Pine Script® v6 Documentation

3 Concepts Part2

TradingView Pine Script Programming Language

Table of Contents

Lines and boxes

Lines and boxes

Introduction

Pine Script® facilitates drawing lines, boxes, and other geometric formations from code using the line, box, and polyline types. These types provide utility for programmatically drawing support and resistance levels, trend lines, price ranges, and other custom formations on a chart.

Unlike plots, the flexibility of these types makes them particularly well-suited for visualizing current calculated data at virtually any available point on the chart, irrespective of the chart bar the script executes on.

Lines, boxes, and polylines are *objects*, like labels, tables, and other *special types*. Scripts reference objects of these types using IDs, which act like *pointers*. As with other objects, line, box, and polyline IDs are qualified as "series" values, and all functions that manage these objects accept "series" arguments.

Lines drawn by a script may be vertical, horizontal, or angled. Boxes are always rectangular. Polylines sequentially connect multiple vertical, horizontal, angled, or curved line segments. Although all of these drawing types have different characteristics, they do have some things in common:

- Lines, boxes, and polylines can have coordinates at any available location on the chart, including ones at future times beyond the last chart bar.
- Objects of these types can use chart.point instances to set their coordinates.
- The x-coordinates of each object can be bar index or time values, depending on their specified `xloc` property.

- Each object can have one of multiple predefined line styles.
- Scripts can call the functions that manage these objects from within the scopes of loops and conditional structures, allowing iterative and conditional control of their drawings.
- There are limits on the number of these objects that a script can reference and display on the chart. A single script instance can display up to 500 lines, 500 boxes, and 100 polylines. Users can specify the maximum number allowed for each type via the `max_lines_count`, `max_boxes_count`, and `max_polylines_count` parameters of the script's indicator() or strategy() declaration statement. If unspecified, the default is ~50. As with label and table types, lines, boxes, and polylines utilize a *garbage collection* mechanism that deletes the oldest objects on the chart when the total number of drawings exceeds the script's limit.

Lines

The built-ins in the `line.*` namespace control the creation and management of line objects:

- The line.new() function creates a new line.
- The `line.set_*()` functions modify line properties.
- The `line.get_*()` functions retrieve values from a line instance.
- The line.copy() function clones a line instance.
- The line.delete() function deletes an existing line instance.
- The line.all variable references a read-only array containing the IDs of all lines displayed by the script. The array's size depends on the `max_lines_count` of the indicator() or strategy() declaration statement and the number of lines the script has drawn.

Scripts can call `line.set_*()`, `line.get_*()`, line.copy(), and line.delete() built-ins as functions or methods.

Creating lines

The line.new() function creates a new line instance to display on the chart. It has the following signatures:

```
line.new(first_point, second_point, xloc, extend, color, style, width,
```

```
line.new(x1, y1, x2, y2, xloc, extend, color, style, width, force_over
```

The first overload of this function contains the `first_point` and `second_point` parameters. The `first_point` is a chart.point representing the start of the line, and the `second_point` is a chart.point representing the line's end. The function copies the information from these chart points to determine the line's coordinates. Whether it uses the `index` or `time` fields from the `first_point` and `second_point` as x-coordinates depends on the function's `xloc` value.

The second overload specifies x1, y1, x2, and y2 values independently, where x1 and x2 are int values representing the starting and ending x-coordinates of the line, and y1 and y2 are float values representing the y-coordinates. Whether the line considers the x values as bar indices or timestamps depends on the `xloc` value in the function call.

Both overloads share the same additional parameters:

`xloc`

Controls whether the x-coordinates of the new line use bar index or time values. Its default value is xloc.bar_index.

When calling the first overload, using an `xloc` value of xloc.bar_index tells the function to use the `index` fields of the `first_point` and `second_point`, and a value of xloc.bar_time tells the function to use the `time` fields of the points.

When calling the second overload, an `xloc` value of xloc.bar_index prompts the function to treat the x1 and x2 arguments as bar index values. When using xloc.bar_time, the function will treat x1 and x2 as time values.

When the specified x-coordinates represent *bar index* values, it's important to note that the minimum x-coordinate allowed is `bar_index - 10000`. For larger offsets, one can use xloc.bar_time.

`extend`

Determines whether the drawn line will infinitely extend beyond its defined start and end coordinates. It accepts one of the following values: extend.left, extend.right, extend.both, or extend.none (default).

`color`

Specifies the color of the line drawing. The default is color.blue.

`style`

Specifies the line's style, which can be any of the options listed in this page's Line styles section. The default value is line.style_solid.

`width`

Controls the width of the line, in pixels. The default value is 1.

`force_overlay`

If `true`, the drawing will display on the main chart pane, even when the script occupies a separate pane. Optional. The default is `false`.

The example below demonstrates how one can draw lines in their simplest form. This script draws a new vertical line connecting the open and close prices at the horizontal center of each chart bar:

image

//@version=6 indicator("Creating lines demo", overlay = true) //@variable The `chart.point` for the start of the line. Contains `index` and `time` information. firstPoint = chart.point.now(open) //@variable The `chart.point` for the end of the line. Contains `index` and `time` information. secondPoint = chart.point.now(close) // Draw a basic line with a `width` of 5 connecting the `firstPoint` to the `secondPoint`. // This line uses the `index` field from each point for its x-coordinates. line.new(firstPoint, secondPoint, width = 5) // Color the background on the unconfirmed bar. bgcolor(barstate.isconfirmed ? na : color.new(color.orange, 70), title = "Unconfirmed bar highlight")

Note that:

- If the `firstPoint` and `secondPoint` reference identical coordinates, the script will *not* display a line since there is no distance between them to draw. However, the line ID will still exist.
- The script will only display approximately the last 50 lines on the chart, as it does not have a specified `max_lines_count` in the indicator() function call. Line drawings persist on the chart until deleted using line.delete() or removed by the garbage collector.
- The script *redraws* the line on the open chart bar (i.e., the bar with an orange background highlight) until it closes. After the bar closes, it will no longer update the drawing.

Let's look at a more involved example. This script uses the previous bar's hl2 price and the current bar's high and low prices to draw a fan with a user-specified number of lines projecting a range of hypothetical price values for the following chart bar. It calls line.new() within a for loop to create `linesPerBar` lines on each bar:

image

//@version=6 indicator("Creating lines demo", "Simple projection fan", true, max_lines_count = 500) //@variable The number of fan lines drawn on each chart bar. int linesPerBar = input.int(20, "Line drawings per bar", 2, 100) // @variable The distance between each y point on the current bar. float step = (high - low) / (linesPerBar - 1) //@variable The `chart.point` for the start of each line. Does not contain `time` information. firstPoint = chart.point.from_index(bar_index - 1, hl2[1]) //@variable The `chart.point` for the end of each line. Does not contain `time` information. secondPoint = chart.point.from_index(bar_index + 1, float(na)) //@variable The stepped y value on the current bar for `secondPoint.price` calculation, starting from the `low`. float barValue = low // Loop to draw the fan. for i = 1 to linesPerBar // Update the `price` of the `secondPoint` using the difference between the `barValue` and `firstPoint.price`. secondPoint.price := 2.0 * barValue - firstPoint.price //@variable Is `color.aqua` when the line's slope is positive, `color.fuchsia` otherwise. color lineColor = secondPoint.price > firstPoint.price ? color.aqua : color.fuchsia // Draw a new `lineColor` line connecting the `firstPoint` and `secondPoint` coordinates. // This line uses the `index` field from each point for its x-coordinates. line.new(firstPoint,

secondPoint, color = lineColor) // Add the `step` to the `barValue`. barValue += step // Color the background on the unconfirmed bar. bgcolor(barstate.isconfirmed ? na : color.new(color.orange, 70), title = "Unconfirmed bar highlight")

Note that:

- We've included `max_lines_count = 500` in the indicator() function call, meaning the script preserves up to 500 lines on the chart.
- Each line.new() call *copies* the information from the chart.point referenced by the `firstPoint` and `secondPoint` variables. As such, the script can change the `price` field of the `secondPoint` on each loop iteration without affecting the y-coordinates in other lines.

Modifying lines

The `line.*` namespace contains multiple *setter* functions that modify the properties of line instances:

- line.set_first_point() and line.set_second_point() respectively update the start and end points of the `id` line using information from the specified `point`.
- line.set_x1() and line.set_x2() set one of the x-coordinates of the `id` line to a new `x` value, which can represent a bar index or time value depending on the line's `xloc` property.
- line.set_y1() and line.set_y2() set one of the y-coordinates of the `id` line to a new `y` value.
- line.set_xy1() and line.set_xy2() update one of the `id` line's points with new `x` and `y` values.
- line.set_xloc() sets the `xloc` of the `id` line and updates both of its x-coordinates with new `x1` and `x2` values.
- line.set_extend() sets the `extend` property of the `id` line.
- line.set_color() updates the `id` line's `color` value.
- line.set_style() changes the `style` of the `id` line.
- line.set_width() sets the `width` of the `id` line.

All setter functions directly modify the `id` line passed into the call and do not return any value. Each setter function accepts "series" arguments, as a script can change a line's properties throughout its execution.

The following example draws lines connecting the opening price of a `timeframe` to its closing price. The script uses the var keyword to declare `periodLine` and the variables that reference chart.point objects (`openPoint` and `closePoint`) only on the *first* chart bar, and it assigns new references to these variables over its execution. After detecting a new bar on the specified timeframe with timeframe.change, the script uses line.set_color() to set the `color` property of the current line referenced by `periodLine`, creates new chart points for `openPoint` and `closePoint` using chart.point.now(), calls line.new() to create another line anchored to those points, then assigns the new line's reference to `periodLine`.

