Pine Script® v6 Documentation

4 Writing Scripts

TradingView Pine Script Programming Language

Table of Contents

Redirect

Pine Script® v6 Documentation

Style guide

Pine Script® v6 Documentation

Style guide

Introduction

This style guide provides recommendations on how to name variables and organize your Pine scripts in a standard way that works well. Scripts that follow our best practices will be easier to read, understand and maintain.

You can see scripts using these guidelines published from the TradingView and PineCoders accounts on the platform.

Naming Conventions

We recommend the use of:

- `camelCase` for all identifiers, i.e., variable or function names: `ma`, `maFast`, `maLengthInput`, `maColor`, `roundedOHLC()`, `pivotHi()`.
- All caps `SNAKE_CASE` for constants: `BULL_COLOR`, `BEAR_COLOR`, `MAX_LOOKBACK`.
- The use of qualifying suffixes when it provides valuable clues about the type or provenance of a variable: `maShowInput`, `bearColor`, `bearColorInput`, `volumesArray`, `maPlotID`, `resultsTable`, `levelsColorArray`.

Script organization

The Pine Script® compiler is quite forgiving of the positioning of specific statements or the version compiler annotation in the script.

While other arrangements are syntactically correct, this is how we recommend organizing scripts:

```
<license>
<version>
<declaration_statement>
<import_statements>
<constant_declarations>
<inputs>
<function_declarations>
<calculations>
<strategy_calls>
<visuals>
<alerts>
```

<license>

If you publish your open-source scripts publicly on TradingView (scripts can also be published privately), your open-source code is by default protected by the Mozilla license. You may choose any other license you prefer.

The reuse of code from those scripts is governed by our House Rules on Script Publishing which preempt the author's license.

The standard license comments appearing at the beginning of scripts are:

// This source code is subject to the terms of the Mozilla Public License 2.0 at https://mozilla.org/MPL/2.0/ // © username

<version>

This is the compiler annotation defining the version of Pine Script the script will use. If none is present, v1 is used. For v6, use:

//@version=6

<declaration_statement>

This is the mandatory declaration statement which defines the type of your script. It must be a call to either indicator(), strategy(), or library().

<import_statements>

If your script uses one or more Pine Script libraries, your import statements belong here.

<constant_declarations>

Scripts can declare variables qualified as “const”, i.e., ones referencing a constant value.

We refer to variables as “constants” when they meet these criteria:

- Their declaration uses the optional `const` keyword (see our User Manual’s section on type qualifiers for more information).
- They are initialized using a literal (e.g., `100` or `"AAPL"`) or a built-in qualified as “const” (e.g., `color.green`).
- Their value does not change during the script’s execution.

We use `SNAKE_CASE` to name these variables and group their declaration near the top of the script. For example:

// ——————— Constants int MS_IN_MIN = 60 * 1000 int MS_IN_HOUR = MS_IN_MIN * 60 int MS_IN_DAY = MS_IN_HOUR * 24 color GRAY = #808080ff color LIME = #00FF00ff color MAROON = #800000ff color ORANGE = #FF8000ff color PINK = #FF0080ff color TEAL = #008080ff color BG_DIV = color.new(ORANGE, 90) color BG_RESETS = color.new(GRAY, 90) string RST1 = "No reset; cumulate since the beginning of the chart" string RST2 = "On a stepped higher timeframe (HTF)" string RST3 = "On a fixed HTF" string RST4 = "At a fixed time" string RST5 = "At the beginning of the regular session" string RST6 = "At the first visible chart bar" string RST7 = "Fixed rolling period" string LTF1 = "Least precise, covering many chart bars" string LTF2 = "Less precise, covering some chart bars" string LTF3 = "More precise, covering less chart bars" string LTF4 = "Most precise, 1min intrabars"

string TT_TOTVOL = "The 'Bodies' value is the transparency of the total volume candle bodies. Zero is opaque, 100 is transparent." string TT_RST_HTF = "This value is used when '" + RST3 +"' is selected." string TT_RST_TIME = "These values are used when '" + RST4 +"' is selected. A reset will occur when the time is greater or equal to the bar's open time, and less than its close time.\nHour: 0-23\nMinute: 0-59" string TT_RST_PERIOD = "This value is used when '" + RST7 +"' is selected."

In this example:

- The RST* and LTF* constants will be used as tuple elements in the `options` argument of `input.*()` calls.
- The TT_* constants will be used as `tooltip` arguments in `input.*()` calls. Note how we use a line continuation for long string literals.
- We do not use var to initialize constants. The Pine Script runtime is optimized to handle declarations on each bar, but using var to initialize a variable only the first time it is declared incurs a minor penalty on script performance because of the maintenance that var variables require on further bars.

Note that:

- Literals used in more than one place in a script should always be declared as a constant. Using the constant rather than the literal makes it more readable if it is given a meaningful name, and the practice makes code easier to maintain. Even though the quantity of milliseconds in a day is unlikely to change in the future, `MS_IN_DAY` is more meaningful than `1000 * 60 * 60 * 24`.
- Constants only used in the local block of a function or if, while, etc., statement for example, can be declared in that local block.

<inputs>

It is **much** easier to read scripts when all their inputs are in the same code section. Placing that section at the beginning of the script also reflects how they are processed at runtime, i.e., before the rest of the script is executed.

Suffixing input variable names with `input` makes them more readily identifiable when they are used later in the script: `maLengthInput`, `bearColorInput`, `showAvgInput`, etc.

// ——————— Inputs string resetInput = input.string(RST2, "CVD Resets", inline = "00", options = [RST1, RST2, RST3, RST4, RST5, RST6, RST7]) string fixedTfInput = input.timeframe("D", " Fixed HTF: ", tooltip = TT_RST_HTF) int hourInput = input.int(9, " Fixed time hour: ", inline = "01", minval = 0, maxval = 23) int minuteInput = input.int(30, "minute", inline = "01", minval = 0, maxval = 59, tooltip = TT_RST_TIME) int fixedPeriodInput = input.int(20, " Fixed period: ", inline = "02", minval = 1, tooltip = TT_RST_PERIOD) string ltfModeInput = input.string(LTF3, "Intrabar precision", inline = "03", options = [LTF1, LTF2, LTF3, LTF4])

<function_declarations>

All user-defined functions must be defined in the script's global scope; nested function definitions are not allowed in Pine Script.

Optimal function design should minimize the use of global variables in the function's scope, as they undermine function portability. When it can't be avoided, those functions must follow the global variable declarations in the code, which entails they can't always be placed in the <function_declarations> section. Such dependencies on global variables should ideally be documented in the function's comments.

It will also help readers if you document the function's objective, parameters and result. The same syntax used in libraries can be used to document your functions. This can make it easier to port your functions to a library should you ever decide to do so:

//@version=6 indicator("<function_declarations>", "", true) string SIZE_LARGE = "Large" string SIZE_NORMAL = "Normal" string SIZE_SMALL = "Small" string sizeInput = input.string(SIZE_NORMAL, "Size", options = [SIZE_LARGE, SIZE_NORMAL, SIZE_SMALL]) // @function Used to produce an argument for the `size` parameter in built-in functions. // @param userSize (simple string) User-selected size. // @returns One of the `size.*` built-in constants. // Dependencies

SIZE_LARGE, SIZE_NORMAL, SIZE_SMALL getSize(simple string userSize) => result = switch userSize SIZE_LARGE => size.large SIZE_NORMAL => size.normal SIZE_SMALL => size.small => size.auto if ta.rising(close, 3) label.new(bar_index, na, yloc = yloc.abovebar, style = label.style_arrowup, size = getSize(sizeInput))

<calculations>

This is where the script's core calculations and logic should be placed. Code can be easier to read when variable declarations are placed near the code segment using the variables. Some programmers prefer to place all their non-constant variable declarations at the beginning of this section, which is not always possible for all variables, as some may require some calculations to have been executed before their declaration.

<strategy_calls>

Strategies are easier to read when strategy calls are grouped in the same section of the script.

<visuals>

This section should ideally include all the statements producing the script's visuals, whether they be plots, drawings, background colors, candle-plotting, etc. See the Pine Script user manual's section on here for more information on how the relative depth of visuals is determined.

<alerts>

Alert code will usually require the script's calculations to have executed before it, so it makes sense to put it at the end of the script.

Spacing

A space should be used on both sides of all operators, except unary operators (`-1`). A space is also recommended after all commas and when using named function arguments, as in `plot(series = close)`:

int a = close > open ? 1 : -1 var int newLen = 2 newLen := min(20, newlen + 1) float a = -b float c = d > e ? d - e : d int index = bar_index % 2 == 0 ? 1 : 2 plot(close, color = color.red)

Line wrapping

Line wrapping can make long lines easier to read. Line wraps are defined by using an indentation level that is not a multiple of four, as four spaces or a tab are used to define local blocks. Here we use two spaces:

plot(series = close, title = "Close", color = color.blue, show_last = 10)

Vertical alignment

Vertical alignment using tabs or spaces can be useful in code sections containing many similar lines such as constant declarations or inputs. They can make mass edits much easier using the Pine Editor's multi-cursor feature (`ctrl + alt +  `):

// Colors used as defaults in inputs. color COLOR_AQUA = #0080FFff color COLOR_BLACK = #000000ff color COLOR_BLUE = #013BCAff color COLOR_CORAL = #FF8080ff color COLOR_GOLD = #CCCC00ff

Explicit typing

Including the type of variables when declaring them is not required. However, it helps make scripts easier to read, navigate, and understand. It can help clarify the expected types at each point in a script's execution and distinguish a variable's declaration (using =) from its reassignments (using `:=`). Using explicit typing can also make scripts easier to debug.

Next

Debugging

Pine Script® v6 Documentation

Debugging

Introduction

TradingView's close integration between the Pine Editor and the Supercharts interface enables efficient, interactive debugging of Pine Script® code. Pine scripts can create dynamic outputs in multiple locations, on and off the chart. Programmers can use these outputs to validate their scripts' behaviors and ensure everything works as expected.

Understanding the most effective tools and methods for inspecting a script helps programmers quickly find and fix potential problems in their code, which improves the overall coding experience. This page explains the script outputs that are the most useful for debugging, along with helpful tips and techniques.

Common debug outputs

Pine scripts can create outputs in several ways, each of which has different advantages. While programmers can use any of them to debug their code, some outputs are more optimal for debugging than others.

The functions in the `log.*` namespace log *interactive messages* in the Pine Logs pane. These logging functions are the most convenient and flexible tools for debugging Pine code. Scripts can call `log.*()` functions on any execution from global or local scopes, enabling programmers to analyze historical and realtime script behaviors in depth with minimal code, for example:

image

//@version=6 indicator("Common debug outputs - Pine Logs") // @variable The natural logarithm of the current `high - low` range. float logRange = math.log(high - low) // Plot the `logRange`. plot(logRange, "logRange") if barstate.isconfirmed // Generate an "error" or "info" message on the confirmed bar, depending on whether `logRange` is defined. switch na(logRange) => log.error("Undefined `logRange` value.") => log.info("`logRange` value: " + str.tostring(logRange)) else // Generate a "warning" message for unconfirmed values. log.warning("Unconfirmed `logRange` value: " + str.tostring(logRange))

Pine drawings display visuals in the main chart pane or the script's separate pane. Although they do not output results in other locations, such as the Data Window or Pine Logs pane, drawings provide convenient ways to visualize a script's data and logic within global or local scopes. Labels are the most flexible drawings for debugging, because they can display colored shapes with formatted text and tooltips at any available chart location, for example:

image

//@version=6 indicator("Common debug outputs - Pine drawings", overlay = true) //@variable Is `true` when a new bar opens on the "1D" timeframe. bool newDailyBar = timeframe.change("1D") //@variable The previous bar's `bar_index` from when `newDailyBar` last occurred. int closedIndex = ta.valuewhen(newDailyBar, bar_index - 1, 0) // @variable The previous bar's `close` from when `newDailyBar` last occurred. float closedPrice = ta.valuewhen(newDailyBar, close[1], 0) if newDailyBar // Draw a line from the previous `closedIndex` and `closedPrice` to the current values. line.new(closedIndex[1], closedPrice[1], closedIndex, closedPrice, width = 2) //@variable A string containing debug information to display in a label. string debugText = "'1D' bar closed at: \n(" + str.tostring(closedIndex) + ", " + str.tostring(closedPrice) + ")" //@variable Draws a label at the current `closedIndex` and `closedPrice`. label debugLabel = label.new(closedIndex, closedPrice, debugText, color = color.purple, textcolor = color.white)

The `plot*()` functions can help to debug numeric values, conditions, and colors from a script's global scope. They can output results in up to *four* locations: the main chart pane or the script's pane, the status line, the price scale, and the Data Window. The display on the chart provides a quick view of the series' history, and the numbers in the other output locations show calculated information for specific bars:

image

//@version=6 indicator("Common debug outputs - Plots") // Plot the `bar_index` in all available locations. plot(bar_index, "bar_index", color.teal, 3)

The bgcolor() function displays colors in the background of the main chart pane or the script's pane. The barcolor() function colors the main chart's bars or candles. Although these outputs are less flexible than Pine Logs, drawings, and plots, they provide a quick way to inspect calculated colors and visualize conditions from the global scope:

image

//@version=6 indicator("Common debug outputs - Background and bar colors") //@variable Is `true` if the `close` is rising over 2 bars. bool risingPrice = ta.rising(close, 2) // Highlight the chart background and color the main chart bars based on `risingPrice`. bgcolor(risingPrice ? color.new(color.green, 70) : na, title= "`risingPrice` highlight") barcolor(risingPrice ? color.aqua : chart.bg_color, title = "`risingPrice` bar color")

Programmers can use any of these outputs individually or in combination to debug their scripts, depending on the data types and structures that require inspection. See the sections below for detailed information about these outputs and various debugging techniques.

Pine Logs

Pine Logs are interactive, user-defined messages that scripts can create from within global or local scopes at *any point* during code executions on the chart's dataset or requested datasets. They provide a simple, powerful way for programmers to inspect a script's

calculations, logic, and execution flow with human-readable text. Using Pine Logs is the primary, most universal technique for debugging Pine Script code.

Pine Logs do not appear on the chart or in the Data Window. Instead, scripts print logged messages with prefixed date and time information in the dedicated *Pine Logs pane*. The inspection and filtering options in the Pine Logs pane help users analyze and navigate logs efficiently.

To access the pane, select “Pine Logs” from the Pine Editor’s “More” menu or from the “More” menu in the status line of a script on the chart that uses the `log.*()` functions:

image

Creating logs

Scripts create Pine Logs by calling the functions in the `log.*` namespace: log.info(), log.warning(), or log.error(). All these logging functions have the following two signatures:

```
log.*(message) → void
log.*(formatString, arg0, arg1, ...) → void
```

Where:

- The first overload prints the specified “string” `message` in the Pine Logs pane.
- The second overload creates a formatted string based on its `formatString` and additional arguments, similar to str.format(), then displays the resulting text inside the pane.

Each `log.*()` function has a different *logging level*, allowing programmers to *categorize* the messages shown in the Pine Logs pane:

- The log.info() function creates a message with the “info” level (gray text).
- The log.warning() function creates a message with the “warning” level (orange text).
- The log.error() function creates a message with the “error” level (red text).

This simple script demonstrates the difference between all three `log.*()` functions. It calls log.info(), log.warning(), and log.error() on the first chart bar to print the values of three literal strings in the Pine Logs pane:

image

//@version=6 indicator("Logging levels demo", overlay = true) // Display logs with all three logging levels in the Pine Logs pane on the first bar. if barstate.isfirst log.info("This is an 'info' message.") log.warning("This is a 'warning' message.") log.error("This is an 'error' message.")

Note that:

- The Pine Logs pane can *filter* messages by their logging level using the menu accessible from the rightmost icon above the logs. See the Filtering logs section to learn more.

Scripts can generate logs at *any point* during their executions, allowing programmers to track information from historical bars, and monitor script behaviors on open realtime bars.

During historical executions, scripts log a new message *once* for each `log.*()` call on any bar. During realtime executions, scripts can call the `log.*()` functions to log messages for *any available tick*, regardless of whether the bar is confirmed. The logs created on realtime ticks are *not* subject to rollback. All logs remain available in the Pine Logs pane until the script restarts.

The example script below calculates the average ratio of each bar's `close - open` value to its `high - low` range. When the range is nonzero, the script prints the values of the calculation's variables in the Pine Logs pane using log.info() if the bar is confirmed or log.warning() if the bar is still open (*unconfirmed*). If the bar's range is zero, making the calculated ratio *undefined*, the script logs an "error" message using log.error():

image

//@version=6 indicator("Historical and realtime logs demo", "Average bar ratio") //@variable The current bar's change from the `open` to `close`. float numerator = close - open //@variable The current bar's `low` to `high` range. float denominator = high - low //@variable The ratio of the bar's open-to-close change to its full range. float ratio = numerator / denominator //@variable The average `ratio` over 10 *non-na* values. float average = ta.sma(ratio, 10) // Plot the `average`. plot(average, "average", color.purple, 3) if barstate.isconfirmed switch denominator // Log an "error" message when the `denominator` is 0. 0.0 => log.error("Division by 0 on confirmed bar!\nBar excluded from the average.") // Otherwise, log an "info" message containing a formatted representation of the variables' confirmed values. => log.info("Values (Confirmed): \nnumerator: {0,number,#.########} \ndenominator: {1,number,#.########} \nratio: {2,number,#.########} \naverage: {3,number,#.########}", numerator, denominator, ratio, average) else switch denominator // Log an "error" message for the unconfirmed bar when the `denominator` is 0. 0.0 => log.error("Division by 0 on unconfirmed bar!") // Otherwise, log a "warning" message containing a formatted representation of the unconfirmed values. => log.warning("Values (unconfirmed): \nnumerator: {0,number,#.########} \ndenominator: {1,number,#.########} \nratio: {2,number,#.########} \naverage: {3,number,#.########}", numerator, denominator, ratio, average)

Note that:

- Programmers can use barstate.isconfirmed in the conditions that trigger `log.*()` calls to allow logs for any realtime bar only once, on its *closing tick*, as shown in the example code.
- Users can pause realtime logs by selecting the “Disable logging” button at the top of the Pine Logs pane.
- Allowing logging on any tick of an open bar can result in a large number of logged messages over time. Therefore, we recommend including unique information in the messages or using different logging levels for easy *filtering* from the Pine Logs pane.

- The Pine Logs pane can display the most recent 10,000 logs for historical bars. If a programmer needs to view earlier logs, they can add logic in the code to filter specific `log.*()` calls. See the Custom code filters section for an example.

The following sections use the example script above to demonstrate the Pine Logs pane's log inspection and filtering features.

Inspecting logs

When a script generates a log by calling any `log.*()` function call, the Pine Logs pane automatically prefixes the logged message with an ISO 8601 *timestamp* representing the log's assigned time, expressed in the chart's time zone. The timestamp prefixed to a log on a historical bar represents the bar's *opening time*, whereas the timestamp for a realtime log represents the *system time* of the log event:

image

Additionally, each log includes "Source code" and "Scroll to bar" options, which appear when hovering over the message in the Pine Logs pane. These features provide convenient ways for users to inspect and verify a log's conditions:

image

The "Source code" option opens the script in the Pine Editor and highlights the code line containing the specific `log.*()` call that triggered the log event:

image

The "Scroll to bar" option navigates the chart to the bar where the `log.*()` call occurred, then displays a temporary label above the bar, containing its date and time information:

image

Note that:

- The label's time information depends on the chart's timeframe. For example, the label on a "1D" chart contains only the weekday and date, whereas the label on an intraday chart also includes the time of day.

It's important to note that every script on the chart that generates logs maintains an *independent* log history. The Pine Logs pane shows logs for only *one* script at a time. To inspect the logs from a specific script when multiple are on the chart, select its title from the dropdown menu at the top of the pane:

image

Filtering logs

The Pine Logs pane displays up to 10,000 logged messages from script executions on historical bars. It then appends a new log for each `log.*()` call executed on *any* realtime tick.

To help users navigate high volumes of logs efficiently, the pane includes *filters* that isolate logs based on logging level, start date and time, or search queries. Users can apply these log filters individually or in combination to show only the messages that meet specific criteria. The filters are accessible from the icons below the "x" in the top-right portion of the pane:

image

For custom filtering options, programmers can use conditional logic to activate specific `log.*()` calls selectively across a script's executions. See the Custom code filters section below to learn more.

Logging level

Selecting the rightmost icon above the messages in the Pine Logs pane opens a "Filter levels" dropdown menu containing checkboxes for each logging level ("Info", "Warning", and "Error"). To remove logs with a

specific logging level from the displayed results, uncheck the level from this menu.

In the example below, we deactivated the “info” and “warning” levels for our script’s logs, allowing only “error” messages in the Pine Logs pane:

image

Note that:

- Deactivating logging levels in this menu hides the relevant messages but *does not* stop the execution of those `log.*()` calls in the code. For instance, a log.info() call still executes and adds to the historical log count even when the “Info” option is unchecked.

Start date

The “Start date” option above the logs in the Pine Logs pane opens a dialog box where users can specify a starting date and time to filter the displayed messages:

image

After the user sets the filter in the dialog box, a tag showing the selected date and time appears above the logs, indicating it is active. With this filter, only logs with prefixed timestamps from the specified start point onward appear in the Pine Logs pane:

image

Character and pattern search

The “Search” option above the logs in the Pine Logs pane opens a search bar where users can *match* logs containing specific character sequences or patterns, similar to the Pine Editor’s “Find/Replace” tool for matching code.

When the search bar is not empty, the pane shows only the messages that fully or partially match the text or pattern, with the matched portion of each message highlighted in blue for visual reference.

Below, we searched “Confirmed” to identify all logs from our example script that contain the term anywhere in their text:

image

Note that:

- The filtered results include logs containing “confirmed” with a lowercase “c” because the search filter performs *case-insensitive* matching on ASCII characters by default.
- The results also include logs containing “unconfirmed” because the default filter behavior does not exclusively match *whole-word* terms.

The rightmost icon in the search bar opens a dropdown menu containing three options to adjust the search filter’s behavior: Match case, Whole word, and Regex:

image

Match case

The “Match case” search option activates *case-sensitive* matching. With this setting, the filter’s results include only the logs containing the search query with identical cases for ASCII letter characters.

Here, we enabled the “Match case” setting for our “Confirmed” search, preventing all the script’s logs containing “confirmed” with a *lowercase* “c” from appearing in the results:

image

Note that:

- The “Match case” setting *does not* affect the search behavior for Unicode letter characters outside the ASCII range (U+0000 - U+007F).

Whole word

The “Whole word” search option activates *whole-word* matching. With this setting enabled, the filter includes logs containing the searched term, but only if it is separated from other text by *whitespace* characters or any of the following non-word characters: `.` (period), `,` (comma), `:` (colon), `;` (semicolon), `'` (apostrophe), or `"` (quotation mark).

For example, searching for “Confirmed” in our script’s logs with the “Whole word” setting prevents the messages containing “unconfirmed” from appearing in the results:

image

Note that:

- With the “Whole word” setting active, the search filter *cannot* match terms containing whitespaces or the other non-word characters listed above.
- Whole-word search queries can include other Unicode characters outside the ASCII range.

Regex

The “Regex” search option enables advanced, flexible log filtering with regular expressions (regex). In contrast to plain text searches, which only match *literal* character sequences, regex searches can match variable *text patterns* based on the rules defined by the query’s syntax.

With regular expressions, the Pine Logs search filter can isolate logs containing various text structures, simple or complex, such as dates and times with a defined format, alphanumeric sequences with varying digits or letters, sequences of characters within specified Unicode subsets, and more.

For instance, this regex search query specifies that the displayed logs must contain “average:”, with optional trailing whitespace characters,

followed by a sequence of characters representing a number greater than 0.5 and less than or equal to 1.0:

```
average:\s*(?:0\.5\d*[1-9]\d*|0\.[6-9]\d*|(?:1\.0*|1))
```

image

The more advanced search query below specifies that the logs must contain prefixed timestamps representing any time of day equal to or after 09:30 and before 16:00 in the chart's time zone:

```
(?<=^\[\d{4}-\d{2}-\d{2}\x54)(?:09:3\d:[0-5]\d\.\d{3}|1[1-5]:(?:[0-
```

image

For more information about regular expressions, consult the Regex syntax reference in this manual's Strings page. Most of the described syntax works the same within the Pine Logs search filter, with a few notable differences:

- The strings used as regex arguments in str.match() calls require *two* consecutive backslashes (`\\`) for specifying escape sequences in the pattern (e.g., `"\\w"` means the regex matches a character from the `\w` class). In contrast, the Pine Logs search filter requires only a *single* backslash for escape sequences. Double backslashes in the search bar match the literal `\` character.
- The regex search query can use the syntax `\xhh` or `\uhhhh` to reference Unicode code points in the Basic Multilingual Plane, where each h is a *hexadecimal* digit (e.g., `\x67` and `\u0067` refer to U+0067, the a character). However, the full-range syntax (`\x{...}`) is not supported.
- The search query *cannot* use Unicode property references, such as `\p{Lu}`, `\p{IsGreek}`, etc.
- The search query can use only the `^` and `$` *boundary assertions* to match a logged message's start and end boundaries. The `\A`, `\Z`, and `\z` assertions are *not* supported.
- The search query cannot use *pattern modifiers* globally (e.g., `(?m)^abc`). However, it can use some modifiers locally inside *non-capturing groups* (e.g., `(?m:^abc)`).

