OscType` enum with three fields representing different oscillator choices: `rsi`, `mfi`, and `cci`. The `calcOscillator()` function uses `OscType` members within a switch structure to determine which oscillator it calculates. The script calls this function using the value from an enum input as the `selection` argument and plots the resulting oscillator:

image

```
//@version=6 indicator("Using enums demo") //@enum An enumeration of oscillator choices. enum OscType rsi = "Relative Strength Index" mfi = "Money Flow Index" cci = "Commodity Channel Index" //@variable An enumerator (member) of the `OscType` enum. OscType oscInput = input.enum(OscType.rsi, "Oscillator type") //@function Calculates one of three oscillators based on a specified `selection`. //@param source The series of values to process. //@param length The number of bars in the calculation. //@param selection Determines which oscillator to calculate. calcOscillator(float source, simple int length, OscType selection) => result = switch selection OscType.rsi => ta.rsi(source, length) OscType.mfi => ta.mfi(source, length) OscType.cci =>
```

ta.cci(source, length) // Plot the value of a `calcOscillator()` call with `oscInput` as the `selection`. plot(calcOscillator(close, 20, oscInput))

Note that:

- The `selection` parameter of the `calcOscillator()` function can only take on one of *four* values: `OscType.rsi`, `OscType.mfi`, `OscType.cci`, or na.
- The “Oscillator type” input in the script’s “Settings/Inputs” tab displays all `OscType` field titles in its dropdown. See this section to learn more about enum inputs.

It’s crucial to note that each declared enum represents a *unique* type. Scripts **cannot** compare members of different enums or use such members in expressions requiring a specific enum type, even if the fields have identical names and titles.

In this example, we added an `OscType2` enum to the above script and changed the `oscInput` variable to use a member of that enum. The script now raises a *compilation error* because it can’t use a member of the `OscType2` enum as the `selection` argument in the `calcOscillator()` call:

//@version=6 indicator("Incompatible enums demo") //@enum An enumeration of oscillator choices. enum OscType rsi = "Relative Strength Index" mfi = "Money Flow Index" cci = "Commodity Channel Index" //@enum An enumeration of oscillator choices. Its fields DO NOT represent the same values those in the `OscType` enum. enum OscType2 rsi = "Relative Strength Index" mfi = "Money Flow Index" cci = "Commodity Channel Index" //@variable An enumerator (member) of the `OscType2` enum. OscType2 oscInput = input.enum(OscType2.rsi, "Oscillator type") //@function Calculates one of three oscillators based on a specified `selection`. //@param source The series of values to process. //@param length The number of bars in the calculation. // @param selection Determines which oscillator to calculate. calcOscillator(float source, simple int length, OscType selection) => result = switch selection OscType.rsi => ta.rsi(source, length) OscType.mfi => ta.mfi(source, length) OscType.cci => ta.cci(source, length) // Plot the value of a `calcOscillator()` call with `oscInput` as the `selection`. // Raises a compilation error because only members of `OscType` are allowed. plot(calcOscillator(close, 20, oscInput))

Utilizing field titles

The “string” titles of an enum’s fields allow programmers to add extra information to each member. These field titles appear within a dropdown in the script’s “Settings/Inputs” tab when calling the input.enum() function.

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

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

image

//@version=6 indicator("Utilizing field titles demo") //@enum An enumeration of cryptocurrency exchanges. All field titles are the same as the field names. enum Exchange BINANCE BITSTAMP BITFINEX COINBASE KRAKEN //@enum An enumeration of cryptocurrency pairs. All the field titles are the same as the field names. enum Pair BTCUSD ETHUSD SOLUSD XRPUSD //@variable An enumerator (member) of the `Exchange` enum. Exchange exchangeInput = input.enum(Exchange.BINANCE, "Exchange") //@variable An enumerator (member) of the `Pair` enum. Pair pairInput = input.enum(Pair.BTCUSD, "Pair") //@variable The exchange-symbol pair for the data request. simple string symbol = str.tostring(exchangeInput) + ":" + str.tostring(pairInput) // Plot the `close` value requested from the `symbol` context.

```
plot(request.security(symbol, timeframe.period, close), "Requested close", color.purple, 3)
```

Note that:

- None of the members of the `Exchange` or `Pair` enums have specified titles. Therefore, each field’s title is the “string” representation of its name, as shown by the script’s enum inputs.
- Calling the str.tostring() function on an enum field is the **only** way to retrieve its title for additional calculations. The str.format() and `log.*()` functions *cannot* accept enum members. To use a field’s title in a string formatting function, call str.tostring() on the field first, then pass the resulting “string” to the function.

Collecting enum members

Pine Script collections (arrays, matrices, and maps) can store enum members, allowing strict control over the values they can contain. To declare a collection of enum members, include the enum’s *name* in the collection’s type template.

For example, this code block creates an empty array to hold members of the `FooBar` enum. The only values this array can allow as elements are `FooBar.foo`, `FooBar.bar`, `FooBar.baz`, and na:

```
//@variable An enumeration of miscellaneous named members. enum FooBar foo bar baz //@variable An array that can only contain the following values: `FooBar.foo`, `FooBar.bar`, `FooBar.baz`, `na`. array<FooBar> fooBarArray = array.new<FooBar>()
```

Enums are particularly helpful when working with maps, as unlike other *non-fundamental* types, scripts can declare maps with *keys* of an enum type, enabling strict control over all possible keys allowed in their key-value pairs.

The following example uses a map with enum keys and “int” values to track and count signal states across chart bars. The script’s `Signal` enum contains five fields representing specific named states. The `signalCounters` map uses the `Signal` name as the *first keyword* in its type template to specify that it can only accept `Signal` members as keys.

The script uses a switch structure to calculate a `signalState` variable whose value is a member of the `Signal` enum, which it uses to determine the counter value to update in the `signalCounters` map. It constructs a “string” to represent the key-value pairs of the map and displays the result in a single-cell table on the last chart bar:

image

//@version=6 indicator("Collecting enum members demo", overlay = true) //@enum An enumeration of named signal states. enum Signal strongBuy = "Strong buy" buy = "Buy" neutral = "Neutral" sell = "Sell" strongSell = "Strong sell" //@variable The number of bars in the signal calculation. int lengthInput = input.int(50, "Length", 2) //@variable A map of `Signal.*` keys and "int" values counting the number of bars with each signal state. // Allowed keys: `Signal.strongBuy`, `Signal.buy`, `Signal.neutral`, `Signal.sell`, `Signal.strongSell`, `na`. var map<Signal, float> signalCounters = map.new<Signal, float>() // @variable A single-cell table displaying the key-value pairs of the `signalCounters` map. var table infoTable = table.new(position.top_right, 1, 1, chart.fg_color) if barstate.isfirst // Put `Signal.*`-"int" pairs into the `signalCounters` map to establish insertion order. signalCounters.put(Signal.strongBuy, 0) signalCounters.put(Signal.buy, 0) signalCounters.put(Signal.neutral, 0) signalCounters.put(Signal.sell, 0) signalCounters.put(Signal.strongSell, 0) // Initialize the `infoTable` cell. infoTable.cell(0, 0, text_color = chart.bg_color, text_halign = text.align_left, text_size = size.large) // Calculate the EMA and Percent rank of `source` data over `length` bars. float ema = ta.ema(close, lengthInput) float rank = ta.percentrank(close, lengthInput) //@variable A `Signal` member representing the current signal state based on `ema` and `rank` values. Signal signalState = switch close > ema => rank > 70 ? Signal.strongBuy : rank > 50 ? Signal.buy : Signal.neutral close < ema => rank < 30 ? Signal.strongSell : rank < 50 ? Signal.sell : Signal.neutral => Signal.neutral // Add 1 to the value in the `signalCounters` map associated with the `signalState` key. signalCounters.put(signalState, signalCounters.get(signalState) + 1) //

Update the `infoTable` cell's text using the keys and values from the `signalCounters` map on the last bar. if barstate.islast string tableText = "" for [state, count] in signalCounters tableText += str.tostring(state) + ": " + str.tostring(count) + "\n" infoTable.cell_set_text(0, 0, str.trim(tableText))

Note that:

- The `signalCounters` map can contain up to *six* key-value pairs, as the `Signal` enum has *five* predefined values, plus a possible value of na, and maps cannot contain *repetitive* keys.
- The script declares the `signalCounters` variable using the var keyword, signifying that the assigned map instance persists across executions.
- On the first chart bar, the script uses five map.put() calls to establish the *insertion order* of keys in the `signalCounters` map. See this section of the Maps page for more information.
- To minimize *resource usage*, the script declares the `infoTable` and initializes its cell on the *first bar*, then updates the cell's text on the *latest bar*. See this section of the Profiling and optimization page to learn more.

Shadowing

In contrast to user-defined types (UDTs), which can have names that *shadow* some built-in types or namespaces, enum types require *unique* names that do **not** match any built-in types or namespaces.

For example, this code declares four enums named `Syminfo`, `syminfo`, `polyline`, and `ta`. The last three all cause a compilation error because their names match built-in namespaces:

//@version=6 indicator("Shadowing demo") // Naming an enum "Syminfo" with a capital "S" works without an issue. enum Syminfo abcd // In contrast, the names "syminfo", "polyline", "ta", etc. cause a compilation error because they match // built-in namespaces: enum syminfo abcd enum polyline abcd enum ta abcd

Previous

Next

Methods

Pine Script® v6 Documentation

User Manual / Language / Methods
ADVANCED

Methods

Introduction

Pine Script methods are specialized functions associated with values of specific built-in types, user-defined types, or enum types. They behave the same as regular functions in most regards while offering a shorter, more convenient syntax. Users can access methods using *dot notation* syntax on variables of the associated type, similar to accessing the fields of a Pine Script object.

Built-in methods

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

When using these special types, the expressions:

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

and:

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

are equivalent. For example, rather than using:

array.get(id, index)

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

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

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

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

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

image

```
//@version=6 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.*()` function calls in the script with equivalent method calls:

//@version=6 indicator("Custom Sample BB", overlay = true) float sourceInput = input.source(close, "Source") int samplesInput = input.int(20, "Samples") int n = input.int(10, "Bars") float multiplier = input.float(2.0, "StdDev") var array<float> sourceArray = array.new<float>(samplesInput) var float sampleMean = na var float sampleDev = na // Identify if `n` bars have passed. if bar_index % n == 0 // Update the queue. sourceArray.push(sourceInput) sourceArray.shift() // Update the mean and standard deviaiton values. sampleMean := sourceArray.avg() sampleDev := sourceArray.stdev() * multiplier // Calculate band values. float highBand = sampleMean + sampleDev float lowBand = sampleMean - sampleDev plot(sampleMean, "Basis", color.orange) plot(highBand, "Upper", color.lime) plot(lowBand, "Lower", color.red)

Note that:

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

User-defined methods

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

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

```
[export] method <functionName>(<paramType> <paramName> [= <defaultV
    <functionBlock>
```

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

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

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

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

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

Note that:

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

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

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

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

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

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

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

// Identify if `n` bars have passed. bool newSample = bar_index % n == 0 // Update the queue and compute new BB values on each new sample. [sampleMean, highBand, lowBand] = sourceArray.maintainQueue(sourceInput, newSample).calcBB(multiplier, newSample)

Note that:

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

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