On other bars where the `periodLine` reference is not na, the script assigns a new chart.point reference to the `closePoint` variable, then uses line.set_second_point() and line.set_color() as methods to update the end coordinate and color of the latest line:

image

//@version=6 indicator("Modifying lines demo", overlay = true) //@variable The size of each period. string timeframe = input.timeframe("D", "Timeframe") //@variable A line connecting the period's opening and closing prices. var line periodLine = na //@variable The first point of the line. Contains `time` and `index` information. var chart.point openPoint = chart.point.now(open) //@variable The closing point of the line. Contains `time` and `index` information. var chart.point closePoint = chart.point.now(close) if timeframe.change(timeframe) //@variable The final color of the `periodLine`. color finalColor = switch closePoint.price > openPoint.price => color.green closePoint.price < openPoint.price => color.red => color.gray // Update the color of the current `periodLine` to the `finalColor`. line.set_color(periodLine, finalColor) // Assign new points to the `openPoint` and `closePoint`. openPoint := chart.point.now(open) closePoint := chart.point.now(close) // Assign a new line to the `periodLine`. Uses `time` fields from the `openPoint` and `closePoint` as x-coordinates. periodLine := line.new(openPoint, closePoint, xloc.bar_time, style = line.style_arrow_right, width = 3) else if not na(periodLine) // Assign a new point to the `closePoint`. closePoint := chart.point.now(close) //@variable The color of the developing `periodLine`. color developingColor = switch closePoint.price > openPoint.price => color.aqua closePoint.price < openPoint.price => color.fuchsia => color.gray // Update the coordinates of

the line's second point using the new `closePoint`. // It uses the `time` field from the point for its new x-coordinate. periodLine.set_second_point(closePoint) // Update the color of the line using the `developingColor`. periodLine.set_color(developingColor)

Note that:

- Each line drawing in this example uses the line.style_arrow_right style. See the Line styles section below for an overview of all available style settings.

Line styles

Users can control the style of their scripts' line drawings by passing one of the following variables as the `style` argument in their line.new() or line.set_style() function calls:

Argument	Line
`line.style_solid`	line style solid
`line.style_dotted`	line style dotted
`line.style_dashed`	line style dashed
`line.style_arrow_left`	line_style_arrow_left
`line.style_arrow_right`	line_style_arrow_right
`line.style_arrow_both`	line style arrow both

Note that:

- *Polylines* can also use any of these variables as their `line_style` value. See the Creating polylines section of this page.

Reading line values

The `line.*` namespace includes *getter* functions, which allow a script to retrieve values from a line object for further use:

- line.get_x1() and line.get_x2() respectively get the first and second x-coordinate from the `id` line. Whether the value returned represents a bar index or time value depends on the line's `xloc` property.

- line.get_y1() and line.get_y2() respectively get the `id` line's first and second y-coordinate.
- line.get_price() retrieves the price (y-coordinate) from a line `id` at a specified `x` value, including at bar indices outside the line's start and end points. This function is only compatible with lines that use xloc.bar_index as the `xloc` value.

The script below draws a new line upon the onset of a rising or falling price pattern forming over `length` bars. It uses the var keyword to declare the `directionLine` variable on the first chart bar. The line reference assigned to `directionLine` persists over subsequent bars until the `newDirection` condition occurs, in which case the script assigns a creates a new line with line.new and assigns that line's reference to the variable.

On every bar, the script calls the line.get_y2(), line.get_y1(), line.get_x2(), and line.get_x1() getters as methods to retrieve values from the current line referenced by `directionLine` and calculate its slope, then uses the result to determine the color of each drawing and plot. The script retrieves an extended value of the current line from *beyond* its second point using line.get_price() and plots the returned value on the chart:

image

//@version=6 indicator("Reading line values demo", overlay = true) // @variable The number of bars for rising and falling calculations. int length = input.int(2, "Length", 2) //@variable A line that's drawn whenever `hlc3` starts rising or falling over `length` bars. var line directionLine = na // @variable Is `true` when `hlc3` is rising over `length` bars, `false` otherwise. bool rising = ta.rising(hlc3, length) //@variable Is `true` when `hlc3` is falling over `length` bars, `false` otherwise. bool falling = ta.falling(hlc3, length) //@variable Is `true` when a rising or falling pattern begins, `false` otherwise. bool newDirection = (rising and not rising[1]) or (falling and not falling[1]) // Update the `directionLine` when `newDirection` is `true`. The line uses the default `xloc.bar_index`. if newDirection directionLine := line.new(bar_index - length, hlc3[length], bar_index, hlc3, width = 3) //@variable The slope of the `directionLine`. float slope = (directionLine.get_y2() - directionLine.get_y1()) / (directionLine.get_x2() - directionLine.get_x1()) //@variable The value extrapolated from the `directionLine` at the `bar_index`. float lineValue =

line.get_price(directionLine, bar_index) //@variable Is `color.green` when the `slope` is positive, `color.red` otherwise. color slopeColor = slope > 0 ? color.green : color.red // Update the color of the `directionLine`. directionLine.set_color(slopeColor) // Plot the `lineValue`. plot(lineValue, "Extrapolated value", slopeColor, 3, plot.style_circles)

Note that:

- This example calls the second overload of the line.new() function, which uses x1, y1, x2, and y2 parameters to define the start and end points of the line. The x1 value is `length` bars behind the current bar_index, and the y1 value is the hlc3 value at that index. The x2 and y2 in the function call use the current bar's bar_index and hlc3 values.
- The line.get_price() function call treats the `directionLine` as though it extends infinitely, regardless of its `extend` property.
- The script only displays approximately the last 50 lines on the chart, but the plot of extrapolated values spans throughout the chart's history.

Cloning lines

Scripts can clone a line `id` and all its properties with the line.copy() function. Any changes to the copied line instance do not affect the original.

For example, this script creates a horizontal line at the the bar's open price once every `length` bars, which it assigns to a `mainLine` variable. On all other bars, it creates a `copiedLine` using line.copy() and calls `line.set_*()` functions to modify its properties. As we see below, altering the `copiedLine` does not affect the `mainLine` in any way:

image

//@version=6 indicator("Cloning lines demo", overlay = true, max_lines_count = 500) //@variable The number of bars between each new mainLine assignment. int length = input.int(20, "Length", 2, 500) //@variable The first `chart.point` used by the `mainLine`. Contains `index` and `time` information. firstPoint = chart.point.now(open) //@variable The second `chart.point` used by the `mainLine`. Does not contain `time` information. secondPoint = chart.point.from_index(bar_index + length, open) //@variable A horizontal line drawn at the `open` price once every `length` bars. var line

mainLine = na if bar_index % length == 0 // Assign a new line to the `mainLine` that connects the `firstPoint` to the `secondPoint`. // This line uses the `index` fields from both points as x-coordinates. mainLine := line.new(firstPoint, secondPoint, color = color.purple, width = 2) //@variable A copy of the `mainLine`. Changes to this line do not affect the original. line copiedLine = line.copy(mainLine) // Update the color, style, and second point of the `copiedLine`. line.set_color(copiedLine, color.orange) line.set_style(copiedLine, line.style_dotted) line.set_second_point(copiedLine, chart.point.now(close))

Note that:

- The `index` field of the `secondPoint` is `length` bars beyond the current bar_index. Since the maximum x-coordinate allowed with xloc.bar_index is `bar_index + 500`, we've set the `maxval` of the `length` input to 500.

Deleting lines

To delete a line drawn by a script, use the line.delete() function. This function removes the line instance from the script and its drawing on the chart.

Deleting line instances is often handy when one wants to only keep a specific number of lines on the chart at any given time or conditionally remove drawings as a chart progresses.

For example, this script creates a horizontal line with the `extend` property set to extend.right whenever an RSI crosses its EMA.

The script stores all line IDs in a `lines` array that it uses as a queue to display only a specified number of lines on the chart. When the size of the array exceeds the specified `numberOfLines` value, the script removes the array's oldest line ID using array.shift() and deletes it with line.delete():

image

//@version=6 //@variable The maximum number of lines allowed on the chart. const int MAX_LINES_COUNT = 500 indicator("Deleting lines demo", "RSI cross levels", max_lines_count = MAX_LINES_COUNT) //@variable The length of the RSI. int rsiLength = input.int(14, "RSI length", 2) //@variable The length of the RSI's EMA. int emaLength = input.int(28, "RSI average

length", 2) //@variable The maximum number of lines to keep on the chart. int numberOfLines = input.int(20, "Lines on the chart", 0, MAX_LINES_COUNT) //@variable An array containing the IDs of lines on the chart. var array<line> lines = array.new<line>() //@variable An `rsiLength` RSI of `close`. float rsi = ta.rsi(close, rsiLength) //@variable A `maLength` EMA of the `rsi`. float rsiMA = ta.ema(rsi, emaLength) if ta.cross(rsi, rsiMA) // @variable The color of the horizontal line. color lineColor = rsi > rsiMA ? color.green : color.red // Draw a new horizontal line. Uses the default `xloc.bar_index`. newLine = line.new(bar_index, rsiMA, bar_index + 1, rsiMA, extend = extend.right, color = lineColor, width = 2) // Push the `newLine` into the `lines` array. lines.push(newLine) // Delete the oldest line when the size of the array exceeds the specified `numberOfLines`. if array.size(lines) > numberOfLines line.delete(lines.shift()) // Plot the `rsi` and `rsiMA`. plot(rsi, "RSI", color.new(color.blue, 40)) plot(rsiMA, "EMA of RSI", color.new(color.gray, 30))

Note that:

- We declared a `MAX_LINES_COUNT` variable with the “const int” *qualified type*, which the script uses as the `max_lines_count` in the indicator() function and the `maxval` of the input.int() assigned to the `numberOfLines` variable.
- This example uses the second overload of the line.new() function, which specifies x1, y1, x2, and y2 coordinates independently.