Custom code filters

If the filtering options in the Pine Logs pane are not sufficient, programmers can control specific `log.*()` calls using inputs and conditional logic.

The script below calculates an RMA of close prices and creates a compound condition from four distinct individual conditions. It plots the RMA on the chart and highlights the background when the `compoundCondition` value is `true`. For debugging, the script uses log.info() to display a formatted string representing the `close` and `rma` values, the values of all the "bool" variables that form the compound condition, and the final `compoundCondition` value.

The `filterLogsInput`, `logStartInput`, and `logEndInput` variables define a custom *time filter* for generating logs. When `filterLogsInput` is `true`, the script uses the time inputs assigned to `logStartInput` and `logEndInput` to filter the log.info() calls, allowing a new log only when the bar's time is within the specified range:

image

//@version=6 indicator("Custom code filters demo", overlay = true) //@variable The length for moving average calculations. int lengthInput = input.int(20, "Length", 2) //@variable If `true`, only allows logs within the input time range. bool filterLogsInput = input.bool(true, "Only log in time range", group = "Log filter") //@variable The starting time for logs if `filterLogsInput` is `true`. int logStartInput = input.time(0, "Start time", group = "Log filter", confirm = true) //@variable The ending time for logs if `filterLogsInput` is `true`. int logEndInput = input.time(0, "End time", group = "Log filter", confirm = true) //@variable The RMA of `close` prices. float rma = ta.rma(close, lengthInput) //@variable Is `true` when `close` exceeds the `rma`. bool priceBelow = close <= rma //@variable Is `true` when the current `close` is greater than the max of the previous `hl2` and `close`. bool priceRising = close > math.max(hl2[1], close[1]) //@variable Is `true` when the `rma` is positively accelerating. bool rmaAccelerating = rma - 2.0 * rma[1] + rma[2] > 0.0 //@variable Is `true` when the difference between `rma` and `close` exceeds 2 times the current ATR. bool closeAtThreshold =

```
rma - close > ta.atr(lengthInput) * 2.0 //@variable Is `true` when all the above conditions occur. bool compoundCondition = priceBelow and priceRising and rmaAccelerating and closeAtThreshold // Plot the `rma`. plot(rma, "RMA", color.teal, 3) // Highlight the chart background when the `compoundCondition` occurs. bgcolor(compoundCondition ? color.new(color.aqua, 80) : na, title = "Compound condition highlight") //@variable If `filterLogsInput` is `true`, is only `true` in the input time range. Otherwise, always `true`. bool showLog = filterLogsInput ? time >= logStartInput and time <= logEndInput : true // Log results for a confirmed bar when `showLog` is `true`. if barstate.isconfirmed and showLog
log.info( "\nclose:                    {0,number,#.#####}
\nrma:                    {1,number,#.#####}
\npriceBelow:    {2} \npriceRising:    {3}
\nrmaAccelerating:    {4} \ncloseAtThreshold:    {5} \n
\ncompoundCondition:    {6}", close, rma, priceBelow, priceRising,
rmaAccelerating, closeAtThreshold, compoundCondition )
```

Note that:

- The `input.*()` calls assigned to the `filterLogsInput`, `logStartInput`, and `logEndInput` variables include a `group` argument to group the inputs in the “Settings/Inputs” tab.
- Users can adjust time input values directly on the chart by selecting the script’s status line and moving the displayed time markers with the mouse pointer. Additionally, users can select “Reset points” in the script’s “More” menu to clear the inputs and choose new values.
- The `formatString` argument of the log.info() call uses the Em Space character (U+2003) to align the represented values vertically in the logged text. In contrast to the standard space and tab characters, leading or repeated Em and En spaces are *not* removed from the Pine Logs pane’s displayed messages.

Pine drawings

Pine’s drawing types create chart drawings with specified properties. Scripts can place drawings at any valid chart location during code

executions on any bar. Programmers can use these types in a script's global or local scopes to visualize numeric data, conditions, colors, and strings on the chart. The flexibility of Pine drawings makes them helpful for debugging scripts when other methods do not suffice, namely when a programmer wants to inspect information graphically outside the Pine Logs pane.

However, before debugging a script using drawings, it is crucial to note the following limitations:

- The `expression` argument of a `request.*()` call *cannot* depend on code that creates or modifies drawings. Likewise, an indicator that specifies another context in its declaration statement cannot create drawings from *anywhere* in the code. To debug code that executes on requested data, use Pine Logs instead.
- In contrast to Pine Logs, drawings do not have built-in navigation features. Therefore, users must manually scroll across the chart to inspect drawings created on specific bars.
- Scripts can maintain only a *limited* number of objects of each drawing type. When the number of drawings exceeds the limit, Pine's garbage collector automatically *removes* the oldest ones.

The sections below explain some simple debugging methods using labels and tables. These drawings, especially *labels*, are the most effective for on-chart debugging because they can use dynamic strings to express information from other data types as custom text.

Labels

Labels display colored shapes and text at specified chart coordinates. In contrast to the outputs of the plotshape() and plotchar() functions, labels can display text from "series string" values that change across script executions. Programmers often use labels to visualize the logic of conditional structures and show text representing information from a script's global or local scopes.

The most common techniques for debugging with labels include:

- Drawing a label containing key information anchored to *every* bar that requires inspection.
- Drawing a single label containing information from specific executions at the *end* of the dataset or visible chart.

Drawing on successive bars

When inspecting values of varying magnitudes or different types across bars, a simple approach is to create formatted strings containing the necessary debug information and display them in labels on each bar requiring analysis.

In this example, we've modified the "Average bar ratio" script from the Pine Logs section above. Instead of creating formatted text and displaying information using `log.*()` function calls, this script formats the values separately, then calls label.new() to show the results on the chart within labels anchored to each bar's high:

image

//@version=6 indicator("Drawing on successive bars demo", "Average bar ratio") //@variable The current bar's change from the `open` to `close`. float numerator = close - open //@variable The current bar's `low` to `high` range. float denominator = high - low //@variable The ratio of the bar's open-to-close change to its full range. float ratio = numerator / denominator //@variable The average `ratio` over 10 *non-na* values. float average = ta.sma(ratio, 10) // Plot the `average`. plot(average, "average", color.purple, 3) if barstate.isconfirmed if denominator == 0 string debugText = "Division by 0 on confirmed bar! \nBar excluded from the average." label.new(bar_index, high, debugText, color = color.red, textcolor = #000000, force_overlay = true) else string debugText = str.format("Values (Confirmed): \nnumerator: {0,number,#.########} \ndenominator: {1,number,#.########} \nratio: {2,number,#.########} \naverage: {3,number,#.########}", numerator, denominator, ratio, average) label.new(bar_index, high, debugText, textcolor = #ffffff, force_overlay = true) else if denominator == 0 string debugText

= "Division by 0 on unconfirmed bar!" label.new(bar_index, high, debugText, color = color.red, textcolor = #000000, force_overlay = true) else string debugText = str.format("Values (unconfirmed): \nnumerator: {0,number,#.########} \ndenominator: {1,number,#.########} \nratio: {2,number,#.########} \naverage: {3,number,#.########}", numerator, denominator, ratio, average) label.new(bar_index, high, debugText, color = color.orange, textcolor = #000000, force_overlay = true)

Note that:

- The label.new() calls include `force_overlay = true`, meaning the labels always appear on the main chart pane.
- Unlike the example in the Pine Logs section, this script's outputs **are** subject to *rollback*, meaning the information shown on a bar reflects only the bar's *latest* data. The script does **not** show information for *all* realtime bar updates.

The above example allows users to inspect the script's confirmed values or latest updates on any bar that has a label drawing. However, each bar's results are legible only when the labels do not *overlap*.

An alternative, more *compact* way to display text with labels on successive bars is to utilize the label.new() function's `tooltip` parameter instead of the `text` parameter, as labels show their tooltips only when the mouse pointer *hovers* over them.

In the script version below, we changed all the label.new() calls to use debugText as the `tooltip` argument instead of the `text` argument. Now, we can view a specific bar's information without visual clutter from other nearby labels:

image

//@version=6 indicator("Drawing tooltips on successive bars demo", "Average bar ratio") //@variable The current bar's change from the `open` to `close`. float numerator = close - open //@variable The current bar's `low` to `high` range. float denominator = high - low // @variable The ratio of the bar's open-to-close change to its full range. float ratio = numerator / denominator //@variable The average `ratio`

over 10 *non-na* values. float average = ta.sma(ratio, 10) // Plot the `average`. plot(average, "average", color.purple, 3) if barstate.isconfirmed if denominator == 0 string debugText = "Division by 0 on confirmed bar!\nBar excluded from the average." label.new(bar_index, high, color = color.red, tooltip = debugText, force_overlay = true) else string debugText = str.format("Values (Confirmed): \nnumerator: {0,number,#.########} \ndenominator: {1,number,#.########} \nratio: {2,number,#.########} \naverage: {3,number,#.########}", numerator, denominator, ratio, average) label.new(bar_index, high, tooltip = debugText, force_overlay = true) else if denominator == 0 string debugText = "Division by 0 on unconfirmed bar!" label.new(bar_index, high, color = color.red, tooltip = debugText, force_overlay = true) else string debugText = str.format("Values (unconfirmed): \nnumerator: {0,number,#.########} \ndenominator: {1,number,#.########} \nratio: {2,number,#.########} \naverage: {3,number,#.########}", numerator, denominator, ratio, average) label.new(bar_index, high, color = color.orange, tooltip = debugText, force_overlay = true)

When drawing labels across successive bars, it’s important to note that the maximum number of labels a script can display is 500. As such, the examples above allow users to inspect information for only the most recent 500 chart bars.

For successive labels on earlier bars, programmers can create conditional logic that limits the drawings to specific *time ranges*, e.g.:

```
if time >= startTime and time <= endTime
    <create_drawing_id>
```

Below, we added a condition to the script that draws a label only when the bar’s time is between the chart.left_visible_bar_time and chart.right_visible_bar_time values. This logic restricts the drawings to *visible* chart bars, allowing us to scroll through the chart and inspect labels on any bar:

image

```
//@version=6 indicator("Drawing in visible ranges demo", "Average bar ratio") //@variable The current bar's change from the `open` to `close`. float numerator = close - open //@variable The current bar's `low` to `high` range. float denominator = high - low //@variable The ratio of the bar's open-to-close change to its full range. float ratio = numerator / denominator //@variable The average `ratio` over 10 *non-na* values. float average = ta.sma(ratio, 10) // Plot the `average`. plot(average, "average", color.purple, 3) if time >= chart.left_visible_bar_time and time <= chart.right_visible_bar_time if barstate.isconfirmed if denominator == 0 string debugText = "Division by 0 on confirmed bar!\nBar excluded from the average." label.new(bar_index, high, color = color.red, tooltip = debugText, force_overlay = true) else string debugText = str.format( "Values (Confirmed): \nnumerator: {0,number,#.########} \ndenominator: {1,number,#.########} \nratio: {2,number,#.########} \naverage: {3,number,#.########}", numerator, denominator, ratio, average ) label.new(bar_index, high, tooltip = debugText, force_overlay = true) else if denominator == 0 string debugText = "Division by 0 on unconfirmed bar!" label.new(bar_index, high, color = color.red, tooltip = debugText, force_overlay = true) else string debugText = str.format( "Values (unconfirmed): \nnumerator: {0,number,#.########} \ndenominator: {1,number,#.########} \nratio: {2,number,#.########} \naverage: {3,number,#.########}", numerator, denominator, ratio, average ) label.new(bar_index, high, color = color.orange, tooltip = debugText, force_overlay = true)
```

Note that:

- The script *restarts* each time the UNIX timestamps of the chart.left_visible_bar_time or chart.right_visible_bar_time variables change after the user scrolls or zooms on the chart.

Drawing at the end of the chart

When debugging information does not change frequently across executions, or only the information from a specific execution requires

inspection, programmers often display it using labels anchored to the *end* of the chart.

The following example displays price and chart information in four separate labels at the end of the chart. The script's `printLabel()` function renders a specified string in a label that always anchors to the last available time in the dataset, regardless of when the function call occurs:

image

//@version=6 indicator("Drawing labels at the end of the chart demo", "Chart info", true) //@function Draws a label to display the `info` text at the latest available time. // Each instance of a call to this function updates its label text across executions. //@param info The string to display. //@param price Optional. The y-coordinate of the label. If `na`, the function draws the label above the last bar. // The default is `na`. // @param textColor Optional. The color of the displayed text. If `na`, the label uses `chart.fg_color`. // The default is `na`. //@param size Optional. The size of the label in typographic points. The default is 18. //@returns A `label` object with dynamic text. printLabel(string info, simple float price = na, simple color textColor = na, simple int size = 18) => var int anchorTime = math.max(last_bar_time, chart.right_visible_bar_time) var color col = nz(textColor, chart.fg_color) var yloc = na(price) ? yloc.abovebar : yloc.price var label result = label.new(anchorTime, price, na, xloc.bar_time, yloc, na, label.style_none, col, size, force_overlay = true) result.set_text(info) // Call `printLabel()` on the first bar to display "Chart info:" and formatted chart information. if barstate.isfirst printLabel("Chart info:" + str.repeat("\n", 6), textColor = color.teal) printLabel(str.format("Symbol: {0}, Type: {1}, Timeframe: {2}\nStandard chart: {3}, Replay active: {4}", ticker.standard(), syminfo.type, timeframe.period, chart.is_standard, str.contains(syminfo.tickerid, "replay")) + str.repeat("\n", 3)) // On the last available bar, call `printLabel()` to display the latest OHLCV values and total bar count. if barstate.islast printLabel(str.format("O: {0,number,#.#####}, H: {1,number,#.#####}, L: {2,number,#.#####}, C: {3,number,#.#####}, V: {4}", open, high,

low, close, str.tostring(volume, format.volume)) + "\n")
printLabel("Total bars: " + str.tostring(bar_index + 1))

Note that:

- The `printLabel()` function draws *one* label per function call instance. The label's x property is the maximum of the last_bar_time and chart.right_visible_bar_time values, ensuring it appears above the last available bar.
- On each execution of a `printLabel()` instance, the label's `text` property updates to reflect the latest `info` value.
- The label.new() call in the `printLabel()` function includes `force_overlay = true`, meaning the drawing always appears in the main chart pane.
- This script uses *four* distinct `printLabel()` calls. The first three append repeated newline characters (`\n`) in the `info` argument to prevent the label text from overlapping.

Tables

Tables display text within cells arranged in columns and rows at fixed locations in the chart pane's visual space. In contrast to other drawing types, which create visuals on the chart at specified *coordinates*, tables appear at one of nine unique, *bar-agnostic* locations defined by the `table.position_*` constants.

Because tables appear at consistent relative locations in the pane, unaffected by scroll or zoom actions, programmers occasionally use them for on-chart debugging. The most common technique is to draw a *single-cell* table containing information from specific script executions.

This example contains a `printTable()` function that calls table.new() and table.cell() to create a single-cell table that displays dynamic text in a relative location on the main chart pane. The script uses a single call to this function to display the same chart information shown by the example script from the previous section:

image

//@version=6 indicator("Debugging with single-cell tables demo", "Chart info", true) //@function Draws a single-cell table to display the `info` text in the top-right corner of the chart. //@param info The string to display. //@param textColor Optional. The color of the displayed text. If `na`, the table uses `chart.fg_color`. // The default is `na`. //@param size Optional. The size of the table's text in typographic points. The default is 18. //@returns A single-cell table with dynamic text. printTable(string info, simple color textColor = na, simple int size = 18) => var color col = nz(textColor, chart.fg_color) var table result = table.new(position.top_right, 1, 1, na, force_overlay = true) table.cell(result, 0, 0, info, text_color = col, text_size = size) // Call `printTable()` on the latest available bar to display chart information in the top-right corner. if barstate.islast printTable(str.format("Chart info: \n\nSymbol: {0}, Type: {1}, Timeframe: {2}\nStandard chart: {3}, Replay active: {4} \n\nO: {5,number,#.#####}, H: {6,number,#.#####}, L: {7,number,#.#####}, C: {8,number,#.#####}, V: {9} \nTotalBars: {10}", ticker.standard(), syminfo.type, timeframe.period, chart.is_standard, str.contains(syminfo.tickerid, "replay"), open, high, low, close, str.tostring(volume, format.volume), bar_index + 1))

Note that:

- Every new table drawing *replaces* any existing one that has the same specified position. Therefore, scripts cannot call the `printTable()` function multiple times to place multiple drawings in a single location, unlike the `printLabel()` function from the previous section.
- This script calls `printTable()` only on the last historical bar and all realtime bars because updating tables on each historical bar is an unnecessary use of runtime resources. See the Reducing drawing updates section of the Profiling and optimization page for more information.

Plots and chart colors

The built-in `plot*()` functions display results from a value's series in up to four locations: the chart pane, the script's status line, the Data

Window, and the price scale. Programmers often use these output functions as a quick way to display the history of a script's numeric values, conditions, and colors. Two other functions, bgcolor() and barcolor(), color a chart pane's background and the main chart's bars or candles. Although not as versatile as other output functions, they offer a quick way to display conditions and colors on the chart.

All these functions, especially plot(), plotchar(), and plotshape(), can serve as helpful tools for debugging a script's calculations and logic. For instance, the outputs of a single plot() call can show the complete available history of a script's series on the chart and provide information for any bar in other locations.

Before using plots or chart colors for debugging, it is important to note the following limitations:

- Unlike Pine Logs or drawings, these outputs *cannot* display results for values that are accessible from local scopes only. Scripts must extract values from local scopes into the *global scope* to debug them with plots or chart colors.
- The only `plot*()` functions that can display *text* on the chart — plotchar() and plotshape() — require "const string" values. Therefore, they cannot display dynamic strings or calculated string conversions of other types.
- Similar to drawings, plots *do not* have built-in navigation features. Users must scroll across the chart to find plotted information for specific bars.
- The maximum *plot count* for any script is 64. Each call to these functions contributes a different number to the total, depending on its arguments. See the Plot limits section of the Limitations page to learn more.

Plotting numbers

One of the simplest methods to inspect global numeric series ("int" or "float" values) is to plot them using the plot(), plotchar(), or plotshape() function. The outputs on the chart pane provide a graphical view of the series' history. The other possible output locations (status line, price

scale, and Data Window) show formatted numbers representing the values calculated on a specific bar.

Let’s look at a simple debugging example. The following script calculates a custom oscillator whose value is the average of three separate oscillators. It displays the `oscillator` value in four output locations using a plot() call:

image

//@version=6 indicator("Plotting numbers demo") //@variable The length of each oscillator. int lengthInput = input.int(20, "Length", 2) // @variable The correlation between `close` and `bar_index` over `lengthInput` bars. float osc1 = ta.correlation(close, bar_index, lengthInput) //@variable The RSI of `close` over `lengthInput` bars, scaled to the range [-1, 1]. float osc2 = (ta.rsi(close, lengthInput) - 50) / 50 //@variable The percent rank of `close` compared to `lengthInput` past values, scaled to the range [-1, 1]. float osc3 = (ta.percentrank(close, lengthInput) - 50) / 50 //@variable The average of `osc1`, `osc2`, and `osc3`. float oscillator = math.avg(osc1, osc2, osc3) // Plot the `oscillator`. plot(oscillator, "Combined oscillator", color.purple, 3)

The above script’s outputs allow inspection of the final `oscillator`, but not the three constituent oscillators that determine its value. Because the script calculates all three series in the global scope, we can inspect them using additional plots. Here, we add three plot() calls to the script to display each oscillator, allowing us to verify the script’s calculated values and understand how they affect the final result:

image

//@version=6 indicator("Plotting numbers demo") //@variable The length of each oscillator. int lengthInput = input.int(20, "Length", 2) // @variable The correlation between `close` and `bar_index` over `lengthInput` bars. float osc1 = ta.correlation(close, bar_index, lengthInput) //@variable The RSI of `close` over `lengthInput` bars, scaled to the range [-1, 1]. float osc2 = (ta.rsi(close, lengthInput) - 50) / 50 //@variable The percent rank of `close` compared to `lengthInput` past values, scaled to the range [-1, 1]. float osc3 =

(ta.percentrank(close, lengthInput) - 50) / 50 //@variable The average of `osc1`, `osc2`, and `osc3`. float oscillator = math.avg(osc1, osc2, osc3) // Plot the `oscillator`. plot(oscillator, "Combined oscillator", color.purple, 3) // Plot the `osc1`, `osc2`, and `osc3` series for inspection. plot(osc1, "osc1", color.red, 2, plot.style_circles, join = true) plot(osc2, "osc2", color.maroon, 2, plot.style_circles, join = true) plot(osc3, "osc3", color.blue, 2, plot.style_circles, join = true)

Note that:

- The numbers in the script's status line and the Data Window represent the values plotted on the bar at the mouse pointer's location. When the pointer is *not* on the chart, these numbers represent the *latest* bar's data.
- The labels in the price scale show the latest *non-na* values available in the plotted series up to the last visible bar. If a plotted series does not have a non-na value at any point before that bar, the price scale does *not* show a label for it.

Plotting without affecting the scale

Debugging multiple numeric series by plotting them on the chart can make the results hard to read if the plots affect the price scale, especially if each plotted series has a significantly different value range. Programmers can specify a plot's *display locations* to avoid distorting the scale by passing a `display.*` constant or expression to the `display` parameter of the `plot*()` call.

Let's look at a simple example that calculates a few numeric series with different ranges. This script calculates a weighted moving average with custom weights and plots the result on the chart:

image

//@version=6 indicator("Plotting without affecting the scale demo", "Weighted average", true, precision = 5) //@variable The number of bars in the average. int lengthInput = input.int(20, "Length", 1) //@variable The weight applied to the price on each bar. float weight = math.pow(close - open, 2) //@variable The numerator of the average.

float numerator = math.sum(weight * close, lengthInput) //@variable The denominator of the average. float denominator = math.sum(weight, lengthInput) //@variable The weighted average over `lengthInput` bars. float average = numerator / denominator // Plot the `average`. plot(average, "Weighted average", linewidth = 3)

Note that:

- This script includes `precision = 5` in the indicator() declaration statement, which specifies that it plots numbers with five fractional digits instead of using the chart's default precision setting.

Suppose we want to inspect *all* the values in the `average` calculation using plots. If we use `plot*()` functions with the default `display` argument (display.all), the plotted results appear in all possible locations, including the chart pane. Unlike the example script from the Plotting numbers section, this script's visuals become hard to read in the pane because each plot has a significantly different range:

image

//@version=6 indicator("Plotting without affecting the scale demo", "Weighted average", true, precision = 5) //@variable The number of bars in the average. int lengthInput = input.int(20, "Length", 1) //@variable The weight applied to the price on each bar. float weight = math.pow(close - open, 2) //@variable The numerator of the average. float numerator = math.sum(weight * close, lengthInput) //@variable The denominator of the average. float denominator = math.sum(weight, lengthInput) //@variable The weighted average over `lengthInput` bars. float average = numerator / denominator // Plot the `average`. plot(average, "Weighted average", linewidth = 3) // Create debug plots for the `weight`, `numerator`, and `denominator`. plot(weight, "weight", color.purple) plot(numerator, "numerator", color.teal) plot(denominator, "denominator", color.maroon)

We can change the `display` argument in each debug plot() call to view all the calculated values while preserving the chart's scale. Below, we set the argument to `display.all - display.pane`, meaning all the debug plots show information in all locations *except* the chart pane.

Now, we can visualize how the calculated values affect each bar's average result without distorting the scale:

image

//@version=6 indicator("Plotting without affecting the scale demo", "Weighted average", true, precision = 5) //@variable The number of bars in the average. int lengthInput = input.int(20, "Length", 1) //@variable The weight applied to the price on each bar. float weight = math.pow(close - open, 2) //@variable The numerator of the average. float numerator = math.sum(weight * close, lengthInput) //@variable The denominator of the average. float denominator = math.sum(weight, lengthInput) //@variable The weighted average over `lengthInput` bars. float average = numerator / denominator // Plot the `average`. plot(average, "Weighted average", linewidth = 3) //@variable The display locations of all debug plots. debugLocations = display.all - display.pane // Create debug plots for the `weight`, `numerator`, and `denominator`. plot(weight, "weight", color.purple, display = debugLocations) plot(numerator, "numerator", color.teal, display = debugLocations) plot(denominator, "denominator", color.maroon, display = debugLocations)

Note that:

- The `display.*` constants support *addition* and *subtraction* operations for customized display settings. This script uses subtraction to remove display.pane from the output locations allowed by display.all. Operations that remove valid display constants more than once do not cause errors. For instance, this script produces the *same* outputs if it subtracts display.pane once, twice, or more times in the `debugLocations` expression.

