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.

Here are the limits imposed in the Pine Script™ environment.

Time

Script compilation

Scripts must compile before they are executed on charts. Compilation occurs when you save a script from the editor or when you add a script to the chart. A two-minute limit is imposed on compilation time, which will depend on the size and complexity of your script, and whether or not a cached version of a previous compilation is available. When a compile exceeds the two-minute limit, a warning is issued. Heed that warning by shortening your script because after three 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 triggering the execution of a script for a list of the events triggering the execution of a script. The time allotted for the script to execute on all bars of a dataset varies with account types. The limit is 20 seconds for basic accounts, 40 for others.

Loop execution

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

Chart visuals

Plot limits

A maximum of 64 plot counts is allowed per script. The functions generating plot counts are:

- plot()
- plotarrow()
- plotbar()
- plotcandle()
- plotchar()
- plotshape()
- alertcondition()
- bgcolor()

The following functions do not generate plot counts:

- hline()

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

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

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

```
//@version=5
indicator("Plot count example")

bool isUp = close > open
color isUpColor = isUp ? color.green : color.red
bool isDn = not isUp
color isDnColor = isDn ? color.red : color.green

// Uses one plot count.
plot(close, color = color.white)

// 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, borde

// Uses five plot counts for the `open`, `high`, `low`, `close`, and `color` series.
plotcandle(open, high, low, close, color = isUpColor, wickcolor = color.white, borderc

// Uses six plot counts for the `open`, `high`, `low`, `close`, `color`, and `wickcolor
plotcandle(open, high, low, close, color = isUpColor, wickcolor = isUpColor , bordercol

// Uses seven plot counts for the `open`, `high`, `low`, `close`, `color`, `wickcolor`,
plotcandle(open, high, low, close, color = isUpColor, wickcolor = isUpColor , bordercol

// 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
```

```
// 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)
```

The example generates a plot count of 54. If you add three instances of the last call to plotcandle() the script will throw an error stating the script now uses 75 plot counts, because the three additional calls to plotcandle() each generate seven plot counts, and 54 + 21 is 75.

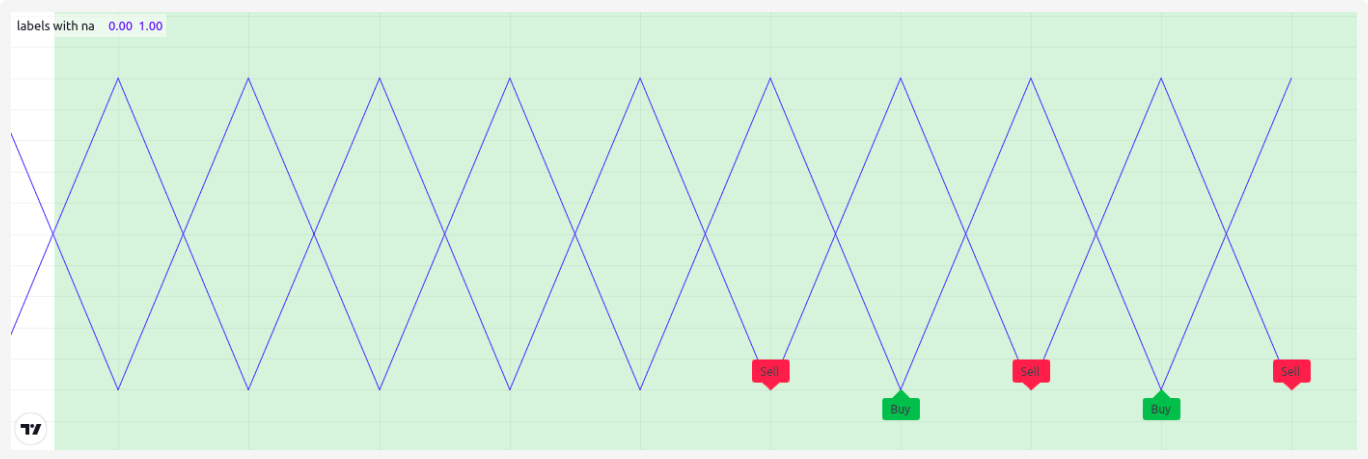
Line, box, and label limits

Contrary to plots which can cover the entire dataset, by default, only the last 50 lines drawn by a script are visible on charts. The same goes for boxes and labels. You can increase the quantity of drawing objects preserved on charts up to a maximum of 500 by using the `max_lines_count`, `max_boxes_count` or `max_labels_count` parameters in the indicator() or strategy() declaration statements.