Filling the space between lines

Scripts can *fill* the space between two line drawings by creating a linefill object that references them with the linefill.new() function. Linefills automatically determine their fill boundaries using the properties from the `line1` and `line2` IDs that they reference.

For example, this script calculates a simple linear regression channel. On the first chart bar, the script declares the `basisLine`, `upperLine`, and `lowerLine` variables to reference the channel’s line IDs, then it makes two linefill.new() calls to create linefill objects that fill the upper and lower portions of the channel. The first linefill fills the space between the `basisLine` and the `upperLine`, and the second fills the space between the `basisLine` and `lowerLine`.

The script updates the coordinates of the lines across subsequent bars. However, notice that the script never needs to update the linefills declared on the first bar. They automatically update their fill regions based on the coordinates of their assigned lines:

image

//@version=6 indicator("Filling the space between lines demo", "Simple linreg channel", true) //@variable The number of bars in the linear regression calculation. int lengthInput = input.int(100) //@variable The basis line of the regression channel. var line basisLine = line.new(na, na, na, na, extend = extend.right, color = chart.fg_color, width = 2) //@variable The channel's upper line. var line upperLine = line.new(na, na, na, na, extend = extend.right, color = color.teal, width = 2) //@variable The channel's lower line. var line lowerLine = line.new(na, na, na, na, extend = extend.right, color = color.maroon, width = 2) //@variable A linefill instance that fills the space between the `basisLine` and `upperLine`. var linefill upperFill = linefill.new(basisLine, upperLine, color.new(color.teal, 80)) //@variable A linefill instance that fills the space between the `basisLine` and `lowerLine`. var linefill lowerFill = linefill.new(basisLine, lowerLine, color.new(color.maroon, 80)) // Update the `basisLine` coordinates with current linear regression values. basisLine.set_xy1(bar_index + 1 - lengthInput, ta.linreg(close, lengthInput, lengthInput - 1)) basisLine.set_xy2(bar_index, ta.linreg(close, lengthInput, 0)) //@variable The channel's standard deviation. float stDev = 0.0 for i = 0 to lengthInput - 1 stDev += math.pow(close[i] - line.get_price(basisLine, bar_index - i), 2) stDev := math.sqrt(stDev / lengthInput) * 2.0 // Update the `upperLine` and `lowerLine` using the values from the `basisLine` and the `stDev`. upperLine.set_xy1(basisLine.get_x1(), basisLine.get_y1() + stDev) upperLine.set_xy2(basisLine.get_x2(), basisLine.get_y2() + stDev) lowerLine.set_xy1(basisLine.get_x1(), basisLine.get_y1() - stDev) lowerLine.set_xy2(basisLine.get_x2(), basisLine.get_y2() - stDev)

To learn more about the linefill type, see this section of the Fills page.

Boxes

The built-ins in the `box.*` namespace create and manage box objects:

- The box.new() function creates a new box.
- The `box.set_*()` functions modify box properties.
- The `box.get_*()` functions retrieve values from a box instance.
- The box.copy() function clones a box instance.
- The box.delete() function deletes a box instance.
- The box.all variable references a read-only array containing the IDs of all boxes displayed by the script. The array's size depends on the `max_boxes_count` of the indicator() or strategy() declaration statement and the number of boxes the script has drawn.

As with lines, users can call `box.set_*()`, `box.get_*()`, box.copy(), and box.delete() built-ins as functions or methods.

Creating boxes

The box.new() function creates a new box object to display on the chart. It has the following signatures:

```
box.new(top_left, bottom_right, border_color, border_width, border_sty

box.new(left, top, right, bottom, border_color, border_width, border_s
```

This function's first overload includes the `top_left` and `bottom_right` parameters, which accept chart.point objects representing the top-left and bottom-right corners of the box, respectively. The function copies the information from these chart points to set the coordinates of the box's corners. Whether it uses the `index` or `time` fields of the `top_left` and `bottom_right` points as x-coordinates depends on the function's `xloc` value.

The second overload specifies `left`, `top`, `right`, and `bottom` edges of the box. The `left` and `right` parameters accept int values specifying the box's left and right x-coordinates, which can be bar index or time values depending on the `xloc` value in the function call. The `top` and `bottom` parameters accept float values representing the box's top and bottom y-coordinates.

The function's additional parameters are identical in both overloads:

`border_color`

Specifies the color of all four of the box's borders. The default is color.blue.

`border_width`

Specifies the width of the borders, in pixels. Its default value is 1.

`border_style`

Specifies the style of the borders, which can be any of the options in the Box styles section of this page.

`extend`

Determines whether the box's borders extend infinitely beyond the left or right x-coordinates. It accepts one of the following values: extend.left, extend.right, extend.both, or extend.none (default).

`xloc`

Determines whether the left and right edges of the box use bar index or time values as x-coordinates. The default is xloc.bar_index.

In the first overload, an `xloc` value of xloc.bar_index means that the function will use the `index` fields of the `top_left` and `bottom_right` chart points, and an `xloc` value of xloc.bar_time means that it will use their `time` fields.

In the second overload, using an `xloc` value of xloc.bar_index means the function treats the `left` and `right` values as bar indices, and xloc.bar_time means it will treat them as timestamps.

When the specified x-coordinates represent *bar index* values, it's important to note that the minimum x-coordinate allowed is `bar_index - 10000`. For larger offsets, one can use xloc.bar_time.

`bgcolor`

Specifies the background color of the space inside the box. The default value is color.blue.

`text`

The text to display inside the box. By default, its value is an empty string.

`text_size`

Specifies the size of the text within the box. It accepts both “int” size values and “string” `size.*` constants. The “int” size can be any positive integer. The `size.*` constants and their equivalent “int” sizes are: size.auto (0), size.tiny (8), size.small (10), size.normal (14), size.large (20), and size.huge (36). The default value is size.auto.

`text_color`

Controls the color of the text. Its default is color.black.

`text_halign`

Specifies the horizontal alignment of the text within the box’s boundaries. It accepts one of the following: text.align_left, text.align_right, or text.align_center (default).

`text_valign`

Specifies the vertical alignment of the text within the box’s boundaries. It accepts one of the following: text.align_top, text.align_bottom, or text.align_center (default).

`text_wrap`

Determines whether the box will wrap the text within it. If its value is text.wrap_auto, the box wraps the text to ensure it does not span past its vertical borders. It also clips the wrapped text when it extends past the bottom. If the value is text.wrap_none, the box displays the text on a single line that can extend beyond its borders. The default is text.wrap_none.

`text_font_family`

Defines the font family of the box’s text. Using font.family_default displays the box’s text with the system’s default font. The font.family_monospace displays the text in a monospace format. The default value is font.family_default.

`force_overlay`

If `true`, the drawing will display on the main chart pane, even when the script occupies a separate pane. Optional. The default is `false`.

`text_formatting`

Specifies the formatting of the box's text. Using text.format_none displays the text with no special formatting. This parameter also accepts the arguments text.format_bold or text.format_italic. Using `text.format_bold + text.format_italic` applies both formats together to display text that is both bold and italicized. The default value is text.format_none.

Let's write a simple script to display boxes on a chart. The example below draws a box projecting each bar's high and low values from the horizontal center of the current bar to the center of the next available bar.

On each bar, the script creates `topLeft` and `bottomRight` points via chart.point.now() and chart.point.from_index(), then calls box.new() to construct a new box and display it on the chart. It also highlights the background on the unconfirmed chart bar using bgcolor() to indicate that it redraws that box until the bar's last update:

image

//@version=6 indicator("Creating boxes demo", overlay = true) //@variable The `chart.point` for the top-left corner of the box. Contains `index` and `time` information. topLeft = chart.point.now(high) //@variable The `chart.point` for the bottom-right corner of the box. Does not contain `time` information. bottomRight = chart.point.from_index(bar_index + 1, low) // Draw a box using the `topLeft` and `bottomRight` corner points. Uses the `index` fields as x-coordinates. box.new(topLeft, bottomRight, color.purple, 2, bgcolor = color.new(color.gray, 70)) // Color the background on the unconfirmed bar. bgcolor(barstate.isconfirmed ? na : color.new(color.orange, 70), title = "Unconfirmed bar highlight")

Note that:

- The `bottomRight` point's `index` field is one bar greater than the `index` in the `topLeft`. If the x-coordinates of the corners were equal, the script would draw a vertical line at the horizontal center of each bar, resembling the example in this page's Creating lines section.

- Similar to lines, if the `topLeft` and `bottomRight` contained identical coordinates, the box wouldn't display on the chart since there would be no space between them to draw. However, its ID would still exist.
- This script only displays approximately the last 50 boxes on the chart, as we have not specified a `max_boxes_count` in the indicator() function call.

Modifying boxes

Multiple *setter* functions exist in the `box.*` namespace, allowing scripts to modify the properties of box objects:

- box.set_top_left_point() and box.set_bottom_right_point() respectively update the top-left and bottom-right coordinates of the `id` box using information from the specified `point`.
- box.set_left() and box.set_right() set the left or right x-coordinate of the `id` box to a new `left/right` value, which can be a bar index or time value depending on the box's `xloc` property.
- box.set_top() and box.set_bottom() set the top or bottom y-coordinate of the `id` box to a new `top/bottom` value.
- box.set_lefttop() sets the `left` and `top` coordinates of the `id` box, and box.set_rightbottom() sets its `right` and `bottom` coordinates.
- box.set_border_color(), box.set_border_width() and box.set_border_style() respectively update the `color`, `width`, and `style` of the `id` box's border.
- box.set_extend() sets the horizontal `extend` property of the `id` box.
- box.set_bgcolor() sets the color of the space inside the `id` box to a new `color`.
- box.set_text(), box.set_text_size(), box.set_text_color(), box.set_text_halign(), box.set_text_valign(), box.set_text_wrap(), box.set_text_font_family(), and box.set_text_formatting() update the `id` box's text-related properties.