Plotting and coloring conditions

Programmers can inspect a script's conditions ("bool" values) with the `plot*()`, bgcolor(), and barcolor() functions in several ways, including:

- Using the "bool" condition as the `series` argument in a plotshape() or plotchar() call. The call creates a shape/character

with specified text on the chart when the condition is `true`, and it shows a *numeric text* representation of the condition in the status line and Data Window (1 for `true` and `0` for `false`).
- Creating a logical expression that returns different “int” or “float” values for the condition’s `true` and `false` states, then using the result as the `series` argument in a `plot*()` call. When using plotchar() or plotshape(), note that these functions show visuals on the chart only when the `series` value is not na or 0.
- Creating a logical expression that returns different “color” values based on the condition’s `true` or `false` state, then using the result to color the chart with bgcolor() or barcolor(), or to color a plot or fill.

The following example uses the above methods to debug a simple condition. The script calculates an RSI with an input length and defines a `crossBelow` condition that is `true` when the RSI crosses 30. It uses plotshape(), plotchar(), and bgcolor() calls to visualize the `crossBelow` condition in different ways:

image

//@version=6 indicator("Plotting and coloring conditions demo") // @variable The length of the RSI. int lengthInput = input.int(14, "Length", 1) //@variable The calculated RSI value. float rsi = ta.rsi(close, lengthInput) //@variable Is `true` when the `rsi` crosses below 30, `false` otherwise. bool crossBelow = ta.crossunder(rsi, 30.0) // Plot the `rsi`. plot(rsi, "RSI", color.rgb(136, 76, 146), linewidth = 3) // Plot a circle near the top of the pane when `crossBelow` is `true`. // The status line and Data Window show 1 when the condition is `true` and 0 when it is `false`. plotshape(crossBelow, "plotshape debug", shape.circle, location.top, color.red, size = size.small) // Plot the `⤭` character at the `rsi` value when `crossBelow` is `true`. // The status line and Data Window show the `rsi` value when the condition is `true` and `na` when it is `false`. plotchar(crossBelow ? rsi : na, "plotchar debug", "⤭", location.absolute, color.maroon, size = size.normal) // Highlight the background when `crossBelow` is `true`. Does not add information to the status line or Data Window. bgcolor(crossBelow ? color.new(color.red, 60) : na, title = "bgcolor debug")

Note that:

- The `plot*()` functions that display text or shapes on the chart — plotshape(), plotchar(), and plotarrow() — *do not* display data in the price scale.
- The plotshape() call uses `crossUnder` as its `series` argument. The chart pane shows a shape at the top when the condition occurs. The status line and Data Window show 1 when the `series` is `true` and 0 when it is `false`.
- The plotchar() call plots the result of a ternary expression that returns the `rsi` when `crossUnder` is `true` and na otherwise. It shows the character U+2930 at the `rsi` location when the expression does not evaluate to na. Because the `series` argument is a "float" value, the number in the status line and Data Window represents that value directly.
- The bgcolor() call highlights the chart's background when `crossUnder` is `true`, but it does not display information in the status line or Data Window.

The plotshape() and plotchar() functions have a `text` parameter that adds "const string" text to the plotted shapes/characters. When debugging multiple global conditions, it is often helpful to call these functions with `text` arguments to label each condition for simple on-chart inspection. The arguments can contain the newline character (`\n` escape sequence), allowing scripts to plot multiple shapes in identical locations with non-overlapping text.

Let's explore a debugging example using this approach. The script below calculates an RSI and its median over `lengthInput` bars. Then, it creates five singular conditions and uses them to form a *compound condition*. The script plots the `rsi` and `median` values with the `plot()` function, and it colors the background with bgcolor() when the `compoundCondition` is `true`:

image

//@version=6 indicator("Plotting and coloring compound conditions demo") //@variable The length of the RSI and median RSI calculations. int lengthInput = input.int(14, "Length", 2) //@variable The RSI of

`close` with a smoothing factor defined by `lengthInput`. float rsi = ta.rsi(close, lengthInput) //@variable The median of the `rsi` over `lengthInput` bars. float median = ta.median(rsi, lengthInput) //@variable Condition #1: Is `true` when the 1-bar `rsi` change switches from 1 to -1. bool changeNegative = ta.change(math.sign(ta.change(rsi))) == -2 //@variable Condition #2: Is `true` when the previous bar's `rsi` is greater than 70. bool prevAbove70 = rsi[1] > 70.0 //@variable Condition #3: Is `true` when the current `close` is lower than the previous bar's `open`. bool closeBelow = close < open[1] //@variable Condition #4: Is `true` when the `rsi` is between 60 and 70. bool betweenLevels = bool(math.max(70.0 - rsi, 0.0) * math.max(rsi - 60.0, 0.0)) //@variable Condition #5: Is `true` when the `rsi` is above the `median`. bool aboveMedian = rsi > median //@variable Is `true` when the first condition occurs alongside conditions 2 and 3 or 4 and 5. bool compundCondition = changeNegative and ((prevAbove70 and closeBelow) or (betweenLevels and aboveMedian)) //Plot the `rsi` and the `median`. plot(rsi, "RSI", color.teal, 3) plot(median, "RSI Median", color.gray, 2) // Highlight the background red when the `compundCondition` occurs. bgcolor(compundCondition ? color.new(color.red, 60) : na, title = "compundCondition")

To verify that the script's logic works as intended, we can inspect each of the conditions that affect the final `compoundCondition` value. Below, we added five plotchar() calls to display information for these conditions, each with the same `location` argument. To label the conditions on the chart, each plotchar() call uses a string containing newline characters (\n) and a digit from 1 to 5 as the `text` argument. With these outputs, we can see which sets of conditions trigger each `compoundCondition` occurrence:

image

//@version=6 indicator("Plotting and coloring compound conditions demo") //@variable The length of the RSI and median RSI calculations. int lengthInput = input.int(14, "Length", 2) //@variable The RSI over `lengthInput` bars. float rsi = ta.rsi(close, lengthInput) //@variable The median of the `rsi` over `lengthInput` bars. float median = ta.median(rsi, lengthInput) //@variable Condition #1: Is `true` when

the 1-bar `rsi` change switches from 1 to -1. bool changeNegative = ta.change(math.sign(ta.change(rsi))) == -2 //@variable Condition #2: Is `true` when the previous bar's `rsi` is greater than 70. bool prevAbove70 = rsi[1] > 70.0 //@variable Condition #3: Is `true` when the current `close` is lower than the previous bar's `open`. bool closeBelow = close < open[1] //@variable Condition #4: Is `true` when the `rsi` is between 60 and 70. bool betweenLevels = bool(math.max(70.0 - rsi, 0.0) * math.max(rsi - 60.0, 0.0)) //@variable Condition #5: Is `true` when the `rsi` is above the `median`. bool aboveMedian = rsi > median //@variable Is `true` when the first condition occurs alongside conditions 2 and 3 or 4 and 5. bool compundCondition = changeNegative and ((prevAbove70 and closeBelow) or (betweenLevels and aboveMedian)) //Plot the `rsi` and the `median`. plot(rsi, "RSI", color.teal, 3) plot(median, "RSI Median", color.gray, 2) // Highlight the background red when the `compundCondition` occurs. bgcolor(compundCondition ? color.new(color.red, 60) : na, title = "compundCondition") // Use `plotshape()` to show `compundCondition` values in the status line and Data Window. plotshape(compundCondition, "compundCondition (1 and (2 and 3) or (4 and 5))", color = chart.fg_color, display = display.all - display.pane) // Plot characters on the chart and numbers in the status line and Data Window when conditions 1-5 occur. plotchar(changeNegative, "changeNegative (1)", "", location.top, text = "1", textcolor = chart.fg_color) plotchar(prevAbove70, "prevAbove70 (2)", "", location.top, text = "\n2", textcolor = chart.fg_color) plotchar(closeBelow, "closeBelow (3)", "", location.top, text = "\n\n3", textcolor = chart.fg_color) plotchar(betweenLevels, "betweenLevels (4)", "", location.top, text = "\n\n\n4", textcolor = chart.fg_color) plotchar(aboveMedian, "aboveMedian (5)", "", location.top, text = "\n\n\n\n5", textcolor = chart.fg_color)

Note that:

- The `char` argument of each plotchar() call is an empty string, meaning the function displays its `text` value without a character above it.
- Because each plotchar() call outputs results at the same relative location (`location.top`), we included different numbers of leading

`\n` sequences in the `text` arguments to move the displayed numerals down and ensure they do not overlap.
- The `title` argument of each plotchar() call contains the condition number to distinguish it in the Data Window.
- The plotshape() call's title describes the compound condition's structure in the Data Window.

To learn more about the plotshape() and plotchar() functions and how their outputs differ from labels, refer to the Text and shapes page.

Tips and techniques

The following sections explain several additional tips and helpful techniques for effective Pine Script debugging.

Decomposing expressions

One of the best practices for efficient debugging is to *split expressions*, especially those with multiple calculations or logical operations, into *smaller parts* assigned to separate variables. Decomposing expressions enables programmers to inspect each critical part individually, making it easier to verify calculations or logic and isolate potential issues in the code. Additionally, complex code broken down into smaller parts is typically simpler to read, maintain, and profile.

The following script calculates a custom oscillator representing the smoothed median change in the differences between the close price and two EMAs over different lengths. The script performs all the calculations in a single expression assigned to the `osc` variable. Then, it creates a compound condition in another expression assigned to the `upSignal` variable and uses that variable to trigger order placement commands. The script plots the `osc` series as columns with different colors based on the `upSignal` value:

image

//@version=6 strategy("Decomposing expressions demo") //@variable The length used for the first part of the oscillator. int length1Input = input.int(20) //@variable The length used for the second part of the

oscillator. int length2Input = input.int(40) //@variable Oscillator smoothing length. int smoothingInput = input.int(10) //@variable The maximum of `length1Input` and `length2Input`. int maxLength = math.max(length1Input, length2Input) //@variable The smoothed median change in the differences between `close` and two EMAs over different lengths. float osc = ta.ema(math.avg(ta.change(close - ta.ema(close, length1Input), length1Input), ta.change(close - ta.ema(close, length2Input), length2Input)), smoothingInput) // @variable `true` if `osc` is positive, above the last two-bar average, and below twice the stdev for `maxLength` bars. bool upSignal = osc < 2 * ta.stdev(osc, maxLength) and osc > 0 and math.avg(osc[1], osc[2]) < osc // Plot the `osc` as columns colored based on the `upSignal`. plot(osc, "Custom oscillator", upSignal ? color.aqua : color.gray, style = plot.style_columns) // Place a "Buy" market order when `upSignal` is `true`, and a closing market order when it is `false`. if upSignal strategy.entry("Buy", strategy.long) else strategy.close("Buy")

Because the `osc` and `upSignal` values depend on multiple calculations and conditions, inspecting only the final values does not provide complete information about the script's behaviors. To verify the script's workings, we can decompose the expressions assigned to `osc` and `upCondition` into smaller parts and inspect them individually.

The script version below declares several extra variables to hold *different parts* of the original `osc` and `upCondition` expressions. With this expanded structure, we can inspect each part of the calculations and logic step-by-step using various outputs. In this script, we included a single log.info() call at the end that displays formatted text containing each variable's information in the Pine Logs pane:

image

//@version=6 strategy("Decomposing expressions demo") //@variable The length used for the first part of the oscillator. int length1Input = input.int(20) //@variable The length used for the second part of the oscillator. int length2Input = input.int(40) //@variable Oscillator smoothing length. int smoothingInput = input.int(10) //@variable The maximum of `length1Input` and `length2Input`. int maxLength = math.max(length1Input, length2Input) //#region Split the `osc`

calculations into smaller parts: // 1. Calculate the EMAS over `length1Input` and `length2Input` bars. float ema1 = ta.ema(close, length1Input), float ema2 = ta.ema(close, length2Input) // 2. Calculate the differences between `close` and `ema1` and `ema2`. float diff1 = close - ema1, float diff2 = close - ema2 // 3. Calculate the changes in `diff1` and `diff2` over `length1Input` and `length2Input` bars. float change1 = ta.change(diff1, length1Input), float change2 = ta.change(diff2, length2Input) // 4. Calculate the median of `change1` and `change2`. float medChange = math.avg(change1, change2) // #endregion //@variable The smoothed median change in the differences between `close` and two EMAs over different lengths. float osc = ta.ema(medChange, smoothingInput) //#region Split the `upSignal` calculations and logic into smaller parts: // 1. Assign the calculations in the expression to separate variables. float oscDev = 2 * ta.stdev(osc, maxLength), float pastAvg = math.avg(osc[1], osc[2]) // 2. Assign each singular condition to a separate variable. bool cond1 = osc < oscDev, bool cond2 = osc > 0, bool cond3 = pastAvg < osc // #endregion //@variable Is `true` if `osc` is positive, above the past two-bar average, and below twice its stdev over `maxLength` bars. bool upSignal = cond1 and cond2 and cond3 // Plot the `osc` as columns colored based on the `upSignal`. plot(osc, "Custom oscillator", upSignal ? color.aqua : color.gray, style = plot.style_columns) // Place a "Buy" market order when `upSignal` is `true`, and a closing market order when it is `false`. if upSignal strategy.entry("Buy", strategy.long) else strategy.close("Buy") // Call `log.info()` to display a formatted message containing debug information in the Pine Logs pane. if barstate.isconfirmed log.info("\nema1: {0,number,0.00000}, diff1: {1,number,0.00000}, change1: {2,number,0.00000} \nema2: {3,number,0.00000}, diff2: {4,number,0.00000}, change2: {5,number,0.00000} \nmedChange: {6,number,0.00000}\n\nosc: {7,number,0.00000}\n---- \noscDev: {8,number,0.00000}\npastAvg: {9,number,0.00000} \ncond1: {10}, cond2: {11}, cond3: {12}\n\nupSignal: {13}", ema1, diff1, change1, ema2, diff2, change2, medChange, osc, oscDev, pastAvg, cond1, cond2, cond3, upSignal)

Note that:

- This script declares some extra variables on the *same line*, separated by commas, to reduce the number of lines added to the code.
- The script calls log.info() only when barstate.isconfirmed is `true`, preventing unnecessary logs on the ticks of unconfirmed bars.
- All the placeholders with the `number` modifier in the log.info() call's formatting string include the `0.00000` pattern, which forces the formatted numbers to always show five fractional digits. Refer to the Formatting strings section of the Strings page for more information.
- The Pine Logs pane displays up to 10,000 historical logs. To view earlier logs, add another condition to the if structure that limits the log.info() call to specific bars. See the Custom code filters section above for an example that restricts `log.*()` calls using time inputs.

Extracting data from local scopes

The *scope* of an identifier (e.g., a variable) refers to the part of a script where it is defined and *accessible* during the script's execution.

All identifiers declared outside user-defined functions, methods, loops, conditional structures, or user-defined type and enum type declarations belong to the *global scope*. Identifiers in the global scope are accessible to most *inner* (local) scopes after declaration. Every Pine script has exactly **one** global scope.

All user-defined functions, methods, loops, and conditional structures in a script create unique, separate *local scopes*. All identifiers within a local scope belong exclusively to that scope, meaning their values or references are *inaccessible* to any *outer* or *containing* scope.

A common practice when debugging variables declared in a local scope is to extract their data to an outer scope or the global scope, making it usable in debugging outputs with different scope requirements.

The following sections explain techniques for extracting data from local scopes using return expressions and reference types. We demonstrate these techniques on the following script, which contains a `customMA()` function that calculates a custom adaptive moving average of a `source` series based on the distance from its current value to its 25th and 75th percentiles over `length` bars. The script contains a local function scope, and a nested block scope from the if structure that sets the `outer`Range value:

image

//@version=6 indicator("Extracting from local scopes initial demo", overlay = true) //@variable The number of bars in the `customMA()` calculation. int lengthInput = input.int(50, "Length", 2) //@function Calculates a moving average that changes only when `source` is outside the first and third quartiles. //@param source The series of values to process. //@param length The number of bars in the quartile calculation. //@returns The adaptive moving average value. customMA(float source, int length) => //@variable The custom moving average. var float result = na // Calculate the 25th and 75th `source` percentiles (first and third quartiles) over `length` bars. float q1 = ta.percentile_linear_interpolation(source, length, 25) float q3 = ta.percentile_linear_interpolation(source, length, 75) //@variable The distance from `source` to its interquartile range. float outerRange = 0.0 // Calculate the `outerRange` value when `source` is not `na`. if not na(source) float upperRange = source - q3 float lowerRange = q1 - source outerRange := math.max(upperRange, lowerRange, 0.0) // @variable The total range of `source` values over `length` bars. float totalRange = ta.range(source, length) //@variable Half the ratio of the `outerRange` to the `totalRange`. float alpha = 0.5 * outerRange / totalRange // Mix the `source` with the `result` based on the `alpha` value. result := (1.0 - alpha) * nz(result, source) + alpha * source // Return the `result`. result //@variable The `customMA()` of `close` over `lengthInput` bars. float maValue = customMA(close, lengthInput) // Plot the `maValue`. plot(maValue, "Custom MA", color.blue, 3)

Extraction using return expressions

In Pine Script, any user-defined function or method call, loop, or conditional structure *returns* the result of the *final* expression or nested structure within its local scope. Scripts can use these structures' returned results, excluding void, by assigning them to variables declared in the *outer* scope.

When debugging functions and conditional structures that contain multiple local variables, a common technique to extract data from their scopes is to return tuples containing the data that requires inspection.

Here, we've modified the previous example script's `customMA()` function to return a tuple containing values calculated from the local scopes. With this change, the script can call the function with a *tuple declaration* to make all the data available to the global scope. The script plots the `q1Dbg` and `q3Dbg` values, highlights the background when `alphaDbg` is 0, and uses log.info() to display a formatted string containing all the extracted data in the Pine Logs pane:

image

//@version=6 indicator("Extraction using return expressions demo", overlay = true) //@variable The number of bars in the `customMA()` calculation. int lengthInput = input.int(50, "Length", 2) //@function Calculates a moving average that changes only when `source` is outside the first and third quartiles. //@param source The series of values to process. //@param length The number of bars in the quartile calculation. //@returns The adaptive moving average value. customMA(float source, int length) => //@variable The custom moving average. var float result = na // Calculate the 25th and 75th `source` percentiles (first and third quartiles) over `length` bars. float q1 = ta.percentile_linear_interpolation(source, length, 25) float q3 = ta.percentile_linear_interpolation(source, length, 75) //@variable The distance from `source` to its interquartile range. float outerRange = 0.0 // To extract `upperRange` and `lowerRange` values, we need to make them accessible to the function's main scope. // Here, we added a tuple at the end of the `if` statement's local block, then declared a tuple in the function's // scope to hold the returned values. [upper,

lower] = if not na(source) float upperRange = source - q3 float lowerRange = q1 - source outerRange := math.max(upperRange, lowerRange, 0.0) [upperRange, lowerRange] //@variable The total range of `source` values over `length` bars. float totalRange = ta.range(source, length) //@variable Half the ratio of the `outerRange` to the `totalRange`. float alpha = 0.5 * outerRange / totalRange // Mix the `source` with the `result` based on the `alpha` value. result := (1.0 - alpha) * nz(result, source) + alpha * source // Return a tuple containing the `result` and other local variables. [result, q1, q3, upper, lower, outerRange, totalRange, alpha] //@variable The `customMA()` of `close` over `lengthInput` bars. [maValue, q1Dbg, q3Dbg, upperDbg, lowerDbg, outerRangeDbg, totalRangeDbg, alphaDbg] = customMA(close, lengthInput) // Plot the `maValue`. plot(maValue, "Custom MA", color.blue, 3) // When the bar is confirmed, log an "info" message containing formatted debug information for each variable. if barstate.isconfirmed log.info("maValue: {0,number,#.#####} \nq1Dbg: {1,number,#.#####}, q3Dbg: {2,number,#.#####} \nupperDbg: {3,number,#.#####}, lowerDbg: {4,number,#.#####} \nouterRangeDbg: {5,number,#.#####}, totalRangeDbg: {6,number,#.#####} \nalphaDbg: {7,number,#.#####}", maValue, q1Dbg, q3Dbg, upperDbg, lowerDbg, outerRangeDbg, totalRangeDbg, alphaDbg) // Display the extracted `q1` and `q3` data in all plot locations. plot(q1Dbg, "q1Dbg", color.new(color.maroon, 50)) plot(q3Dbg, "q3Dbg", color.new(color.teal, 50)) // Highlight the chart's background when the extracted `alpha` value is 0. bgcolor(alphaDbg == 0.0 ? color.new(color.orange, 90) : na, title = "`alpha == 0.0` highlight")

Note that:

- We added a tuple at the *end* of the if structure's block to *return* the `upperRange` and `lowerRange` values from its local scope. The function assigns the result to a two-variable tuple in its main scope, enabling it to include the if structure's local values in the return expression.

Extraction using reference types

Reference types, including all special types and user-defined types (UDTs), serve as structures for creating *objects*. Each object has an associated *reference* that distinguishes it and provides access to its data. Unlike fundamental types, variables of reference types **do not** store values directly. Instead, they hold the references for specific objects in memory.

An advanced, flexible way to extract data from local scopes is to initialize reference-type objects — such as instances of collections or UDTs — in the global scope and store local variable data in their elements or fields.

This technique is especially useful for extracting data from user-defined functions and methods. Although functions can access global variables, they cannot *reassign* them like global conditional structures and loops can. Consequently, they *cannot update* the data held by global variables of fundamental types. However, scripts do not modify reference types by reassigning their variables; they access objects via their references and use *methods* or *field reassignments* to update their data. As such, scripts *can* update global collections or UDT instances from inside function scopes.

For example, this modified version of our initial script declares a global `debugData` variable that holds the reference of a map with “string” keys and “float” values. Each map.put() call inside the `customMA()` scope modifies the map by adding a *key-value pair* containing a local variable’s *name* and *value*. After calling `customMA()`, the script uses map.get() calls on `debugData` to retrieve the stored information for its debugging outputs:

image

//@version=6 indicator("Extraction using reference types demo", overlay = true) //@variable The number of bars in the `customMA()` calculation. int lengthInput = input.int(50, "Length", 2) //@variable A global map of "string" keys and "float" values to store debug information from local scopes. var map<string, float> debugData =

map.new<string, float>() //@function Calculates a moving average that changes only when `source` is outside the first and third quartiles. //@param source The series of values to process. //@param length The number of bars in the quartile calculation. //@returns The adaptive moving average value. customMA(float source, int length) => //@variable The custom moving average. var float result = na // Calculate the 25th and 75th percentiles (first and third quartiles). float q1 = ta.percentile_linear_interpolation(source, length, 25), debugData.put("q1", q1) float q3 = ta.percentile_linear_interpolation(source, length, 75), debugData.put("q3", q3) //@variable The distance from `source` to its interquartile range. float outerRange = 0.0 // Calculate the `outerRange` value when `source` is not `na`. if not na(source) float upperRange = source - q3, debugData.put("upperRange", upperRange) float lowerRange = q1 - source, debugData.put("lowerRange", lowerRange) outerRange := math.max(upperRange, lowerRange, 0.0), debugData.put("outerRange", outerRange) //@variable The total range of `source` values over `length` bars. float totalRange = ta.range(source, length), debugData.put("totalRange", totalRange) //@variable Half the ratio of the `outerRange` to the `totalRange`. float alpha = 0.5 * outerRange / totalRange, debugData.put("alpha", alpha) // Mix the `source` with the `result` based on the `alpha` value. result := (1.0 - alpha) * nz(result, source) + alpha * source // Return the `result`. result //@variable The `customMA()` of `close` over `lengthInput` bars. float maValue = customMA(close, lengthInput) // Plot the `maValue`. plot(maValue, "Custom MA", color.blue, 3) // When the bar is confirmed, log an "info" message containing formatted debug information for each value. if barstate.isconfirmed log.info("maValue: {0,number,#.#####}\nq1: {1,number,#.#####}, q3: {2,number,#.#####} \nupperRange: {3,number,#.#####}, lowerRange: {4,number,#.#####} \nouterRange: {5,number,#.#####}, totalRange: {6,number,#.#####} \nalpha: {7,number,#.#####}", maValue, debugData.get("q1"), debugData.get("q3"), debugData.get("upperRange"), debugData.get("lowerRange"), debugData.get("outerRange"), debugData.get("totalRange"), debugData.get("alpha")) // Display the extracted `q1` and `q3` data in all plot locations.

plot(debugData.get("q1"), "q1", color.new(color.maroon, 50))
plot(debugData.get("q3"), "q3", color.new(color.teal, 50)) // Highlight the chart's background when the extracted `alpha` value is 0.
bgcolor(debugData.get("alpha") == 0.0 ? color.new(color.orange, 90) : na, title = "`alpha == 0.0` highlight")

Note that:

- The script declares `debugData` with the var keyword, meaning the assigned map reference *persists* across script executions.
- A function executes its local code only when the script calls it. Therefore, the `debugData` map contains new information only after the `customMA()` call.
- Because the map.put() calls in `customMA()` assign keys to the map that do not change across executions, each `customMA()` call *replaces* the `debugData` map's existing data. Programmers can *preserve* data from specific executions with this technique by making a *copy* of the global collection after the function call.

