Execution model

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

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

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

All symbol/timeframe pairs have a dataset comprising a limited number of bars. When you scroll a chart to the left to see the dataset's earlier bars, the corresponding bars are loaded on the chart. The loading process stops when there are no more bars for that particular symbol/timeframe pair or the maximum number of bars your account type permits has been loaded [1]. You can scroll the chart to the left until the very first bar of the dataset, which has an index value of 0 (see bar_index).

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

Calculation based on historical bars

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

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

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

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

```
src = close
```

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

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

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

Calculation based on realtime bars

The behavior of a Pine Script™ on the realtime bar is very different than on historical bars. Recall that the realtime

bar is the rightmost bar on the chart when trading is active on the chart's symbol. Also, recall that strategies can behave in two different ways in the realtime bar. By default, they only execute when the realtime bar closes, but the `calc_on_every_tick` parameter of the `strategy` declaration statement can be set to true to modify the strategy's behavior so that it executes each time the realtime bar updates, as indicators do. The behavior described here for indicators will thus only apply to strategies using `calc_on_every_tick=true`.

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

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

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

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

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

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

To summarize the realtime bar process:

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

Events triggering the execution of a script

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

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

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

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

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

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

More information

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

Execution of Pine Script™ functions and historical context inside function blocks

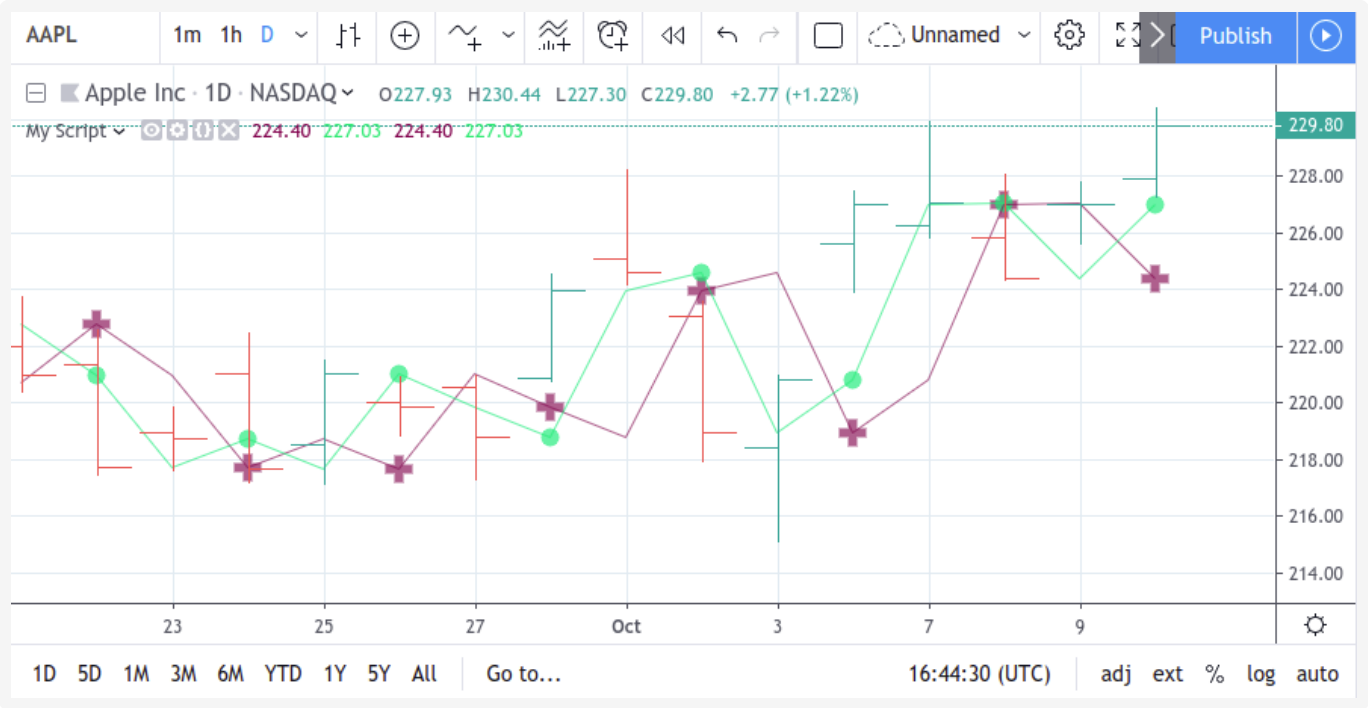
The history of series variables used inside Pine Script™ functions is created through each successive call to the function. If the function is not called on each bar the script runs on, this will result in disparities between the historic values of series inside vs outside the function's local block. Hence, series referenced inside and outside the function using the same index value will not refer to the same point in history if the function is not called on each bar.