As with setter functions in the `line.*` namespace, all box setters modify the `id` box directly without returning a value, and each setter function accepts "series" arguments.

Note that, unlike lines, the box.* namespace does not contain a setter function to modify a box's xloc. Users must create a new box with the desired xloc setting for such cases.

This example uses boxes to visualize the ranges of upward and downward bars with the highest volume over a user-defined timeframe. When the script detects a new bar on the specified timeframe with timeframe.change(), it assigns new boxes to the upBox and downBox variables, resets the upVolume and downVolume values, and highlights the chart background.

When an upward or downward bar's volume exceeds the upVolume or downVolume, the script updates the volume-tracking variables and calls box.set_top_left_point() and box.set_bottom_right_point() to update the upBox or downBox coordinates. The setters use the information from the chart points created with chart.point.now() and chart.point.from_time() to project that bar's high and low values from the current time to the closing time of the specified timeframe:

image

//@version=6 indicator("Modifying boxes demo", "High volume boxes", true, max_boxes_count = 100) //@variable The timeframe of the calculation. string timeframe = input.timeframe("D", "Timeframe") //@variable A box projecting the range of the upward bar with the highest `volume` over the `timeframe`. var box upBox = na //@variable A box projecting the range of the downward bar with the lowest `volume` over the `timeframe`. var box downBox = na // @variable The highest volume of upward bars over the `timeframe`. var float upVolume = na //@variable The highest volume of downward bars over the `timeframe`. var float downVolume = na // Color variables. var color upBorder = color.teal var color upFill = color.new(color.teal, 90) var color downBorder = color.maroon var color downFill = color.new(color.maroon, 90) //@variable The closing time of the `timeframe`. int closeTime = time_close(timeframe) //@variable Is `true` when a new bar starts on the `timeframe`. bool changeTF = timeframe.change(timeframe) //@variable The `chart.point` for the top-left corner of the boxes. Contains `index` and `time` information. topLeft = chart.point.now(high) //@variable The `chart.point` for the bottom-right corner of the boxes. Does not contain `index` information. bottomRight = chart.point.from_time(closeTime, low) if changeTF and not na(volume) if close > open // Update `upVolume` and `downVolume` values.

upVolume := volume downVolume := 0.0 // Draw a new `upBox` using `time` and `price` info from the `topLeft` and `bottomRight` points. upBox := box.new(topLeft, bottomRight, upBorder, 3, xloc = xloc.bar_time, bgcolor = upFill) // Draw a new `downBox` with `na` coordinates. downBox := box.new(na, na, na, na, downBorder, 3, xloc = xloc.bar_time, bgcolor = downFill) else // Update `upVolume` and `downVolume` values. upVolume := 0.0 downVolume := volume // Draw a new `upBox` with `na` coordinates. upBox := box.new(na, na, na, na, upBorder, 3, xloc = xloc.bar_time, bgcolor = upFill) // Draw a new `downBox` using `time` and `price` info from the `topLeft` and `bottomRight` points. downBox := box.new(topLeft, bottomRight, downBorder, 3, xloc = xloc.bar_time, bgcolor = downFill) // Update the ``upVolume`` and change the ``upBox`` coordinates when volume increases on an upward bar. else if close > open and volume > upVolume upVolume := volume box.set_top_left_point(upBox, topLeft) box.set_bottom_right_point(upBox, bottomRight) // Update the ``downVolume`` and change the ``downBox`` coordinates when volume increases on a downward bar. else if close <= open and volume > downVolume downVolume := volume box.set_top_left_point(downBox, topLeft) box.set_bottom_right_point(downBox, bottomRight) // Highlight the background when a new `timeframe` bar starts. bgcolor(changeTF ? color.new(color.orange, 70) : na, title = "Timeframe change highlight")

Note that:

- The indicator() function call contains `max_boxes_count = 100`, meaning the script will preserve the last 100 boxes on the chart.
- We utilized *both overloads* of box.new() in this example. On the first bar of the `timeframe`, the script calls the first overload for the upBox when the bar is rising, and it calls that overload for the downBox when the bar is falling. It uses the second overload to assign a new box with na values to the other box variable on that bar.

Box styles

Users can include one of the following `line.style_*` variables in their box.new() or box.set_border_style() function calls to set the border styles of boxes drawn by their scripts:

Argument	Box
`line.style_solid`	box_style_solid
`line.style_dotted`	box_style_dotted
`line.style_dashed`	box style dashed

Reading box values

The `box.*` namespace features *getter* functions that allow scripts to retrieve coordinate values from a box instance:

- box.get_left() and box.get_right() respectively get the x-coordinates of the left and right edges of the `id` box. Whether the value returned represents a bar index or time value depends on the box's `xloc` property.
- box.get_top() and box.get_bottom() respectively get the top and bottom y-coordinates of the `id` box.

The example below draws boxes to visualize hypothetical price ranges over a period of `length` bars. At the start of each new period, it uses the average candle range multiplied by the `scaleFactor` input to calculate the corner points of a box centered at the hl2 price with an `initialRange` height. After drawing the first box, it creates `numberOfBoxes - 1` new boxes inside a for loop.

Within each loop iteration, the script gets the `lastBoxDrawn` reference by retrieving the last element from the read-only box.all array, then calls box.get_top() and box.get_bottom() to get the y-coordinates of the referenced box. It uses these values to calculate the coordinates for a new box that's `scaleFactor` times taller than the previous:

image

//@version=6 indicator("Reading box values demo", "Nested boxes", overlay = true, max_boxes_count = 500) //@variable The number of bars in the range calculation. int length = input.int(10, "Length", 2, 500) //@variable The number of nested boxes drawn on each period. int numberOfBoxes = input.int(5, "Nested box count", 1) //@variable The scale factor applied to each box. float scaleFactor = input.float(1.6, "Scale factor", 1) //@variable The initial box range. float initialRange = scaleFactor * ta.sma(high - low,

length) if bar_index % length == 0 //@variable The top-left `chart.point` for the initial box. Does not contain `time` information. topLeft = chart.point.from_index(bar_index, hl2 + initialRange / 2) //@variable The bottom-right `chart.point` for the initial box. Does not contain `time` information. bottomRight = chart.point.from_index(bar_index + length, hl2 - initialRange / 2) // Calculate border and fill colors of the boxes. borderColor = color.rgb(math.random(100, 255), math.random(0, 100), math.random(100, 255)) bgColor = color.new(borderColor, math.max(100 * (1 - 1/ numberOfBoxes), 90)) // Draw a new box using the `topLeft` and `bottomRight` points. Uses their `index` fields as x-coordinates. box.new(topLeft, bottomRight, borderColor, 2, bgcolor = bgColor) if numberOfBoxes > 1 // Loop to create additional boxes. for i = 1 to numberOfBoxes - 1 //@variable The last box drawn by the script. box lastBoxDrawn = box.all.last() //@variable The top price of the last box. float top = box.get_top(lastBoxDrawn) //@variable The bottom price of the last box. float bottom = box.get_bottom(lastBoxDrawn) //@variable The scaled range of the new box. float newRange = scaleFactor * (top - bottom) * 0.5 // Update the `price` fields of the `topLeft` and `bottomRight` points. // This does not affect the coordinates of previous boxes. topLeft.price := hl2 + newRange bottomRight.price := hl2 - newRange // Draw a new box using the updated `topLeft` and `bottomRight` points. box.new(topLeft, bottomRight, borderColor, 2, bgcolor = bgColor)

Note that:

- The indicator() function call uses `max_boxes_count = 500`, meaning the script can display up to 500 boxes on the chart.
- Each drawing has a `right` index `length` bars beyond the `left` index. Since the x-coordinates of these drawings can be up to 500 bars into the future, we've set the `maxval` of the `length` input to 500.
- On each new period, the script uses randomized color.rgb() values for the `border_color` and `bgcolor` of the boxes.
- Each box.new() call copies the coordinates from the chart.point objects assigned to the `topLeft` and `bottomRight` variables, which is why the script can modify their `price` fields on each loop iteration without affecting the other boxes.

Cloning boxes

To clone a specific box `id`, use box.copy(). This function copies the box and its properties. Any changes to the copied box do not affect the original.

For example, this script declares an `originalBox` variable on the first bar and assigns a new box reference to it once every `length` bars. On other bars, it uses box.copy() to create a copy of the box, assigns that copy to the `copiedBox` variable, then calls `box.set_*()` functions to modify the copy's properties. As shown on the chart below, changes to the copied box do not modify the box referenced by `originalBox`:

image

//@version=6 indicator("Cloning boxes demo", overlay = true, max_boxes_count = 500) //@variable The number of bars between each new mainLine assignment. int length = input.int(20, "Length", 2) //@variable The `chart.point` for the top-left of the `originalBox`. Contains `time` and `index` information. topLeft = chart.point.now(high) //@variable The `chart.point` for the bottom-right of the `originalBox`. Does not contain `time` information. bottomRight = chart.point.from_index(bar_index + 1, low) //@variable A new box with `topLeft` and `bottomRight` corners on every `length` bars. var box originalBox = na //@variable Is teal when the bar is rising, maroon when it's falling. color originalColor = close > open ? color.teal : color.maroon if bar_index % length == 0 // Assign a new box using the `topLeft` and `bottomRight` info to the `originalBox`. // This box uses the `index` fields from the points as x-coordinates. originalBox := box.new(topLeft, bottomRight, originalColor, 2, bgcolor = color.new(originalColor, 60)) else // @variable A clone of the `originalBox`. box copiedBox = box.copy(originalBox) // Modify the `copiedBox`. These changes do not affect the `originalBox`. box.set_top(copiedBox, high) box.set_bottom_right_point(copiedBox, bottomRight) box.set_border_color(copiedBox, color.gray) box.set_border_width(copiedBox, 1) box.set_bgcolor(copiedBox, na)

Deleting boxes

To delete boxes drawn by a script, use box.delete(). As with `*.delete()` functions in other drawing namespaces, this function is handy for

conditionally removing boxes or maintaining a specific number of boxes on the chart.