```
//@version=6 indicator("Custom Sample BB", overlay = true) float
sourceInput = input.source(close, "Source") int samplesInput =
input.int(20, "Samples") int n = input.int(10, "Bars") float multiplier =
input.float(2.0, "StdDev") var array<float> sourceArray =
array.new<float>(samplesInput) // @function Maintains a queue of the
size of `srcArray`. // It appends a `value` to the array and removes its
oldest element at position zero. // @param srcArray (array<float>) The
array where the queue is maintained. // @param value (float) The new
value to be added to the queue. // The queue's oldest value is also
removed, so its size is constant. // @param takeSample (bool) A new
`value` is only pushed into the queue if this is true. // @returns
(array<float>) `srcArray` object. method maintainQueue(array<float>
srcArray, float value, bool takeSample = true) => if takeSample
srcArray.push(value) srcArray.shift() srcArray // @function Computes
Bollinger Band values from an array of data. // @param srcArray
(array<float>) The array where the queue is maintained. // @param
multiplier (float) Standard deviaiton multiplier. // @param calcuate
(bool) The method will only calculate new values when this is true. //
@returns A tuple containing the basis, upper band, and lower band
respectively. method calcBB(array<float> srcArray, float mult, bool
calculate = true) => var float mean = na var float dev = na if
calculate // Compute the mean and standard deviation of the array.
mean := srcArray.avg() dev := srcArray.stdev() * mult [mean, mean +
dev, mean - dev] // Identify if `n` bars have passed. bool newSample =
bar_index % n == 0 // Update the queue and compute new BB values
on each new sample. [sampleMean, highBand, lowBand] =
sourceArray.maintainQueue(sourceInput,
newSample).calcBB(multiplier, newSample) plot(sampleMean, "Basis",
color.orange) plot(highBand, "Upper", color.lime) plot(lowBand,
"Lower", color.red)
```

Method overloading

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

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

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

// @function Identifies an object's type. // @param this Object to inspect. // @returns (string) A string representation of the type. method getType(int this) => na(this) ? "int(na)" : "int" method getType(float this) => na(this) ? "float(na)" : "float" method getType(bool this) => // "bool" values only have two states, \`true\` and \`false\`, but never \`na\`. "bool" method getType(color this) => na(this) ? "color(na)" : "color" method getType(string this) => na(this) ? "string(na)" : "string"

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

image

//@version=6 indicator("Type Inspection") // @function Identifies an object's type. // @param this Object to inspect. // @returns (string) A string representation of the type. method getType(int this) => na(this) ? "int(na)" : "int" method getType(float this) => na(this) ? "float(na)" : "float" method getType(bool this) => na(this) ? "bool(na)" : "bool" method getType(color this) => na(this) ? "color(na)" : "color" method getType(string this) => na(this) ? "string(na)" : "string" a = 1 b = 1.0 c = true d = color.white e = "1" // Inspect variables and format results. results = str.format("a: {0}\nb: {1}\nc: {2}\nd: {3}\ne: {4}", a.getType(), b.getType(), c.getType(), d.getType(), e.getType()) var label lbl = label.new(0, 0) lbl.set_x(bar_index) lbl.set_text(results)

Note that:

- The underlying type of each variable determines which overload of `getType()` the compiler will use.

- The method will append “(na)” to the output string when a variable is na to demarcate that it is empty.

Advanced example

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

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

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

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

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

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

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

Note that:

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

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

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

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

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

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

Note that:

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

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

image

//@version=6 indicator("Empirical Distribution", overlay = true) float sourceInput = input.source(close, "Source") int length = input.int(20, "Length") int n = input.int(20, "Steps") // @function Maintains a queue of the size of `srcArray`. // It appends a `value` to the array and removes its oldest element at position zero. // @param srcArray (array<float>) The array where the queue is maintained. // @param value (float) The new value to be added to the queue. // The queue's oldest value is also removed, so its size is constant. // @param takeSample (bool) A new `value` is only pushed into the queue if this is true. // @returns (array<float>) `srcArray` object. method maintainQueue(array<float> srcArray, float value, bool takeSample = true) => if takeSample srcArray.push(value) srcArray.shift() srcArray //

@function Replaces elements in a `srcArray` between `lowerBound` and `upperBound` with an `innerValue`, // and replaces elements outside the range with an `outerValue`. // @param srcArray (array<float>) Array to modify. // @param innerValue (float) Value to replace elements within the range with. // @param outerValue (float) Value to replace elements outside the range with. // @param lowerBound (float) Lowest value to replace with `innerValue`. // @param upperBound (float) Highest value to replace with `innerValue`. // @returns (array<float>) `srcArray` object. method fill(array<float> srcArray, float innerValue, float outerValue, float lowerBound, float upperBound) => for [i, element] in srcArray if (element >= lowerBound or na(lowerBound)) and (element <= upperBound or na(upperBound)) srcArray.set(i, innerValue) else srcArray.set(i, outerValue) srcArray // @function Estimates the empirical CDF of a `srcArray`. // @param srcArray (array<float>) Array to calculate on. // @param steps (int) Number of steps in the estimation. // @returns (array<float>) Array of estimated CDF ratios. method eCDF(array<float> srcArray, int steps) => float min = srcArray.min() float rng = srcArray.range() / steps array<float> cdfArray = array.new<float>() // Add averages of `srcArray` filtered by value region to the `cdfArray`. float val = min for i = 1 to steps val += rng cdfArray.push(srcArray.copy().fill(1.0, 0.0, min, val).avg()) cdfArray // @function Rescales the elements within a `srcArray` to the interval [0, 1]. // @param srcArray (array<float>) Array to normalize. // @returns (array<float>) Normalized copy of the `srcArray`. method featureScale(array<float> srcArray) => float min = srcArray.min() float rng = srcArray.range() array<float> scaledArray = array.new<float>() // Push normalized `element` values into the `scaledArray`. for element in srcArray scaledArray.push((element - min) / rng) scaledArray // @function Draws a label containing eCDF estimates in the format "{price}: {percent}%" // @param srcArray (array<float>) Array of source values. // @param cdfArray (array<float>) Array of CDF estimates. // @returns (void) makeLabel(array<float> srcArray, array<float> cdfArray) => float max = srcArray.max() float rng = srcArray.range() / cdfArray.size() string results = "" var label lbl = label.new(0, 0, "", style = label.style_label_left, text_font_family = font.family_monospace) // Add percentage strings to `results` starting from the `max`. cdfArray.reverse() for [i, element] in cdfArray results += str.format("{0}: {1}%\n", max - i * rng, element * 100) // Update `lbl` attributes. lbl.set_xy(bar_index + 1, srcArray.avg()) lbl.set_text(results) var array<float> sourceArray = array.new<float>(length) // Add background color for the last `length` bars. bgcolor(bar_index > last_bar_index - length ? color.new(color.orange, 80) : na) // Queue `sourceArray`, feature scale, then estimate the distribution over `n` steps. array<float> distArray = sourceArray.maintainQueue(sourceInput).featureScale().eCDF(n) // Draw label. makeLabel(sourceArray, distArray)

Previous

Next

Arrays

Pine Script® v6 Documentation

User Manual / Language / Arrays

ADVANCED

Arrays

Introduction

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

All elements in an array must be of the same built-in type, user-defined type, or enum type.

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

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

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>(), array.from(), or array.copy(). For `array.new<type>()` calls, the arguments of the function’s `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 reference returned by the `array.new\<float\>()` function call is assigned to the `prices` variable:

prices = array.new<float>(0)

The `initial_value` parameter of the `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 execution of the script on the first chart bar. Array variables declared using these keywords point to the same array instances until explicitly reassigned, allowing an array and its element references to persist across bars.

When declaring an array variable using these keywords and pushing a new value to the end of the referenced array on each bar, the array will grow by one on each bar and be of size `bar_index + 1` (bar_index starts at zero) by the time the script executes on the last bar, as this code demonstrates:

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

The same code without the var keyword would re-declare the array on each bar. In this case, after execution of the array.push() call, the array.size() *method* call (`a.size()`) would return a value of 1.

Reading and writing array elements

Scripts can write values to existing individual array elements using array.set(), and read using array.get(). 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() 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:

image

//@version=6 indicator("Distance from high", "", true) lookbackInput = input.int(100) FILL_COLOR = color.green // Declare array and set its values on the first bar only. var fillColors = array.new<color>(5) if barstate.isfirst // Initialize the array elements with progressively lighter shades of the fill color. fillColors.set(0, color.new(FILL_COLOR, 70)) fillColors.set(1, color.new(FILL_COLOR, 75)) fillColors.set(2, color.new(FILL_COLOR, 80)) fillColors.set(3, color.new(FILL_COLOR, 85)) fillColors.set(4, color.new(FILL_COLOR, 90)) // Find the offset to highest high. Change its sign because the function returns a negative value. lastHiBar = - ta.highestbars(high, lookbackInput) // Convert the offset to an array index, capping it to 4 to avoid a runtime error. // The index used by `array.get()` will be the equivalent of `floor(fillNo)`. fillNo = math.min(lastHiBar / (lookbackInput / 5), 4) // Set background to a progressively lighter fill with increasing distance from location of highest high. bgcolor(array.get(fillColors, fillNo)) // Plot key values to the Data Window for debugging. plotchar(lastHiBar, "lastHiBar", "", location.top, size = size.tiny) plotchar(fillNo, "fillNo", "", location.top, size = size.tiny)

Another technique for initializing the elements in an array is to create an *empty array* (an array with no elements), then use array.push() to append **new** elements to the end of the array, increasing the size of the array by one on each call. The following code is functionally identical to the initialization section from the preceding script:

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

This code is equivalent to the one above, but it uses array.unshift() to insert new elements at the *beginning* of the `fillColors` array:

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

We can also use array.from() to create the same `fillColors` array with a single function call:

```
//@version=6 indicator("Using `var`") FILL_COLOR = color.green var
array<color> fillColors = array.from( color.new(FILL_COLOR, 70),
color.new(FILL_COLOR, 75), color.new(FILL_COLOR, 80),
color.new(FILL_COLOR, 85), color.new(FILL_COLOR, 90) ) // Cycle
background through the array's colors. bgcolor(array.get(fillColors,
bar_index % (fillColors.size())))
```

The array.fill() 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 the array.fill() function's last parameter, `index_to`, must have a value one greater than the last index the function will fill. The remaining elements will hold na values, as the array.new<type>() function call does not contain an `initial_value` argument.

Looping through array elements

When looping through an array's element indices and the array's size is unknown, one can use the array.size() function to get the maximum index value. For example:

```
//@version=6 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=6 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=6 indicator("`while` loop", overlay = true) array<float> a = request.security_lower_tf(syminfo.tickerid, "1", close) string labelText = "" int i = 0 while i < array.size(a) labelText += str.tostring(array.get(a, i)) + "\n" i += 1 label.new(bar_index, high, text = labelText)
```

Scope

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

image

```
//@version=6 indicator("Bands", "", true) //@variable The distance ratio between plotted price levels. factorInput = 1 + (input.float(-2., "Step %") / 100) //@variable A single-value array holding the lowest `ohlc4`
```

```
value within a 50 bar window from 10 bars back. level =
array.new<float>(1, ta.lowest(ohlc4, 50)[10]) nextLevel(val) =>
newLevel = level.get(0) * val // Write new level to the global `level`
array so we can use it as the base in the next function call. level.set(0,
newLevel) newLevel plot(nextLevel(1)) plot(nextLevel(factorInput))
plot(nextLevel(factorInput)) plot(nextLevel(factorInput))
```

History referencing

The 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 an array.get() method call 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:

image

```
//@version=6 indicator("History referencing") //@variable A single-
value array declared on each bar. a = array.new<float>(1) // Set the
value of the only element in `a` to `close`. array.set(a, 0, close) //
@variable The array instance assigned to `a` on the previous bar.
previous = a[1] previousClose1 = na(previous) ? na : previous.get(0)
previousClose2 = close[1] plot(previousClose1, "previousClose1",
color.gray, 6) plot(previousClose2, "previousClose2", color.white, 2)
```

Inserting and removing array elements

Inserting

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

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

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

image

```
//@version=6 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=6 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:

image

//@version=6 indicator("Lows from new highs", "", true) var lows = array.new<float>(0) flushLows = false //@function Removes the last element from the `id` 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:

image

//@version=6 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 `pivotLegsInput` bars behind the current `bar_index`. 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))

Negative indexing

The array.get(), array.set(), array.insert(), and array.remove() functions support *negative indexing*, which references elements starting from the end of the array. An index of `-1` refers to the last element in the array, an index of `-2` refers to the second to last element, and so on.