Let's look at this example script where the `f()` and `f2()` functions are called every second bar:

```
//@version=5
indicator("My Script", overlay = true)

// Returns the value of "a" the last time the function was called 2 bars ago.
f(a) => a[1]
// Returns the value of last bar's "close", as expected.
f2() => close[1]

oneBarInTwo = bar_index % 2 == 0
plot(oneBarInTwo ? f(close) : na, color = color.maroon, linewidth = 6, style = plot.st
plot(oneBarInTwo ? f2() : na, color = color.lime, linewidth = 6, style = plot.style_ci
plot(close[2], color = color.maroon)
plot(close[1], color = color.lime)
```



As can be seen with the resulting plots, `a[1]` returns the previous value of a in the function's context, so the last time `f()` was called two bars ago — not the close of the previous bar, as `close[1]` does in `f2()`. This results in `a[1]` in the function block referring to a different past value than `close[1]` even though they use the same index of 1.

Why this behavior?

This behavior is required because forcing execution of functions on each bar would lead to unexpected results, as would be the case for a label.new() function call inside an if branch, which must not execute unless the if condition requires it.

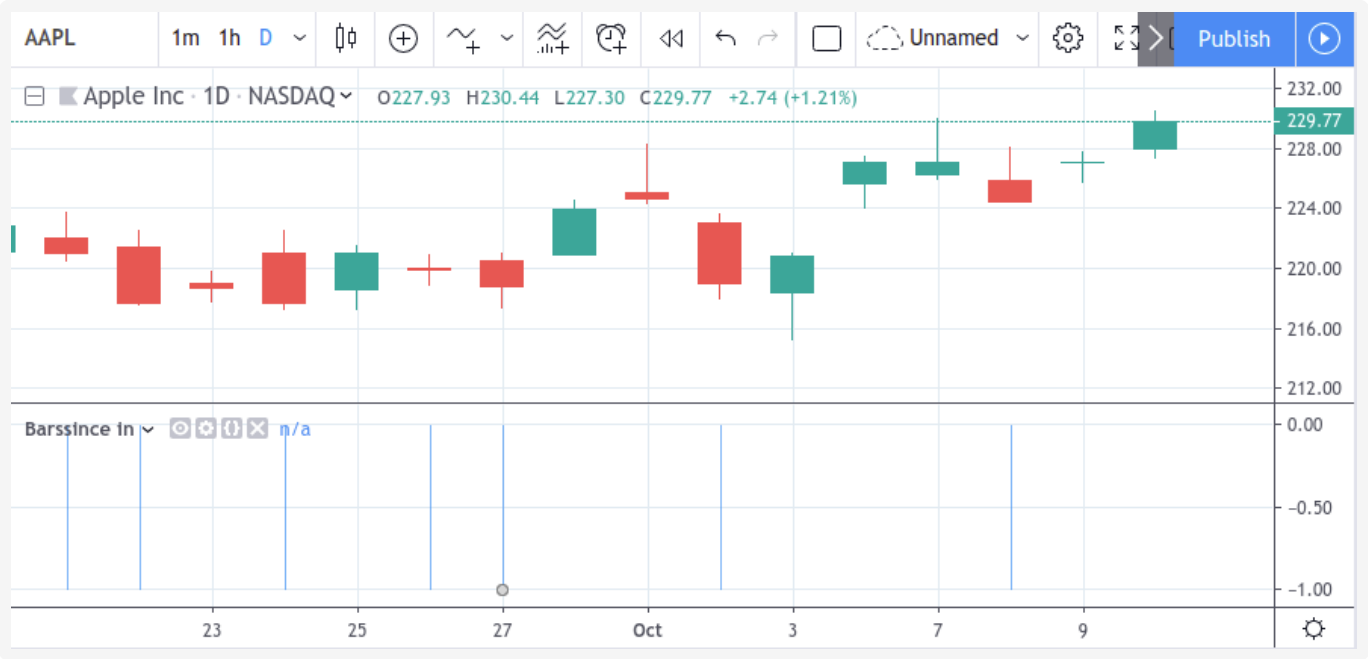
On the other hand, this behavior leads to unexpected results with certain built-in functions which require being executed each bar to correctly calculate their results. Such functions will not return expected results if they are placed in contexts where they are not executed every bar, such as if branches.

The solution in these cases is to take those function calls outside their context so they can be executed on every bar.