This example displays boxes representing periodic cumulative volume values. The script creates a new box ID and stores it in a `boxes` array once every `length` bars. If the array's size exceeds the `numberOfBoxes` value, the script removes the oldest box from the array using array.shift() and deletes it using box.delete().

On other bars, it accumulates volume over each period by modifying the `top` property of the last box in the `boxes` array. The script then uses `for` loops to find the `highestTop` of all the array's boxes and set the `bgcolor` of each box with a gradient color created using color.from_gradient() based on its box.get_top() value relative to the `highestTop`:

image

//@version=6 //@variable The maximum number of boxes to show on the chart. const int MAX_BOXES_COUNT = 500 indicator("Deleting boxes demo", "Cumulative volume boxes", format = format.volume, max_boxes_count = MAX_BOXES_COUNT) //@variable The number of bars in each period. int length = input.int(20, "Length", 1) //@variable The maximum number of volume boxes in the calculation. int numberOfBoxes = input.int(10, "Number of boxes", 1, MAX_BOXES_COUNT) //@variable An array containing the ID of each box displayed by the script. var boxes = array.new<box>() if bar_index % length == 0 // Push a new box into the `boxes` array. The box has the default `xloc.bar_index` property. boxes.push(box.new(bar_index, 0, bar_index + 1, 0, #000000, 2, text_color = #000000)) // Shift the oldest box out of the array and delete it when the array's size exceeds the `numberOfBoxes`. if boxes.size() > numberOfBoxes box.delete(boxes.shift()) //@variable The last box drawn by the script as of the current chart bar. box lastBox = boxes.last() // Add the current bar's volume to the top of the `lastBox` and update the `right` index. lastBox.set_top(lastBox.get_top() + volume) lastBox.set_right(bar_index + 1) // Display the top of the `lastBox` as volume-formatted text. lastBox.set_text(str.tostring(lastBox.get_top(), format.volume)) //@variable The highest `top` of all boxes in the `boxes` array. float highestTop = 0.0 for id in boxes highestTop := math.max(id.get_top(), highestTop) // Set the `bgcolor` of each `id` in `boxes` with a gradient based on the ratio of its

`top` to the `highestTop`. for id in boxes
id.set_bgcolor(color.from_gradient(id.get_top() / highestTop, 0, 1, color.purple, color.orange))

Note that:

- At the top of the code, we've declared a `MAX_BOXES_COUNT` variable with the "const int" *qualified type*. We use this value as the `max_boxes_count` in the indicator() function and the maximum possible value of the `numberOfBoxes` input.
- This script uses the second overload of the box.new() function, which specifies the box's `left`, `top`, `right`, and `bottom` coordinates separately.
- We've included format.volume as the `format` argument in the indicator() call, which tells the script that the y-axis of the chart pane represents *volume* values. Each box also displays its `top` coordinate as volume-formatted text.

Polylines

Pine Script polylines are **advanced** drawings that sequentially connect the coordinates from an array of chart.point instances using straight or *curved* line segments.

These powerful drawings can connect up to 10,000 points at any available location on the chart, allowing scripts to draw custom series, polygons, and other complex geometric formations that are otherwise difficult or impossible to draw using line or box objects.

The `polyline.*` namespace features the following built-ins for creating and managing polyline objects:

- The polyline.new() function creates a new polyline instance.
- The polyline.delete() function deletes an existing polyline instance.
- The polyline.all variable references a read-only array containing the IDs of all polylines displayed by the script. The array's size depends on the `max_polylines_count` of the indicator() or strategy() declaration statement and the number of polylines drawn by the script.

Unlike lines or boxes, polylines do not have functions for modification or reading their properties. To redraw a polyline on the chart, one can *delete* the existing instance and *create* a new polyline with the desired changes.

Creating polylines

The polyline.new() function creates a new polyline instance to display on the chart. It has the following signature:

```
polyline.new(points, curved, closed, xloc, line_color, fill_color, lin
```

The following eight parameters affect the behavior of a polyline drawing:

`points`

Accepts an array of chart.point objects that determine the coordinates of each point in the polyline. The drawing connects the coordinates from each element in the array sequentially, starting from the *first*. Whether the polyline uses the `index` or `time` field from each chart point for its x-coordinates depends on the `xloc` value in the function call.

`curved`

Specifies whether the drawing uses curved line segments to connect each chart.point in the `points` array. The default value is `false`, meaning it uses straight line segments.

`closed`

Controls whether the polyline will connect the last chart.point in the `points` array to the first, forming a *closed polyline*. The default value is `false`.

`xloc`

Specifies which field from each chart.point in the `points` array the polyline uses for its x-coordinates. When its value is xloc.bar_index, the function uses the `index` fields to create the polyline. When its value is xloc.bar_time, the function uses the `time` fields. The default value is xloc.bar_index.

`line_color`

Specifies the color of all line segments in the polyline drawing. The default is `color.blue`.

`fill_color`

Controls the color of the closed space filled by the polyline drawing. Its default value is na.

`line_style`

Specifies the style of the polyline, which can be any of the available options in the Line styles section of this page. The default is line.style_solid.

`line_width`

Specifies the width of the polyline, in pixels. The default value is 1.

`force_overlay`

If `true`, the drawing will display on the main chart pane, even when the script occupies a separate pane. Optional. The default is `false`.

This script demonstrates a simple example of drawing a polyline on the chart. It uses array.push to push the reference of a new chart.point object with an alternating `price` value into a `points` array and colors the background with bgcolor() once every `length` bars.

On the last confirmed historical bar, where barstate.islastconfirmedhistory is `true`, the script creates a new polyline with polyline.new(). The polyline drawing passes through the coordinates of each chart point in the `points` array in order, starting from the first point:

image

//@version=6 indicator("Creating polylines demo", "Oscillating polyline") // @variable The number of bars between each point in the drawing. int length = input.int(20, "Length between points", 2) //@variable An array of `chart.point` objects to sequentially connect with a polyline. var points = array.new<chart.point>() //@variable The y-coordinate of each point in the `points`. Alternates between 1 and -1 on each `newPoint`. var int yValue = 1 //@variable Is `true` once every `length` bars, `false` otherwise. bool newPoint = bar_index % length == 0 if newPoint // Push a new `chart.point`

into the `points`. The new point contains `time` and `index` info. points.push(chart.point.now(yValue)) // Change the sign of the `yValue`. yValue *= -1 // Draw a new `polyline` on the last confirmed historical chart bar. // The polyline uses the `time` field from each `chart.point` in the `points` array as x-coordinates. if barstate.islastconfirmedhistory polyline.new(points, xloc = xloc.bar_time, line_color = #9151A6, line_width = 3) // Highlight the chart background on every `newPoint` condition. bgcolor(newPoint ? color.new(color.gray, 70) : na, title = "New point highlight")

Note that:

- This script uses only *one* polyline to connect each chart point from the array with straight line segments, and this drawing spans throughout the available chart data, starting from the first bar.
- While one can achieve a similar effect using lines, doing so would require a new line instance on each occurrence of the `newPoint` condition, and such a drawing would be limited to a maximum of 500 line segments. This single unclosed polyline drawing, on the other hand, can contain up to 9,999 line segments.

Curved drawings

Polylines can draw *curves* that are otherwise impossible to produce with lines or boxes. When enabling the `curved` parameter of the polyline.new() function, the resulting polyline interpolates *nonlinear* values between the coordinates from each chart.point in its array of `points` to generate a curvy effect.

For instance, the “Oscillating polyline” script in our previous example uses *straight* line segments to produce a drawing resembling a triangle wave, meaning a waveform that zig-zags between its peaks and valleys. If we set the `curved` parameter in the polyline.new() call from that example to `true`, the resulting drawing would connect the points using *curved* segments, producing a smooth, nonlinear shape similar to a sine wave:

image

//@version=6 indicator("Curved drawings demo", "Smooth oscillating polyline") //@variable The number of bars between each point in the drawing. int length = input.int(20, "Length between points", 2) //@variable An array of `chart.point` objects to sequentially connect with a polyline. var points = array.new<chart.point>() //@variable The y-coordinate of each point in the `points`. Alternates between 1 and -1 on each `newPoint`. var int yValue = 1 //@variable Is `true` once every `length` bars, `false` otherwise. bool newPoint = bar_index % length == 0 if newPoint // Push a new `chart.point` into the `points`. The new point contains `time` and `index` info. points.push(chart.point.now(yValue)) // Change the sign of the `yValue`. yValue *= -1 // Draw a new curved `polyline` on the last confirmed historical chart bar. // The polyline uses the `time` field from each `chart.point` in the `points` array as x-coordinates. if barstate.islastconfirmedhistory polyline.new(points, curved = true, xloc = xloc.bar_time, line_color = #9151A6, line_width = 3) // Highlight the chart background on every `newPoint` condition. bgcolor(newPoint ? color.new(color.gray, 70) : na, title = "New point highlight")

Notice that in this example, the smooth curves have relatively consistent behavior, and no portion of the drawing extends past its defined coordinates, which is not always the case when drawing curved polylines. The data used to construct a polyline heavily impacts the smooth, piecewise function it interpolates between its points. In some cases, the interpolated curve *can* reach beyond its actual coordinates.