When using a *positive* index, functions traverse the array *forwards* from the beginning of the array (*first to last* element). The first element's index is `0`, and the last element's index is `array.size() - 1`. When using a *negative* index, functions traverse the array *backwards* from the end of the array (*last to first* element). The last element's index is `-1`, and the first element's index is `–array.size()`:

array<string> myArray = array.from("first", "second", "third", "fourth", "last") // Positive indexing: Indexes forwards from the beginning of the array. myArray.get(0) // Returns "first" element myArray.get(myArray.size() - 1) // Returns "last" element myArray.get(4) // Returns "last" element // Negative indexing: Indexes backwards from the end of the array. myArray.get(-1) // Returns "last" element myArray.get(-myArray.size()) // Returns "first" element myArray.get(-5) // Returns "first" element

Like positive indexing, negative indexing is bound by the size of the array. For example, functions operating on an array of 5 elements only accept indices of 0 to 4 (first to last element) or -1 to -5 (last to first element). Any other indices are out of bounds and will raise a runtime error.

We can use negative indices to retrieve, update, add, and remove array elements. This simple script creates an "int" `countingArray` and calls the array.get(), array.set(), array.insert(), and array.remove() functions to perform various array operations using negative indices. It displays each array operation and its corresponding result using a table:

image

```
//@version=6 indicator("Negative indexing demo", overlay = false) //
@variable A `table` that displays various array operations and their
results. var table displayTable = table.new( position.middle_center, 2,
15, bgcolor = color.white, frame_color = color.black, frame_width = 1,
border_width = 1 ) //@function Initializes a `displayTable` row to output
a "string" of an `arrayOperation` and the `operationResult`.
displayRow(int rowID, string arrayOperation, operationResult) => //
@variable Is white if the `rowID` is even, light blue otherwise. Used to
set alternating table row colors. color rowColor = rowID % 2 == 0 ?
color.white : color.rgb(33, 149, 243, 75) // Display the `arrayOperation`
in the row's first cell. displayTable.cell(0, rowID, arrayOperation,
text_color = color.black, text_halign = text.align_left, bgcolor =
rowColor, text_font_family = font.family_monospace ) // Display the
`operationResult` in the row's second cell. displayTable.cell(1, rowID,
str.tostring(operationResult), text_color = color.black, text_halign =
text.align_right, bgcolor = rowColor ) if
barstate.islastconfirmedhistory //@variable Array of "int" numbers.
Holds six multiples of 10, counting from 10 to 60. array<int>
countingArray = array.from(10, 20, 30, 40, 50, 60) // Initialize the
table's header cells. displayTable.cell(0, 0, "ARRAY OPERATION")
displayTable.cell(1, 0, "RESULT") // Display the initial `countingArray`
values. displayTable.cell(0, 1, "Initial `countingArray`", text_color =
color.black, text_halign = text.align_center, bgcolor = color.yellow)
displayTable.cell(1, 1, str.tostring(countingArray), text_color =
color.black, text_halign = text.align_right, bgcolor = color.yellow) //
Retrieve array elements using negative indices in `array.get()`.
displayRow(2, "`countingArray.get(0)`", countingArray.get(0))
displayRow(3, "`countingArray.get(-1)`", countingArray.get(-1))
displayRow(4, "`countingArray.get(-countingArray.size())`",
countingArray.get(-countingArray.size())) // Update array elements
using negative indices in `array.set()` and `array.insert()`.
countingArray.set(-2, 99) displayRow(5, "`countingArray.set(-2, 99)`",
countingArray) countingArray.insert(-5, 878) displayRow(6,
```

"`countingArray.insert(-5, 878)`", countingArray) // Remove array elements using negative indices in `array.remove()`. countingArray.remove(-3) displayRow(7, "`countingArray.remove(-3)`", countingArray)

Note that not all array operations can use negative indices. For example, search functions like array.indexof() and array.binary_search() return the *positive* index of an element if it's found in the array. If the value is not found, the functions return `-1`. However, this returned value is **not** a negative index, and using it as one would incorrectly reference the last array element. If a script needs to use a search function's returned index in subsequent array operations, it must appropriately differentiate between this `-1` result and other valid indices.

Calculations on arrays

While series variables can be viewed as a horizontal set of values stretching back in time, Pine Script's one-dimensional arrays can be viewed as vertical structures residing on each bar. As an array's set of elements is not a time series, Pine Script's usual mathematical functions are not allowed on them. Special-purpose functions must be used to operate on all of an array's values. The available functions are: array.abs(), array.avg(), array.covariance(), array.min(), array.max(), array.median(), array.mode(), array.percentile_linear_interpolation(), array.percentile_nearest_rank(), array.percentrank(), array.range(), array.standardize(), array.stdev(), array.sum(), array.variance().

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

- When all array elements have na value or the array contains no elements, na is returned. `array.standardize()` however, will return an empty array.
- `array.mode()` will return na when no mode is found.

Manipulating arrays

Concatenation

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

image

```
//@version=6 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:

image

```
//@version=6 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

The array.join() function converts an “int”, “float”, or “string” array’s elements into strings, then *joins* each one to form a single “string” value with a specified `separator` inserted between each combined value. It provides a convenient alternative to converting values to strings with str.tostring() and performing repeated string concatenation operations.

The following script demonstrates the array.join() function’s behaviors. It requests tuples of “string”, “int”, and “float” values from three different contexts with request.security() calls, creates separate arrays for each type with array.from(), then creates joined strings with the array.join() function. Lastly, it creates another array from those strings with array.from() and joins them with another array.join() call, using a newline as the separator, and displays the final string in the table:

image

```
//@version=6 indicator("Joining demo") //@function Returns a tuple
containing the ticker ID ("string"), bar index ("int"), and closing price
("float"). dataRequest() => [syminfo.tickerid, bar_index, close] if
barstate.islast //@variable A single-cell table displaying the results of
`array.join()` calls. var table displayTable =
table.new(position.middle_center, 1, 1, color.blue) // Request data for
three symbols. [ticker1, index1, price1] = request.security("SPY", "",
dataRequest()) [ticker2, index2, price2] = request.security("GLD", "",
dataRequest()) [ticker3, index3, price3] = request.security("TLT", "",
dataRequest()) // Create separate "string", "int", and "float" arrays to
hold the requested data. array<string> tickerArray =
array.from(ticker1, ticker2, ticker3) array<int> indexArray =
array.from(index1, index2, index3) array<float> priceArray =
array.from(price1, price2, price3) // Convert each array's data to strings
and join them with different separators. string joined1 =
array.join(tickerArray, ", ") string joined2 = indexArray.join("|") string
joined3 = priceArray.join("\n") //@variable A joined "string" containing
the `joined1`, `joined2`, and `joined3` values. string displayText =
array.from(joined1, joined2, joined3).join("\n---\n") // Initialize a cell to
```

show the `displayText`. displayTable.cell(0, 0, displayText, text_color = color.white, text_size = 36)

Note that:

- Each array.join() call inserts the specified separator only between each element string. It does *not* include the separator at the start or end of the returned value.
- The array.join() function uses the same numeric format as the default for str.tostring(). See the String conversion and formatting section of the Strings page to learn more.
- Calls to array.join() cannot directly convert elements of “bool”, “color”, or other types to strings. Scripts must convert data of these types separately.

Sorting

Scripts can sort arrays containing “int”, “float”, or “string” elements in ascending or descending order using the array.sort() function. The direction in which the function sorts the array’s elements depends on its `order` parameter, which accepts the order.ascending or order.descending constants. The default argument is order.ascending, meaning the function sorts the elements in ascending order of value.

The function sorts arrays of “int” and “float” elements based on their *numeric* values.

The example below declares two arrays with references assigned to the a and b variables, and it concatenates those arrays to form a combined c array. The script creates Pine Logs showing formatted text representing the unsorted arrays, and the results of using array.sort() to sort all three arrays in ascending and descending order:

image

//@version=6 indicator("Sorting numeric arrays demo") if barstate.isfirst //@variable A formatting string. string formatString = "\n{0}:\n{1}\n{2}\n{3}" // Create two three-element arrays. array<float> a = array.from(2.1, 0.5, 1.2) array<float> b = array.from(0.1, 1.4, 0.6) //@variable A combined array containing the elements from `a` and `b`. array<float> c = array.copy(a).concat(b) // Log formatted text showing the unsorted `a`, `b`, and `c` arrays. log.info(formatString, "Unsorted", a, b, c) // Sort the `a`, `b`, and `c` arrays in ascending order (default). array.sort(a) array.sort(b) c.sort() // Log formatted text showing the `a`, `b`, and `c` arrays sorted in ascending order. log.info(formatString, "Ascending", a, b, c) // Sort the `a`, `b`, and `c` arrays in descending order. a.sort(order.descending) b.sort(order.descending) c.sort(order.descending) // Log formatted text showing the `a`, `b`, and `c` arrays sorted in descending order. log.info(formatString, "Descending", a, b, c)

Note that:

- Each array.sort() call directly *modifies* the order of the elements in the original array. To get sorted elements *without* reorganizing the original array, use the array.sort_indices() function. This function returns a new array of “int” values representing the *indices* of the elements sorted in ascending or descending order.

The array.sort() function sorts arrays of “string” values based on the *Unicode values* of their characters. The sorting algorithm starts with each element’s *first* character position, then successively uses additional characters if multiple elements have matching characters at the same position.

This example creates an array of arbitrary strings on the first bar, then sorts the array’s contents in ascending order with an array.sort() call. The script logs formatted representations of the array in the Pine Logs pane before and after calling the array.sort() function:

image

//@version=6 indicator("Sorting string arrays demo") if barstate.isfirst // @variable An array of arbitrary "string" values. array<string> stringArray = array.from("abC", "Abc", "ABc", "ABC", "!", "123", "12.3", " ") // Log the original `stringArray`. log.info("Unsorted: {0}", stringArray) // Sort the array in ascending order (default) and log the result. stringArray.sort() log.info("Ascending: {0}", stringArray)

Note that:

- Whitespace and control characters have lower Unicode values than other characters, which is why the `" "` element appears first in the sorted array.
- Some ASCII punctuation marks and symbols have lower Unicode values than digit or letter characters. The `"!"` element comes before the elements with word characters because its Unicode value is U+0021. However, some other ASCII punctuation and symbol characters, such as the Left Curly Bracket `{` (U+007B), have higher Unicode values than ASCII digits and letters.
- ASCII digits have lower Unicode values than letter characters. For example, the `1` character's value is U+0031, and the `A` character's value is U+0041.
- Uppercase ASCII letters come *before* lowercase characters in the Unicode Standard. For instance, the `a` character has the Unicode value U+0061, which is larger than the value for `A`.

Reversing

Use array.reverse() to reverse an array:

```
//@version=6 indicator("`array.reverse()`") a = array.new<float>(0)
array.push(a, 0) array.push(a, 1) array.push(a, 2) if barstate.islast
array.reverse(a) label.new(bar_index, 0, "a: " + str.tostring(a))
```

Slicing

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

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

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

image

```
//@version=6 indicator("`array.slice()`") a = array.new<float>(0)
array.push(a, 0) array.push(a, 1) array.push(a, 2) array.push(a, 3) if
barstate.islast // Create a shadow of elements at index 1 and 2 from
array `a`. sliceOfA = array.slice(a, 0, 3) label.new(bar_index, 0,
"BEFORE\na: " + str.tostring(a) + "\nsliceOfA: " +
str.tostring(sliceOfA)) // Remove first element of parent array `a`.
array.remove(a, 0) // Add a new element at the end of the shallow copy,
thus also affecting the original array `a`. array.push(sliceOfA, 4)
label.new(bar_index, 0, "AFTER\na: " + str.tostring(a) + "\nsliceOfA: " +
str.tostring(sliceOfA), style = label.style_label_up)
```

Searching arrays

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

```
//@version=6 indicator("Searching in arrays") valueInput = input.int(1)
a = array.new<float>(0) array.push(a, 0) array.push(a, 1) array.push(a,
2) array.push(a, 1) if barstate.islast valueFound = array.includes(a,
valueInput) firstIndexFound = array.indexof(a, valueInput)
lastIndexFound = array.lastindexof(a, valueInput) label.new(bar_index,
0, "a: " + str.tostring(a) + "\nFirst " + str.tostring(valueInput) +
(firstIndexFound != -1 ? " value was found at index: " +
str.tostring(firstIndexFound) : " value was not found.") + "\nLast " +
```

str.tostring(valueInput) + (lastIndexFound != -1 ? " value was found at index: " + str.tostring(lastIndexFound) : " value was not found."))