Inspecting loops

Loops are structures that execute a local code block repeatedly based on a *counter* (for), the *contents* of a collection (for...in), or a *condition* (while). These structures allow scripts to perform repetitive tasks without redundant lines of code.

Because loops can execute their local code multiple times, programmers must use techniques to track local variables *across iterations* to debug them effectively. As with other structures, there are many ways to inspect loops. These sections cover two helpful techniques: collecting loop information and tracing loop executions.

Collecting loop information

One of the most effective loop inspection techniques is to use collections or strings to gather information from the local scope on each iteration requiring inspection, then use the information in output functions *after* the loop terminates.

Let’s look at a simple loop debugging example using this technique. The following script calculates the average rate of change in the close price over lengths from 1 to `lookbackInput` bars inside a for loop. It declares an `aroc` variable in the global scope, sums the rates of change inside the loop, and then divides the sum by the `lookbackInput` to calculate the average:

image

//@version=6 indicator("Collecting loop information demo", "Average ROC") //@variable The number of past bars in the calculation. int lookbackInput = input.int(20, "Lookback", 1) //@variable The average ROC of `close` prices over each length from 1 to `lookbackInput` bars. float aroc = 0.0 // Calculation loop. for length = 1 to lookbackInput // @variable The `close` value `length` bars ago. float pastClose = close[length] //@variable The `close` rate of change over `length` bars. float roc = (close - pastClose) / pastClose // Add the `roc` to the `aroc` value. aroc += roc // Divide `aroc` by the `lookbackInput` to get the average. aroc /= lookbackInput // Plot the `aroc` series. plot(aroc, "aroc", color.blue, 3)

To debug the script’s loop and ensure it works as intended, we can collect data from the local scope on each iteration and pass the result to the available output functions after the loop ends. In the script version below, we demonstrate two extraction methods. The first declares a global `logText` variable and concatenates formatted strings containing each loop iteration’s `length` and `roc` values. The second declares a global `rocArray` variable and pushes each iteration’s `roc` value into the referenced array.

After terminating the loop, the script calls log.info() to display the `logText` in the Pine Logs pane if the bar is confirmed. It then displays a “string” representation of the `rocArray` inside label tooltips. Lastly, it shows the array’s *first* and *last* element values in all possible plot locations with the plot() function:

image

//@version=6 indicator("Collecting loop information demo", "Average ROC", max_labels_count = 500) //@variable The number of bars in the

calculation. int lookbackInput = input.int(20, "Lookback", 1) //@variable An array containing the `roc` value from each loop iteration. array<float> debugValues = array.new<float>() //@variable A string containing information about the `roc` value on each iteration. string logText = "" //@variable The average ROC of `close` over lags from 1 to `lookbackInput` bars. float aroc = 0.0 // Calculation loop. for length = 1 to lookbackInput //@variable The `close` value `length` bars ago. float pastClose = close[length] //@variable The `close` rate of change over `length` bars. float roc = (close - pastClose) / pastClose // Add the `roc` to `aroc`. aroc += roc // Concatenate a new "string" representation with the `debugText`. logText += "\nlength: " + str.tostring(length) + ", roc: " + str.tostring(roc) // Push the `roc` value into the `debugValues` array. array.push(debugValues, roc) // Divide `aroc` by the `lookbackInput`. aroc /= lookbackInput // Plot the `aroc`. plot(aroc, "aroc", color.blue, 3) // Log the `logText` in the Pine Logs pane when the bar is confirmed. if barstate.isconfirmed log.info(logText) // Draw a label with a tooltip containing a "string" representation of the `debugValues` array. label.new(bar_index, aroc, color = color.new(color.blue, 70), tooltip = str.tostring(debugValues)) // Plot the `roc` values from the first and last iteration. plot(array.first(debugValues), "First iteration roc", color.new(color.teal, 50), 2) plot(array.last(debugValues), "Last iteration roc", color.new(color.maroon, 50), 2)

Note that:

- Scripts can generate Pine Logs and drawings directly from within a loop's local scope. However, because loops usually execute their local code *more than once*, calling `log.*()` or label.new() functions inside the scope can result in numerous logs or labels per bar. Logging on each iteration helps trace execution patterns, but it also limits the number of *historical* bars with available debug data. See the next section, Tracing loop executions, for an example.
- Strings can contain up to *4096* characters, and large strings or repeated concatenation can impact a script's performance. Therefore, extracting loop information with string concatenation is suitable for relatively *small* loops or inspecting specific variables.

> To extract large amounts of data from loops, use collections instead.

Tracing loop executions

An alternative way to inspect a loop, without collecting information for use in the *outer scope*, is to add `log.*()` calls directly to the loop's *local block*. Each iteration that activates the call results in a new message in the Pine Logs pane, allowing programmers to trace the loop's execution pattern in detail.

This simple script calculates a random sample from a binomial distribution using a for loop. The plotted `sample` series represents the number of math.random() calls across `trialsInput` iterations that return a value not exceeding the `probabilityInput` value. On each iteration where `success` is `false`, the loop *skips* the rest of its block and moves to the next iteration. On other iterations, it increments the `sample` value by one:

image

//@version=6 indicator("Tracing loop executions demo", "Binomial sample") //@variable The probability that each random trial succeeds. float probabilityInput = input.float(0.5, "Success probability", 0.0, 1.0) // @variable The number of random trials to test. float trialsInput = input.int(10, "Trials", 1) //@variable Random sample from a binomial distribution, i.e., the number of successes from `trialsInput` random trials. int sample = 0 // Execute `trialsInput` loop iterations to calculate the `sample`. for trial = 1 to trialsInput //@variable A pseudorandom value between 0 and 1. float randValue = math.random() //@variable `true` if the `randValue` is less than or equal to the `probabilityInput`, `false` otherwise. bool success = randValue <= probabilityInput // Skip the rest of the iteration if `success` is `false`. if not success continue // Otherwise, add 1 to the `sample`. sample += 1 // Plot the `sample` as teal columns. plot(sample, "Binomial sample", color.teal, 1, plot.style_columns)

Below, we added `log.*()` function calls to generate Pine Logs at specific points in the loop's local block across iterations. Each loop

iteration creates two new logs. The first log shows formatted text containing the local `trial`, `randValue`, and `success` variables' values. The second log depends on the if statement. When the statement's local code executes, the log is a `"CONTINUE"` message with the "warning" level. Otherwise, the second log is an "info" message containing the current iteration's `sample` value:

image

```
//@version=6 indicator("Tracing loop executions demo") //@variable The probability that each random trial succeeds. float probabilityInput = input.float(0.5, "Success probability", 0.0, 1.0) //@variable The number of random trials to test. float trialsInput = input.int(10, "Trials", 1) //@variable Random sample from a binomial distribution, i.e., the number of successes from `trialsInput` random trials. int sample = 0 // Log a message to mark the point before the start of the loop. log.warning("---------------- LOOP START (bar {0,number,#})", bar_index) // Execute `trialsInput` loop iterations to calculate the `sample`. for trial = 1 to trialsInput //@variable A pseudorandom value between 0 and 1. float randValue = math.random() //@variable `true` if the `randValue` is less than or equal to the `probabilityInput`, `false` otherwise. bool success = randValue <= probabilityInput // Log a message containing the `trial`, `randValue`, and `success` information. log.info("trial: {0}, randValue: {1,number,#.########}, success: {2}", trial, randValue, success) // Skip the rest of the iteration if `success` is `false`. if not success // Log a message before the `continue` statement. log.warning("CONTINUE") continue // Otherwise, add 1 to the `sample`. sample += 1 // Log a message showing the iteration's `sample` value. log.info("sample: {0}", sample) // Log a message to mark the point after the loop ends. log.warning("---------------- LOOP END\n\n") // Plot the `sample` as teal columns. plot(sample, "Binomial sample", color.teal, 1, plot.style_columns)
```

Note that:

- The script includes log.warning() calls before and after the loop to mark its start and end in the Pine Logs pane. The message

marking the start of the loop also displays the current bar_index value.
- The Pine Logs pane shows only the most recent 10,000 logs created on historical bars. Because this script creates multiple logs per bar, the earliest message in the pane is from less than 10,000 bars back. Programmers can use conditional logic that limits `log.*()` calls in order to inspect a loop's execution flow on earlier bars with this technique. See the Custom code filters section to learn more.

Debugging collections

Collections are data structures that store values or references as *elements*, which scripts access using indices or keys, depending on the type. These structures can contain a lot of information, as the maximum number of elements across all instances of each collection type is 100,000.

Programmers can inspect a collection's data using various techniques, depending on the types they contain and their sizes. The most common approaches include:

- Creating a "string" representation of the collection with str.tostring() and displaying the result using Pine Logs or other text outputs.
- Retrieving specific elements from the collection, then creating formatted strings for logging, or using the element values or references in other output processes.

Displaying collection strings

The simplest way to inspect the data of arrays and matrices of "int", "float", "bool", and "string" types is to generate "string" representations with the str.tostring() function, then display the results using Pine Logs or other "string" outputs.

The following script calls request.security_lower_tf() to retrieve a "float" array containing close prices for each lower-timeframe bar within the current chart bar, which it uses to calculate an average intrabar price.

Then, it calculates the ratio of the difference between the bar's price and the intrabar average to the bar's total range. The script plots the resulting ratio and its EMA in a separate pane:

image

//@version=6 indicator("Displaying collection strings demo") // @variable The length of the EMA. int lengthInput = input.int(20, "EMA length", 1) //@variable An array of `close` prices requested for the chart's symbol at the 1-minute timeframe. array<float> intrabarPrices = request.security_lower_tf("", "1", close) //@variable The average `close` price of the intrabars within the current chart bar. float avgPrice = intrabarPrices.avg() //@variable The bar's total range. float barRange = high - low //@variable The difference between `close` and `avgPrice`, normalized by the `barRange`. float ratio = (close - avgPrice) / barRange //@variable The EMA of the `ratio`. float smoothed = ta.ema(ratio, lengthInput) // Plot the `ratio` series as conditionally-colored columns. plot(ratio, "", ratio > 0 ? color.teal : color.maroon, 1, plot.style_columns) // Display the `smoothed` series as a translucent orange area plot. plot(smoothed, "", color.new(color.orange, 40), 1, plot.style_area)

To verify the ratio's calculations, we can inspect the data stored in the `intrabarPrices` array by converting it to a "string" value and displaying the result for each bar.

The script version below declares a debugText variable that holds a formatted string representing the `intrabarPrices` array, the array's size, and the avgPrice value. The script calls the `log.*()` functions to display the debugText value for each bar in the Pine Logs pane:

image

//@version=6 indicator("Displaying collection strings demo") // @variable The length of the EMA. int lengthInput = input.int(20, "EMA length", 1) //@variable An array of `close` prices requested for the chart's symbol at the 1-minute timeframe. array<float> intrabarPrices = request.security_lower_tf("", "1", close) //@variable The average `close` price of the intrabars within the current chart bar. float avgPrice = intrabarPrices.avg() //@variable The bar's total range. float barRange

= high - low //@variable The difference between `close` and `avgPrice`, normalized by the `barRange`. float ratio = (close - avgPrice) / barRange //@variable The EMA of the `ratio`. float smoothed = ta.ema(ratio, lengthInput) // Plot the `ratio` series as conditionally-colored columns. plot(ratio, "", ratio > 0 ? color.teal : color.maroon, 1, plot.style_columns) // Display the `smoothed` series as a translucent orange area plot. plot(smoothed, "", color.new(color.orange, 40), 1, plot.style_area) //@variable A "string" representation of `intrabarPrices`, `intrabarPrices.size()`, and the `avgPrice`. string debugText = str.format("\nintrabarPrices: {0}\nsize: {1}\navgPrice: {2,number,#.#####}", str.tostring(intrabarPrices), intrabarPrices.size(), avgPrice) // Log the `debugText` with the "info" or "warning" level, depending on whether the bar is confirmed. switch barstate.isconfirmed => log.info(debugText) => log.warning(debugText)

Note that:

- The script calls log.info() on confirmed bars and log.warning() on the open bar. Users can filter the logs by logging level to inspect confirmed and unconfirmed bars' logs separately.
- For larger collections whose "string" representations *exceed* 4096 characters or cause excessive memory use, programmers can split them into *smaller parts* and convert them to strings separately. Alternatively, they can inspect individual elements via the `*.get()` method or for...in loops.

Inspecting individual elements

Collections of "color" or non-fundamental types (e.g., labels) *do not* have built-in "string" representations. Consequently, the technique described in the Displaying collection strings section does not work for them.

To inspect a collection that does not have a built-in "string" format, programmers can retrieve elements individually within for...in loops or using methods such as `*.get()`, then use those elements in custom "string" constructions or other output routines.

Consider the following example, which calculates the ratio of close changes to the overall close range over `lengthInput` bars. It plots the resulting `osc` in a separate pane, and it draws a label on the main chart pane each time the variable's absolute value is 1:

image

//@version=6 indicator("Inspecting individual elements demo") // @variable The number of bars in the calculation. int lengthInput = input.int(20, "Length", 2) //@variable The change in price across `lengthInput` - 1 bars. float priceChange = ta.change(close, lengthInput - 1) //@variable The total `close` range over `lengthInput` bars. float priceRange = ta.range(close, lengthInput) //@variable The ratio of the `priceChange` to the `priceRange`. float osc = priceChange / priceRange //@variable Teal if `osc` is positive, maroon otherwise. color oscColor = osc > 0 ? color.teal : color.maroon // Draw a label at the current bar's `bar_index` and `close` displaying `priceChange` when `osc` is 1 or -1. if math.abs(osc) == 1 string labelText = str.format("priceChange: {0,number,#.####}", priceChange) label.new(bar_index, close, labelText, color = oscColor, textcolor = color.white, force_overlay = true) // Plot the `osc` using the `oscColor`. plot(osc, "Oscillator", oscColor, 1, plot.style_area)

When a script creates labels, it *automatically* maintains an array containing each active label's reference. Programmers can access this array using the label.all variable, and thus inspect each individual label's properties on any bar.

In the version below, the script executes a log.info() call to display the current bar_index and the size of the label.all array for the latest bar. Then, it iterates through the array with a for...in loop. On each iteration, the script calls log.info() to log formatted text containing the array index and the corresponding label's `x`, `y`, and `text` properties. Additionally, the script plots the oldest and newest active labels' y-coordinates on each bar:

image

//@version=6 indicator("Inspecting individual elements demo") // @variable The number of bars in the calculation. int lengthInput =

input.int(20, "Length", 2) //@variable The change in price across `lengthInput` - 1 bars. float priceChange = ta.change(close, lengthInput - 1) //@variable The total `close` range over `lengthInput` bars. float priceRange = ta.range(close, lengthInput) //@variable The ratio of the `priceChange` to the `priceRange`. float osc = priceChange / priceRange //@variable Teal if `osc` is positive, maroon otherwise. color oscColor = osc > 0 ? color.teal : color.maroon // Draw a label at the current bar's `bar_index` and `close` displaying `priceChange` when `osc` is 1 or -1. if math.abs(osc) == 1 string labelText = str.format("priceChange: {0,number,#.####}", priceChange) label.new(bar_index, close, labelText, color = oscColor, textcolor = color.white, force_overlay = true) // Plot the `osc` using the `oscColor`. plot(osc, "Oscillator", oscColor, 1, plot.style_area) // On the first or last tick of the latest bar, inspect all labels on the chart. if barstate.islast and (barstate.isnew or barstate.isconfirmed) // Log a message containing the current `bar_index` and `label.all.size()`. log.info("Current bar: {0,number,#}, Active labels: {1}", bar_index, label.all.size()) // Loop through the `label.all` array. for [i, lbl] in label.all // Log a message containing the array index (`i`) and the label's `x`, `y`, and `text` properties. log.info("{0}, x: {1,number,#}, y: {2,number,#.#####}, text: {3}", i, lbl.get_x(), lbl.get_y(), lbl.get_text()) // Initialize variables for the oldest and newest active labels. label oldestLabel = na label newestLabel = na // Reassign the variables to the first and last labels in `label.all` when the array is not empty. if label.all.size() > 0 oldestLabel := label.all.first() newestLabel := label.all.last() // Plot the y-coordinate history of the `oldestLabel` and `newestLabel`. plot(label.get_y(oldestLabel), "oldestLabel y-coordinate", color.fuchsia, force_overlay = true) plot(label.get_y(newestLabel), "newestLabel y-coordinate", color.aqua, force_overlay = true)

Note that:

- It is not possible to obtain *all* properties from drawing objects. For example, there is no built-in method to retrieve a label's color. Some other types, such as table, do not have `*.get_*()` methods. If an object's properties are not directly accessible, programmers can create separate variables for the arguments of the drawing's

`*.new()` or `*.set_*()` function, and then use those variables for debugging.
- In the above image, the logs show that the label.all array contains 55 elements. By default, Pine limits the number of labels to approximately 50, but the *precise* number of active labels varies. Programmers can increase the label drawing limit using the `max_labels_count` parameter of the indicator() or strategy() declaration statement.

Debugging objects of UDTs

User-defined types (UDTs) define the structures of objects. Objects contain a fixed set of *fields*, where each field can hold a separate value or reference to another specified type, even to another instance of the *same* user-defined type.

Because UDT objects can organize values and references to an arbitrary number of various *different* types, Pine does not have a built-in method to convert UDT objects to strings. Instead, to debug these structures, programmers must retrieve data from each *field* that requires inspection.

The following example defines a custom `Data` type with three fields. The first two fields reference arrays that hold successive price and time values. The third field specifies the number of bars between each new data sample. The script creates a new object of this type with a randomized `length` field on the first bar, then updates its arrays on bars whose bar_index values are divisible by that field.

The script uses array.covariance() and array.variance() on the object's `prices` and `times` arrays to calculate a time-based slope of the collected data, and then plots the result on the chart:

image

//@version=6 indicator("Debugging objects of UDTs demo") //@type A structure for storing time and price information once every `sampleMult` bars. //@field prices References an array of "float" price values. //@field times References an array of "int" UNIX timestamps. //

@field sampleMult Number of bars per sample. type Data array<float> prices array<int> times int sampleMult //@variable The initial seed for the `math.random()` function. int seedInput = input.int(1234, "Seed", 1) //@variable References a `Data` object with arrays of 10 elements and a random `sampleMult` value. var Data data = Data.new(array.new<float>(10), array.new<int>(10), int(math.random(1, 11, seedInput))) // Queue new data through the `prices` and `times` arrays of the `Data` object once every `data.sampleMult` bars. if bar_index % data.sampleMult == 0 data.prices.push(close) data.times.push(time) data.prices.shift() data.times.shift() //@variable The time-based slope calculated from the `data` array fields. float slope = array.covariance(data.prices, data.times) / data.times.variance() // Plot the `slope` value. plot(slope, "Slope", slope > 0 ? color.teal : color.maroon, 3)

To verify and understand the script's calculations, we can *extract* information from the `Data` object's *fields* and inspect the data with Pine Logs or other outputs.

The script version below includes a log.info() call inside the if structure. The call displays formatted text representing information from the `Data` object's `prices`, `times`, and `length` fields in the Pine Logs pane. Now, we can view each change to the object's data to confirm the script's behavior:

image

//@version=6 indicator("Debugging objects of UDTs demo") //@type A structure for storing time and price information once every `sampleMult` bars. //@field prices References an array of "float" price values. //@field times References an array of "int" UNIX timestamps. //@field sampleMult Number of bars per sample. type Data array<float> prices array<int> times int sampleMult //@variable The initial seed for the `math.random()` function. int seedInput = input.int(1234, "Seed", 1) //@variable References a `Data` object with arrays of 10 elements and a random `sampleMult` value. var Data data = Data.new(array.new<float>(10), array.new<int>(10), int(math.random(1, 11, seedInput))) // Queue new data through the `prices` and `times` arrays of the `Data` object once every

`data.sampleMult` bars. if bar_index % data.sampleMult == 0 data.prices.push(close) data.times.push(time) data.prices.shift() data.times.shift() // Log formatted text containing information from the `Data` object's `prices`, `times`, and `sampleMult` // fields with the "info" or "warning" level. string fString = "Data object fields:\n\nprices: {0}\n\ntimes: {1}\n\nsampleMult: {2}\n-------" switch barstate.isconfirmed => log.info(fString, str.tostring(data.prices), str.tostring(data.times), data.sampleMult) => log.warning(fString, str.tostring(data.prices), str.tostring(data.times), data.sampleMult) // @variable The time-based slope calculated from the `data` array fields. float slope = array.covariance(data.prices, data.times) / data.times.variance() // Plot the `slope` value. plot(slope, "Slope", slope > 0 ? color.teal : color.maroon, 3)

Note that:

- The script calls log.info() on confirmed bars and log.warning() on open bars, allowing users to filter the results by logging level in the Pine Logs pane.

Organization and readability

Source code that is organized and easy to read is typically simpler to debug. Furthermore, well-written code is more straightforward for programmers to maintain and improve over time. Therefore, we recommend prioritizing organization and readability throughout the script-writing process, especially while debugging.

Below are a few helpful coding recommendations based on our Style guide and best practices:

- Follow the script organization guidelines. Organizing scripts based on this structure makes different parts of the code simple to locate and inspect.
- Use *identifiers* that you can read, distinguish, and understand. When a code contains unclear identifiers, it is often harder to debug efficiently. See our Naming conventions to learn our recommended identifier format.

- Use *type keywords* to signify the qualified types that variables and parameters can accept. Although Pine can usually *infer* variable and parameter types, declaring them explicitly improves readability and helps programmers distinguish between assignment and reassignment operations. Plus, it enables Pine's autosuggest feature to display more relevant type-based suggestions.
- Document the code using comments and compiler annotations (`//@function`, `//@variable`, etc.). The Pine Editor's autosuggest displays the text from annotations when the mouse pointer hovers over identifiers, making it simple to recall what different parts of the code represent.

Previous

Next

Profiling and optimization

Pine Script® v6 Documentation

Profiling and optimization

Introduction

Pine Script® is a cloud-based compiled language geared toward efficient repeated script execution. When a user adds a Pine script to a chart, it executes *numerous* times, once for each available bar or tick in the data feeds it accesses, as explained in this manual's Execution model page.

The Pine Script compiler automatically performs several internal optimizations to accommodate scripts of various sizes and help them run smoothly. However, such optimizations *do not* prevent performance bottlenecks in script executions. As such, it's up to programmers to profile a script's runtime performance and identify ways to modify critical code blocks and lines when they need to improve execution times.

This page covers how to profile and monitor a script's runtime and executions with the Pine Profiler and explains some ways programmers can modify their code to optimize runtime performance.

For a quick introduction, see the following video, where we profile an example script and optimize it step-by-step, examining several common script inefficiencies and explaining how to avoid them along the way:

Pine Profiler

Before diving into optimization, it's prudent to evaluate a script's runtime and pinpoint *bottlenecks*, i.e., areas in the code that substantially impact overall performance. With these insights, programmers can ensure they focus on optimizing where it truly matters instead of spending time and effort on low-impact code.

Enter the *Pine Profiler*, a powerful utility that analyzes the executions of all significant code lines and blocks in a script and displays helpful performance information next to the lines inside the Pine Editor. By inspecting the Profiler's results, programmers can gain a clearer perspective on a script's overall runtime, the distribution of runtime across its significant code regions, and the critical portions that may need extra attention and optimization.

Profiling a script

The Pine Profiler can analyze the runtime performance of any *editable* script coded in Pine Script v6. To profile a script, add it to the chart, open the source code in the Pine Editor, and turn on the "Profiler mode" switch in the dropdown accessible via the "More" option in the top-right corner:

image

We will use the script below for our initial profiling example, which calculates a custom `oscillator` based on average distances from the close price to upper and lower percentiles over `lengthInput` bars. It includes a few different types of *significant* code regions, which come with some differences in interpretation while profiling:

//@version=6 indicator("Pine Profiler demo") //@variable The number of bars in the calculations. int lengthInput = input.int(100, "Length", 2) // @variable The percentage for upper percentile calculation. float upperPercentInput = input.float(75.0, "Upper percentile", 50.0, 100.0) // @variable The percentage for lower percentile calculation. float lowerPercentInput = input.float(25.0, "Lower percentile", 0.0, 50.0) // Calculate percentiles using the linear interpolation method. float

upperPercentile = ta.percentile_linear_interpolation(close, lengthInput, upperPercentInput) float lowerPercentile = ta.percentile_linear_interpolation(close, lengthInput, lowerPercentInput) // Declare arrays for upper and lower deviations from the percentiles on the same line. var upperDistances = array.new<float>(lengthInput), var lowerDistances = array.new<float>(lengthInput) // Queue distance values through the `upperDistances` and `lowerDistances` arrays based on excessive price deviations. if math.abs(close - 0.5 * (upperPercentile + lowerPercentile)) > 0.5 * (upperPercentile - lowerPercentile) array.push(upperDistances, math.max(close - upperPercentile, 0.0)) array.shift(upperDistances) array.push(lowerDistances, math.max(lowerPercentile - close, 0.0)) array.shift(lowerDistances) // @variable The average distance from the `upperDistances` array. float upperAvg = upperDistances.avg() //@variable The average distance from the `lowerDistances` array. float lowerAvg = lowerDistances.avg() //@variable The ratio of the difference between the `upperAvg` and `lowerAvg` to their sum. float oscillator = (upperAvg - lowerAvg) / (upperAvg + lowerAvg) //@variable The color of the plot. A green-based gradient if `oscillator` is positive, a red-based gradient otherwise. color oscColor = oscillator > 0 ? color.from_gradient(oscillator, 0.0, 1.0, color.gray, color.green) : color.from_gradient(oscillator, -1.0, 0.0, color.red, color.gray) // Plot the `oscillator` with the `oscColor`. plot(oscillator, "Oscillator", oscColor, style = plot.style_area)

Once enabled, the Profiler collects information from all executions of the script’s significant code lines and blocks, then displays bars and approximate runtime percentages to the left of the code lines inside the Pine Editor:

image

Note that:

- The Profiler tracks every execution of a significant code region, including the executions on *realtime ticks*. Its information updates over time as new executions occur.