In this example we set the maximum quantity of last labels shown on the chart to 100:

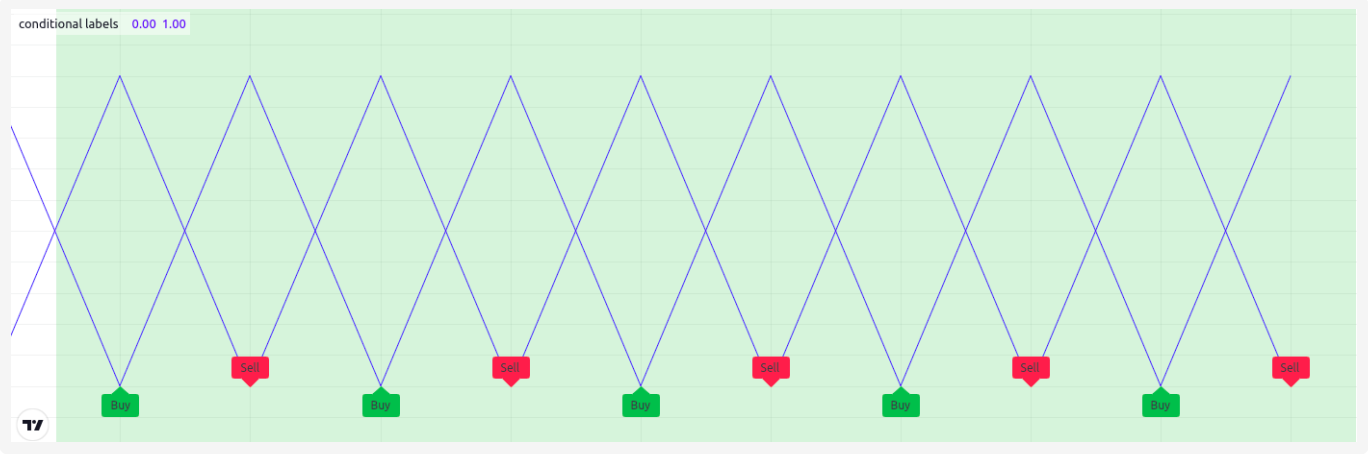
```
//@version=5
indicator("Label limits example", max_labels_count = 100, overlay = true)
label.new(bar_index, high, str.tostring(high, format.mintick))
```

It's important to note that when you set any of the attributes of a drawing object to na, it still counts as a drawing on the chart and thus contributes to a script's drawing totals. To demonstrate this, the following script draws a "Buy" and "Sell" label on each bar with `x` values determined by the `longCondition` and `shortCondition` variables. The "Buy" label's `x` value is na when the bar index is even, and the "Sell" label's `x` value is na when the bar index is odd. Although the `max_labels_count` is 10 in this example, we can see that the script displays fewer than ten labels on the chart since the ones with na values also count toward the total:



```
//@version=5

// Approximate maximum number of label drawings
MAX_LABELS = 10

indicator("labels with na", overlay = false, max_labels_count = MAX_LABELS)

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

longCondition =  bar_index % 2 != 0
shortCondition = bar_index % 2 == 0

// Add "Buy" and "Sell" labels on each new bar.
label.new(longCondition ? bar_index : na,  0, text = "Buy", color = color.new(color.gre
label.new(shortCondition ? bar_index : na, 0, text = "Sell", color = color.new(color.re

plot(longCondition  ? 1 : 0)
plot(shortCondition ? 1 : 0)
```

If we want the script to display the desired number of labels, we need to eliminate the ones with na `x` values so that they don't add to the script's label count. This example conditionally draws the "Buy" and "Sell" labels rather than always drawing them and setting their attributes to na on alternating bars:



```
//@version=5

// Approximate maximum number of label drawings
MAX_LABELS = 10

indicator("conditional labels", overlay = false, max_labels_count = MAX_LABELS)

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

longCondition =  bar_index % 2 != 0
shortCondition = bar_index % 2 == 0

// Add a "Buy" label when `longCondition` is true.
if longCondition
    label.new(bar_index,  0, text = "Buy", color = color.new(color.green, 0), style = l
// Add a "Sell" label when `shortCondition` is true.
if shortCondition
    label.new(bar_index, 0, text = "Sell", color = color.new(color.red, 0), style = lab

plot(longCondition  ? 1 : 0)
plot(shortCondition ? 1 : 0)
```

Table limits

A maximum of nine tables can be displayed by a script, one for each of the possible locations: position.bottom_center, position.bottom_left, position.bottom_right, position.middle_center, position.middle_left, position.middle_right, position.top_center, position.top_left, or position.top_right. If you place two tables in the same position, only the most recently added table will be visible.