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

Error handling

Malformed `array.*()` call syntax in Pine scripts will cause the usual **compiler** error messages to appear in Pine Editor’s console, at the bottom of the window, when you save a script. Refer to the Pine Script v6 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 error is the most frequent one programmers encounter when using arrays. The error occurs when the script references a *nonexistent* array index. The “xx” value represents the out-of-bounds index the function tried to use, and “yy” represents the array’s size. Recall that array indices start at zero — not one — and end at the array’s size, minus one. For instance, the last valid index in a three-element array is 2.

To avoid this error, you must make provisions in your code logic to prevent using an index value outside the array’s boundaries. This code example generates the error because the last `i` value in the loop's iterations is beyond the valid index range for the a array:

```
//@version=6 indicator("Out of bounds index") a = array.new<float>(3) for i = 1 to 3 array.set(a, i, i) plot(array.pop(a))
```

To resolve the error, last `i` value in the loop statement should be less than or equal to 2:

```
for i = 0 to 2
```

To iterate over all elements in an array of *unknown* size with a for loop, set the loop counter's final value to one less than the array.size() value:

```
//@version=6 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 sizing arrays dynamically using a field in the script's *Settings/ Inputs* tab, protect the boundaries of that value using input.int()'s `minval` and `maxval` parameters:

```
//@version=6 indicator("Protected array size") sizeInput = input.int(10, "Array size", minval = 1, maxval = 100000) a = array.new<float>(sizeInput) for i = 0 to sizeInput - 1 array.set(a, i, i) plot(array.size(a))
```

See the Looping through array elements section of this page for more information.

Cannot call array methods when ID of array is 'na'

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

```
//@version=6 indicator("Array methods on `na` array") 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 pointing to the “window” at the parent array’s indices 3 to 4, is pointing out of the parent array’s boundaries:

```
//@version=6 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)
```

Previous

Next

Matrices

Pine Script® v6 Documentation

User Manual / Language / Matrices

ADVANCED

Matrices

Introduction

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

Matrices reference their elements using *two* indices: one index for their *rows* and the other for their *columns*. Each index starts at 0 and extends to the number of rows/columns in the matrix minus one. Matrices in Pine can have dynamic numbers of rows and columns that vary across bars. The total number of elements within a matrix is the *product* of the number of rows and columns it contains (e.g., a 5x5 matrix contains 25 elements). Like arrays, the total number of elements in a matrix cannot exceed 100,000.

Declaring a matrix

Pine Script uses the following syntax for matrix declaration:

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

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

When initializing a matrix variable with 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 an initial reference of na. It explicitly declares the variable as `matrix<float>`, which tells the compiler that the variable can accept only matrix objects containing float values:

```
matrix<float> myMatrix = na
```

When a matrix variable is not initialized with na, the matrix keyword and its type template are optional, as the compiler will use the type information from the matrix instance that 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 to reference 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’s value on the chart. As we see from the plot, the value of the matrix.get() method call persists across bars, never returning to the initial value of 0:

image

//@version=6 indicator("var matrix demo") //@variable A 1x2 rectangular matrix declared only at `bar_index == 0`, i.e., the first bar. var m = matrix.new<int>(1, 2, 0) //@variable Is `true` on every 20th bar. bool update = bar_index % 20 == 0 if update int currentValue = m.get(0, 0) // Get the current value of the first row and column. m.set(0, 0, currentValue + 1) // Set the first row and column element value to `currentValue + 1`. plot(m.get(0, 0), linewidth = 3) // Plot the value from the first row and column.

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 matrix.get() and matrix.set() method calls. 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:

image

//@version=6 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 the result of a math.random() call:

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

Note when using matrix.fill() with matrices of *reference 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 reference on the first bar. On each bar, the script sets the x property of the label referenced at row 0, column 0 to bar_index, and the `text` property 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 refers to the *same* label object, resulting in only one label on the chart with coordinates and displayed text that update on each bar:

image

//@version=6 indicator("Object matrix fill demo") //@variable A 4x4 label matrix. var matrix<label> m = matrix.new<label>(4, 4) // Fill `m` with a new label object on the first bar. if bar_index == 0 m.fill(label.new(0, 0, textcolor = color.white, size = size.huge)) // @variable The number of label objects on the chart. int numLabels = label.all.size() // Set the `x` of the label from the first row and column to `bar_index`. m.get(0, 0).set_x(bar_index) // Set the `text` of the label at the last row and column to the number of labels. m.get(3, 3).set_text(str.format("Total labels on the chart: {0}", numLabels))

Rows and columns

Retrieving

Scripts can retrieve all the data from a specific row or column in a matrix via the matrix.row() and matrix.col() functions. These functions return the row or column contents as an array sized according to the other dimension of the matrix. The size of a matrix.row() array equals the number of columns (matrix.columns()), and the size of a matrix.col() array equals the number of rows matrix.rows().

The script below populates a 3x2 `m` matrix with the values 1 - 6 on the first chart bar. It uses matrix.row() and matrix.col() method calls 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:

image

//@version=6 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 2, column 0 value to 5. m.set(2, 1, 6.0) // Set row 2, 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 matrix.rows() and matrix.columns() methods rather than array.size() to demonstrate that the size of the `row0` array equals the number of matrix columns and the size of the `column0` array equals the number of matrix rows.

The matrix.row() and matrix.col() functions copy the contents of 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:

image

//@version=6 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 constructed from 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 reference 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, because both elements refer to the same object:

```
//@version=6 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 values or 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 referenced by `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 one.

The script below declares an empty `m` matrix and inserts rows and columns by calling matrix.add_row() and matrix.add_col() as 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 the matrix after each row/column insertion with a user-defined `debugLabel()` function to visualize the process:

image

//@version=6 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")

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 one.

For this example, we’ve added these lines of code to our “Rows and columns demo” script from the Inserting 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 matrix.remove_row() and matrix.remove_col() method calls, then displays the rows in a label at `bar_index + 30`. As we can see, the matrix has a 3x3 shape after the script executes this block, and the index values for all existing rows are reduced by 1:

image

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 positions of the elements at the `row1/column1` and `row2/column2` indices.

Let’s add another set of code lines to the example from the removing section. The following lines swap the first and last rows of the `m` matrix 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:

image

Replacing

It may be desirable in some cases to completely *replace* a row or column in a matrix. To do so, insert another array's elements 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 matrix.add_row() function to insert the new `values` at the `row` index, and the matrix.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:

image

//@version=6 indicator("Replacing rows demo") //@function Displays the rows of a matrix in a label with a note. //@param this The matrix to display. //@param barIndex The `bar_index` to display the label at. //@param bgColor The background color of the label. //@param textColor The color of the label's text. //@param note The text to display above the rows. method debugLabel(matrix<float> this, int barIndex = bar_index, color bgColor = color.blue, color textColor = color.white, string note = "") => labelText = note + "\n" + str.tostring(this) if barstate.ishistory label.new(barIndex, 0, labelText, color = bgColor, style = label.style_label_center, textcolor = textColor, size = size.huge) //@function Replaces the `row` of `this` matrix with a new array of `values`. //@param row The row index to replace. //@param values The array of values to insert. method replaceRow(matrix<float> this, int row, array<float> values) => this.add_row(row, values) // Inserts a copy of the `values` array at the `row`. this.remove_row(row + 1) // Removes the old elements previously at the `row`. //@variable A 3x3 matrix. var matrix<float> m = matrix.new<float>(3, 3, 0.0) if bar_index == last_bar_index - 1 m.debugLabel(note = "Original") // Replace each row of `m`. m.replaceRow(0, array.from(1.0, 2.0, 3.0)) m.replaceRow(1, array.from(4.0, 5.0, 6.0)) m.replaceRow(2, array.from(7.0, 8.0, 9.0)) m.debugLabel(bar_index + 10, note = "Replaced rows")

Looping through a matrix

for

When a script only needs to iterate over the row/column indices in a matrix, the most common method is to use for loops. For example, this line creates a loop with a `row` value that starts at 0 and increases by one until it reaches one less than the number of rows in the `m` matrix (i.e., the last row index):

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

To iterate over all index values in the `m` matrix, we can create a *nested* loop that iterates over each `column` index on each `row` value:

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

Let’s use this nested structure to create a method that visualizes matrix elements. In the script below, we’ve defined a `toTable()` method that displays the elements of a matrix within a table object. It iterates over each `row` index and over each `column` index on every `row`. Within the loop, it converts each element to a string to display in the corresponding table cell.

On the first bar, the script creates an empty `m` matrix, populates it with rows, and calls `m.toTable()` to display its elements:

image

//@version=6 indicator("for loop demo", "Matrix to table") //@function Displays the elements of `this` matrix in a table. //@param this The matrix to display. //@param position The position of the table on the chart. //@param bgColor The background color of the table. //@param textColor The color of the text in each cell. //@param note A note string to display on the bottom row of the table. //@returns A new `table` object with cells corresponding to each element of `this` matrix. method toTable(matrix<float> this, string position =

position.middle_center, color bgColor = color.blue, color textColor = color.white, string note = na) => //@variable The number of rows in `this` matrix. int rows = this.rows() //@variable The number of columns in `this` matrix. int columns = this.columns() //@variable A table that displays the elements of `this` matrix with an optional `note` cell. table result = table.new(position, columns, rows + 1, bgColor) // Iterate over each row index of `this` matrix. for row = 0 to rows - 1 // Iterate over each column index of `this` matrix on each `row`. for col = 0 to columns - 1 //@variable The element from `this` matrix at the `row` and `col` index. float element = this.get(row, col) // Initialize the corresponding `result` cell with the `element` value. result.cell(col, row, str.tostring(element), text_color = textColor, text_size = size.huge) // Initialize a merged cell on the bottom row if a `note` is provided. if not na(note) result.cell(0, rows, note, text_color = textColor, text_size = size.huge) result.merge_cells(0, rows, columns - 1, rows) result // Return the `result` table. //@variable A 3x4 matrix of values. var m = matrix.new<float>() if bar_index == 0 // Add rows to `m`. m.add_row(0, array.from(1, 2, 3)) m.add_row(1, array.from(5, 6, 7)) m.add_row(2, array.from(9, 10, 11)) // Add a column to `m`. m.add_col(3, array.from(4, 8, 12)) // Display the elements of `m` in a table. m.toTable()

`for...in`

When a script needs to iterate over and retrieve the rows of a matrix, using the for...in structure is often preferred over the standard for loop. This structure directly references the row arrays in a matrix, making it a more convenient option for such use cases. For example, this line creates a loop that returns the reference of an array representing a row in the `m` matrix on each iteration:

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() value for each `row` array.

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 by calling matrix.add_col() and matrix.remove_col() as methods on each subsequent bar. It uses `m.rowWiseAvg()` to calculate the array of row-wise averages, then it plots the value of each array element on the chart:

image

```
//@version=6 indicator("for...in loop demo", "Average OHLC", overlay =
true) //@variable The number of terms in the average. int length =
input.int(20, "Length", minval = 1) //@function Calculates the average
of each matrix row. method rowWiseAvg(matrix<float> this) => //
@variable An array with elements corresponding to each row's
average. array<float> result = array.new<float>() // Iterate over each
`row` of `this` matrix. for row in this // Push the average of each `row`
into the `result`. result.push(row.avg()) result // Return the resulting
array. //@variable A 4x`length` matrix of values. var matrix<float> m
= matrix.new<float>(4, length) // Add a new column containing OHLC
values to the matrix. m.add_col(m.columns(), array.from(open, high,
low, close)) // Remove the first column. m.remove_col(0) //@variable An
array containing averages of `open`, `high`, `low`, and `close` over
`length` bars. array<float> averages = m.rowWiseAvg()
plot(averages.get(0), "Average Open", color.blue, 2)
plot(averages.get(1), "Average High", color.green, 2)
plot(averages.get(2), "Average Low", color.red, 2) plot(averages.get(3),
"Average Close", color.orange, 2)
```

Note that:

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

Copying a matrix

Shallow copies

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

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

image

//@version=6 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 have the same values or references as the original matrix. When matrices contain references to 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 matrix.

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 object referenced in the copied matrix affects the object referenced in the original matrix:

image

```
//@version=6 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 refer to copies of the objects referenced by the original matrix) by explicitly copying each element in the matrix.