- Profiler results **do not** appear for script declaration statements, type declarations, other *insignificant* code lines such as variable declarations with no tangible impact, *unused code* that the script's outputs do not depend on, or *repetitive code* that the compiler optimizes during translation. See this section for more information.

When a script contains at least *four* significant lines of code, the Profiler will include "flame" icons next to the *top three* code regions with the highest performance impact. If one or more of the highest-impact code regions are *outside* the lines visible inside the Pine Editor, a "flame" icon and a number indicating how many critical lines are outside the view will appear at the top or bottom of the left margin. Clicking the icon will vertically scroll the Editor's window to show the nearest critical line:

image

Hovering the mouse pointer over the space next to a line highlights the analyzed code and exposes a tooltip with additional information, including the time spent and the number of executions. The information shown next to each line and in the corresponding tooltip depends on the profiled code region. The section below explains different types of code the Profiler analyzes and how to interpret their performance results.

image

Interpreting profiled results

Single-line results

For a code line containing single-line expressions, the Profiler bar and displayed percentage represent the relative portion of the script's total runtime spent on that line. The corresponding tooltip displays three fields:

- The "Line number" field indicates the analyzed code line.

- The “Time” field shows the runtime percentage for the line of code, the runtime spent on that line, and the script’s total runtime.
- The “Executions” field shows the number of times that specific line executed while running the script.

Here, we hovered the pointer over the space next to line 12 of our profiled code to view its tooltip:

image

float upperPercentile = ta.percentile_linear_interpolation(close, lengthInput, upperPercentInput)

Note that:

- The time information for the line represents the time spent completing *all* executions, **not** the time spent on a single execution.
- To estimate the *average* time spent per execution, divide the line’s time by the number of executions. In this case, the tooltip shows that line 12 took about 14.1 milliseconds to execute 20,685 times, meaning the average time per execution was approximately 14.1 ms / 20685 = 0.0006816534 milliseconds (0.6816534 microseconds).

When a line of code consists of more than one expression separated by commas, the number of executions shown in the tooltip represents the *sum* of each expression’s total executions, and the time value displayed represents the total time spent evaluating all the line’s expressions.

For instance, this global line from our initial example includes two variable declarations separated by commas. Each uses the var keyword, meaning the script only executes them once on the first available bar. As we see in the Profiler tooltip for the line, it counted *two* executions (one for each expression), and the time value shown is the *combined* result from both expressions on the line:

image

var upperDistances = array.new<float>(lengthInput), var lowerDistances = array.new<float>(lengthInput)

Note that:

- When analyzing scripts with more than one expression on the same line, we recommend moving each expression to a *separate line* for more detailed insights while profiling, namely if they may contain *higher-impact* calculations.

When using line wrapping for readability or stylistic purposes, the Profiler considers all portions of a wrapped line as part of the *first line* where it starts in the Pine Editor.

For example, although this code from our initial script occupies more than one line in the Pine Editor, it's still treated as a *single* line of code, and the Profiler tooltip displays single-line results, with the "Line number" field showing the *first* line in the Editor that the wrapped line occupies:

image

color oscColor = oscillator > 0 ? color.from_gradient(oscillator, 0.0, 1.0, color.gray, color.green) : color.from_gradient(oscillator, -1.0, 0.0, color.red, color.gray)

Code block results

For a line at the start of a loop or conditional structure, the Profiler bar and percentage represent the relative portion of the script's runtime spent on the **entire code block**, not just the single line. The corresponding tooltip displays four fields:

- The "Code block range" field indicates the range of lines included in the structure.
- The "Time" field shows the code block's runtime percentage, the time spent on all block executions, and the script's total runtime.
- The "Line time" field shows the runtime percentage for the block's initial line, the time spent on that line, and the script's total runtime. The interpretation differs for switch blocks or if blocks

with `else if` statements, as the values represent the total time spent on **all** the structure's conditional statements. See below for more information.
- The "Executions" field shows the number of times the code block executed while running the script.

Here, we hovered over the space next to line 19 in our initial script, the beginning of a simple if structure *without* `else if` statements. As we see below, the tooltip shows performance information for the entire code block and the current line:

image

if math.abs(close - 0.5 * (upperPercentile + lowerPercentile)) > 0.5 * (upperPercentile - lowerPercentile) array.push(upperDistances, math.max(close - upperPercentile, 0.0)) array.shift(upperDistances) array.push(lowerDistances, math.max(lowerPercentile - close, 0.0)) array.shift(lowerDistances)

Note that:

- The "Time" field shows that the total time spent evaluating the structure 20,685 times was 7.2 milliseconds.
- The "Line time" field indicates that the runtime spent on the *first line* of this if structure was about three milliseconds.

Users can also inspect the results from lines and nested blocks within a code block's range to gain more granular performance insights. Here, we hovered over the space next to line 20 within the code block to view its single-line result:

image

Note that:

- The number of executions shown is *less than* the result for the entire code block, as the condition that controls the execution of this line does not return `true` all the time. The opposite applies to the code inside loops since each execution of a loop statement can trigger **several** executions of the loop's local block.

When profiling a switch structure or an if structure that includes `else if` statements, the “Line time” field will show the time spent executing **all** the structure’s conditional expressions, **not** just the block’s first line. The results for the lines inside the code block range will show runtime and executions for each **local block**. This format is necessary for these structures due to the Profiler’s calculation and display constraints. See this section for more information.

For example, the “Line time” for the switch structure in this script represents the time spent evaluating *all four* conditional statements within its body, as the Profiler *cannot* track them separately. The results for each line in the code block’s range represent the performance information for each *local block*:

image

//@version=6 indicator("`switch` and `if...else if` results demo") // @variable The upper band for oscillator calculation. var float upperBand = close //@variable The lower band for oscillator calculation. var float lowerBand = close // Update the `upperBand` and `lowerBand` based on the proximity of the `close` to the current band values. // The "Line time" field on line 11 represents the time spent on all 4 conditional expressions in the structure. switch close > upperBand => upperBand := close close < lowerBand => lowerBand := close upperBand - close > close - lowerBand => upperBand := 0.9 * upperBand + 0.1 * close close - lowerBand > upperBand - close => lowerBand := 0.9 * lowerBand + 0.1 * close //@variable The ratio of the difference between `close` and `lowerBand` to the band range. float oscillator = 100.0 * (close - lowerBand) / (upperBand - lowerBand) // Plot the `oscillator` as columns with a dynamic color. plot(oscillator, "Oscillator", oscillator > 50.0 ? color.teal : color.maroon, style = plot.style_columns, histbase = 50.0)

When the conditional logic in such structures involves significant calculations, programmers may require more granular performance information for each calculated condition. An effective way to achieve this analysis is to use *nested* if blocks instead of the more compact switch or `if...else if` structures. For example, instead of:

switch <expression1> => <localBlock1> <expression2> => <localBlock2> => <localBlock3>

or:

if <expression1> <localBlock1> else if <expression2> <localBlock2> else <localBlock3>

one can use nested if blocks for more in-depth profiling while maintaining the same logical flow:

if <expression1> <localBlock1> else if <expression2> <localBlock2> else <localBlock3>

Below, we changed the previous switch example to an equivalent nested if structure. Now, we can view the runtime and executions for each significant part of the conditional pattern individually:

image

//@version=6 indicator("`switch` and `if...else if` results demo") // @variable The upper band for oscillator calculation. var float upperBand = close //@variable The lower band for oscillator calculation. var float lowerBand = close // Update the `upperBand` and `lowerBand` based on the proximity of the `close` to the current band values. if close > upperBand upperBand := close else if close < lowerBand lowerBand := close else if upperBand - close > close - lowerBand upperBand := 0.9 * upperBand + 0.1 * close else if close - lowerBand > upperBand - close lowerBand := 0.9 * lowerBand + 0.1 * close //@variable The ratio of the difference between `close` and `lowerBand` to the band range. float oscillator = 100.0 * (close - lowerBand) / (upperBand - lowerBand) // Plot the `oscillator` as columns with a dynamic color. plot(oscillator, "Oscillator", oscillator > 50.0 ? color.teal : color.maroon, style = plot.style_columns, histbase = 50.0)

Note that:

- This same process can also apply to ternary operations. When a complex ternary expression’s operands contain significant calculations, reorganizing the logic into a nested if structure allows

more detailed Profiler results, making it easier to spot critical parts.

User-defined function calls

User-defined functions and methods are functions written by users. They encapsulate code sequences that a script may execute several times. Users often write functions and methods for improved code modularity, reusability, and maintainability.

The indented lines of code within a function represent its *local scope*, i.e., the sequence that executes *each time* the script calls it. Unlike code in a script's global scope, which a script evaluates once on each execution, the code inside a function may activate zero, one, or *multiple times* on each script execution, depending on the conditions that trigger the calls, the number of calls that occur, and the function's logic.

This distinction is crucial to consider while interpreting Profiler results. When a profiled code contains user-defined function or method calls:

- The results for each *function call* reflect the runtime allocated toward it and the total number of times the script activated that specific call.
- The time and execution information for all local code *inside* a function's scope reflects the combined results from **all** calls to the function.

This example contains a user-defined `similarity()` function that estimates the similarity of two series, which the script calls only *once* from the global scope on each execution. In this case, the Profiler's results for the code inside the function's body correspond to that specific call:

image

//@version=6 indicator("User-defined function calls demo") //@function Estimates the similarity between two standardized series over `length` bars. // Each individual call to this function activates its local scope. similarity(float sourceA, float sourceB, int length) => // Standardize

`sourceA` and `sourceB` for comparison. float normA = (sourceA - ta.sma(sourceA, length)) / ta.stdev(sourceA, length) float normB = (sourceB - ta.sma(sourceB, length)) / ta.stdev(sourceB, length) // Calculate and return the estimated similarity of `normA` and `normB`. float abSum = math.sum(normA * normB, length) float a2Sum = math.sum(normA * normA, length) float b2Sum = math.sum(normB * normB, length) abSum / math.sqrt(a2Sum * b2Sum) // Plot the similarity between the `close` and an offset `close` series. plot(similarity(close, close[1], 100), "Similarity 1", color.red)

Let’s increase the number of times the script calls the function each time it executes. Here, we changed the script to call our user-defined function *five times*:

//@version=6 indicator("User-defined function calls demo") //@function Estimates the similarity between two standardized series over `length` bars. // Each individual call to this function activates its local scope. similarity(float sourceA, float sourceB, int length) => // Standardize `sourceA` and `sourceB` for comparison. float normA = (sourceA - ta.sma(sourceA, length)) / ta.stdev(sourceA, length) float normB = (sourceB - ta.sma(sourceB, length)) / ta.stdev(sourceB, length) // Calculate and return the estimated similarity of `normA` and `normB`. float abSum = math.sum(normA * normB, length) float a2Sum = math.sum(normA * normA, length) float b2Sum = math.sum(normB * normB, length) abSum / math.sqrt(a2Sum * b2Sum) // Plot the similarity between the `close` and several offset `close` series. plot(similarity(close, close[1], 100), "Similarity 1", color.red) plot(similarity(close, close[2], 100), "Similarity 2", color.orange) plot(similarity(close, close[4], 100), "Similarity 3", color.green) plot(similarity(close, close[8], 100), "Similarity 4", color.blue) plot(similarity(close, close[16], 100), "Similarity 5", color.purple)

In this case, the local code results no longer correspond to a *single* evaluation per script execution. Instead, they represent the *combined* runtime and executions of the local code from **all five** calls. As we see below, the results after running this version of the script across the same data show 137,905 executions of the local code, *five times* the number from when the script only contained one `similarity()` function call:

image

When requesting other contexts

Pine scripts can request data from other *contexts*, i.e., different symbols, timeframes, or data modifications than what the chart's data uses by calling the `request.*()` family of functions or specifying an alternate `timeframe` in the indicator() declaration statement.

When a script requests data from another context, it evaluates all required scopes and calculations within that context, as explained in the Other timeframes and data page. This behavior can affect the runtime of a script's code regions and the number of times they execute.

The Profiler information for any code line or block represents the results from executing the code in *all necessary contexts*, which may or may not include the chart's data. Pine Script determines which contexts to execute code within based on the calculations required by a script's data requests and outputs.

Let's look at a simple example. This initial script only uses the chart's data for its calculations. It declares a `pricesArray` variable with the varip keyword, meaning the array assigned to it persists across the data's history and all available realtime ticks. On each execution, the script calls array.push() to push a new close value into the array, and it plots the array's size.

After profiling the script across all the bars on an intraday chart, we see that the number of elements in the `pricesArray` corresponds to the number of executions the Profiler shows for the array.push() call on line 8:

image

//@version=6 indicator("When requesting other contexts demo") // @variable An array containing the `close` value from every available price update. varip array<float> pricesArray = array.new<float>() // Push a new `close` value into the `pricesArray` on each update.

array.push(pricesArray, close) // Plot the size of the `pricesArray`. plot(array.size(pricesArray), "Total number of chart price updates")

Now, let's try evaluating the size of the `pricesArray` from *another context* instead of using the chart's data. Below, we've added a request.security() call with array.size(pricesArray) as its `expression` argument to retrieve the value calculated on the "1D" timeframe and plotted that result instead.

In this case, the number of executions the Profiler shows on line 8 still corresponds to the number of elements in the `pricesArray`. However, it did not execute the same number of times since the script did not require the *chart's data* in the calculations. It only needed to initialize the array and evaluate array.push() across all the requested *daily data*, which has a different number of price updates than our current intraday chart:

image

//@version=6 indicator("When requesting other contexts demo") // @variable An array containing the `close` value from every available price update. varip array<float> pricesArray = array.new<float>() // Push a new `close` value into the `pricesArray` on each update. array.push(pricesArray, close) // Plot the size of the `pricesArray` requested from the daily timeframe. plot(request.security(syminfo.tickerid, "1D", array.size(pricesArray)), "Total number of daily price updates")

Note that:

- The requested EOD data in this example had fewer data points than our intraday chart, so the array.push() call required fewer executions in this case. However, EOD feeds *do not* have history limitations, meaning it's also possible for requested HTF data to span **more** bars than a user's chart, depending on the timeframe, the data provider, and the user's plan.

If this script were to plot the array.size() value directly in addition to the requested daily value, it would then require the creation of *two* arrays (one for each context) and the execution of array.push() across both

the chart's data *and* the data from the daily timeframe. As such, the declaration on line 5 will execute *twice*, and the results on line 8 will reflect the time and executions accumulated from evaluating the array.push() call across **both separate datasets**:

image

//@version=6 indicator("When requesting other contexts demo") // @variable An array containing the `close` value from every available price update. varip array<float> pricesArray = array.new<float>() // Push a new `close` value into the `pricesArray` on each update. array.push(pricesArray, close) // Plot the size of the `pricesArray` from the daily timeframe and the chart's context. // Including both in the outputs requires executing line 5 and line 8 across BOTH datasets. plot(request.security(syminfo.tickerid, "1D", array.size(pricesArray)), "Total number of daily price updates") plot(array.size(pricesArray), "Total number of chart price updates")

It's important to note that when a script calls a user-defined function or method that contains `request.*()` calls in its local scope, the script's *translated form* extracts the `request.*()` calls **outside** the scope and encapsulates the expressions they depend on within **separate functions**. When the script executes, it evaluates the required `request.*()` calls first, then *passes* the requested data to a *modified form* of the user-defined function.

Since the translated script executes a user-defined function's data requests separately **before** evaluating non-requested calculations in its local scope, the Profiler's results for lines containing calls to the function **will not** include the time spent on its `request.*()` calls or their required expressions.

As an example, the following script contains a user-defined `getCompositeAvg()` function with a request.security() call that requests the math.avg() of 10 ta.wma() calls with different `length` arguments from a specified `symbol`. The script uses the function to request the average result using a Heikin Ashi ticker ID:

//@version=6 indicator("User-defined functions with `request.*()` calls demo", overlay = true) int multInput = input.int(10, "Length multiplier",

1) string tickerID = ticker.heikinashi(syminfo.tickerid) getCompositeAvg(string symbol, int lengthMult) => request.security(symbol, timeframe.period, math.avg(ta.wma(close, lengthMult), ta.wma(close, 2 * lengthMult), ta.wma(close, 3 * lengthMult), ta.wma(close, 4 * lengthMult), ta.wma(close, 5 * lengthMult), ta.wma(close, 6 * lengthMult), ta.wma(close, 7 * lengthMult), ta.wma(close, 8 * lengthMult), ta.wma(close, 9 * lengthMult), ta.wma(close, 10 * lengthMult))) plot(getCompositeAvg(tickerID, multInput), "Composite average", linewidth = 3)

After profiling the script, users might be surprised to see that the runtime results shown inside the function's body heavily **exceed** the results shown for the *single* `getCompositeAvg()` call:

image

The results appear this way since the translated script includes internal modifications that *moved* the request.security() call and its expression **outside** the function's scope, and the Profiler has no way to represent the results from those calculations other than displaying them next to the request.security() line in this scenario. The code below roughly illustrates how the translated script looks:

//@version=6 indicator("User-defined functions with `request.*()` calls demo", overlay = true) int multInput = input.int(10, "Length multiplier") string tickerID = ticker.heikinashi(syminfo.tickerid) secExpr(int lengthMult)=> math.avg(ta.wma(close, lengthMult), ta.wma(close, 2 * lengthMult), ta.wma(close, 3 * lengthMult), ta.wma(close, 4 * lengthMult), ta.wma(close, 5 * lengthMult), ta.wma(close, 6 * lengthMult), ta.wma(close, 7 * lengthMult), ta.wma(close, 8 * lengthMult), ta.wma(close, 9 * lengthMult), ta.wma(close, 10 * lengthMult)) float sec = request.security(tickerID, timeframe.period, secExpr(multInput)) getCompositeAvg(float s) => s plot(getCompositeAvg(sec), "Composite average", linewidth = 3)

Note that:

- The `secExpr()` code represents the *separate function* used by request.security() to calculate the required expression in the requested context.
- The request.security() call takes place in the **outer scope**, outside the `getCompositeAvg()` function.
- The translation substantially reduced the local code of `getCompositeAvg()`. It now solely returns a value passed into it, as all the function's required calculations take place **outside** its scope. Due to this reduction, the function call's performance results **will not** reflect any of the time spent on the data request's required calculations.

Insignificant, unused, and redundant code

When inspecting a profiled script's results, it's crucial to understand that *not all* code in a script necessarily impacts runtime performance. Some code has no direct performance impact, such as a script's declaration statement and type declarations. Other code regions with insignificant expressions, such as most `input.*()` calls, variable references, or variable declarations without significant calculations, have little to *no effect* on a script's runtime. Therefore, the Profiler will **not** display performance results for these types of code.

Additionally, Pine scripts do not execute code regions that their *outputs* (plots, drawings, logs, etc.) do not depend on, as the compiler automatically **removes** them during translation. Since unused code regions have *zero* impact on a script's performance, the Profiler will **not** display any results for them.

The following example contains a `barsInRange` variable and a for loop that adds 1 to the variable's value for each historical close price between the current high and low over `lengthInput` bars. However, the script **does not use** these calculations in its outputs, as it only plots the close price. Consequently, the script's compiled form **discards** that unused code and only considers the plot(close) call.

The Profiler does not display **any** results for this script since it does not execute any **significant** calculations:

image

//@version=6 indicator("Unused code demo") //@variable The number of historical bars in the calculation. int lengthInput = input.int(100, "Length", 1) //@variable The number of closes over `lengthInput` bars between the current bar's `high` and `low`. int barsInRange = 0 for i = 1 to lengthInput //@variable The `close` price from `i` bars ago. float pastClose = close[i] // Add 1 to `barsInRange` if the `pastClose` is between the current bar's `high` and `low`. if pastClose > low and pastClose < high barsInRange += 1 // Plot the `close` price. This is the only output. // Since the outputs do not require any of the above calculations, the compiled script will not execute them. plot(close)

Note that:

- Although this script does not use the input.int() from line 5 and discards all its associated calculations, the “Length” input *will* still appear in the script’s settings, as the compiler **does not** completely remove unused inputs.

If we change the script to plot the `barsInRange` value instead, the declared variables and the for loop are no longer unused since the output depends on them, and the Profiler will now display performance information for that code:

image

//@version=6 indicator("Unused code demo") //@variable The number of historical bars in the calculation. int lengthInput = input.int(100, "Length", 1) //@variable The number of closes over `lengthInput` bars between the current bar's `high` and `low`. int barsInRange = 0 for i = 1 to lengthInput //@variable The `close` price from `i` bars ago. float pastClose = close[i] // Add 1 to `barsInRange` if the `pastClose` is between the current bar's `high` and `low`. if pastClose > low and pastClose < high barsInRange += 1 // Plot the `barsInRange` value. The above calculations will execute since the output requires them. plot(barsInRange, "Bars in range")

Note that:

- The Profiler does not show performance information for the `lengthInput` declaration on line 5 or the `barsInRange` declaration on line 8 since the expressions on these lines do not impact the script's performance.

When possible, the compiler also simplifies certain instances of *redundant code* in a script, such as some forms of identical expressions with the same fundamental type values. This optimization allows the compiled script to only execute such calculations *once*, on the first occurrence, and *reuse* the calculated result for each repeated instance that the outputs depend on.

If a script contains repetitive code and the compiler simplifies it, the Profiler will only show results for the **first occurrence** of the code since that's the only time the script requires the calculation.

For example, this script contains a code line that plots the value of ta.sma(close, 100) and 12 code lines that plot the value of ta.sma(close, 500):

//@version=6 indicator("Redundant calculations demo", overlay = true) // Plot the 100-bar SMA of `close` values one time. plot(ta.sma(close, 100), "100-bar SMA", color.teal, 3) // Plot the 500-bar SMA of `close` values 12 times. After compiler optimizations, only the first `ta.sma(close, 500)` // call on line 9 requires calculation in this case. plot(ta.sma(close, 500), "500-bar SMA", #001aff, 12) plot(ta.sma(close, 500), "500-bar SMA", #4d0bff, 11) plot(ta.sma(close, 500), "500-bar SMA", #7306f7, 10) plot(ta.sma(close, 500), "500-bar SMA", #920be9, 9) plot(ta.sma(close, 500), "500-bar SMA", #ae11d5, 8) plot(ta.sma(close, 500), "500-bar SMA", #c618be, 7) plot(ta.sma(close, 500), "500-bar SMA", #db20a4, 6) plot(ta.sma(close, 500), "500-bar SMA", #eb2c8a, 5) plot(ta.sma(close, 500), "500-bar SMA", #f73d6f, 4) plot(ta.sma(close, 500), "500-bar SMA", #fe5053, 3) plot(ta.sma(close, 500), "500-bar SMA", #ff6534, 2) plot(ta.sma(close, 500), "500-bar SMA", #ff7a00, 1)

Since the last 12 lines all contain identical ta.sma() calls, the compiler can automatically simplify the script so that it only needs to evaluate

ta.sma(close, 500) *once* per execution rather than repeating the calculation 11 more times.

As we see below, the Profiler only shows results for lines 5 and 9. These are the only parts of the code requiring significant calculations since the ta.sma() calls on lines 10-20 are redundant in this case:

image

Another type of repetitive code optimization occurs when a script contains two or more user-defined functions or methods with identical compiled forms. In such a case, the compiler simplifies the script by **removing** the redundant functions, and the script will treat all calls to the redundant functions as calls to the **first** defined version. Therefore, the Profiler will only show local code performance results for the *first* function since the discarded “clones” will never execute.