Let’s add some variation to the chart points in our example’s `points` array to demonstrate this behavior. In the version below, the script multiplies `yValue` by a pseudorandom value in each chart.point.now() call.

To visualize the behavior, this script also creates a horizontal line at the `price` value from each chart.point instance in the `points` array, and it displays another polyline connecting the same points with straight line segments. As we see on the chart, both polylines pass through all coordinates from the `points` array. However, the curvy polyline occasionally reaches *beyond* the vertical boundaries indicated by the horizontal lines, whereas the polyline drawn using straight segments does not:

image

//@version=6 indicator("Curved drawings demo", "Random oscillating polylines") //@variable The number of bars between each point in the drawing. int length = input.int(20, "Length between points", 2) //@variable An array of `chart.point` objects to sequentially connect with a polyline. var points = array.new<chart.point>() //@variable The sign of each `price` in the `points`. Alternates between 1 and -1 on each `newPoint`. var int yValue = 1 //@variable Is `true` once every `length` bars. bool newPoint = bar_index % length == 0 if newPoint // Push a new `chart.point` with a randomized `price` into the `points`. // The new point contains `time` and `index` info. points.push(chart.point.now(yValue * math.random())) // Change the sign of the `yValue`. yValue *= -1 //@variable The newest `chart.point`. lastPoint = points.last() // Draw a horizontal line at the `lastPoint.price`. This line uses the default `xloc.bar_index`. line.new(lastPoint.index - length, lastPoint.price, lastPoint.index + length, lastPoint.price, color = color.red) // Draw two `polyline` instances on the last confirmed chart bar. // Both polylines use the `time` field from each `chart.point` in the `points` array as x-coordinates. if barstate.islastconfirmedhistory polyline.new(points, curved = false, xloc = xloc.bar_time, line_color = #EB8A3B, line_width = 2) polyline.new(points, curved = true, xloc = xloc.bar_time, line_color = #9151A6, line_width = 3) // Highlight the chart background on every `newPoint` condition. bgcolor(newPoint ? color.new(color.gray, 70) : na, title = "New point highlight")

Closed shapes

Since a single polyline can contain numerous straight or curved line segments, and the `closed` parameter allows the drawing to connect the coordinates from the first and last chart.point in its array of `points`, programmers can use polylines to draw many different types of closed polygonal shapes.

Let’s draw some polygons in Pine. The following script periodically draws randomized polygons centered at hl2 price values.

On each occurrence of the `newPolygon` condition, the script clears the `points` array with array.clear(), calculates `numberOfSides` and `rotationOffset` values based on values from math.random() calls, then uses a `for` loop to push `numberOfSides` new chart points into the array. The chart points

contain stepped coordinates from an elliptical path with `xScale` and `yScale` semi-axes. The script draws the polygon by connecting each point from the `points` array using a *closed polyline* with straight line segments:

image

//@version=6 indicator("Closed shapes demo", "N-sided polygons", true) // @variable The size of the horizontal semi-axis. float xScale = input.float(3.0, "X scale", 1.0) //@variable The size of the vertical semi-axis. float yScale = input.float(1.0, "Y scale") * ta.atr(2) //@variable An array of `chart.point` objects containing vertex coordinates. var points = array.new<chart.point>() //@variable The condition that triggers a new polygon drawing. Based on the horizontal axis to prevent overlaps. bool newPolygon = bar_index % int(math.round(2 * xScale)) == 0 and barstate.isconfirmed if newPolygon // Clear the `points` array. points.clear() // @variable The number of sides and vertices in the new polygon. int numberOfSides = int(math.random(3, 7)) //@variable A random rotation offset applied to the new polygon, in radians. float rotationOffset = math.random(0.0, 2.0) * math.pi //@variable The size of the angle between each vertex, in radians. float step = 2 * math.pi / numberOfSides //@variable The counter-clockwise rotation angle of each vertex. float angle = rotationOffset for i = 1 to numberOfSides //@variable The approximate x-coordinate from an ellipse at the `angle`, rounded to the nearest integer. int xValue = int(math.round(xScale * math.cos(angle))) + bar_index //@variable The y-coordinate from an ellipse at the `angle`. float yValue = yScale * math.sin(angle) + hl2 // Push a new `chart.point` containing the `xValue` and `yValue` into the `points` array. // The new point does not contain `time` information. points.push(chart.point.from_index(xValue, yValue)) // Add the `step` to the `angle`. angle += step // Draw a closed polyline connecting the `points`. // The polyline uses the `index` field from each `chart.point` in the `points` array. polyline.new(points, closed = true, line_color = color.navy, fill_color = color.new(color.orange, 50), line_width = 3)

Note that:

- This example shows the last ~50 polylines on the chart, as we have not specified a `max_polylines_count` value in the indicator() function call.
- The `yScale` calculation multiplies an input.float() by `ta.atr(2)` to adapt the vertical scale of the drawings to recent price ranges.

- The resulting polygons have a maximum width of twice the horizontal semi-axis (`2 * xScale`), rounded to the nearest integer. The `newPolygon` condition uses this value to prevent the polygon drawings from overlapping.
- The script rounds the `xValue` calculation to the nearest integer because the `index` field of a chart.point only accepts an int value, as the x-axis of the chart does not include fractional bar indices.

Deleting polylines

To delete a specific polyline, use polyline.delete(). This function removes the polyline object from the script and its drawing on the chart.

As with other drawing objects, we can use polyline.delete() to maintain a specific number of polyline drawings or conditionally remove drawings from a chart.

For example, the script below periodically draws approximate arithmetic spirals and stores their polyline references in an array, which it uses as a queue to manage the number of drawings it displays.

When the `newSpiral` condition occurs, the script creates a `points` array and adds chart points within a `for` loop. On each loop iteration, it calls the `spiralPoint()` user-defined function to create a new chart.point containing stepped values from an elliptical path that grows with respect to the `angle` value. The script then creates a randomly colored *curved polyline* connecting the coordinates from the `points` and pushes its reference into the `polylines` array.

When the array's size exceeds the `numberOfSpirals` value, the script removes the oldest polyline reference using array.shift() and deletes the object using polyline.delete():

image

//@version=6 //@variable The maximum number of polylines allowed on the chart. const int MAX_POLYLINES_COUNT = 100 indicator("Deleting polylines example", "Spirals", true, max_polylines_count = MAX_POLYLINES_COUNT) //@variable The number of spiral drawings on the chart. int numberOfSpirals = input.int(10, "Spirals shown", 1, MAX_POLYLINES_COUNT) //@variable The

number of full spiral rotations to draw. int rotations = input.int(5, "Rotations", 1) //@variable The scale of the horizontal semi-axis. float xScale = input.float(1.0, "X scale") //@variable The scale of the vertical semi-axis. float yScale = input.float(0.2, "Y scale") * ta.atr(2) //@function Calculates an approximate point from an elliptically-scaled arithmetic spiral. //@returns A `chart.point` with `index` and `price` information. spiralPoint(float angle, int xOffset, float yOffset) => result = chart.point.from_index(int(math.round(angle * xScale * math.cos(angle))) + xOffset, angle * yScale * math.sin(angle) + yOffset) //@variable An array of polylines. var polylines = array.new<polyline>() //@variable The condition to create a new spiral. bool newSpiral = bar_index % int(math.round(4 * math.pi * rotations * xScale)) == 0 if newSpiral //@variable An array of `chart.point` objects for the `spiral` drawing. points = array.new<chart.point>() //@variable The counter-clockwise angle between calculated points, in radians. float step = math.pi / 2 //@variable The rotation angle of each calculated point on the spiral, in radians. float theta = 0.0 // Loop to create the spiral's points. Creates 4 points per full rotation. for i = 0 to rotations * 4 //@variable A new point on the calculated spiral. chart.point newPoint = spiralPoint(theta, bar_index, ohlc4) // Add the `newPoint` to the `points` array. points.push(newPoint) // Add the `step` to the `theta` angle. theta += step //@variable A random color for the new `spiral` drawing. color spiralColor = color.rgb(math.random(150, 255), math.random(0, 100), math.random(150, 255)) //@variable A new polyline connecting the spiral points. Uses the `index` field from each point as x-coordinates. polyline spiral = polyline.new(points, true, line_color = spiralColor, line_width = 3) // Push the new `spiral` into the `polylines` array. polylines.push(spiral) // Shift the first polyline out of the array and delete it when the array's size exceeds the `numberOfSpirals`. if polylines.size() > numberOfSpirals polyline.delete(polylines.shift()) // Highlight the background when `newSpiral` is `true`. bgcolor(newSpiral ? color.new(color.blue, 70) : na, title = "New drawing highlight")

Note that:

- We declared a `MAX_POLYLINES_COUNT` global variable with a constant value of 100. The script uses this constant as the `max_polylines_count` value in the indicator() function and the `maxval` of the `numberOfSpirals` input.