In this script, ta.barssince() is not called on every bar because it is inside a ternary operator's conditional branch:

```
//@version=5
indicator("Barssince", overlay = false)
res = close > close[1] ? ta.barssince(close < close[1]) : -1
plot(res, style = plot.style_histogram, color=res >= 0 ? color.red : color.blue)
```

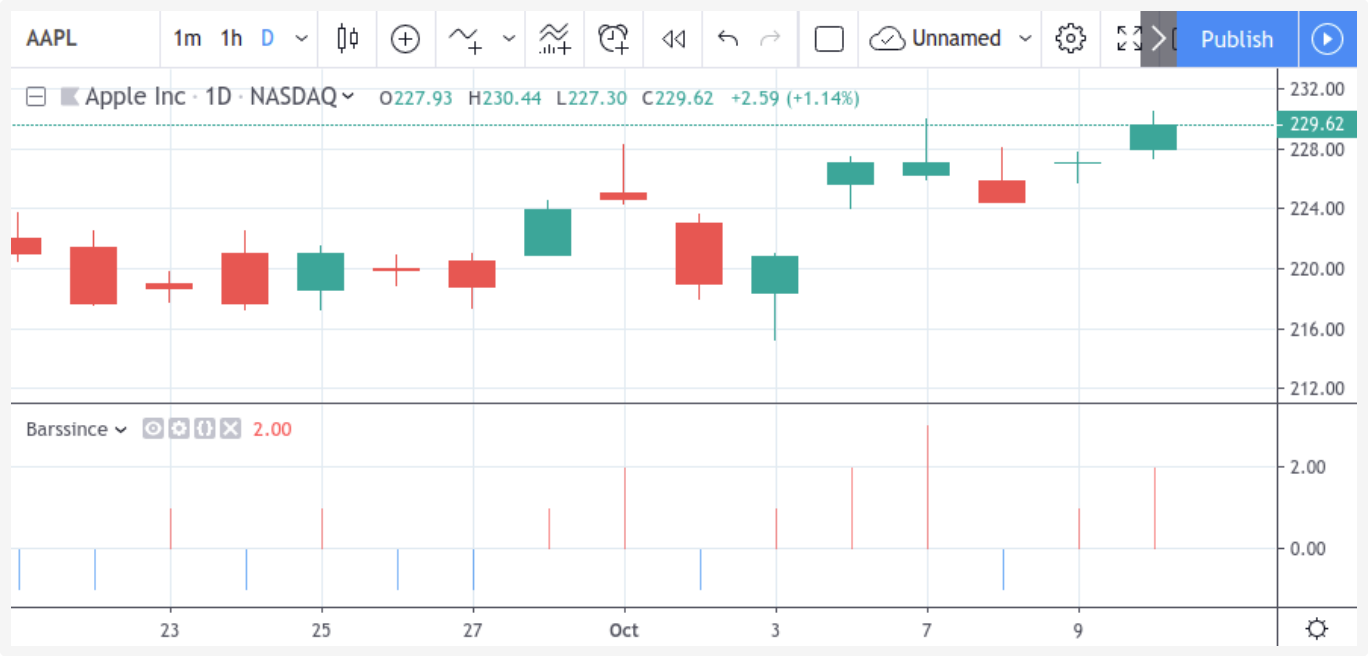
This leads to incorrect results because ta.barssince() is not executed on every bar:



The solution is to take the ta.barssince() call outside the conditional branch to force its execution on every bar:

```
//@version=5
indicator("Barssince", overlay = false)
b = ta.barssince(close < close[1])
res = close > close[1] ? b : -1
plot(res, style = plot.style_histogram, color = res >= 0 ? color.red : color.blue)
```

Using this technique we get the expected output:



Exceptions

Not all built-in functions need to be executed every bar. These are the functions which do not require it, and so do not need special treatment:

```
dayofmonth, dayofweek, hour, linebreak, math.abs, math.acos, math.asin, math.atan, math
math.cos, math.exp, math.floor, math.log, math.log10, math.max, math.min, math.pow, ma
math.sign, math.sin, math.sqrt, math.tan, minute, month, na, nz, second, str.tostring,
ticker.heikinashi, ticker.kagi, ticker.new, ticker.renko, time, timestamp, weekofyear,
```

Note

Functions called from within a for loop use the same context in each of the loop's iterations. In the example below, each ta.lowest() call on the same bar uses the value that was passed to it, i.e., bar_index, so function calls used in loops do not require special treatment.

```
//@version=5
indicator("My Script")
va = 0.0
for i = 1 to 2 by 1
    if (i + bar_index) % 2 == 0
        va := ta.lowest(bar_index, 10)  // same context on each call
plot(va)
```

Footnotes

[1] The upper limit for the total number of historical bars is about 10000 for *Pro*/*Pro+* users and about 20000 for *Premium* users. *Free* users are able to see about 5000 bars.

Language

Time series