For instance, the script below contains two user-defined functions, `metallicRatio()` and `calcMetallic()`, that calculate a metallic ratio of a given order raised to a specified exponent:

//@version=6 indicator("Redundant functions demo") //@variable Controls the base ratio for the `calcMetallic()` call. int order1Input = input.int(1, "Order 1", 1) //@variable Controls the base ratio for the `metallicRatio()` call. int order2Input = input.int(2, "Order 2", 1) // @function Calculates the value of a metallic ratio with a given `order`, raised to a specified `exponent`. //@param order Determines the base ratio used. 1 = Golden Ratio, 2 = Silver Ratio, 3 = Bronze Ratio, and so on. //@param exponent The exponent applied to the ratio. metallicRatio(int order, float exponent) => math.pow((order + math.sqrt(4.0 + order * order)) * 0.5, exponent) //@function A function with the same signature and body as `metallicRatio()`. // The script discards this function and treats `calcMetallic()` as an alias for `metallicRatio()`. calcMetallic(int ord, float exp) => math.pow((ord + math.sqrt(4.0 + ord * ord)) * 0.5, exp) // Plot the results from a `calcMetallic()` and `metallicRatio()` call. plot(calcMetallic(order1Input, bar_index % 5), "Ratio 1", color.orange, 3) plot(metallicRatio(order2Input, bar_index % 5), "Ratio 2", color.maroon)

Despite the differences in the function and parameter names, the two functions are otherwise identical, which the compiler detects while translating the script. In this case, it **discards** the redundant `calcMetallic()` function, and the compiled script treats the `calcMetallic()` call as a `metallicRatio()` call.

As we see here, the Profiler shows performance information for the `calcMetallic()` and `metallicRatio()` calls on lines 21 and 22, but it does **not** show any results for the local code of the `calcMetallic()` function on line 18. Instead, the Profiler's information on line 13 within the `metallicRatio()` function reflects the local code results from **both** function calls:

image

A look into the Profiler's inner workings

The Pine Profiler wraps all necessary code regions with specialized *internal functions* to track and collect required information across script executions. It then passes the information to additional calculations that organize and display the performance results inside the Pine Editor. This section gives users a peek into how the Profiler applies internal functions to wrap Pine code and collect performance data.

There are two main internal **(non-Pine)** functions the Profiler wraps significant code with to facilitate runtime analysis. The first function retrieves the current system time at specific points in the script's execution, and the second maps cumulative elapsed time and execution data to specific code regions. We represent these functions in this explanation as `System.timeNow()` and `registerPerf()` respectively.

When the Profiler detects code that requires analysis, it adds `System.timeNow()` above the code to get the initial time before execution. Then, it adds `registerPerf()` below the code to map and accumulate the elapsed time and number of executions. The elapsed time added on each `registerPerf()` call is the `System.timeNow()` value *after* the execution minus the value *before* the execution.

The following *pseudocode* outlines this process for a single line of code, where `_startX` represents the starting time for the `lineX` line:

long _startX = System.timeNow() <code_line_to_analyze> registerPerf(System.timeNow() - _startX, lineX)

The process is similar for code blocks. The difference is that the `registerPerf()` call maps the data to a *range of lines* rather than a single line. Here, `lineX` represents the *first* line in the code block, and `lineY` represents the block's *last* line:

long _startX = System.timeNow() <code_block_to_analyze> registerPerf(System.timeNow() - _startX, lineX, lineY)

Note that:

- In the above snippets, `long`, `System.timeNow()`, and `registerPerf()` represent *internal code*, **not** Pine Script code.

Let's now look at how the Profiler wraps a full script and all its significant code. We will start with this script, which calculates three pseudorandom series and displays their average result. The script utilizes an object of a user-defined type to store a pseudorandom state, a method to calculate new values and update the state, and an if...else if structure to update each series based on generated values:

//@version=6 indicator("Profiler's inner workings demo") int seedInput = input.int(12345, "Seed") type LCG float state method generate(LCG this, int generations = 1) => float result = 0.0 for i = 1 to generations this.state := 16807 * this.state % 2147483647 result += this.state / 2147483647 result / generations var lcg = LCG.new(seedInput) var float val0 = 1.0 var float val1 = 1.0 var float val2 = 1.0 if lcg.generate(10) < 0.5 val0 *= 1.0 + (2.0 * lcg.generate(50) - 1.0) * 0.1 else if lcg.generate(10) < 0.5 val1 *= 1.0 + (2.0 * lcg.generate(50) - 1.0) * 0.1 else if lcg.generate(10) < 0.5 val2 *= 1.0 + (2.0 * lcg.generate(50) - 1.0) * 0.1 plot(math.avg(val0, val1, val2), "Average pseudorandom result", color.purple)

The Profiler will wrap the entire script and all necessary code regions, excluding any insignificant, unused, or redundant code, with the

aforementioned **internal** functions to collect performance data. The *pseudocode* below demonstrates how this process applies to the above script:

long _startMain = System.timeNow() // Start time for the script's overall execution. // <Additional internal code executes here> //@version=6 indicator("Profiler's inner workings demo") // Declaration statements do not require profiling. int seedInput = input.int(12345, "Seed") // Variable declaration without significant calculation. type LCG // Type declarations do not require profiling. float state method generate(LCG this, int generations = 1) => // Function signature does not affect runtime. float result = 0.0 // Variable declaration without significant calculation. long _start11 = System.timeNow() // Start time for the loop block that begins on line 11. for i = 1 to generations // Loop header calculations are not independently wrapped. long _start12 = System.timeNow() // Start time for line 12. this.state := 16807 * this.state % 2147483647 registerPerf(System.timeNow() - _start12, line12) // Register performance info for line 12. long _start13 = System.timeNow() // Start time for line 13. result += this.state / 2147483647 registerPerf(System.timeNow() - _start13, line13) // Register performance info for line 13. registerPerf(System.timeNow() - _start11, line11, line13) // Register performance info for the block (line 11 - 13). long _start14 = System.timeNow() // Start time for line 14. result / generations registerPerf(System.timeNow() - _start14, line14) // Register performance info for line 14. long _start16 = System.timeNow() // Start time for line 16. var lcg = LCG.new(seedInput) registerPerf(System.timeNow() - _start16, line16) // Register performance info for line 16. var float val0 = 1.0 // Variable declarations without significant calculations. var float val1 = 1.0 var float val2 = 1.0 long _start22 = System.timeNow() // Start time for the `if` block that begins on line 22. if lcg.generate(10) < 0.5 // `if` statement is not independently wrapped. long _start23 = System.timeNow() // Start time for line 23. val0 *= 1.0 + (2.0 * lcg.generate(50) - 1.0) * 0.1 registerPerf(System.timeNow() - _start23, line23) // Register performance info for line 23. else if lcg.generate(10) < 0.5 // `else if` statement is not independently wrapped. long _start25 = System.timeNow() // Start time for line 25. val1 *= 1.0 + (2.0 * lcg.generate(50) - 1.0) * 0.1 registerPerf(System.timeNow() - _start25,

line25) // Register performance info for line 25. else if lcg.generate(10) < 0.5 // `else if` statement is not independently wrapped. long _start27 = System.timeNow() // Start time for line 27. val2 *= 1.0 + (2.0 * lcg.generate(50) - 1.0) * 0.1 registerPerf(System.timeNow() - _start27, line27) // Register performance info for line 27. registerPerf(System.timeNow() - _start22, line22, line28) // Register performance info for the block (line 22 - 28). long _start29 = System.timeNow() // Start time for line 29. plot(math.avg(val0, val1, val2), "Average pseudorandom result", color.purple) registerPerf(System.timeNow() - _start29, line29) // Register performance info for line 29. // <Additional internal code executes here> registerPerf(System.timeNow() - _startMain, total) // Register the script's overall performance info.

Note that:

- This example is **pseudocode** that provides a basic outline of the **internal calculations** the Profiler applies to collect performance data. Saving this example in the Pine Editor will result in a compilation error since `long`, `System.timeNow()`, and `registerPerf()` **do not** represent Pine Script code.
- These internal calculations that the Profiler wraps a script with require **additional** computational resources, which is why a script's runtime **increases** while profiling. Programmers should always interpret the results as **estimates** since they reflect a script's performance with the extra calculations included.

After running the wrapped script to collect performance data, *additional* internal calculations organize the results and display relevant information inside the Pine Editor:

image

The *"Line time"* calculation for code blocks also occurs at this stage, as the Profiler cannot individually wrap loop headers or the conditional statements in if or switch structures. This field's value represents the *difference* between a block's total time and the sum of its local code times, which is why the "Line time" value for a switch block or an if block with `else if` expressions represents the time spent on **all** the

structure's conditional statements, not just the block's *initial line* of code. If a programmer requires more granular information for each conditional expression in such a block, they can reorganize the logic into a *nested* if structure, as explained here.

Profiling across configurations

When a code's time complexity is not constant or its execution pattern varies with its inputs, function arguments, or available data, it's often wise to profile the code across *different configurations* and data feeds for a more well-rounded perspective on its general performance.

For example, this simple script uses a for loop to calculate the sum of squared distances between the current close price and `lengthInput` previous prices, then plots the square root of that sum on each bar. In this case, the `lengthInput` directly impacts the calculation's runtime since it determines the number of times the loop executes its local code:

//@version=6 indicator("Profiling across configurations demo") // @variable The number of previous bars in the calculation. Directly affects the number of loop iterations. int lengthInput = input.int(25, "Length", 1) //@variable The sum of squared distances from the current `close` to `lengthInput` past `close` values. float total = 0.0 // Look back across `lengthInput` bars and accumulate squared distances. for i = 1 to lengthInput float distance = close - close[i] total += distance * distance // Plot the square root of the `total`. plot(math.sqrt(total))

Let's try profiling this script with different `lengthInput` values. First, we'll use the default value of 25. The Profiler's results for this specific run show that the script completed 20,685 executions in about 96.7 milliseconds:

image

Here, we've increased the input's value to 50 in the script's settings. The results for this run show that the script's total runtime was 194.3 milliseconds, close to *twice* the time from the previous run:

image

In the next run, we changed the input's value to 200. This time, the Profiler's results show that the script finished all executions in approximately 0.8 seconds, around *four times* the previous run's time:

image

We can see from these observations that the script's runtime appears to scale *linearly* with the `lengthInput` value, excluding other factors that may affect performance, as one might expect since the bulk of the script's calculations occur within the loop and the input's value controls how many times the loop must execute.

Repetitive profiling

The runtime resources available to a script *vary* over time. Consequently, the time it takes to evaluate a code region, even one with constant complexity, *fluctuates* across executions, and the cumulative performance results shown by the Profiler **will vary** with each independent script run.

Users can enhance their analysis by *restarting* a script several times and profiling each independent run. Averaging the results from each profiled run and evaluating the dispersion of runtime results can help users establish more robust performance benchmarks and reduce the impact of *outliers* (abnormally long or short runtimes) in their conclusions.

Incorporating a *dummy input* (i.e., an input that does nothing) into a script's code is a simple technique that enables users to *restart* it while profiling. The input will not directly affect any calculations or outputs. However, as the user changes its value in the script's settings, the script restarts and the Profiler re-analyzes the executed code.

For example, this script queues pseudorandom values with a constant seed through an array with a fixed size, and it calculates and plots the array.avg() value on each bar. For profiling purposes, the script includes a `dummyInput` variable with an input.int() value assigned to it. The input does nothing in the code aside from allowing us to *restart* the script each time we change its value:

//@version=6 indicator("Repetitive profiling demo") //@variable An input not connected to script calculations. Changing its value in the "Inputs" tab restarts the script. int dummyInput = input.int(0, "Dummy input") //@variable An array of pseudorandom values. var array<float> randValues = array.new<float>(2500, 0.0) // Push a new `math.random()` value with a fixed `seed` into the `randValues` array and remove the oldest value. array.push(randValues, math.random(seed = 12345)) array.shift(randValues) // Plot the average of all elements in the `randValues` array. plot(array.avg(randValues), "Pseudorandom average")

After the first script run, the Profiler shows that it took 308.6 milliseconds to execute across all of the chart's data:

image

Now, let's change the dummy input's value in the script's settings to restart it without changing the calculations. This time, it completed the same code executions in 424.6 milliseconds, 116 milliseconds longer than the previous run:

image

Restarting the script again yields another new result. On the third run, the script finished all code executions in 227.4 milliseconds, the shortest time so far:

image

After repeating this process several times and documenting the results from each run, one can manually calculate their *average* to estimate the script's expected total runtime:

```
AverageTime = (time1 + time2 + ... + timeN) / N
```

Optimization

Code optimization, not to be confused with indicator or strategy optimization, involves modifying a script's source code for improved execution time, resource efficiency, and scalability. Programmers may

use various approaches to optimize a script when they need enhanced runtime performance, depending on what a script's calculations entail.

Fundamentally, most techniques one will use to optimize Pine code involve *reducing* the number of times critical calculations occur or *replacing* significant calculations with simplified formulas or built-ins. Both of these paradigms often overlap.

The following sections explain several straightforward concepts programmers can apply to optimize their Pine Script code.

Using built-ins

Pine Script features a variety of *built-in* functions and variables that help streamline script creation. Many of Pine's built-ins feature internal optimizations to help maximize efficiency and minimize execution time. As such, one of the simplest ways to optimize Pine code is to utilize these efficient built-ins in a script's calculations when possible.

Let's look at an example where one can replace user-defined calculations with a concise built-in call to substantially improve performance. Suppose a programmer wants to calculate the highest value of a series over a specified number of bars. Someone not familiar with all of Pine's built-ins might approach the task using a code like the following, which uses a loop on each bar to compare `length` historical values of a `source` series:

//@variable A user-defined function to calculate the highest `source` value over `length` bars. pineHighest(float source, int length) => float result = na if bar_index + 1 >= length result := source if length > 1 for i = 1 to length - 1 result := math.max(result, source[i]) result

Alternatively, one might devise a more optimized Pine function by reducing the number of times the loop executes, as iterating over the history of the `source` to achieve the result is only necessary when specific conditions occur:

//@variable A faster user-defined function to calculate the highest `source` value over `length` bars. // This version only requires a loop when the highest value is removed from the window, the `length` //

changes, or when the number of bars first becomes sufficient to calculate the result. fasterPineHighest(float source, int length) => var float result = na if source[length] == result or length != length[1] or bar_index + 1 == length result := source if length > 1 for i = 1 to length - 1 result := math.max(result, source[i]) else result := math.max(result, source) result

The built-in ta.highest() function will outperform **both** of these implementations, as its internal calculations are highly optimized for efficient execution. Below, we created a script that plots the results of calling `pineHighest()`, `fasterPineHighest()`, and ta.highest() to compare their performance using the Profiler:

//@version=6 indicator("Using built-ins demo") //@variable A user-defined function to calculate the highest `source` value over `length` bars. pineHighest(float source, int length) => float result = na if bar_index + 1 >= length result := source if length > 1 for i = 1 to length - 1 result := math.max(result, source[i]) result //@variable A faster user-defined function to calculate the highest `source` value over `length` bars. // This version only requires a loop when the highest value is removed from the window, the `length` // changes, or when the number of bars first becomes sufficient to calculate the result. fasterPineHighest(float source, int length) => var float result = na if source[length] == result or length != length[1] or bar_index + 1 == length result := source if length > 1 for i = 1 to length - 1 result := math.max(result, source[i]) else result := math.max(result, source) result plot(pineHighest(close, 20)) plot(fasterPineHighest(close, 20)) plot(ta.highest(close, 20))

The profiled results over 20,735 script executions show the call to `pineHighest()` took the most time to execute, with a runtime of 57.9 milliseconds, about 69.3% of the script's total runtime. The `fasterPineHighest()` call performed much more efficiently, as it only took about 16.9 milliseconds, approximately 20.2% of the total runtime, to calculate the same values.

The most efficient *by far*, however, was the ta.highest() call, which only required 3.2 milliseconds (~3.8% of the total runtime) to execute across all the chart's data and compute the same values in this run:

image

While these results effectively demonstrate that the built-in function outperforms our user-defined functions with a small `length` argument of 20, it's crucial to consider that the calculations required by the functions *will vary* with the argument's value. Therefore, we can profile the code while using different arguments to gauge how its runtime scales.

Here, we changed the `length` argument in each function call from 20 to 200 and profiled the script again to observe the changes in performance. The time spent on the `pineHighest()` function in this run increased to about 0.6 seconds (~86% of the total runtime), and the time spent on the `fasterPineHighest()` function increased to about 75 milliseconds. The ta.highest() function, on the other hand, *did not* experience a substantial runtime change. It took about 5.8 milliseconds this time, only a couple of milliseconds more than the previous run.

In other words, while our user-defined functions experienced significant runtime growth with a higher `length` argument in this run, the change in the built-in ta.highest() function's runtime was relatively marginal in this case, thus further emphasizing its performance benefits:

image

Note that:

- In many scenarios, a script's runtime can benefit from using built-ins where applicable. However, the relative performance edge achieved from using built-ins depends on a script's *high-impact code* and the specific built-ins used. In any case, one should always profile their scripts, preferably several times, when exploring optimized solutions.
- The calculations performed by the functions in this example also depend on the sequence of the chart's data. Therefore, programmers can gain further insight into their general performance by profiling the script across different datasets as well.

Reducing repetition

The Pine Script compiler can automatically simplify some types of repetitive code without a programmer's intervention. However, this automatic process has its limitations. If a script contains repetitive calculations that the compiler *cannot* reduce, programmers can reduce the repetition *manually* to improve their script's performance.

For example, this script contains a `valuesAbove()` method that counts the number of elements in an array above the element at a specified index. The script plots the number of values above the element at the last index of a `data` array with a calculated `plotColor`. It calculates the `plotColor` within a switch structure that calls `valuesAbove()` in all 10 of its conditional expressions:

//@version=6 indicator("Reducing repetition demo") //@function Counts the number of elements in `this` array above the element at a specified `index`. method valuesAbove(array<float> this, int index) => int result = 0 float reference = this.get(index) for [i, value] in this if i == index continue if value > reference result += 1 result //@variable An array containing the most recent 100 `close` prices. var array<float> data = array.new<float>(100) data.push(close) data.shift() //@variable Returns `color.purple` with a varying transparency based on the `valuesAbove()`. color plotColor = switch data.valuesAbove(99) <= 10 => color.new(color.purple, 90) data.valuesAbove(99) <= 20 => color.new(color.purple, 80) data.valuesAbove(99) <= 30 => color.new(color.purple, 70) data.valuesAbove(99) <= 40 => color.new(color.purple, 60) data.valuesAbove(99) <= 50 => color.new(color.purple, 50) data.valuesAbove(99) <= 60 => color.new(color.purple, 40) data.valuesAbove(99) <= 70 => color.new(color.purple, 30) data.valuesAbove(99) <= 80 => color.new(color.purple, 20) data.valuesAbove(99) <= 90 => color.new(color.purple, 10) data.valuesAbove(99) <= 100 => color.new(color.purple, 0) // Plot the number values in the `data` array above the value at its last index. plot(data.valuesAbove(99), color = plotColor, style = plot.style_area)

The profiled results for this script show that it spent about 2.5 seconds executing 21,201 times. The code regions with the highest impact on the script's runtime are the for loop within the `valuesAbove()` local scope starting on line 8 and the switch block that starts on line 21:

image

Notice that the number of executions shown for the local code within `valuesAbove()` is substantially *greater* than the number shown for the code in the script's global scope, as the script calls the method up to 11 times per execution, and the results for a function's local code reflect the *combined* time and executions from each separate call:

image

Although each `valuesAbove()` call uses the *same* arguments and returns the *same* result, the compiler cannot automatically reduce this code for us during translation. We will need to do the job ourselves. We can optimize this script by assigning the value of `data.valuesAbove(99)` to a *variable* and *reusing* the value in all other areas requiring the result.

In the version below, we modified the script by adding a `count` variable to reference the `data.valuesAbove(99)` value. The script uses this variable in the `plotColor` calculation and the plot() call:

//@version=6 indicator("Reducing repetition demo") //@function Counts the number of elements in `this` array above the element at a specified `index`. method valuesAbove(array<float> this, int index) => int result = 0 float reference = this.get(index) for [i, value] in this if i == index continue if value > reference result += 1 result //@variable An array containing the most recent 100 `close` prices. var array<float> data = array.new<float>(100) data.push(close) data.shift() //@variable The number values in the `data` array above the value at its last index. int count = data.valuesAbove(99) // @variable Returns `color.purple` with a varying transparency based on the `valuesAbove()`. color plotColor = switch count <= 10 => color.new(color.purple, 90) count <= 20 => color.new(color.purple, 80) count <= 30 => color.new(color.purple, 70) count <= 40 => color.new(color.purple, 60) count <= 50 => color.new(color.purple, 50)

count <= 60 => color.new(color.purple, 40) count <= 70 => color.new(color.purple, 30) count <= 80 => color.new(color.purple, 20) count <= 90 => color.new(color.purple, 10) count <= 100 => color.new(color.purple, 0) // Plot the `count`. plot(count, color = plotColor, style = plot.style_area)

With this modification, the profiled results show a significant improvement in performance, as the script now only needs to evaluate the `valuesAbove()` call **once** per execution rather than up to 11 separate times:

image

Note that:

- Since this script only calls `valuesAbove()` once, the method's local code will now reflect the results from that specific call. See this section to learn more about interpreting profiled function and method call results.

Minimizing `request.*()` calls

The built-in functions in the `request.*()` namespace allow scripts to retrieve data from other contexts. While these functions provide utility in many applications, it's important to consider that each call to these functions can have a significant impact on a script's resource usage.

A single script can contain up to 40 calls to the `request.*()` family of functions. However, users should strive to keep their scripts' `request.*()` calls well *below* this limit to keep the performance impact of their data requests as low as possible.

When a script requests the values of several expressions from the *same* context with multiple request.security() or request.security_lower_tf() calls, one effective way to optimize such requests is to *condense* them into a single `request.*()` call that uses a tuple as its `expression` argument. This optimization not only helps improve the runtime of the requests; it also helps reduce the script's *memory usage* and compiled size.

As a simple example, the following script requests nine ta.percentrank() values with different lengths from a specified symbol using nine separate calls to request.security(). It then plots all nine requested values on the chart to utilize them in the outputs:

//@version=6 indicator("Minimizing `request.*()` calls demo") // @variable The symbol to request data from. string symbolInput = input.symbol("BINANCE:BTCUSDT", "Symbol") // Request 9 `ta.percentrank()` values from the `symbolInput` context using 9 `request.security()` calls. float reqRank1 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 10)) float reqRank2 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 20)) float reqRank3 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 30)) float reqRank4 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 40)) float reqRank5 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 50)) float reqRank6 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 60)) float reqRank7 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 70)) float reqRank8 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 80)) float reqRank9 = request.security(symbolInput, timeframe.period, ta.percentrank(close, 90)) // Plot the `reqRank*` values. plot(reqRank1) plot(reqRank2) plot(reqRank3) plot(reqRank4) plot(reqRank5) plot(reqRank6) plot(reqRank7) plot(reqRank8) plot(reqRank9)

The results from profiling the script show that it took the script 340.8 milliseconds to complete its requests and plot the values in this run:

image

Since all the request.security() calls request data from the **same context**, we can optimize the code's resource usage by merging all of them into a single request.security() call that uses a tuple as its `expression` argument:

//@version=6 indicator("Minimizing `request.*()` calls demo") // @variable The symbol to request data from. string symbolInput =

input.symbol("BINANCE:BTCUSDT", "Symbol") // Request 9 `ta.percentrank()` values from the `symbolInput` context using a single `request.security()` call. [reqRank1, reqRank2, reqRank3, reqRank4, reqRank5, reqRank6, reqRank7, reqRank8, reqRank9] = request.security(symbolInput, timeframe.period, [ta.percentrank(close, 10), ta.percentrank(close, 20), ta.percentrank(close, 30), ta.percentrank(close, 40), ta.percentrank(close, 50), ta.percentrank(close, 60), ta.percentrank(close, 70), ta.percentrank(close, 80), ta.percentrank(close, 90)]) // Plot the `reqRank*` values. plot(reqRank1) plot(reqRank2) plot(reqRank3) plot(reqRank4) plot(reqRank5) plot(reqRank6) plot(reqRank7) plot(reqRank8) plot(reqRank9)

As we see below, the profiled results from running this version of the script show that it took 228.3 milliseconds this time, a decent improvement over the previous run:

image

Note that:

- The computational resources available to a script **fluctuate** over time. As such, it's typically a good idea to profile a script multiple times to help solidify performance conclusions.
- Another way to request multiple values from the same context with a single `request.*()` call is to pass an object of a user-defined type (UDT) as the `expression` argument. See this section of the Other timeframes and data page to learn more about requesting UDTs.
- Programmers can also reduce the total runtime of a request.security(), request.security_lower_tf(), or request.seed() call by passing an argument to the function's `calc_bars_count` parameter, which *restricts* the number of *historical* data points it can access from a context and execute required calculations on. In general, if calls to these `request.*()` functions retrieve *more* historical data than what a script *needs*, limiting the requests with `calc_bars_count` can help improve the script's performance.

Avoiding redrawing

Pine Script's drawing types allow scripts to draw custom visuals on a chart that one cannot achieve through other outputs such as plots. While these types provide greater visual flexibility, they also have a *higher* runtime and memory cost, especially when a script unnecessarily *recreates* drawings instead of directly updating their properties to change their appearance.

Most drawing types, excluding polylines, feature built-in *setter functions* in their namespaces that allow scripts to modify a drawing *without* deleting and recreating it. Utilizing these setters is typically less computationally expensive than creating a new drawing object when only *specific properties* require modification.

For example, the script below compares deleting and redrawing boxes to using `box.set*()` functions. On the first bar, it declares the `redrawnBoxes` and `updatedBoxes` arrays and executes a loop to push 25 box elements into them.