- As with our “N-sided polygons” example in the previous section, we round the calculation of x-coordinates to the nearest integer since the `index` field of a chart.point can only accept an int value.
- Despite the smooth appearance of the drawings, each polyline’s `points` array only contains *four* chart.point objects per spiral rotation. Since the polyline.new() call includes `curved = true`, each polyline uses *smooth curves* to connect their `points`, producing a visual approximation of the spiral’s actual curvature.
- The width of each spiral is approximately `4 * math.pi * rotations * xScale`, rounded to the nearest integer. We use this value in the `newSpiral` condition to space each drawing and prevent overlaps.

Redrawing polylines

It may be desirable in some cases to change a polyline drawing throughout a script’s execution. While the `polyline.*` namespace does not contain built-in setter functions, we can *redraw* polylines referenced by variables or collections by *deleting* the existing polylines and assigning *new instances* with the desired changes.

The following example uses polyline.delete() and polyline.new() calls to update the value of a polyline variable.

This script draws closed polylines that connect the open, high, low, and close points of periods containing `length` bars. It creates a `currentDrawing` variable on the first bar and assigns a polyline reference to it on every chart bar. It uses the `openPoint`, `highPoint`, `lowPoint`, and `closePoint` variables to reference chart points that track the period’s developing OHLC values. As new values emerge, the script assigns new chart.point objects to the variables, collects them in an array using array.from(), then creates a new polyline connecting the coordinates from the array’s points with polyline.new().

When the `newPeriod` condition is `false` (i.e., the current period is not complete), the script deletes the polyline referenced by the `currentDrawing` variable before creating a new one, resulting in a dynamic drawing that changes over the developing period:

image

```
//@version=6 indicator("Redrawing polylines demo", "OHLC polygons", true, max_polylines_count = 100) //@variable The length of the period. int length = input.int(100, "Length", 1) //@variable A `chart.point` representing the start of each period. var chart.point openPoint = na //@variable A `chart.point` representing the highest point of each period. var chart.point highPoint = na //@variable A `chart.point` representing the lowest point of each period. var chart.point lowPoint = na //@variable A `chart.point` representing the current bar's closing point. closePoint = chart.point.now(close) //@variable The current period's polyline drawing. var polyline currentDrawing = na //@variable Is `true` once every `length` bars. bool newPeriod = bar_index % length == 0 if newPeriod // Assign new chart points to the `openPoint`, `highPoint`, and `closePoint`. openPoint := chart.point.now(open) highPoint := chart.point.now(high) lowPoint := chart.point.now(low) else // Assign a new `chart.point` to the `highPoint` when the `high` is greater than its `price`. if high > highPoint.price highPoint := chart.point.now(high) // Assign a new `chart.point` to the `lowPoint` when the `low` is less than its `price`. if low < lowPoint.price lowPoint := chart.point.now(low) //@variable Is teal when the `closePoint.price` is greater than the `openPoint.price`, maroon otherwise. color drawingColor = closePoint.price > openPoint.price ? color.teal : color.maroon // Delete the polyline assigned to the `currentDrawing` if it's not a `newPeriod`. if not newPeriod polyline.delete(currentDrawing) // Assign a new polyline to the `currentDrawing`. // Uses the `index` field from each `chart.point` in its array as x-coordinates. currentDrawing := polyline.new( array.from(openPoint, highPoint, closePoint, lowPoint), closed = true, line_color = drawingColor, fill_color = color.new(drawingColor, 60) )
```

Realtime behavior

Lines, boxes, and polylines are subject to both *commit* and *rollback* actions, which affect the behavior of a script when it executes on a realtime bar. See the page on Pine Script's Execution model.

This script demonstrates the effect of rollback when it executes on the realtime, *unconfirmed* chart bar:

image

//@version=6 indicator("Realtime behavior demo", overlay = true) // @variable Is orange when the `line` is subject to rollback and gray after the `line` is committed. color lineColor = barstate.isconfirmed ? color.gray : color.orange line.new(bar_index, hl2, bar_index + 1, hl2, color = lineColor, width = 4)

The line.new() call in this example creates a new line ID on each iteration when values change on the unconfirmed bar. The script automatically deletes the objects created on each change in that bar because of the *rollback* before each iteration. It only *commits* the last line created before the bar closes, and that line instance is the one that persists on the confirmed bar.

Limitations

Total number of objects

Lines, boxes, and polylines consume server resources, which is why there are limits on the total number of drawings per script. When a script creates more drawing objects than the allowed limit, the Pine Script runtime automatically deletes the oldest ones in a process referred to as *garbage collection*.

A single script can contain up to 500 lines, 500 boxes, and 100 polylines. Users can control the garbage collection limits by specifying the `max_lines_count`, `max_boxes_count`, and `max_polylines_count` values in their script's indicator() or strategy() declaration statement.

This script demonstrates how garbage collection works in Pine. It creates a new line, box, and polyline on each chart bar. We haven't specified values for the `max_lines_count`, `max_boxes_count`, or `max_polylines_count` parameters in the indicator() function call, so the script will maintain the most recent ~50 lines, boxes, and polylines on the chart, as this is the default setting for each parameter:

image

//@version=6 indicator("Garbage collection demo", overlay = true) // @variable A new `chart.point` at the current `bar_index` and `high`. firstPoint = chart.point.now(high) //@variable A new `chart.point` one bar into the future at the current `low`. secondPoint =

chart.point.from_index(bar_index + 1, low) //@variable A new `chart.point` one bar into the future at the current `high`. thirdPoint = chart.point.from_index(bar_index + 1, high) // Draw a new `line` connecting the `firstPoint` to the `secondPoint`. line.new(firstPoint, secondPoint, color = color.red, width = 2) // Draw a new `box` with the `firstPoint` top-left corner and `secondPoint` bottom-right corner. box.new(firstPoint, secondPoint, color.purple, 2, bgcolor = na) // Draw a new `polyline` connecting the `firstPoint`, `secondPoint`, and `thirdPoint` sequentially. polyline.new(array.from(firstPoint, secondPoint, thirdPoint), true, line_width = 2)

Note that:

- We've used TradingView's "Measure" drawing tool to measure the number of bars covered by the script's drawing objects.

Past and future references with `xloc.bar_index`

Objects positioned using xloc.bar_index can contain x-coordinates no further than 500 bars into the future or 10,000 bars into the past.

Other contexts

Scripts cannot use lines, boxes, or polylines in `request.*()` functions. Instances of these types can use the values from `request.*()` calls, but scripts can only create and draw them in the chart's context.

This limitation is also why drawing objects will not work when using the `timeframe` parameter in the indicator() declaration statement.

Historical buffer and `max_bars_back`

Using barstate.isrealtime in combination with drawings may sometimes produce unexpected results. For example, the intention of this script is to ignore all historical bars and draw horizontal lines spanning 300 bars back on *realtime* bars:

//@version=6 indicator("Historical buffer demo", overlay = true) //@variable A `chart.point` at the `bar_index` from 300 bars ago and current `close`.

firstPoint = chart.point.from_index(bar_index[300], close) //@variable The current bar's `chart.point` containing the current `close`. secondPoint = chart.point.now(close) // Draw a new line on realtime bars. if barstate.isrealtime line.new(firstPoint, secondPoint)

However, it will fail at runtime and raise an error. The script fails because it cannot determine the buffer size for historical values of the underlying time series. Although the code doesn’t contain the built-in time variable, the built-in bar_index uses the time series in its inner workings. Therefore, accessing the value of the bar_index from 300 bars back requires the history buffer of the time series to be at least 300 bars.

Pine Script includes a mechanism that detects the required historical buffer size automatically in most cases. It works by letting the script access historical values any number of bars back for a limited duration. In this script’s case, using barstate.isrealtime to control the drawing of lines prevents it from accessing the historical series, so it cannot infer the required historical buffer size, and the script fails.

The simple solution to this issue is to use the max_bars_back() function to *explicitly define* the historical buffer of the time series before evaluating the conditional structure:

//@version=6 indicator("Historical buffer demo", overlay = true) //@variable A `chart.point` at the `bar_index` from 300 bars ago and current `close. firstPoint = chart.point.from_index(bar_index[300], close) //@variable The current bar's `chart.point` containing the current `close`. secondPoint = chart.point.now(close) // Explicitly set the historical buffer of the `time` series to 300 bars. max_bars_back(time, 300) // Draw a new line on realtime bars. if barstate.isrealtime line.new(firstPoint, secondPoint)

Such issues can be confusing, but they’re quite rare. The Pine Script team hopes to eliminate them over time.

Previous

Next

Non-standard charts data

Non-standard charts data

Introduction

Pine Script® features several `ticker.*()` functions that generate *ticker identifiers* for requesting data from *non-standard* chart feeds. The available functions that create these ticker IDs are ticker.heikinashi(), ticker.renko(), ticker.linebreak(), ticker.kagi(), and ticker.pointfigure(). Scripts can use these functions' returned values as the `symbol` argument in request.security() calls to access non-standard chart data while running on *any* chart type.

ticker.heikinashi()

Heikin-Ashi means *average bar* in Japanese. The open/high/low/close values of Heikin-Ashi candlesticks are synthetic; they are not actual market prices. They are calculated by averaging combinations of real OHLC values from the current and previous bar. The calculations used make Heikin-Ashi bars less noisy than normal candlesticks. They can be useful to make visual assessments, but are unsuited to backtesting or automated trading, as orders execute on market prices — not Heikin-Ashi prices.

The ticker.heikinashi() function creates a special ticker identifier for requesting Heikin-Ashi data with the request.security() function.

This script requests the close value of Heikin-Ashi bars and plots them on top of the normal candlesticks:

image

//@version=6 indicator("HA Close", "", true) haTicker = ticker.heikinashi(syminfo.tickerid) haClose = request.security(haTicker, timeframe.period, close) plot(haClose, "HA Close", color.black, 3)

Note that:

- The close values for Heikin-Ashi bars plotted as the black line are very different from those of real candles using market prices. They act more like a moving average.
- The black line appears over the chart bars because we have selected “Visual Order/Bring to Front” from the script’s “More” menu.