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 matrix.copy(), we see that there are now two labels on the chart, and any changes to the label referenced by the copied matrix do not affect the one referenced by the original matrix:

image

//@version=6 indicator("Deep copy demo") //@function Returns a deep copy of a label matrix. method deepCopy(matrix<label> this) => // @variable A deep copy of `this` matrix. matrix<label> that = this.copy() for row = 0 to that.rows() - 1 for column = 0 to that.columns() - 1 // Assign the element at each `row` and `column` of `that` matrix to a copy of the retrieved label. that.set(row, column, that.get(row, column).copy()) that //@variable Initial value of the original matrix. var label newLabel = label.new(bar_index, 2, "Original", color = color.blue, textcolor = color.white, size = size.huge) //@variable A 1x1 matrix containing a new `label` instance. var matrix<label> myMatrix = matrix.new<label>(1, 1, newLabel) // @variable A deep copy of `myMatrix`. var matrix<label> myCopy = myMatrix.deepCopy() //@variable The first label from the `myCopy` matrix. label testLabel = myCopy.get(0, 0) // Change the `text`, `style`, and `x` values of `testLabel`. Does not affect the `newLabel`. testLabel.set_text("Copy") testLabel.set_style(label.style_label_up) testLabel.set_x(bar_index) // Change the `x` value of `newLabel`. newLabel.set_x(bar_index) // Plot the total number of labels. plot(label.all.size(), linewidth = 3)

Submatrices

In Pine, a *submatrix* is a shallow copy of an existing matrix that only includes the rows and columns specified by the `from_row/column` and `to_row/column` parameters. In essence, it is a sliced copy of a matrix.

For example, the script below creates an `mSub` matrix from the `m` matrix via the matrix.submatrix() method, then calls our user-defined `debugLabel()` function to display the rows of both matrices in labels:

image

//@version=6 indicator("Submatrix demo") //@function Displays the rows of a matrix in a label with a note. //@param this The matrix to display. //@param barIndex The `bar_index` to display the label at. // @param bgColor The background color of the label. //@param textColor The color of the label's text. //@param note The text to display above the rows. method debugLabel(matrix<float> this, int barIndex = bar_index, color bgColor = color.blue, color textColor = color.white, string note = "") => labelText = note + "\n" + str.tostring(this) if barstate.ishistory label.new(barIndex, 0, labelText, color = bgColor, style = label.style_label_center, textcolor = textColor, size = size.huge) //@variable A 3x3 matrix of values. var m = matrix.new<float>() if bar_index == last_bar_index - 1 // Add columns to `m`. m.add_col(0, array.from(9, 6, 3)) m.add_col(1, array.from(8, 5, 2)) m.add_col(2, array.from(7, 4, 1)) // Display the rows of `m`. m.debugLabel(note = "Original Matrix") //@variable A 2x2 submatrix of `m` containing the first two rows and columns. matrix<float> mSub = m.submatrix(from_row = 0, to_row = 2, from_column = 0, to_column = 2) // Display the rows of `mSub` debugLabel(mSub, bar_index + 10, bgColor = color.green, note = "Submatrix")

Scope and history

Matrix variables leave historical trails on each bar, allowing scripts to use the history-referencing operator [] to interact with past matrix instances previously assigned to a variable. Additionally, scripts can modify matrices assigned to global variables from within the scopes of user-defined 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 a matrix created in the global scope, and the `calcAvg()` function references previous matrices assigned to the `globalMatrix` variable using the [] operator to calculate a matrix of averages:

image

//@version=6 indicator("Scope and history demo", "Bar ratio comparison") int length = input.int(10, "Length", 1) //@variable A global matrix. matrix<float> globalMatrix = matrix.new<float>() // @function Calculates the ratio of body range to candle range. bodyRatio() => math.abs(close - open) / (high - low) //@function

Calculates the ratio of upper wick range to candle range. upperWickRatio() => (high - math.max(open, close)) / (high - low) // @function Calculates the ratio of lower wick range to candle range. lowerWickRatio() => (math.min(open, close) - low) / (high - low) // @function Adds data to the `globalMatrix`. addData() => // Add a new column of data at `column` 0. globalMatrix.add_col(0, array.from(bodyRatio(), upperWickRatio(), lowerWickRatio())) // @variable The column of `globalMatrix` from index 0 `length` bars ago. array<float> pastValues = globalMatrix.col(0)[length] // Add `pastValues` to the `globalMatrix`, or an array of `na` if `pastValues` is `na`. if na(pastValues) globalMatrix.add_col(1, array.new<float>(3)) else globalMatrix.add_col(1, pastValues) //@function Returns the `length`-bar average of matrices assigned to `globalMatrix` on historical bars. calcAvg() => //@variable The sum historical `globalMatrix` matrices. matrix<float> sums = matrix.new<float>(globalMatrix.rows(), globalMatrix.columns(), 0.0) for i = 0 to length - 1 //@variable The `globalMatrix` matrix `i` bars before the current bar. matrix<float> previous = globalMatrix[i] // Break the loop if `previous` is `na`. if na(previous) sums.fill(na) break // Assign the sum of `sums` and `previous` to `sums`. sums := matrix.sum(sums, previous) // Divide the `sums` matrix by the `length`. result = sums.mult(1.0 / length) // Add data to the `globalMatrix`. addData() //@variable The historical average of the `globalMatrix` matrices. globalAvg = calcAvg() //@variable A `table` displaying information from the `globalMatrix`. var table infoTable = table.new(position.middle_center, globalMatrix.columns() + 1, globalMatrix.rows() + 1, bgcolor = color.navy) // Define value cells. for [i, row] in globalAvg for [j, value] in row color textColor = value > 0.333 ? color.orange : color.gray infoTable.cell(j + 1, i + 1, str.tostring(value), text_color = textColor, text_size = size.huge) // Define header cells. infoTable.cell(0, 1, "Body ratio", text_color = color.white, text_size = size.huge) infoTable.cell(0, 2, "Upper wick ratio", text_color = color.white, text_size = size.huge) infoTable.cell(0, 3, "Lower wick ratio", text_color = color.white, text_size = size.huge) infoTable.cell(1, 0, "Current average", text_color = color.white, text_size = size.huge) infoTable.cell(2, 0, str.format("{0} bars ago", length), text_color = color.white, text_size = size.huge)

Note that:

- The `addData()` and `calcAvg()` functions have no parameters, as they directly interact with the `globalMatrix` and `length` variables declared in the outer scope.
- The `calcAvg()` functions calculates the averages 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 the Matrix calculations section below.

Inspecting a matrix

The ability to inspect the shape of a matrix and patterns within its elements is crucial, as it helps reveal important information about a matrix and its compatibility with various calculations and transformations. Pine Script includes several built-ins for matrix inspection, including matrix.is_square(), matrix.is_identity(), matrix.is_diagonal(), matrix.is_antidiagonal(), matrix.is_symmetric(), matrix.is_antisymmetric(), matrix.is_triangular(), matrix.is_stochastic(), matrix.is_binary(), and matrix.is_zero().

To demonstrate these features, this example contains a custom `inspect()` method that uses conditional blocks with `matrix.is_*()` functions to return information about a matrix. It displays a string representation of an `m` matrix and the description returned from `m.inspect()` in labels on the chart:

image

//@version=6 indicator("Matrix inspection demo") //@function Inspects a matrix using `matrix.is_*()` functions and returns a `string` describing some of its features. method inspect(matrix<int> this)=> // @variable A string describing `this` matrix. string result = "This matrix: \n" if this.is_square() result += "- Has an equal number of rows and columns.\n" if this.is_binary() result += "- Contains only 1s and 0s.\n" if this.is_zero() result += "- Is filled with 0s.\n" if this.is_triangular() result += "- Contains only 0s above and/or below its main diagonal.\n" if this.is_diagonal() result += "- Only has nonzero values in its main diagonal.\n" if this.is_antidiagonal() result += "- Only has nonzero values in its main antidiagonal.\n" if this.is_symmetric() result += "- Equals its transpose.\n" if this.is_antisymmetric() result += "- Equals the negative of its transpose.\n" if this.is_identity() result += "- Is the identity matrix.\n" result //@variable A 4x4 identity matrix. matrix<int> m = matrix.new<int>() // Add rows to the matrix. m.add_row(0, array.from(1, 0, 0, 0)) m.add_row(1, array.from(0, 1, 0, 0)) m.add_row(2, array.from(0, 0, 1, 0)) m.add_row(3, array.from(0, 0, 0, 1)) if bar_index == last_bar_index - 1 // Display the `m` matrix in a blue label. label.new(bar_index, 0, str.tostring(m), color = color.blue, style = label.style_label_right, textcolor = color.white, size = size.huge) // Display the result of `m.inspect()` in a purple label. label.new(bar_index, 0, m.inspect(), color = color.purple, style = label.style_label_left, textcolor = color.white, size = size.huge)

Manipulating a matrix

Reshaping

The shape of a matrix can determine its compatibility with various matrix operations. In some cases, it is necessary to change the dimensions of a matrix without affecting the number of elements or the values they reference, otherwise known as *reshaping*. To reshape a matrix in Pine, use the matrix.reshape() function.

This example demonstrates the results of multiple reshaping operations on a matrix. The initial `m` matrix has a 1x8 shape (one row and eight columns). Through successive calls to the matrix.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:

image

//@version=6 indicator("Reshaping example") //@function Displays the rows of a matrix in a label with a note. //@param this The matrix to display. //@param barIndex The `bar_index` to display the label at. //@param bgColor The background color of the label. //@param textColor

The color of the label's text. //@param note The text to display above the rows. method debugLabel(matrix<float> this, int barIndex = bar_index, color bgColor = color.blue, color textColor = color.white, string note = "") => labelText = note + "\n" + str.tostring(this) if barstate.ishistory label.new(barIndex, 0, labelText, color = bgColor, style = label.style_label_center, textcolor = textColor, size = size.huge) //@variable A matrix containing the values 1-8. matrix<int> m = matrix.new<int>() if bar_index == last_bar_index - 1 // Add the initial vector of values. m.add_row(0, array.from(1, 2, 3, 4, 5, 6, 7, 8)) m.debugLabel(note = "Initial 1x8 matrix") // Reshape. `m` now has 2 rows and 4 columns. m.reshape(2, 4) m.debugLabel(bar_index + 10, note = "Reshaped to 2x4") // Reshape. `m` now has 4 rows and 2 columns. m.reshape(4, 2) m.debugLabel(bar_index + 20, note = "Reshaped to 4x2") // Reshape. `m` now has 8 rows and 1 column. m.reshape(8, 1) m.debugLabel(bar_index + 30, note = "Reshaped to 8x1")

Note that:

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

Reversing

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

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

image

```
//@version=6 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 by calling matrix.transpose() as a method, then displays strings representing 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 matrix:

image

//@version=6 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:

image

//@version=6 indicator("Sorting rows example") //@function Displays the rows of a matrix in a label with a note. //@param this The matrix to display. //@param barIndex The `bar_index` to display the label at. //@param bgColor The background color of the label. //@param textColor

The color of the label's text. //@param note The text to display above the rows. method debugLabel(matrix<float> this, int barIndex = bar_index, color bgColor = color.blue, color textColor = color.white, string note = "") => labelText = note + "\n" + str.tostring(this) if barstate.ishistory label.new(barIndex, 0, labelText, color = bgColor, style = label.style_label_center, textcolor = textColor, size = size.huge) //@variable A 3x3 matrix. matrix<int> m = matrix.new<int>() if bar_index == last_bar_index - 1 // Add rows to `m`. m.add_row(0, array.from(3, 2, 4)) m.add_row(1, array.from(1, 9, 6)) m.add_row(2, array.from(7, 8, 9)) m.debugLabel(note = "Original") // Copy `m` and sort rows in ascending order based on the first column (default). matrix<int> m1 = m.copy() m1.sort() m1.debugLabel(bar_index + 10, color.green, note = "Sorted using col 0\n(Ascending)") // Copy `m` and sort rows in descending order based on the second column. matrix<int> m2 = m.copy() m2.sort(1, order.descending) m2.debugLabel(bar_index + 20, color.red, note = "Sorted using col 1\n(Descending)")

It’s important to note that matrix.sort() does not sort the columns of a matrix. However, one *can* use this function to sort matrix columns with the help of matrix.transpose().

As an example, this script contains a `sortColumns()` method that uses the matrix.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:

image

//@version=6 indicator("Sorting columns example") //@function Displays the rows of a matrix in a label with a note. //@param this The matrix to display. //@param barIndex The `bar_index` to display the label at. //@param bgColor The background color of the label. //@param textColor The color of the label's text. //@param note The text to display above the rows. method debugLabel(matrix<float> this, int barIndex = bar_index, color bgColor = color.blue, color textColor = color.white, string note = "") => labelText = note + "\n" + str.tostring(this) if barstate.ishistory label.new(barIndex, 0, labelText, color = bgColor, style = label.style_label_center, textcolor = textColor, size = size.huge) //@function Sorts the columns of `this` matrix based on the values in the specified `row`. method sortColumns(matrix<int> this, int row = 0, bool ascending = true) => //@variable The transpose of `this` matrix. matrix<int> thisT = this.transpose() //@variable Is `order.ascending` when `ascending` is `true`, `order.descending` otherwise. order = ascending ? order.ascending : order.descending // Sort the rows of `thisT` using the `row` column. thisT.sort(row, order) //@variable A copy of `this` matrix with sorted columns. result = thisT.transpose() //@variable A 3x3 matrix. matrix<int> m = matrix.new<int>() if bar_index == last_bar_index - 1 // Add rows to `m`. m.add_row(0, array.from(3, 2, 4)) m.add_row(1, array.from(1, 9, 6)) m.add_row(2, array.from(7, 8, 9)) m.debugLabel(note = "Original") // Sort the columns of `m` based on the first row and display the result. m.sortColumns(0).debugLabel(bar_index + 10, note = "Sorted using row 0\n(Ascending)")

Concatenating

Scripts can *concatenate* two matrices using matrix.concat(). This function appends the rows of an `id2` matrix to the end of an `id1` matrix with the same number of columns.

To create a matrix with elements representing the *columns* of a matrix appended to another, transpose both matrices, use matrix.concat() on the transposed matrices, then transpose() the result.

For example, this script appends the rows of the `m2` matrix to the `m1` matrix and appends their columns using *transposed copies* of the matrices. It displays the `m1` and `m2` matrices and the results after concatenating their rows and columns in labels using the custom `debugLabel()` method:

image

//@version=6 indicator("Concatenation demo") //@function Displays the rows of a matrix in a label with a note. //@param this The matrix to display. //@param barIndex The `bar_index` to display the label at. //@param bgColor The background color of the label. //@param textColor The color of the label's text. //@param note The text to display above the rows. method debugLabel(matrix<float> this, int barIndex = bar_index, color bgColor = color.blue, color textColor = color.white, string note = "") => labelText = note + "\n" + str.tostring(this) if barstate.ishistory label.new(barIndex, 0, labelText, color = bgColor, style = label.style_label_center, textcolor = textColor, size = size.huge) //@variable A 2x3 matrix filled with 1s. matrix<int> m1 = matrix.new<int>(2, 3, 1) //@variable A 2x3 matrix filled with 2s. matrix<int> m2 = matrix.new<int>(2, 3, 2) //@variable The transpose of `m1`. t1 = m1.transpose() //@variable The transpose of `m2`. t2 = m2.transpose() if bar_index == last_bar_index - 1 // Display the original matrices. m1.debugLabel(note = "Matrix 1") m2.debugLabel(bar_index + 10, note = "Matrix 2") // Append the rows of `m2` to the end of `m1` and display `m1`. m1.concat(m2) m1.debugLabel(bar_index + 20, color.blue, note = "Appended rows") // Append the rows of `t2` to the end of `t1`, then display the transpose of `t1`. t1.concat(t2) t1.transpose().debugLabel(bar_index + 30, color.purple, note = "Appended columns")

Matrix calculations

Element-wise calculations

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

The script below uses `*.avg()`, `*.max()`, and `*.min()` methods to calculate developing averages and extremes of OHLC data in a period. It adds a new column of open, high, low, and close values to the end of the `ohlcData` matrix whenever `queueColumn` is `true`. When `false`, the script uses the matrix.get() and matrix.set() 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 column 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:

image

```
//@version=6 indicator("Element-wise calculations example",
"Developing values", overlay = true) //@variable The number of data
points in the averages. int length = input.int(3, "Length", 1) //@variable
The timeframe of each reset period. string timeframe =
input.timeframe("D", "Reset Timeframe") //@variable A 4x`length`
matrix of OHLC values. var matrix<float> ohlcData =
matrix.new<float>(4, length) //@variable Is `true` at the start of a new
bar at the `timeframe`. bool queueColumn =
timeframe.change(timeframe) if queueColumn // Add new values to the
end column of `ohlcData`. ohlcData.add_col(length, array.from(open,
high, low, close)) // Remove the oldest column from `ohlcData`.
ohlcData.remove_col(0) else // Adjust the last element of column 1 for
new highs. if high > ohlcData.get(1, length - 1) ohlcData.set(1, length -
1, high) // Adjust the last element of column 2 for new lows. if low <
ohlcData.get(2, length - 1) ohlcData.set(2, length - 1, low) // Adjust the
last element of column 3 for the new closing price. ohlcData.set(3,
length - 1, close) //@variable The `matrix.avg()` of all elements in
`ohlcData`. avgOHLC4 = ohlcData.avg() //@variable The `matrix.avg()`
of all elements in rows 1 and 2, i.e., the average of all `high` and `low`
values. avgHL2 = ohlcData.submatrix(from_row = 1, to_row =
3).avg() //@variable The `matrix.max()` of all values in `ohlcData`.
Equivalent to `ohlcData.row(1).max()`. maxHigh = ohlcData.max() //
@variable The `array.min()` of all `low` values in `ohlcData`.
Equivalent to `ohlcData.min()`. minLow = ohlcData.row(2).min() //
@variable The `array.avg()` of the last column in `ohlcData`, i.e., the
current OHLC4. ohlc4Value = ohlcData.col(length - 1).avg()
plot(avgOHLC4, "Average OHLC4", color.purple, 2) plot(avgHL2,
"Average HL2", color.navy, 2) plot(maxHigh, "Max High", color.green)
plot(minLow, "Min Low", color.red) plot(ohlc4Value, "Current OHLC4",
color.blue)
```

Note that:

- In this example, we used `array.*()` and `matrix.*()` methods interchangeably to demonstrate their similarities in syntax and behavior.
- Users can calculate the matrix equivalent of array.sum() by multiplying the values of matrix.avg() and 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() results using the two matrices. This example displays the original matrix, its transpose, and the resulting sum and difference matrices in labels on the chart:

image

//@version=6 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^T") // 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^T") // 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^T")

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. The functions matrix.is_symmetric() and matrix.is_antisymmetric() test a matrix for these conditions.

matrix.mult()

Scripts can multiply two matrices via the matrix.mult() function. This function can also multiply 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 the `id1` matrix and the same number of columns as the `id2` matrix. 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 the `id1` maxtrix and column in the `id2` matrix:

image

//@version=6 indicator("Matrix mult example") //@function Displays the rows of a matrix in a label with a note. //@param this The matrix to display. //@param barIndex The `bar_index` to display the label at. //@param bgColor The background color of the label. //@param textColor The color of the label's text. //@param note The text to display above the rows. method debugLabel(matrix<float> this, int barIndex = bar_index, color bgColor = color.blue, color textColor = color.white, string note = "") => labelText = note + "\n" + str.tostring(this) if barstate.ishistory label.new(barIndex, 0, labelText, color = bgColor, style = label.style_label_center, textcolor = textColor, size = size.huge) //@variable A 2x3 matrix. a = matrix.new<float>() //@variable A 3x4 matrix. b = matrix.new<float>() // Add rows to `a`. a.add_row(0, array.from(1, 2, 3)) a.add_row(1, array.from(4, 5, 6)) // Add rows to `b`. b.add_row(0, array.from(0.5, 1.0, 1.5, 2.0)) b.add_row(1, array.from(2.5, 3.0, 3.5, 4.0)) b.add_row(0, array.from(4.5, 5.0, 5.5, 6.0)) if bar_index == last_bar_index - 1 //@variable The result of `a` * `b`. matrix<float> ab = a.mult(b) // Display `a`, `b`, and `ab` matrices. debugLabel(a, note = "A") debugLabel(b, bar_index + 10, note = "B") debugLabel(ab, bar_index + 20, color.green, note = "A * B")

Note that:

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

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

image

//@version=6 indicator("Determinants example", "Cramer's Rule") // @function Solves a system of linear equations with a matching number of unknowns using Cramer's rule. //@param this An augmented matrix containing the coefficients for each unknown and the results of // the equations. For example, a row containing the values 2, -1, and 3 represents the equation // `2 * x0 + (-1) * x1 = 3`, where `x0` and `x1` are the unknown values in the system. //@returns An array containing solutions for each variable in the system. solve(matrix<float> this) => //@variable The coefficient matrix for the system of equations. matrix<float> coefficients = this.submatrix(from_column = 0, to_column = this.columns() - 1) //@variable The array of resulting constants for each equation. array<float> constants = this.col(this.columns() - 1) //@variable An array containing solutions for each unknown in the system. array<float> result = array.new<float>() //@variable The determinant value of the coefficient matrix. float baseDet = coefficients.det() matrix<float> modified = na for col = 0 to coefficients.columns() - 1 modified := coefficients.copy() modified.add_col(col, constants) modified.remove_col(col + 1) // Calculate the solution for the column's unknown by dividing the determinant of `modified` by the `baseDet`. result.push(modified.det() / baseDet) result //@variable A 3x4 matrix containing coefficients and results for a system of three equations. m = matrix.new<float>() // Add rows for the following equations: // Equation 1: 3 * x0 + 4 * x1 - 1 * x2 = 8 // Equation 2: 5 * x0 - 2 * x1 + 1 * x2 = 4 // Equation 3: 2 * x0 - 2 * x1 + 1 * x2 = 1 m.add_row(0, array.from(3.0, 4.0, -1.0, 8.0)) m.add_row(1, array.from(5.0, -2.0, 1.0, 4.0)) m.add_row(2, array.from(2.0, -2.0, 1.0, 1.0)) //@variable An array of solutions to the unknowns in the system of equations represented by `m`. solutions = solve(m) plot(solutions.get(0), "x0", color.red, 3) // Plots 1. plot(solutions.get(1), "x1", color.green, 3) // Plots 2. plot(solutions.get(2), "x2", color.blue, 3) // Plots 3.

Note that:

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

matrix.inv() and matrix.pinv()

For any non-singular square matrix, there is an inverse matrix that yields the identity matrix when multiplied by the original. Inverses have use 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 calls matrix.inv() and matrix.pinv() as methods to calculate the inverse or pseudoinverse of `m`. The script displays strings representing the original matrix, its inverse or pseudoinverse, and their product in labels on the chart:

image

//@version=6 indicator("Inverse example") // Element inputs for the 2x2 matrix. float r0c0 = input.float(4.0, "Row 0, Col 0") float r0c1 = input.float(3.0, "Row 0, Col 1") float r1c0 = input.float(2.0, "Row 1, Col 0") float r1c1 = input.float(1.0, "Row 1, Col 1") //@function Displays the rows of a matrix in a label with a note. //@param this The matrix to display. //@param barIndex The `bar_index` to display the label at. // @param bgColor The background color of the label. //@param textColor The color of the label's text. //@param note The text to display above the rows. method debugLabel(matrix<float> this, int barIndex = bar_index, color bgColor = color.blue, color textColor = color.white, string note = "") => labelText = note + "\n" + str.tostring(this) if barstate.ishistory label.new(barIndex, 0, labelText, color = bgColor, style = label.style_label_center, textcolor = textColor, size = size.huge) //@variable A 2x2 matrix of input values. m = matrix.new<float>() // Add input values to `m`. m.add_row(0, array.from(r0c0, r0c1)) m.add_row(1, array.from(r1c0, r1c1)) //