`request.*()` calls

Number of calls

A script cannot make more than 40 calls to `request.*()` functions. All instances of calls to these functions are counted, even if they are included in code blocks or functions that are never actually used in the script's logic. The functions counting towards this limit are: request.security(), request.security_lower_tf(), request.quandl(), request.financial(), request.dividends(), request.earnings() and request.splits().

Intrabars

When accessing lower timeframes, with request.security() or request.security_lower_tf(), a maximum of 100,000 intrabars can be used in calculations.

The quantity of chart bars covered with 100,000 intrabars will vary with the ratio of the chart's timeframe to the lower timeframe used, and with the average number of intrabars contained in each chart bar. For example, when using a 1min lower timeframe, chart bars at the 60min timeframe of an active 24x7 market will usually contain 60 intrabars each. Because 100,000 / 60 = 1666.67, the quantity of chart bars covered by the 100,000 intrabars will typically be 1666. On markets where 60min chart bars do not always contain 60 1min intrabars, more chart bars will be covered.

Tuple element limit

All the `request.*()` functions in one script taken together cannot return more than 127 tuple values. Below we have an example showing what can cause this error and how to work around it:

```
//@version=5
indicator("Tuple values error")

// CAUSES ERROR:
[v1, v2, v3,...] = request.security(syminfo.tickerid, "1D", [s1, s2, s3,...])

// Works fine:
type myType
    int v1
    int v2
    int v3
    ...

myObj = request.security(syminfo.tickerid, "1D", myType.new())
```

Note that:

- In this example, we have a request.security() function with at least three values in our tuple, and we could either have more than 127 values in our tuple above or more than 127 values between multiple request.security() functions to throw this error.
- We get around the error by simply creating a User-defined object that can hold the same values without throwing an error.
- Using the `myType.new()` function is functionally the same as listing the same values in our `[s1, s2, s3,...]` tuple.

Script size and memory

Compiled tokens

Before a script is executed, it is compiled into a tokenized Intermediate Language (IL). Using an IL allows Pine Script™ to accommodate longer scripts by applying various optimizations before it is executed. The compiled form of indicators and strategies is limited to 60,000 tokens; libraries have a limit of 1 million tokens. There is no way to inspect the number of tokens created during compilation; you will only know your script exceeds the limit when the compiler reaches it.

Replacing code repetitions with function calls and using libraries to offload some of the workload are the most efficient ways to decrease the number of tokens your compiled script will generate.

The size of variable names and comments do not affect the number of compiled tokens.

Local blocks

Local blocks are segments of indented code used in function definitions or in if, switch, for or while structures, which allow for one or more local blocks.

Scripts are limited to 500 local blocks.

Variables

A maximum of 1000 variables are allowed per scope. Pine scripts always contain one global scope, and can contain

zero or more local scopes. Local scopes are created by indented code such as can be found in functions or if, switch, for or while structures, which allow for one or more local blocks. Each local block counts as one local scope.

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

Arrays and matrices

Arrays and matrices are limited to 100,000 elements.

Other limitations

Maximum bars back

References to past values using the [] history-referencing operator are dependent on the size of the historical buffer maintained by the Pine Script™ runtime, which is limited to a maximum of 5000 bars. This Help Center page discusses the historical buffer and how to change its size using either the `max_bars_back` parameter or the max_bars_back() function.

Maximum bars forward

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

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

```
//@version=5
indicator("Max bars forward example", overlay = true)

// This function draws a `line` using bar index x-coordinates.
drawLine(bar1, y1, bar2, y2) =>
    // Only execute this code on the last bar.
    if barstate.islast
        // Create the line only the first time this function is executed on the last ba
        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 = 5

// 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
drawLine(leftBar, leftY, rightBar, rightY)
```

Chart bars

The number of bars appearing on charts is dependent on the amount of historical data available for the chart's symbol

and timeframe, and on the type of account you hold. When the required historical date is available, the minimum number of chart bars is:

- 20,000 bars for the Premium plan.
- 10,000 bars for Pro and Pro+ plans.
- 5000 bars for other plans.

Trade orders in backtesting

A maximum of 9000 orders can be placed when backtesting strategies. When using Deep Backtesting, the limit is 200,000.

Publishing scripts

FAQ