The script uses a separate for loop to iterate across the arrays and update the drawings on each execution. It *recreates* the boxes in the `redrawnBoxes` array using box.delete() and box.new(), whereas it *directly modifies* the properties of the boxes in the `updatedBoxes` array using box.set_lefttop() and box.set_rightbottom(). Both approaches achieve the same visual result. However, the latter is more efficient:

//@version=6 indicator("Avoiding redrawing demo") //@variable An array of `box` IDs deleted with `box.delete()` and redrawn with `box.new()` on each execution. var array<box> redrawnBoxes = array.new<box>() //@variable An array of `box` IDs with properties that update across executions update via `box.set*()` functions. var array<box> updatedBoxes = array.new<box>() // Populate both arrays with 25 elements on the first bar. if barstate.isfirst for i = 1 to 25 array.push(redrawnBoxes, box(na)) array.push(updatedBoxes, box.new(na, na, na, na)) for i = 0 to 24 // Calculate coordinates. int x = bar_index - i float y = close[i + 1] - close // Get the `box` ID from each array at the `i` index. box redrawnBox = redrawnBoxes.get(i) box updatedBox = updatedBoxes.get(i) // Delete the `redrawnBox`, create

a new `box` ID, and replace that element in the `redrawnboxes` array. box.delete(redrawnBox) redrawnBox := box.new(x - 1, y, x, 0.0) array.set(redrawnBoxes, i, redrawnBox) // Update the properties of the `updatedBox` rather than redrawing it. box.set_lefttop(updatedBox, x - 1, y) box.set_rightbottom(updatedBox, x, 0.0)

The results from profiling this script show that line 24, which contains the box.new() call, is the *heaviest* line in the code block that executes on each bar, with a runtime close to **double** the combined time spent on the box.set_lefttop() and box.set_rightbottom() calls on lines 27 and 28:

image

Note that:

- The number of executions shown for the loop's *local code* is 25 times the number shown for the code in the script's *global scope*, as each execution of the loop statement triggers 25 executions of the local block.
- This script updates its drawings over *all bars* in the chart's history for **testing** purposes. However, it does **not** actually need to execute all these historical updates since users will only see the **final** result from the *last historical bar* and the changes across *realtime bars*. See the next section to learn more.

Reducing drawing updates

When a script produces drawing objects that change across *historical bars*, users will only ever see their **final results** on those bars since the script completes its historical executions when it first loads on the chart. The only time one will see such drawings *evolve* across executions is during *realtime bars*, as new data flows in.

Since the evolving outputs from dynamic drawings on historical bars are **never visible** to a user, one can often improve a script's performance by *eliminating* the historical updates that don't impact the final results.

For example, this script creates a table with two columns and 21 rows to visualize the history of an RSI in a paginated, tabular format. The script initializes the cells of the `infoTable` on the first bar, and it references the history of the calculated `rsi` to update the `text` and `bgcolor` of the cells in the second column within a for loop on each bar:

//@version=6 indicator("Reducing drawing updates demo") //@variable The first offset shown in the paginated table. int offsetInput = input.int(0, "Page", 0, 249) * 20 //@variable A table that shows the history of RSI values. var table infoTable = table.new(position.top_right, 2, 21, border_color = chart.fg_color, border_width = 1) // Initialize the table's cells on the first bar. if barstate.isfirst table.cell(infoTable, 0, 0, "Offset", text_color = chart.fg_color) table.cell(infoTable, 1, 0, "RSI", text_color = chart.fg_color) for i = 0 to 19 table.cell(infoTable, 0, i + 1, str.tostring(offsetInput + i)) table.cell(infoTable, 1, i + 1) float rsi = ta.rsi(close, 14) // Update the history shown in the `infoTable` on each bar. for i = 0 to 19 float historicalRSI = rsi[offsetInput + i] table.cell_set_text(infoTable, 1, i + 1, str.tostring(historicalRSI)) table.cell_set_bgcolor(infoTable, 1, i + 1, color.from_gradient(historicalRSI, 30, 70, color.red, color.green)) plot(rsi, "RSI")

After profiling the script, we see that the code with the highest impact on performance is the for loop that starts on line 20, i.e., the code block that updates the table's cells:

image

This critical code region executes **excessively** across the chart's history, as users will only see the table's **final** historical result. The only time that users will see the table update is on the **last historical bar** and across all subsequent **realtime bars**. Therefore, we can optimize this script's resource usage by restricting the executions of this code to only the last available bar.

In this script version, we placed the loop that updates the table cells within an if structure that uses barstate.islast as its condition, effectively restricting the code block's executions to only the last historical bar and all realtime bars. Now, the script *loads* more

efficiently since all the table's calculations only require **one** historical execution:

image

//@version=6 indicator("Reducing drawing updates demo") //@variable The first offset shown in the paginated table. int offsetInput = input.int(0, "Page", 0, 249) * 20 //@variable A table that shows the history of RSI values. var table infoTable = table.new(position.top_right, 2, 21, border_color = chart.fg_color, border_width = 1) // Initialize the table's cells on the first bar. if barstate.isfirst table.cell(infoTable, 0, 0, "Offset", text_color = chart.fg_color) table.cell(infoTable, 1, 0, "RSI", text_color = chart.fg_color) for i = 0 to 19 table.cell(infoTable, 0, i + 1, str.tostring(offsetInput + i)) table.cell(infoTable, 1, i + 1) float rsi = ta.rsi(close, 14) // Update the history shown in the `infoTable` on the last available bar. if barstate.islast for i = 0 to 19 float historicalRSI = rsi[offsetInput + i] table.cell_set_text(infoTable, 1, i + 1, str.tostring(historicalRSI)) table.cell_set_bgcolor(infoTable, 1, i + 1, color.from_gradient(historicalRSI, 30, 70, color.red, color.green)) plot(rsi, "RSI")

Note that:

- The script will still update the cells when new **realtime** updates come in, as users can observe those changes on the chart, unlike the changes that the script used to execute across historical bars.

Storing calculated values

When a script performs a critical calculation that changes *infrequently* throughout all executions, one can reduce its runtime by **saving the result** to a variable declared with the var or varip keywords and **only** updating the value if the calculation changes. If the script calculates *multiple* values excessively, one can store them within collections, matrices, and maps or objects of user-defined types.

Let's look at an example. This script calculates a weighted moving average with custom weights based on a generalized window function. The `numerator` is the sum of weighted close values, and the

`denominator` is the sum of the calculated weights. The script uses a for loop that iterates `lengthInput` times to calculate these sums, then it plots their ratio, i.e., the resulting average:

//@version=6 indicator("Storing calculated values demo", overlay = true) //@variable The number of bars in the weighted average calculation. int lengthInput = input.int(50, "Length", 1, 5000) // @variable Window coefficient. float coefInput = input.float(0.5, "Window coefficient", 0.0, 1.0, 0.01) //@variable The sum of weighted `close` prices. float numerator = 0.0 //@variable The sum of weights. float denominator = 0.0 //@variable The angular step in the cosine calculation. float step = 2.0 * math.pi / lengthInput // Accumulate weighted sums. for i = 0 to lengthInput - 1 float weight = coefInput - (1 - coefInput) * math.cos(step * i) numerator += close[i] * weight denominator += weight // Plot the weighted average result. plot(numerator / denominator, "Weighted average", color.purple, 3)

After profiling the script's performance over our chart's data, we see that it took about 241.3 milliseconds to calculate the default 50-bar average across 20,155 chart updates, and the critical code with the *highest impact* on the script's performance is the loop block that starts on line 17:

image

Since the number of loop iterations *depends* on the `lengthInput` value, let's test how its runtime scales with another configuration requiring heavier looping. Here, we set the value to 2500. This time, the script took about 12 seconds to complete all of its executions:

image

Now that we've pinpointed the script's *high-impact* code and established a benchmark to improve, we can inspect the critical code block to identify optimization opportunities. After examining the calculations, we can observe the following:

- The only value that causes the `weight` calculation on line 18 to vary across loop iterations is the *loop index*. All other values in its calculation remain consistent. Consequently, the `weight`

calculated on each loop iteration **does not vary** across chart bars. Therefore, rather than calculating the weights on **every update**, we can calculate them **once**, on the first bar, and **store them** in a collection for future access across subsequent script executions.
- Since the weights never change, the resulting `denominator` never changes. Therefore, we can add the var keyword to the variable declaration and only calculate its value **once** to reduce the number of executed addition assignment (+=) operations.
- Unlike the `denominator`, we **cannot** store the `numerator` value to simplify its calculation since it consistently *changes* over time.

In the modified script below, we've added a `weights` variable to reference an array that stores each calculated `weight`. This variable and the `denominator` both include the var keyword in their declarations, meaning the values assigned to them will *persist* throughout all script executions until explicitly reassigned. The script calculates their values using a for loop that executes only on the first chart bar. Across all other bars, it calculates the `numerator` using a for...in loop that references the *saved values* from the `weights` array:

//@version=6 indicator("Storing calculated values demo", overlay = true) //@variable The number of bars in the weighted average calculation. int lengthInput = input.int(50, "Length", 1, 5000) // @variable Window coefficient. float coefInput = input.float(0.5, "Window coefficient", 0.0, 1.0, 0.01) //@variable An array that stores the `weight` values calculated on the first chart bar. var array<float> weights = array.new<float>() //@variable The sum of weighted `close` prices. float numerator = 0.0 //@variable The sum of weights. The script now only calculates this value on the first bar. var float denominator = 0.0 //@variable The angular step in the cosine calculation. float step = 2.0 * math.pi / lengthInput // Populate the `weights` array and calculate the `denominator` only on the first bar. if barstate.isfirst for i = 0 to lengthInput - 1 float weight = coefInput - (1 - coefInput) * math.cos(step * i) array.push(weights, weight) denominator += weight // Calculate the `numerator` on each bar using the stored `weights`. for [i, w] in weights numerator += close[i] * w //

Plot the weighted average result. plot(numerator / denominator, "Weighted average", color.purple, 3)

With this optimized structure, the profiled results show that our modified script with a high `lengthInput` value of 2500 took about 5.9 seconds to calculate across the same data, about *half* the time of our previous version:

image

Note that:

- Although we've significantly improved this script's performance by saving its *execution-invariant* values to variables, it does still involve a higher computational cost with **large** `lengthInput` values due to the remaining loop calculations that execute on each bar.
- Another, more *advanced* way one can further enhance this script's performance is by storing the weights in a *single-row* matrix on the first bar, using an array as a queue to hold recent close values, then replacing the for...in loop with a call to matrix.mult(). See the Matrices page to learn more about working with `matrix.*()` functions.

Eliminating loops

Loops allow Pine scripts to perform *iterative* calculations on each execution. Each time a loop activates, its local code may execute *several times*, often leading to a *substantial increase* in resource usage.

Pine loops are necessary for *some* calculations, such as manipulating elements within collections or looking backward through a dataset's history to calculate values *only* obtainable on the current bar. However, in many other cases, programmers use loops when they **don't need to**, leading to suboptimal runtime performance. In such cases, one may eliminate unnecessary loops in any of the following ways, depending on what their calculations entail:

- Identifying simplified, **loop-free expressions** that achieve the same result without iteration

- Replacing a loop with optimized built-ins where possible
- Distributing a loop's iterations *across bars* when feasible rather than evaluating them all at once

This simple example contains an `avgDifference()` function that calculates the average difference between the current bar's `source` value and all the values from `length` previous bars. The script calls this function to calculate the average difference between the current close price and `lengthInput` previous prices, then it plots the result on the chart:

//@version=6 indicator("Eliminating loops demo") //@variable The number of bars in the calculation. int lengthInput = input.int(20, "Length", 1) //@function Calculates the average difference between the current `source` and `length` previous `source` values. avgDifference(float source, int length) => float diffSum = 0.0 for i = 1 to length diffSum += source - source[i] diffSum / length plot(avgDifference(close, lengthInput))

After inspecting the script's profiled results with the default settings, we see that it took about 64 milliseconds to execute 20,157 times:

image

Since we use the `lengthInput` as the `length` argument in the `avgDifference()` call and that argument controls how many times the loop inside the function must iterate, our script's runtime will **grow** with the `lengthInput` value. Here, we set the input's value to 2000 in the script's settings. This time, the script completed its executions in about 3.8 seconds:

image

As we see from these results, the `avgDifference()` function can be costly to call, depending on the specified `lengthInput` value, due to its for loop that executes on each bar. However, loops are **not** necessary to achieve the output. To understand why, let's take a closer look at the loop's calculations. We can represent them with the following expression:

(source - source[1]) + (source - source[2]) + ... + (source - source[length])

Notice that it adds the *current* `source` value `length` times. These iterative additions are not necessary. We can simplify that part of the expression to `source * length`, which reduces it to the following:

source * length - source[1] - source[2] - ... - source[length]

or equivalently:

source * length - (source[1] + source[2] + ... + source[length])

After simplifying and rearranging this representation of the loop's calculations, we see that we can compute the result in a simpler way and **eliminate** the loop by subtracting the previous bar's rolling sum (math.sum()) of `source` values from the `source * length` value, i.e.:

source * length - math.sum(source, length)[1]

The `fastAvgDifference()` function below is a **loop-free** alternative to the original `avgDifference()` function that uses the above expression to calculate the sum of `source` differences, then divides the expression by the `length` to return the average difference:

//@function A faster way to calculate the `avgDifference()` result. // Eliminates the `for` loop using the relationship: // `(x - x[1]) + (x - x[2]) + ... + (x - x[n]) = x * n - math.sum(x, n)[1]`. fastAvgDifference(float source, int length) => (source * length - math.sum(source, length)[1]) / length

Now that we've identified a potential optimized solution, we can compare the performance of `fastAvgDifference()` to the original `avgDifference()` function. The script below is a modified form of the previous version that plots the results from calling both functions with the `lengthInput` as the `length` argument:

//@version=6 indicator("Eliminating loops demo") //@variable The number of bars in the calculation. int lengthInput = input.int(20, "Length", 1) //@function Calculates the average difference between the current `source` and `length` previous `source` values.

avgDifference(float source, int length) => float diffSum = 0.0 for i = 1 to length diffSum += source - source[i] diffSum / length //@function A faster way to calculate the `avgDifference()` result. // Eliminates the `for` loop using the relationship: // `(x - x[1]) + (x - x[2]) + ... + (x - x[n]) = x * n - math.sum(x, n)[1]`. fastAvgDifference(float source, int length) => (source * length - math.sum(source, length)[1]) / length plot(avgDifference(close, lengthInput)) plot(fastAvgDifference(close, lengthInput))

The profiled results for the script with the default `lengthInput` of 20 show a substantial difference in runtime spent on the two function calls. The call to the original function took about 47.3 milliseconds to execute 20,157 times on this run, whereas our optimized function only took 4.5 milliseconds:

image

Now, let's compare the performance with the *heavier* `lengthInput` value of 2000. As before, the runtime spent on the `avgDifference()` function increased significantly. However, the time spent executing the `fastAvgDifference()` call remained very close to the result from the previous configuration. In other words, while our original function's runtime scales directly with its `length` argument, our optimized function demonstrates relatively *consistent* performance since it does not require a loop:

image

Optimizing loops

Although Pine's execution model and the available built-ins often *eliminate* the need for loops in many cases, there are still instances where a script **will** require loops for some types of tasks, including:

- Manipulating collections or executing calculations over a collection's elements when the available built-ins **will not** suffice
- Performing calculations across historical bars that one **cannot** achieve with simplified *loop-free* expressions or optimized *built-ins*
- Calculating values that are **only** obtainable through iteration

When a script uses loops that a programmer cannot eliminate, there are several techniques one can use to reduce their performance impact. This section explains two of the most common, useful techniques that can help improve a required loop's efficiency.

Reducing loop calculations

The code executed within a loop's local scope can have a **multiplicative** impact on its overall runtime, as each time a loop statement executes, it will typically trigger *several* iterations of the local code. Therefore, programmers should strive to keep a loop's calculations as simple as possible by eliminating unnecessary structures, function calls, and operations to minimize the performance impact, especially when the script must evaluate its loops *numerous times* throughout all its executions.

For example, this script contains a `filteredMA()` function that calculates a moving average of up to `length` unique `source` values, depending on the `true` elements in a specified `mask` array. The function queues the unique `source` values into a `data` array, uses a for...in loop to iterate over the `data` and calculate the `numerator` and `denominator` sums, then returns the ratio of those sums. Within the loop, it only adds values to the sums when the `data` element is not na and the `mask` element at the `index` is `true`. The script utilizes this user-defined function to calculate the average of up to 100 unique close prices filtered by a `randMask` and plots the result on the chart:

//@version=6 indicator("Reducing loop calculations demo", overlay = true) //@function Calculates a moving average of up to `length` unique `source` values filtered by a `mask` array. filteredMA(float source, int length, array<bool> mask) => // Raise a runtime error if the size of the `mask` doesn't equal the `length`. if mask.size() != length runtime.error("The size of the `mask` array used in the `filteredMA()` call must match the `length`.") //@variable An array containing `length` unique `source` values. var array<float> data = array.new<float>(length) // Queue unique `source` values into the `data` array. if not data.includes(source) data.push(source) data.shift() // The numerator and denominator of the average. float

numerator = 0.0 float denominator = 0.0 // Loop to calculate sums. for item in data if na(item) continue int index = array.indexof(data, item) if mask.get(index) numerator += item denominator += 1.0 // Return the average, or the last non-`na` average value if the current value is `na`. fixnan(numerator / denominator) //@variable An array of 100 pseudorandom "bool" values. var array<bool> randMask = array.new<bool>(100, true) // Push the first element from `randMask` to the end and queue a new pseudorandom value. randMask.push(randMask.shift()) randMask.push(math.random(seed = 12345) < 0.5) randMask.shift() // Plot the `filteredMA()` of up to 100 unique `close` values filtered by the `randMask`. plot(filteredMA(close, 100, randMask))

After profiling the script, we see it took about two seconds to execute 21,778 times. The code with the highest performance impact is the expression on line 37, which calls the `filteredMA()` function. Within the `filteredMA()` function's scope, the for...in loop has the highest impact, with the `index` calculation in the loop's scope (line 22) contributing the most to the loop's runtime:

image

The above code demonstrates suboptimal usage of a for...in loop, as we **do not** need to call array.indexof() to retrieve the `index` in this case. The array.indexof() function can be *costly* to call within a loop since it must search through the array's contents and locate the corresponding element's index *each time* the script calls it.

To eliminate this costly call from our for...in loop, we can use the *second form* of the structure, which produces a *tuple* containing the **index** and the element's value on each iteration:

for [index, item] in data

In this version of the script, we removed the array.indexof() call on line 22 since it is **not** necessary to achieve the intended result, and we changed the for...in loop to use the alternative form:

//@version=6 indicator("Reducing loop calculations demo", overlay = true) //@function Calculates a moving average of up to `length` unique

`source` values filtered by a `mask` array. filteredMA(float source, int length, array<bool> mask) => // Raise a runtime error if the size of the `mask` doesn't equal the `length`. if mask.size() != length runtime.error("The size of the `mask` array used in the `filteredMA()` call must match the `length`.") //@variable An array containing `length` unique `source` values. var array<float> data = array.new<float>(length) // Queue unique `source` values into the `data` array. if not data.includes(source) data.push(source) data.shift() // The numerator and denominator of the average. float numerator = 0.0 float denominator = 0.0 // Loop to calculate sums. for [index, item] in data if na(item) continue if mask.get(index) numerator += item denominator += 1.0 // Return the average, or the last non-`na` average value if the current value is `na`. fixnan(numerator / denominator) //@variable An array of 100 pseudorandom "bool" values. var array<bool> randMask = array.new<bool>(100, true) // Push the first element from `randMask` to the end and queue a new pseudorandom value. randMask.push(randMask.shift()) randMask.push(math.random(seed = 12345) < 0.5) randMask.shift() // Plot the `filteredMA()` of up to 100 unique `close` values filtered by the `randMask`. plot(filteredMA(close, 100, randMask))

With this simple change, our loop is much more efficient, as it no longer needs to redundantly search through the array on each iteration to keep track of the index. The profiled results from this script run show that it took only 0.6 seconds to complete its executions, a significant improvement over the previous version’s result:

image

Loop-invariant code motion

Loop-invariant code is any code region within a loop’s scope that produces an **unchanging** result on each iteration. When a script’s loops contain loop-invariant code, it can substantially impact performance in some cases due to excessive, **unnecessary** calculations.

Programmers can optimize a loop with invariant code by *moving* the unchanging calculations **outside** the loop's scope so the script only needs to evaluate them once per execution rather than repetitively.

The following example contains a `featureScale()` function that creates a rescaled version of an array. Within the function's for...in loop, it scales each element by calculating its distance from the array.min() and dividing the value by the array.range(). The script uses this function to create a `rescaled` version of a `prices` array, then plots the difference between the array's array.first() and array.avg() method call results on the chart:

```
//@version=6 indicator("Loop-invariant code motion demo") // @function Returns a feature scaled version of `this` array. featureScale(array<float> this) => array<float> result = array.new<float>() for item in this result.push((item - array.min(this)) / array.range(this)) result //@variable An array containing the most recent 100 `close` prices. var array<float> prices = array.new<float>(100, close) // Queue the `close` through the `prices` array. prices.unshift(close) prices.pop() //@variable A feature scaled version of the `prices` array. array<float> rescaled = featureScale(prices) // Plot the difference between the first element and the average value in the `rescaled` array. plot(rescaled.first() - rescaled.avg())
```

As we see below, the profiled results for this script after 20,187 executions show it completed its run in about 3.3 seconds. The code with the highest impact on performance is the line containing the `featureScale()` function call, and the function's critical code is the for...in loop block starting on line 7:

image

Upon examining the loop's calculations, we can see that the array.min() and array.range() calls on line 8 are **loop-invariant**, as they will always produce the **same result** across each iteration. We can make our loop much more efficient by assigning the results from these calls to variables **outside** its scope and referencing them as needed.

The `featureScale()` function in the script below assigns the array.min() and array.range() values to `minValue` and `rangeValue` variables *before* executing the for...in loop. Inside the loop's local scope, it *references* the variables across its iterations rather than repetitively calling these `array.*()` functions:

//@version=6 indicator("Loop-invariant code motion demo") // @function Returns a feature scaled version of `this` array. featureScale(array<float> this) => array<float> result = array.new<float>() float minValue = array.min(this) float rangeValue = array.range(this) for item in this result.push((item - minValue) / rangeValue) result //@variable An array containing the most recent 100 `close` prices. var array<float> prices = array.new<float>(100, close) // Queue the `close` through the `prices` array. prices.unshift(close) prices.pop() //@variable A feature scaled version of the `prices` array. array<float> rescaled = featureScale(prices) // Plot the difference between the first element and the average value in the `rescaled` array. plot(rescaled.first() - rescaled.avg())

As we see from the script's profiled results, moving the *loop-invariant* calculations outside the loop leads to a substantial performance improvement. This time, the script completed its executions in only 289.3 milliseconds:

image

Minimizing historical buffer calculations

Pine scripts create *historical buffers* for all variables and function calls their outputs depend on. Each buffer contains information about the range of historical values the script can access with the history-referencing operator [].

A script *automatically* determines the required buffer size for all its variables and function calls by analyzing the historical references executed during the **first 244 bars** in a dataset. When a script only references the history of a calculated value *after* those initial bars, it will **restart** its executions repetitively across previous bars with successively larger historical buffers until it either determines the

appropriate size or raises a runtime error. Those repetitive executions can significantly increase a script's runtime in some cases.

When a script *excessively* executes across a dataset to calculate historical buffers, one effective way to improve its performance is *explicitly* defining suitable buffer sizes using the max_bars_back() function. With appropriate buffer sizes declared explicitly, the script does not need to re-execute across past data to determine the sizes.

For example, the script below uses a polyline to draw a basic histogram representing the distribution of calculated `source` values over 500 bars. On the last available bar, the script uses a for loop to look back through historical values of the calculated `source` series and determine the chart points used by the polyline drawing. It also plots the value of `bar_index + 1` to verify the number of bars it executed across:

//@version=6 indicator("Minimizing historical buffer calculations demo", overlay = true) //@variable A polyline with points that form a histogram of `source` values. var polyline display = na //@variable The difference Q3 of `high` prices and Q1 of `low` prices over 500 bars. float innerRange = ta.percentile_nearest_rank(high, 500, 75) - ta.percentile_nearest_rank(low, 500, 25) // Calculate the highest and lowest prices, and the total price range, over 500 bars. float highest = ta.highest(500) float lowest = ta.lowest(500) float totalRange = highest - lowest //@variable The source series for histogram calculation. Its value is the midpoint between the `open` and `close`. float source = math.avg(open, close) if barstate.islast polyline.delete(display) // Calculate the number of histogram bins and their size. int bins = int(math.round(5 * totalRange / innerRange)) float binSize = totalRange / bins //@variable An array of chart points for the polyline. array<chart.point> points = array.new<chart.point>(bins, chart.point.new(na, na, na)) // Loop to build the histogram. for i = 0 to 499 //@variable The histogram bin number. Uses past values of the `source` for its calculation. // The script must execute across all previous bars AGAIN to determine the historical buffer for // `source`, as initial references to the calculated series occur AFTER the first 244 bars. int index = int((source[i] - lowest) / binSize) if na(index) continue chart.point currentPoint = points.get(index) if na(currentPoint.index) points.set(index, chart.point.from_index(bar_index + 1, (index + 0.5) *

binSize + lowest)) continue currentPoint.index += 1 // Add final points to the `points` array and draw the new `display` polyline. points.unshift(chart.point.now(lowest)) points.push(chart.point.now(highest)) display := polyline.new(points, closed = true) plot(bar_index + 1, "Number of bars", display = display.data_window)

Since the script *only* references past `source` values on the *last bar*, it will **not** construct a suitable historical buffer for the series within the first 244 bars on a larger dataset. Consequently, it will **re-execute** across all historical bars to identify the appropriate buffer size.