```
@variable Is `true` if `m` is square with a nonzero determinant,
indicating invertibility. bool isInvertible = m.is_square() and m.det() !=
0 //@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 calls `m.inv()` only 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`) using matrix.rank() 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:

image

```
//@version=6 indicator("Matrix rank example") //@variable A 3x3 full-
rank matrix. m1 = matrix.new<float>() //@variable A 3x3 rank-
deficient matrix. m2 = matrix.new<float>() // Add linearly independent
vectors to `m1`. m1.add_row(0, array.from(3, 2, 3)) m1.add_row(1,
array.from(4, 6, 6)) m1.add_row(2, array.from(7, 4, 9)) // Add linearly
dependent vectors to `m2`. m2.add_row(0, array.from(1, 2, 3))
```

m2.add_row(1, array.from(2, 4, 6)) m2.add_row(2, array.from(3, 6, 9)) // Plot `matrix.rank()` values. plot(m1.rank(), color = color.green, linewidth = 3) plot(m2.rank(), color = color.red, linewidth = 3)

Note that:

- The highest rank value a matrix can have is the minimum of its number of rows and columns. A matrix with the maximum possible rank is known as a *full-rank* matrix, and any matrix without full rank is known as a *rank-deficient* matrix.
- The determinants of full-rank square matrices are nonzero, and such matrices have inverses. Conversely, the determinant of a rank-deficient matrix is always 0.
- For any matrix that contains nothing but the same value in each of its elements (e.g., a matrix filled with 0), the rank is always 0 since none of the vectors hold unique information. For any other matrix with distinct values, the minimum possible rank is 1.

Error handling

In addition to usual **compiler** errors, which occur during a script's compilation due to improper syntax, scripts using matrices can raise specific **runtime** errors during their execution. When a script raises a runtime error, it displays a red exclamation point next to the script title. Users can view the error message by clicking this icon.

In this section, we discuss runtime errors that users may encounter while utilizing matrices in their scripts.

The row/column index (xx) is out of bounds, row/column size is (yy).

This runtime error occurs when trying to access indices outside the matrix dimensions with functions including matrix.get(), matrix.set(), matrix.fill(), and matrix.submatrix(), as well as some of the functions relating to the rows and columns of a matrix.

For example, this code contains two lines that will produce this runtime error. The m.set() method references a row index that doesn't exist (2).

The m.submatrix() method references all column indices up to `to_column - 1`. A `to_column` value of 4 results in a runtime error because the last column index referenced (3) does not exist in `m`:

//@version=6 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=6 indicator("Invalid array size demo") // Declare an empty matrix. m = matrix.new<float>() m.add_col(0, array.from(1, 2)) // Add a column. Changes the shape of `m` to 2x1. m.add_col(1, array.from(1, 2, 3)) // Raises a runtime error because `m` has 2 rows, not 3. plot(m.col(0).get(1))

Note that:

- When `m` is empty, one can insert a row or column array of *any* size, as shown in the first `m.add_col()` line.

Cannot call matrix methods when the ID of matrix is ‘na’.

When a matrix variable is assigned to na, it means that the variable doesn’t reference an existing object. Consequently, one cannot use built-in `matrix.*()` functions and methods with it. For example:

```
//@version=6 indicator("na matrix methods demo") //@variable A `matrix` variable assigned to `na`. matrix<float> m = na mCopy = m.copy() // Raises a runtime error. You can't copy a matrix that doesn't exist. if bar_index == last_bar_index - 1 label.new(bar_index, 0, str.tostring(mCopy), color = color.navy, textcolor = color.white, size = size.huge)
```

To resolve this error, assign `m` to a valid matrix instance before using `matrix.*()` functions.

Matrix is too large. Maximum size of the matrix is 100,000 elements.

The total number of elements in a matrix (matrix.elements_count()) cannot exceed **100,000**, regardless of its shape. For example, this script will raise an error because it inserts 1000 rows with 101 elements into the `m` matrix:

```
//@version=6 indicator("Matrix too large demo") var matrix<float> m = matrix.new<float>() if bar_index == 0 for i = 1 to 1000 // This raises an error because the script adds 101 elements on each iteration. // 1000 rows * 101 elements per row = 101000 total elements. This is too large. m.add_row(m.rows(), array.new<float>(101, i)) plot(m.get(0, 0))
```

The row/column index must be 0 <= from_row/column < to_row/column.

When using `matrix.*()` functions with `from_row/column` and `to_row/column` indices, the `from_*` values must be less than the corresponding `to_*` values, with the minimum possible value being 0. Otherwise, the script will raise a runtime error.

For example, this script shows an attempt to declare a submatrix from a 4x4 `m` matrix with a `from_row` value of 2 and a `to_row` value of 2, which will result in an error:

```
//@version=6 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=6 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=6 indicator("Invalid mult dimensions demo") //@variable A
2x3 matrix. matrix<float> m1 = matrix.new<float>(2, 3, 1) //
@variable A 2x3 matrix. matrix<float> m2 = matrix.new<float>(2, 3,
2) mSum = matrix.mult(m1, m2) // Raises an error. The number of columns in `m1` and rows in `m2` aren't equal. plot(mSum.get(0, 0))
```

Operation not available for non-square matrices.

Some matrix operations, including matrix.inv(), matrix.det(), matrix.eigenvalues(), and matrix.eigenvectors() only work with **square** matrices, i.e., matrices with the same number of rows and columns. When attempting to execute such functions on non-square matrices, the script will raise an error stating the operation isn't available or that it cannot calculate the result for the matrix `id`. For example:

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

Previous

Next

Maps

Pine Script® v6 Documentation

User Manual / Language / Maps

ADVANCED

Maps

Introduction

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

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

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

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

Declaring a map

Pine Script uses the following syntax to declare maps:

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

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

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

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

```
map<string, float> myMap = na
```

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

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

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

Using var and varip keywords

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

For example, this script declares a `colorMap` variable to reference a map that holds pairs of string keys and color values on the first chart bar. The script plots a calculated `oscillator` series in a separate pane. It uses the "color" values put into the map on the *first* bar to color the plot on *every* bar:

image

```
//@version=6 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)

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 `id` map.

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 that pair’s value on the chart:

image

//@version=6 indicator("Putting and getting demo") //@variable The length of the `ta.rising()` and `ta.falling()` calculation. int length =

```
input.int(2, "Length") //@variable A map associating `string` keys with
`int` values. var data = map.new<string, int>() // Put a new ("Rising",
`bar_index`) pair into the `data` map when `close` is rising. if
ta.rising(close, length) data.put("Rising", bar_index) // Put a new
("Falling", `bar_index`) pair into the `data` map when `close` is falling.
if ta.falling(close, length) data.put("Falling", bar_index) // Put the
"Difference" between current "Rising" and "Falling" values into the
`data` map. data.put("Difference", data.get("Rising") -
data.get("Falling")) //@variable The difference between the last "Rising"
and "Falling" `bar_index`. int index = data.get("Difference") //@variable
Returns `color.green` when `index` is positive, `color.red` when
negative, and `color.gray` otherwise. color indexColor = index > 0 ?
color.green : index < 0 ? color.red : color.gray plot(index, color =
indexColor, style = plot.style_columns)
```

Note that:

- This script replaces the values associated with the “Rising”, “Falling”, and “Difference” keys on successive map.put() method 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 the reference for an instance 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` refers to that same object without copying it. Modifying the object referenced by a key-value pair directly modifies 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 corresponding to the “B” key affects the one referenced by the pair with the “A” key, as shown by the candle plot of the “A” object’s fields:

image

//@version=6 indicator("Putting and getting objects demo") //@type A custom type to hold OHLC data. type ChartData float o float h float l float c //@function Updates the fields of a `ChartData` object. method update(ChartData this) => this.o := open this.h := high this.l := low this.c := close //@variable A new `ChartData` instance declared on the first bar. var myData = ChartData.new() //@variable A map associating `string` keys with `ChartData` instances. var myMap = map.new<string, ChartData>() // Put a new pair with the "A" key into `myMap` only on the first bar. if bar_index == 0 myMap.put("A", myData) // Put a pair with the "B" key into `myMap` on every bar. myMap.put("B", myData) //@variable The `ChartData` value associated with the "A" key in `myMap`. ChartData oldest = myMap.get("A") // @variable The `ChartData` value associated with the "B" key in `myMap`. ChartData newest = myMap.get("B") // Update `newest`. Also affects `oldest` and `myData` since they all reference the same `ChartData` object. newest.update() // Plot the fields of `oldest` as candles. plotcandle(oldest.o, oldest.h, oldest.l, oldest.c)

Note that:

- This script would behave differently if it passed a copy of `myData` into each map.put() call. For more information, see the Copying objects section of the Objects page.

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 keys/values within an `id` map 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`:

image

```
//@version=6 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 map.keys() and map.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 that calls map.put() as a method to insert 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 key-value pair into the map *after* the inserting one with the “Third” key, the pair’s key and value are still second in the keys and `values` arrays, because the key is already present in the map *before* the new call:

image

```
//@version=6 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 )
```

`map.contains()`

To check if a specific key exists within an `id` map, 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:

image

```
//@version=6 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 an `id` map, 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 specified key. Otherwise, it returns na.

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

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

image

//@version=6 indicator("Removing key-value pairs demo") //@function Returns a label to display the keys and values from a map. method debugLabel(map<string, int> this, int barIndex = bar_index, color bgColor = color.blue, string note = "") => //@variable A string representing the size, keys, and values in `this` map. string repr = str.format("{0}\nSize: {1}\nKeys: {2}\nValues: {3}", note, this.size(), str.tostring(this.keys()), str.tostring(this.values())) label.new(barIndex, 0, repr, color = bgColor, style = label.style_label_center, textcolor = color.white, size = size.huge) if bar_index == last_bar_index - 1 // @variable A map containing `string` keys and `int` values. m = map.new<string, int>() // Put key-value pairs into `m`. for [i, key] in array.from("A", "B", "C", "D", "E") m.put(key, i) m.debugLabel(bar_index, color.green, "Added pairs") //@variable An array of keys to remove from `m`. array<string> removeKeys = array.from("B", "B", "D", "F", "a") // Remove each `key` in `removeKeys` from `m`. for key in removeKeys m.remove(key) m.debugLabel(bar_index + 10, color.red, "Removed pairs") // Remove all remaining keys from `m`. m.clear() m.debugLabel(bar_index + 20, color.purple, "Cleared the map")

Note that:

- Not all strings in the `removeKeys` array are present in the keys of the map. 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 calls map.put_all() as a method 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:

image

//@version=6 indicator("Combining maps demo", "Hex map") // @variable An array of string hex digits. var array<string> hexDigits = str.split("0123456789ABCDEF", "") //@function Returns a hexadecimal string for the specified `value`. hex(int value) => //@variable A string representing the hex form of the `value`. string result = "" //@variable A temporary value for digit calculation. int tempValue = value while tempValue > 0 //@variable The next integer digit. int digit = tempValue % 16 // Add the hex form of the `digit` to the `result`. result := hexDigits.get(digit) + result // Divide the `tempValue` by the base. tempValue := int(tempValue / 16) result //@function Returns a map holding the `numbers` as keys and their `hex` strings as values. hexMap(array<int> numbers) => //@variable A map associating `int` keys with `string` values. result = map.new<int, string>() for number in numbers // Put a pair containing the `number` and its `hex()` representation into the `result`. result.put(number, hex(number)) result //@function Returns a label to display the keys and values of a hex map. debugLabel(map<int, string> this, int barIndex = bar_index, color bgColor = color.blue, string style = label.style_label_center, string note = "") => string repr = str.format("{0}\nDecimal: {1}\nHex: {2}", note, str.tostring(this.keys()), str.tostring(this.values())) label.new(barIndex, 0, repr, color = bgColor, style = style, textcolor = color.white, size = size.huge) if bar_index == last_bar_index - 1 // @variable A map with decimal `int` keys and hexadecimal `string` values. map<int, string> mapA = hexMap(array.from(101, 202, 303, 404)) debugLabel(mapA, bar_index, color.navy, label.style_label_down, "A") //@variable A map containing key-value pairs to add to `mapA`. map<int, string> mapB = hexMap(array.from(303, 404, 505, 606, 707, 808)) debugLabel(mapB, bar_index, color.maroon, label.style_label_up, "B") // Put all pairs from `mapB` into `mapA`. mapA.put_all(mapB) debugLabel(mapA, bar_index + 10, color.purple, note = "Merge B into A")

Looping through a map

There are several ways scripts can iteratively access the keys and values in a map. For example, one could loop through the map.keys() array and use map.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:

image

//@version=6 indicator("Looping through a map demo", "Table of averages") //@variable The length of the moving average. int length = input.int(20, "Length") //@variable The size of the table text. string txtSize = input.string(size.huge, "Text size", options = [size.auto, size.tiny, size.small, size.normal, size.large, size.huge]) //@function Displays the pairs of `this` map within a table. //@param this A map with `string` keys and `float` values. //@param position The position of the table on the chart. //@param header The string to display on the top row of the table. //@param textSize The size of the text in the table. //@returns A new `table` object with cells displaying each pair in `this`. method toTable(map<string, float> this, string position = position.middle_center, string header = na, string textSize = size.huge) => // Color variables borderColor = #000000 headerColor = color.rgb(1, 88, 80) pairColor = color.maroon textColor = color.white //@variable A table that displays the key-value pairs of `this` map. table result = table.new(position, this.size() + 1, 3, border_width = 2, border_color = borderColor) // Initialize top and side header cells. result.cell(1, 0, header, bgcolor = headerColor, text_color = textColor, text_size = textSize) result.merge_cells(1, 0, this.size(), 0) result.cell(0, 1, "Key", bgcolor = headerColor, text_color = textColor, text_size = textSize) result.cell(0, 2, "Value", bgcolor = headerColor, text_color = textColor, text_size = textSize) //@variable The column index of the table. Updates on each loop iteration. int col = 1 // Loop over each `key` and `value` from `this` map in the insertion order. for [key, value] in this // Initialize a `key` cell in the `result` table on row 1. result.cell(col, 1, str.tostring(key), bgcolor = color.maroon, text_color = color.white, text_size = textSize) // Initialize a `value` cell in the `result` table on row 2. result.cell(col, 2, str.tostring(value), bgcolor = color.maroon, text_color = color.white, text_size = textSize) // Move to the next column index. col += 1 result // Return the `result` table. // @variable A map with `string` keys and `float` values to hold `length`-bar averages. averages = map.new<string, float>() // Put key-value pairs into the `averages` map. averages.put("Open", ta.sma(open, length)) averages.put("High", ta.sma(high, length)) averages.put("Low", ta.sma(low, length)) averages.put("Close", ta.sma(close, length)) averages.put("Volume", ta.sma(volume, length)) //@variable The text to display at the top of the table. string headerText = str.format("{0} {1}-bar averages", "'" + syminfo.tickerid + "'", length) // Display the `averages` map in a `table` with the `headerText`. averages.toTable(header = headerText, textSize = txtSize)

Copying a map

Shallow copies

Scripts can make a *shallow copy* of an `id` map by 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 the `m` map and updates the values associated with its keys. The script displays the key-value pairs from the two maps on the chart using our custom `debugLabel()` method:

image

```
//@version=6 indicator("Shallow copy demo") //@function Displays the
key-value pairs of `this` map in a label. method
debugLabel( map<string, float> this, int barIndex = bar_index, color
bgColor = color.blue, color textColor = color.white, string note = "" )
=> //@variable The text to display in the label. labelText = note + "\n{"
for [key, value] in this labelText += str.format("{0}: {1}, ", key, value)
labelText := str.replace(labelText, ", ", "}", this.size() - 1) if
barstate.ishistory label result = label.new( barIndex, 0, labelText, color
= bgColor, style = label.style_label_center, textcolor = textColor, size
= size.huge ) if bar_index == last_bar_index - 1 //@variable A map of
`string` keys and random `float` values. m = map.new<string,
float>() // Assign random values to an array of keys in `m`. for key in
array.from("A", "B", "C", "D") m.put(key, math.random(0, 10)) //
@variable A shallow copy of `m`. mCopy = m.copy() // Assign the
insertion order value `i` to each `key` in `mCopy`. for [i, key] in
mCopy.keys() mCopy.put(key, i) // Display the labels.
m.debugLabel(bar_index, note = "Original")
mCopy.debugLabel(bar_index + 10, color.purple, note = "Copied and
changed")
```

Deep copies

While a shallow copy will suffice when copying maps that have values of a fundamental type or enum type, it’s crucial to understand that shallow copies of a map with elements 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’s elements affect the objects 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 to store string key elements and label value elements, then puts a key-value pair into it. The script copies the map to a `shallow` variable via the built-in map.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:

image

```
//@version=6 indicator("Deep copy demo") //@function Returns a deep
copy of `this` map. method deepCopy(map<string, label> this) => //
@variable A deep copy of `this` map. result = map.new<string,
label>() // Add key-value pairs with copies of each `value` to the
`result`. for [key, value] in this result.put(key, value.copy()) result //
Return the `result`. //@variable A map containing `string` keys and
`label` values. var original = map.new<string, label>() if bar_index ==
last_bar_index - 1 // Put a new key-value pair into the `original` map.
map.put( original, "Test", label.new(bar_index, 0, "Original", textcolor =
color.white, size = size.huge) ) //@variable A shallow copy of the
`original` map. map<string, label> shallow = original.copy() //
@variable A deep copy of the `original` map. map<string, label> deep
= original.deepCopy() //@variable The "Test" label from the `shallow`
copy. label shallowLabel = shallow.get("Test") //@variable The "Test"
label from the `deep` copy. label deepLabel = deep.get("Test") //
Modify the "Test" label's `y` attribute in the `original` map. // This also
affects the `shallowLabel`. original.get("Test").set_y(label.all.size()) //
Modify the `shallowLabel`. Also modifies the "Test" label in the
`original` map. shallowLabel.set_text("Shallow copy")
shallowLabel.set_color(color.red)
shallowLabel.set_style(label.style_label_up) // Modify the `deepLabel`.
Does not modify any other label instance. deepLabel.set_text("Deep
copy") deepLabel.set_color(color.navy)
deepLabel.set_style(label.style_label_left) deepLabel.set_x(bar_index +
5)
```

Note that:

- The `deepCopy()` method loops through the `original` map, copying each `value` and putting key-value pairs containing the copies into a new map instance.

Scope and history

As with other collections in Pine, map variables leave historical trails on each bar, allowing a script to access past map instances assigned to a variable using the history-referencing operator []. Scripts can also assign maps to global variables and interact with them from the scopes of user-defined 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 map.values() array and the value from `globalData.get(50)` (i.e., the 50-bar EMA):

image

```
//@version=6 indicator("Scope and history demo", overlay = true) // @variable The source value for EMA calculation. float source = input.source(close, "Source") //@variable A map containing global key-value pairs. globalData = map.new<int, float>() //@function Calculates a set of EMAs and updates the key-value pairs in `globalData`. update() => //@variable The previous map instance assigned to `globalData`. map<int, float> previous = globalData[1] // Put key-value pairs with keys 10-200 into `globalData` if `previous` is `na`. if na(previous) for i = 10 to 200 globalData.put(i, source) else // Iterate each `key` and `value` in the `previous` map. for [key, value] in previous //@variable The smoothing parameter for the `key`-length EMA. float alpha = 2.0 / (key + 1.0) //@variable The `key`-length EMA value. float ema = (1.0 -
alpha) * value + alpha * source // Put the `key`-length `ema` into the
`globalData` map. globalData.put(key, ema) // Update the `globalData`
map. update() //@variable The array of values from `globalData` in
their insertion order. array<float> values = globalData.values() // Plot
the max EMA, min EMA, and 50-bar EMA values. plot(values.max(),
"Max EMA", color.green, 2) plot(values.min(), "Min EMA", color.red, 2)
plot(globalData.get(50), "50-bar EMA", color.orange, 3)
```

Maps of other collections

Maps cannot directly store references to other maps, arrays, or matrices as values, but they can hold references to objects of a user-defined type whose fields reference other collections.

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 directly reference other collections, we will first create a *wrapper type* with a field to reference 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 containing string keys and `Wrapper` references as values, where the `data` field of each object referenced by the map points to a `map<string, float>` instance:

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

The script below uses this concept to construct a map to reference other 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 map referenced by the `mapOfMaps` variable, creates a string representation of the nested maps with a user-defined `toString()` method, then displays the string on the chart using a label:

image

//@version=6 indicator("Nested map demo") //@variable The timeframe of the requested data. string tf = input.timeframe("D", "Timeframe") // Symbol inputs. string symbol1 = input.symbol("EURUSD", "Symbol 1") string symbol2 = input.symbol("GBPUSD", "Symbol 2") string symbol3 = input.symbol("EURGBP", "Symbol 3") //@type A wrapper type for maps with `string` keys and `float` values. type Wrapper map<string, float> data //@function Returns a wrapped map containing OHLCV data from the `tickerID` at the `timeframe`. requestData(string tickerID, string timeframe) => // Request a tuple of OHLCV values from the specified ticker and timeframe. [o, h, l, c, v] = request.security(tickerID, timeframe, [open, high, low, close, volume]) //@variable A map containing requested OHLCV data. result = map.new<string, float>() // Put key-value pairs into the `result`. result.put("Open", o) result.put("High", h) result.put("Low", l) result.put("Close", c) result.put("Volume", v) //Return the wrapped `result`. Wrapper.new(result) //@function Returns a string representing `this` map of `string` keys and `Wrapper` values. method toString(map<string, Wrapper> this) => //@variable A string representation of `this` map. string result = "{" // Iterate over each `key1` and associated `wrapper` in `this`. for [key1, wrapper] in this // Add `key1` to the `result`. result += key1 //@variable A string representation of the `wrapper.data` map. string innerStr = ": {" // Iterate over each `key2` and associated `value` in the wrapped map. for [key2, value] in wrapper.data // Add the key-value pair's representation to `innerStr`. innerStr += str.format("{0}: {1}, ", key2, str.tostring(value)) // Replace the end of `innerStr` with "}" and add to `result`. result += str.replace(innerStr, ", ", "},\n", wrapper.data.size() - 1) // Replace the blank line at the end of `result` with "}". result := str.replace(result, ",\n", "}", this.size() - 1) result //@variable A map of wrapped maps containing OHLCV data from multiple tickers. var mapOfMaps = map.new<string, Wrapper>() //@variable A label showing the contents of the `mapOfMaps`. var debugLabel = label.new(bar_index, 0, color = color.navy, textcolor = color.white, size = size.huge, style = label.style_label_center, text_font_family = font.family_monospace) // Put wrapped maps into `mapOfMaps`.

```
mapOfMaps.put(symbol1, requestData(symbol1, tf))
mapOfMaps.put(symbol2, requestData(symbol2, tf))
mapOfMaps.put(symbol3, requestData(symbol3, tf)) // Update the
label. debugLabel.set_text(mapOfMaps.toString())
debugLabel.set_x(bar_index)
```

Previous