As we see from the profiled results after running the script across 20,320 bars, the number of *global* code executions was 162,560, which is **eight times** the number of chart bars. In other words, the script had to *repeat* the historical executions **seven more times** to determine the appropriate buffer for the `source` series in this case:

image

This script will only reference the most recent 500 `source` values on the last historical bar and all realtime bars. Therefore, we can help it establish the correct buffer *without* re-execution by defining a 500-bar referencing length with max_bars_back().

In the following script version, we added max_bars_back(source, 500) after the variable declaration to explicitly specify that the script will access up to 500 historical `source` values throughout its executions:

//@version=6 indicator("Minimizing historical buffer calculations demo", overlay = true) //@variable A polyline with points that form a histogram of `source` values. var polyline display = na //@variable The difference Q3 of `high` prices and Q1 of `low` prices over 500 bars. float innerRange = ta.percentile_nearest_rank(high, 500, 75) - ta.percentile_nearest_rank(low, 500, 25) // Calculate the highest and lowest prices, and the total price range, over 500 bars. float highest = ta.highest(500) float lowest = ta.lowest(500) float totalRange = highest - lowest //@variable The source series for histogram calculation. Its value is the midpoint between the `open` and `close`. float source = math.avg(open, close) // Explicitly define a 500-bar historical buffer for

the `source` to prevent recalculation. max_bars_back(source, 500) if barstate.islast polyline.delete(display) // Calculate the number of histogram bins and their size. int bins = int(math.round(5 * totalRange / innerRange)) float binSize = totalRange / bins //@variable An array of chart points for the polyline. array<chart.point> points = array.new<chart.point>(bins, chart.point.new(na, na, na)) // Loop to build the histogram. for i = 0 to 499 //@variable The histogram bin number. Uses past values of the `source` for its calculation. // Since the `source` now has an appropriate predefined buffer, the script no longer needs // to recalculate across previous bars to determine the referencing length. int index = int((source[i] - lowest) / binSize) if na(index) continue chart.point currentPoint = points.get(index) if na(currentPoint.index) points.set(index, chart.point.from_index(bar_index + 1, (index + 0.5) * binSize + lowest)) continue currentPoint.index += 1 // Add final points to the `points` array and draw the new `display` polyline. points.unshift(chart.point.now(lowest)) points.push(chart.point.now(highest)) display := polyline.new(points, closed = true) plot(bar_index + 1, "Number of bars", display = display.data_window)

With this change, our script no longer needs to re-execute across all the historical data to determine the buffer size. As we see in the profiled results below, the number of global code executions now aligns with the number of chart bars, and the script took substantially less time to complete all of its historical executions:

image

Note that:

- This script only requires up to the most recent 501 historical bars to calculate its drawing output. In this case, another way to optimize resource usage is to include `calc_bars_count = 501` in the indicator() function, which reduces unnecessary script executions by restricting the historical data the script can calculate across to 501 bars.

Tips

Working around Profiler overhead

Since the Pine Profiler must perform *extra calculations* to collect performance data, as explained in this section, the time it takes to execute a script **increases** while profiling.

Most scripts will run as expected with the Profiler's overhead included. However, when a complex script's runtime approaches a plan's limit, using the Profiler on it may cause its runtime to *exceed* the limit. Such a case indicates that the script likely needs optimization, but it can be challenging to know where to start without being able to profile the code. The most effective workaround in this scenario is reducing the number of bars the script must execute on. Users can achieve this reduction in any of the following ways:

- Selecting a dataset that has fewer data points in its history, e.g., a higher timeframe or a symbol with limited data
- Using conditional logic to limit code executions to a specific time or bar range
- Including a `calc_bars_count` argument in the script's declaration statement to specify how many recent historical bars it can use

Reducing the number of data points works in most cases because it directly decreases the number of times the script must execute, typically resulting in less accumulated runtime.

As a demonstration, this script contains a `gcd()` function that uses a *naive* algorithm to calculate the greatest common divisor of two integers. The function initializes its `result` using the smallest absolute value of the two numbers. Then, it reduces the value of the `result` by one within a while loop until it can divide both numbers without remainders. This structure entails that the loop will iterate up to *N* times, where *N* is the smallest of the two arguments.

In this example, the script plots the value of `gcd(10000, 10000 + bar_index)`. The smallest of the two arguments is always 10,000 in this case, meaning the while loop within the function will require up to

10,000 iterations per script execution, depending on the bar_index value:

//@version=6 indicator("Script takes too long while profiling demo") // @function Calculates the greatest common divisor of `a` and `b` using a naive algorithm. gcd(int a, int b) => //@variable The greatest common divisor. int result = math.max(math.min(math.abs(a), math.abs(b)), 1) // Reduce the `result` by 1 until it divides `a` and `b` without remainders. while result > 0 if a % result == 0 and b % result == 0 break result -= 1 // Return the `result`. result plot(gcd(10000, 10000 + bar_index), "GCD")

When we add the script to our chart, it takes a while to execute across our chart's data, but it does not raise an error. However, *after* enabling the Profiler, the script raises a runtime error stating that it exceeded the Premium plan's runtime limit (40 seconds):

image

Our current chart has over 20,000 historical bars, which may be too many for the script to handle within the alloted time while the Profiler is active. We can try limiting the number of historical executions to work around the issue in this case.

Below, we included `calc_bars_count = 10000` in the indicator() function, which limits the script's available history to the most recent 10,000 historical bars. After restricting the script's historical executions, it no longer exceeds the Premium plan's limit while profiling, so we can now inspect its performance results:

image

//@version=6 indicator("Script takes too long while profiling demo", calc_bars_count = 10000) //@function Calculates the greatest common divisor of `a` and `b` using a naive algorithm. gcd(int a, int b) => //@variable The greatest common divisor. int result = math.max(math.min(math.abs(a), math.abs(b)), 1) // Reduce the `result` by 1 until it divides `a` and `b` without remainders. while result > 0 if a % result == 0 and b % result == 0 break result -= 1 // Return the `result`. result plot(gcd(10000, 10000 + bar_index), "GCD")

Previous

Next

Limitations

Pine Script® v6 Documentation

Limitations

Introduction

As is mentioned in our Welcome page:

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

If you develop complex scripts using Pine Script®, sooner or later you will run into some of the limitations we impose. This section provides you with an overview of the limitations that you may encounter. There are currently no means for Pine Script programmers to get data on the resources consumed by their scripts. We hope this will change in the future.

In the meantime, when you are considering large projects, it is safest to make a proof of concept in order to assess the probability of your script running into limitations later in your project.

Below, we describe the limits imposed in the Pine Script environment.

Time

Script compilation

Scripts must compile before they are executed on charts. Compilation occurs when you save a script from the Pine Editor or when you add a script to the chart. A two-minute limit is imposed on compilation time, which will depend on the size and complexity of your script, and whether or not a cached version of a previous compilation is available. When a compile exceeds the two-minute limit, a warning is issued. Heed that warning by shortening your script because after three consecutives warnings a one-hour ban on compilation attempts is enforced. The first thing to consider when optimizing code is to avoid repetitions by using functions to encapsulate oft-used segments, and call functions instead of repeating code.

Script execution

Once a script is compiled it can be executed. See the Events that trigger script executions section of the Execution model page for a list of the events triggering the execution of a script. The time allotted for the script to execute on all bars of a dataset varies with account types. The limit is 20 seconds for basic accounts, 40 for others.

Loop execution

The execution time for any loop on any single bar is limited to 500 milliseconds. The outer loop of embedded loops counts as one loop, so it will time out first. Keep in mind that even though a loop may execute under the 500 ms time limit on a given bar, the time it takes to execute on all the dataset’s bars may nonetheless cause your script to exceed the total execution time limit. For example, the limit on total execution time will make it impossible for you script to execute a 400 ms loop on each bar of a 20,000-bar dataset because your script would then need 8000 seconds to execute.

Chart visuals

Plot limits

A maximum of 64 plot counts are allowed per script. The functions that generate plot counts are:

- plot()
- plotarrow()
- plotbar()
- plotcandle()
- plotchar()
- plotshape()
- alertcondition()
- bgcolor()
- fill(), but only if its `color` is of the series form.

The following functions do not generate plot counts:

- hline()
- line.new()
- label.new()
- table.new()
- box.new()

One function call can generate up to seven plot counts, depending on the function and how it is called. When your script exceeds the maximum of 64 plot counts, the runtime error message will display the plot count generated by your script. Once you reach that point, you can determine how many plot counts a function call generates by commenting it out in a script. As long as your script still throws an error, you will be able to see how the actual plot count decreases after you have commented out a line.

The following example shows different function calls and the number of plot counts each one will generate:

//@version=6 indicator("Plot count example") bool isUp = close > open color isUpColor = isUp ? color.green : color.red bool isDn = not isUp

color isDnColor = isDn ? color.red : color.green // Uses one plot count each. p1 = plot(close, color = color.white) p2 = plot(open, color = na) // Uses two plot counts for the `close` and `color` series. plot(close, color = isUpColor) // Uses one plot count for the `close` series. plotarrow(close, colorup = color.green, colordown = color.red) // Uses two plot counts for the `close` and `colorup` series. plotarrow(close, colorup = isUpColor) // Uses three plot counts for the `close`, `colorup`, and the `colordown` series. plotarrow(close - open, colorup = isUpColor, colordown = isDnColor) // Uses four plot counts for the `open`, `high`, `low`, and `close` series. plotbar(open, high, low, close, color = color.white) // Uses five plot counts for the `open`, `high`, `low`, `close`, and `color` series. plotbar(open, high, low, close, color = isUpColor) // Uses four plot counts for the `open`, `high`, `low`, and `close` series. plotcandle(open, high, low, close, color = color.white, wickcolor = color.white, bordercolor = color.purple) // Uses five plot counts for the `open`, `high`, `low`, `close`, and `color` series. plotcandle(open, high, low, close, color = isUpColor, wickcolor = color.white, bordercolor = color.purple) // Uses six plot counts for the `open`, `high`, `low`, `close`, `color`, and `wickcolor` series. plotcandle(open, high, low, close, color = isUpColor, wickcolor = isUpColor , bordercolor = color.purple) // Uses seven plot counts for the `open`, `high`, `low`, `close`, `color`, `wickcolor`, and `bordercolor` series. plotcandle(open, high, low, close, color = isUpColor, wickcolor = isUpColor , bordercolor = isUp ? color.lime : color.maroon) // Uses one plot count for the `close` series. plotchar(close, color = color.white, text = "|", textcolor = color.white) // Uses two plot counts for the `close`` and `color` series. plotchar(close, color = isUpColor, text = "—", textcolor = color.white) // Uses three plot counts for the `close`, `color`, and `textcolor` series. plotchar(close, color = isUpColor, text = "O", textcolor = isUp ? color.yellow : color.white) // Uses one plot count for the `close` series. plotshape(close, color = color.white, textcolor = color.white) // Uses two plot counts for the `close` and `color` series. plotshape(close, color = isUpColor, textcolor = color.white) // Uses three plot counts for the `close`, `color`, and `textcolor` series. plotshape(close, color = isUpColor, textcolor = isUp ? color.yellow : color.white) // Uses one plot count. alertcondition(close > open, "close > open", "Up bar alert") // Uses one plot count. bgcolor(isUp ?

color.yellow : color.white) // Uses one plot count for the `color` series. fill(p1, p2, color = isUpColor)

This example generates a plot count of 56. If we were to add two more instances of the last call to plotcandle(), the script would throw an error stating that the script now uses 70 plot counts, as each additional call to plotcandle() generates seven plot counts, and 56 + (7 * 2) is 70.

Line, box, polyline, and label limits

Contrary to plots, which can cover the chart's entire dataset, scripts will only show the last 50 lines, boxes, polylines, and labels on the chart by default. One can increase the maximum number for each of these drawing types via the `max_lines_count`, `max_boxes_count`, `max_polylines_count`, and `max_labels_count` parameters of the script's indicator() or strategy() declaration statement. The maximum number of line, box, and label IDs is 500, and the maximum number of polyline IDs is 100.

In this example, we set the maximum number of recent labels shown on the chart to 100:

//@version=6 indicator("Label limits example", max_labels_count = 100, overlay = true) label.new(bar_index, high, str.tostring(high, format.mintick))

It's important to note when setting any of a drawing object's properties to na that its ID still exists and thus contributes to a script's drawing totals. To demonstrate this behavior, the following script draws a "Buy" and "Sell" label on each bar, with x values determined by the `longCondition` and `shortCondition` variables.

The "Buy" label's x value is na when the bar index is even, and the "Sell" label's x value is na when the bar index is odd. Although the `max_labels_count` is 10 in this example, we can see that the script displays fewer than 10 labels on the chart since the ones with na values also count toward the total:

image

//@version=6 // Approximate maximum number of label drawings MAX_LABELS = 10 indicator("labels with na", overlay = false, max_labels_count = MAX_LABELS) // Add background color for the last MAX_LABELS bars. bgcolor(bar_index > last_bar_index - MAX_LABELS ? color.new(color.green, 80) : na) longCondition = bar_index % 2 != 0 shortCondition = bar_index % 2 == 0 // Add "Buy" and "Sell" labels on each new bar. label.new(longCondition ? bar_index : na, 0, text = "Buy", color = color.new(color.green, 0), style = label.style_label_up) label.new(shortCondition ? bar_index : na, 0, text = "Sell", color = color.new(color.red, 0), style = label.style_label_down) plot(longCondition ? 1 : 0) plot(shortCondition ? 1 : 0)

To display the desired number of labels, we must eliminate label drawings we don’t want to show rather than setting their properties to na. The example below uses an if structure to conditionally draw the “Buy” and “Sell” labels, preventing the script from creating new label IDs when it isn’t necessary:

image

//@version=6 // Approximate maximum number of label drawings MAX_LABELS = 10 indicator("conditional labels", overlay = false, max_labels_count = MAX_LABELS) // Add background color for the last MAX_LABELS bars. bgcolor(bar_index > last_bar_index - MAX_LABELS ? color.new(color.green, 80) : na) longCondition = bar_index % 2 != 0 shortCondition = bar_index % 2 == 0 // Add a "Buy" label when `longCondition` is true. if longCondition label.new(bar_index, 0, text = "Buy", color = color.new(color.green, 0), style = label.style_label_up) // Add a "Sell" label when `shortCondition` is true. if shortCondition label.new(bar_index, 0, text = "Sell", color = color.new(color.red, 0), style = label.style_label_down) plot(longCondition ? 1 : 0) plot(shortCondition ? 1 : 0)

Table limits

Scripts can display a maximum of nine tables on the chart, one for each of the possible locations: position.bottom_center, position.bottom_left, position.bottom_right, position.middle_center, position.middle_left, position.middle_right, position.top_center,

position.top_left, and position.top_right. When attempting to place two tables in the same location, only the newest instance will show on the chart.

request.*() calls

Number of calls

A script can use up to 40 *unique* calls to the functions in the `request.*()` namespace, or up to 64 unique calls if you have a Professional plan. A subsequent call to the same `request.*()` function with the same arguments is not unique. This limitation applies when using any `request.*()` functions, including:

- request.security()
- request.security_lower_tf()
- request.currency_rate()
- request.dividends()
- request.splits()
- request.earnings()
- request.quandl()
- request.financial()
- request.economic()
- request.seed()

When a script executes two or more identical `request.*()` function calls, only the *first* call counts toward this limit. The repeated calls do not count because they *reuse* the data from the first call rather than executing a redundant request. Note that when a script imports library functions containing `request.*()` calls within their scopes, those calls **do** count toward this limit, even if the script already calls the same `request.*()` function with the same arguments in its main scope.

The script below calls request.security() with the same arguments 50 times within a for loop. Although the script contains more than 40 `request.*()` calls, it *does not* raise an error because each call is **identical**. In this case, it reuses the data from the first iteration's

request.security() call for the repeated calls on all subsequent iterations:

//@version=6 indicator("`request.*()` call limit demo") //@variable The sum of values requested from all `request.security()` calls. float reqSum = 0.0 // Call `request.security()` 50 times within a loop. // More than 40 `request.*()` calls occur, but each call is identical. Redundant calls do not count toward the limit. for i = 1 to 50 reqSum += request.security(syminfo.tickerid, "1D", close) plot(reqSum)

Here, we modified the above script to call request.security() with a different `timeframe` argument on each iteration, meaning all 50 calls are now **unique**. This time, the script will reach the `request.*()` call limit while executing the loop and raise a runtime error because it requests a *distinct* dataset on each iteration:

//@version=6 indicator("`request.*()` call limit demo") //@variable The sum of values requested from all `request.security()` calls. float reqSum = 0.0 // Call `request.security()` 50 times within a loop with different `timeframe` arguments. // This loop causes a runtime error when `i == 41` because each iteration executes a unique request. for i = 1 to 50 reqSum += request.security(syminfo.tickerid, str.tostring(i), close) plot(reqSum)

Note that:

- These example scripts can call request.security() within a loop and allow “series string” `timeframe` arguments because Pine v6 scripts enable dynamic requests by default. See this section of the Other timeframes and data page for more information.

Intrabars

Scripts can retrieve up to the most recent 200,000 *intrabars* (lower-timeframe bars) via the request.security() or request.security_lower_tf() functions, depending on the user’s plan:

- All non-professional plans — Basic, Essential, Plus, and Premium — can request up to 100K bars of data.
- Expert plans have access to 125K bars of data.

- Ultimate plans can request 200K lower timeframe bars.

The `request.*()` functions limit requested data via the `calc_bars_count` parameter. If this parameter is not specified, the default is used, which is 100,000 bars. If the plan permits more, this limit can be increased by passing a greater value.

The number of bars on the chart's timeframe covered by 100,000 intrabars varies with the number of intrabars each chart bar contains. For example, requesting data from the 1-minute timeframe while running the script on a 60-minute chart means each chart bar can contain up to 60 intrabars. In this case, the minimum number of chart bars covered by the intrabar request is 1,666, as 100,000 / 60 = 1,666.67. It's important to note, however, that a provider may not report data for *every* minute within an hour. Therefore, such a request may cover more chart bars, depending on the available data.

Tuple element limit

All the `request.*()` function calls in a script taken together cannot return more than 127 tuple elements. When the combined tuple size of all `request.*()` calls will exceed 127 elements, one can instead utilize user-defined types (UDTs) to request a greater number of values.

The example below outlines this limitation and the way to work around it. The first request.security() call represents using a tuple with 128 elements as the `expression` argument. Since the number of elements is greater than 127, it would result in an error.

To avoid the error, we can use those same values as *fields* within an object of a UDT and pass its ID to the `expression` instead:

//@version=6 indicator("Tuple element limit") s1 = close s2 = close * 2 ... s128 = close * 128 // Causes an error. [v1, v2, v3, ..., v128] = request.security(syminfo.tickerid, "1D", [s1, s2, s3, ..., s128]) // Works fine: type myType float v1 float v2 float v3 ... float v128 myObj = request.security(syminfo.tickerid, "1D", myType.new(s1, s2, s3, ..., s128))

Note that:

- This example outlines a scenario where the script tries to evaluate 128 tuple elements in a single request.security() call. The same limitation applies if we were to split the tuple request across *multiple* calls. For example, two request.security() calls that each retrieve a tuple with 64 elements will also cause an error.

Script size and memory

Compiled tokens

Before the execution of a script, the compiler translates it into a tokenized *Intermediate Language* (IL). Using an IL allows Pine Script to accommodate larger scripts by applying various memory and performance optimizations. The compiler determines the size of a script based on the *number of tokens* in its IL form, **not** the number of characters or lines in the code viewable in the Pine Editor.

The compiled form of each indicator, strategy, and library script is limited to 80,000 tokens. When a script imports libraries, the total number of tokens from all imported libraries cannot exceed 1 million. There is no way to inspect a script's compiled form, nor its IL token count. As such, you will only know your script exceeds the size limit when the compiler reaches it.

In most cases, a script's compiled size will likely not reach the limit. However, if a compiled script does reach the token limit, the most effective ways to decrease compiled tokens are to reduce repetitive code, encapsulate redundant calls within functions, and utilize libraries when possible.

It's important to note that the compilation process omits any *unused* variables, functions, types, etc. from the final IL form, where "unused" refers to anything that *does not* affect the script's outputs. This optimization prevents superfluous elements in the code from contributing to the script's IL token count.

For example, the script below declares a user-defined type and a user-defined method and defines a sequence of calls using them:

//@version=6 indicator("My Script") plot(close) type myType float field = 10.0 method m(array<myType> a, myType v) => a.push(v) var arr = array.new<myType>() arr.push(myType.new(25)) arr.m(myType.new())

Despite the inclusion of `array.new\<myType\>()`, `myType.new()`, and `arr.m()` calls in the script, the only thing actually **output** by the script is `plot(close)`. The rest of the code does not affect the output. Therefore, the compiled form of this script will have the *same* number of tokens as:

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

Variables per scope

Scripts can contain up to 1,000 variables in each of its scopes. Pine scripts always contain one global scope, represented by non-indented code, and they may contain zero or more local scopes. Local scopes are sections of indented code representing procedures executed within functions and methods, as well as if, switch, for, for...in, and while structures, which allow for one or more local blocks. Each local block counts as one local scope.

The branches of a conditional expression using the ?: ternary operator do not count as local blocks.

Compilation request size

The size of the compilation request for a script cannot exceed 5MB. The compilation request is all of the information that is sent to the compiler. This information comprises the script itself and any libraries the script imports.

Unlike the limit for compiled tokens, the request size limit includes unused parts of code. This is because the script is not compiled yet, so any unused code has not yet been optimized out.

To reduce the compilation request size, you can:

- Reduce the size of the script by optimizing the code.
- Reduce the number of script inputs (script inputs are counted separately).
- Remove any imported libraries that are not needed.
- Use smaller libraries. The entire library is sent for compilation, regardless of which functions are called.

Collections

Pine Script collections (arrays, matrices, and maps) can have a maximum of 100,000 elements. Each key-value pair in a map contains two elements, meaning maps can contain a maximum of 50,000 key-value pairs.

Other limitations

Maximum bars back

References to past values using the [] history-referencing operator are dependent on the size of the historical buffer maintained by the Pine Script runtime, which is limited to a maximum of 5000 bars for most series. This Help Center page discusses the historical buffer and how to change its size using either the `max_bars_back` parameter or the max_bars_back() function. Drawings using `xloc.bar_index` can be positioned a maximum of 10,000 bars in the past.

Maximum bars forward

When positioning drawings using `xloc.bar_index`, it is possible to use bar index values greater than that of the current bar as *x* coordinates. A maximum of 500 bars in the future can be referenced.

This example shows how we use the [maxval] parameter in our input.int() function call to cap the user-defined number of bars forward we draw a projection line so that it never exceeds the limit:

//@version=6 indicator("Max bars forward example", overlay = true) // This function draws a `line` using bar index x-coordinates. drawLine(bar1, y1, bar2, y2) => // Only execute this code on the last bar. if barstate.islast // Create the line only the first time this function is executed on the last bar. var line lin = line.new(bar1, y1, bar2, y2, xloc.bar_index) // Change the line's properties on all script executions on the last bar. line.set_xy1(lin, bar1, y1) line.set_xy2(lin, bar2, y2) // Input determining how many bars forward we draw the `line`. int forwardBarsInput = input.int(10, "Forward Bars to Display", minval = 1, maxval = 500) // Calculate the line's left and right points. int leftBar = bar_index[2] float leftY = high[2] int rightBar = leftBar + forwardBarsInput float rightY = leftY + (ta.change(high)[1] * forwardBarsInput) // This function call is executed on all bars, but it only draws the `line` on the last bar. drawLine(leftBar, leftY, rightBar, rightY)

Chart bars

The number of bars appearing on charts is dependent on the amount of historical data available for the chart's symbol and timeframe, and on the type of account you hold. When the required historical date is available, the minimum number of chart bars is:

- 40000 historical bars for the Ultimate plan.
- 25000 historical bars for the Expert plan.
- 20000 historical bars for the Premium plan.
- 10000 historical bars for Essential and Plus plans.
- 5000 historical bars for other plans.

Trade orders in backtesting

A script can place a maximum of 9000 orders when backtesting strategies. Once it reaches that limit, the earlier orders are *trimmed* to store the information of new orders. Programmers can use the strategy.closedtrades.first_index variable to reference the index of the earliest untrimmed trade.

When using Deep Backtesting, the order limit is 1,000,000.

Previous