If you wanted to omit values for extended hours in the last example, an intermediary ticker without extended session information would need to be created first:

image

//@version=6 indicator("HA Close", "", true) regularSessionTicker = ticker.new(syminfo.prefix, syminfo.ticker, session.regular) haTicker = ticker.heikinashi(regularSessionTicker) haClose = request.security(haTicker, timeframe.period, close, gaps = barmerge.gaps_on) plot(haClose, "HA Close", color.black, 3, plot.style_linebr)

Note that:

- We use the ticker.new() function first, to create a ticker without extended session information.
- We use that ticker instead of syminfo.tickerid in our ticker.heikinashi() call.
- In our request.security() call, we set the `gaps` parameter’s value to `barmerge.gaps_on`. This instructs the function not to use previous values to fill slots where data is absent. This makes it possible for it to return na values outside of regular sessions.
- To be able to see this on the chart, we also need to use a special `plot.style_linebr` style, which breaks the plots on na values.

This script plots Heikin-Ashi candles under the chart:

image

```
//@version=6 indicator("Heikin-Ashi candles") CANDLE_GREEN = #26A69A
CANDLE_RED = #EF5350 haTicker = ticker.heikinashi(syminfo.tickerid) [haO,
haH, haL, haC] = request.security(haTicker, timeframe.period, [open, high,
low, close]) candleColor = haC >= haO ? CANDLE_GREEN : CANDLE_RED
plotcandle(haO, haH, haL, haC, color = candleColor)
```

Note that:

- We use a tuple with request.security() to fetch four values with the same call.
- We use plotcandle() to plot our candles. See the Bar plotting page for more information.

ticker.renko()

Renko bars only plot price movements, without taking time or volume into consideration. They look like bricks stacked in adjacent columns. A new brick is only drawn after the price passes the top or bottom by a predetermined amount. The ticker.renko() function creates a ticker id which can be used with request.security() to fetch Renko values, but there is no Pine Script function to draw Renko bars on the chart:

```
//@version=6 indicator("", "", true) renkoTicker =
ticker.renko(syminfo.tickerid, "ATR", 10) renkoLow =
request.security(renkoTicker, timeframe.period, low) plot(renkoLow)
```

ticker.linebreak()

The *Line Break* chart type displays a series of vertical boxes that are based on price changes. The ticker.linebreak() function creates a ticker id which can be used with request.security() to fetch “Line Break” values, but there is no Pine Script function to draw such bars on the chart:

```
//@version=6 indicator("", "", true) lineBreakTicker =
ticker.linebreak(syminfo.tickerid, 3) lineBreakClose =
request.security(lineBreakTicker, timeframe.period, close)
plot(lineBreakClose)
```

ticker.kagi()

Kagi charts are made of a continuous line that changes directions. The direction changes when the price changes beyond a predetermined amount. The ticker.kagi() function creates a ticker id which can be used with request.security() to fetch “Kagi” values, but there is no Pine Script function to draw such bars on the chart:

//@version=6 indicator("", "", true) kagiBreakTicker = ticker.linebreak(syminfo.tickerid, 3) kagiBreakClose = request.security(kagiBreakTicker, timeframe.period, close) plot(kagiBreakClose)

ticker.pointfigure()

Point and Figure (PnF) charts only plot price movements, without taking time into consideration. A column of X’s is plotted as the price rises, and O’s are plotted when price drops. The ticker.pointfigure() function creates a ticker id which can be used with request.security() to fetch “PnF” values, but there is no Pine Script function to draw such bars on the chart. Every column of X’s or O’s is represented with four numbers. You may think of them as synthetic OHLC PnF values:

//@version=6 indicator("", "", true) pnfTicker = ticker.pointfigure(syminfo.tickerid, "hl", "ATR", 14, 3) [pnfO, pnfC] = request.security(pnfTicker, timeframe.period, [open, close], barmerge.gaps_on) plot(pnfO, "PnF Open", color.green, 4, plot.style_linebr) plot(pnfC, "PnF Close", color.red, 4, plot.style_linebr)

Previous

Next

Other timeframes and data

Other timeframes and data

Introduction

Pine Script® allows users to request data from sources and contexts other than those their charts use. The functions we present on this page can fetch data from a variety of alternative sources:

- request.security() retrieves data from another symbol, timeframe, or other context.
- request.security_lower_tf() retrieves *intrabar* data, i.e., data from a timeframe lower than the chart timeframe.
- request.currency_rate() requests a *daily rate* to convert a value expressed in one currency to another.
- request.dividends(), request.splits(), and request.earnings() respectively retrieve information about an issuing company's dividends, splits, and earnings.
- request.financial() retrieves financial data from FactSet.
- request.economic() retrieves economic and industry data.
- request.seed() retrieves data from a *user-maintained* GitHub repository.

These are the signatures of the functions in the `request.*` namespace:

```
request.security(symbol, timeframe, expression, gaps, lookahead, ignor

request.security_lower_tf(symbol, timeframe, expression, ignore_invali

request.currency_rate(from, to, ignore_invalid_currency) → series floa

request.dividends(ticker, field, gaps, lookahead, ignore_invalid_symbo
```

```
request.splits(ticker, field, gaps, lookahead, ignore_invalid_symbol)

request.earnings(ticker, field, gaps, lookahead, ignore_invalid_symbol

request.financial(symbol, financial_id, period, gaps, ignore_invalid_s

request.economic(country_code, field, gaps, ignore_invalid_symbol) → s

request.seed(source, symbol, expression, ignore_invalid_symbol, calc_b
```

The `request.*()` family of functions has numerous potential applications. Throughout this page, we discuss in detail these functions and some of their typical use cases.

Common characteristics

Many functions in the `request.*()` namespace share some common properties and parameters. Before we explore each function in depth, let's familiarize ourselves with these characteristics.

Behavior

All `request.*()` functions have similar internal behavior, even though they do not all share the same required parameters. Every unique `request.*()` call in a script requests a dataset from a defined *context* (i.e., ticker ID and timeframe) and evaluates an *expression* across the retrieved data.

The request.security() and request.security_lower_tf() functions allow programmers to specify the context of a request and the expression directly via the `symbol`, `timeframe`, and `expression` parameters, making them suitable for a wide range of data requests.

For example, the request.security() call in this simple script requests daily "AMEX:SPY" data, and it calculates the slope of a 20-bar linear regression line using the retrieved hl2 prices. The first two arguments specify the context of the request, and the third specifies the expression to evaluate across the requested data:

image

//@version=6 indicator("Behavior of `request.security()` demo") //@variable The 20-bar linear regression slope of `hl2` prices from the "AMEX:SPY" symbol on the "1D" timeframe. float requestedSlope = request.security("AMEX:SPY", "1D", ta.linreg(hl2, 20, 0) - ta.linreg(hl2, 20, 1)) //@variable Is `color.teal` when the `requestedSlope` is positive, and `color.maroon` otherwise. color plotColor = requestedSlope > 0 ? color.teal : color.maroon // Plot the `requestedSlope` with the `plotColor`. plot(requestedSlope, "Requested slope", plotColor, 1, plot.style_area)

Other functions within the `request.*()` namespace *do not* allow programmers to directly define the full context of a request or the evaluated expression. Instead, these functions determine some of the necessary information *internally* because they perform only specific types of requests.

For instance, request.financial() exclusively retrieves periodic financial data. Its required parameters (`symbol`, `financial_id`, and `period`) all define parts of a specific financial *ticker ID*. The function does not allow specification of the timeframe or expression, as it determines these details internally. The script below demonstrates a simple call to this function that retrieves the annual cost of goods data for the chart symbol’s issuing company:

image

//@version=6 indicator("Behavior of `request.financial()` demo", format = format.volume) //@variable The annual cost of goods sold by the chart symbol's issuing company. float costOfGoods = request.financial(syminfo.tickerid, "COST_OF_GOODS", "FY") // Plot the `costOfGoods`. plot(costOfGoods, "Cost of goods", color.purple, 3, plot.style_stepline_diamond)

Scripts can perform up to 40 unique requests using any combination of `request.*()` function calls. Only unique `request.*()` calls count toward this limit because they are the only calls that fetch *new data*. Redundant calls to the same `request.*()` function with identical arguments *do not* retrieve new data. Instead, they *reuse* the data from the first executed call. See this section of the Limitations page for more information.

gaps

When using a `request.*()` function to retrieve data from another context, the data may not come in on each new bar as it would with the current chart. The `gaps` parameter of a `request.*()` function controls how the function responds to nonexistent values in the requested series.

Suppose we have a script executing on an 1-minute chart that requests hourly data for the chart's symbol using request.security(). The function call returns new values only on the 1-minute bars that cover the opening and closing times of the symbol's hourly bars. On other chart bars, we can decide whether the function returns na values or the last available values via the `gaps` parameter.

When the `gaps` parameter uses barmerge.gaps_on, the function returns na results on all chart bars where new data is not yet confirmed from the requested context. Otherwise, when the parameter uses barmerge.gaps_off, the function fills the gaps in the requested data with the last confirmed values on historical bars and the most recent developing values on realtime bars.

The script below demonstrates the difference in behavior by plotting the results from two request.security() calls that fetch the close price of the current symbol from the hourly timeframe on a 1-minute chart. The first call uses `gaps = barmerge.gaps_off` and the second uses `gaps = barmerge.gaps_on`:

image

//@version=6 indicator("`gaps` demo", overlay = true) //@variable The `close` requested from the hourly timeframe without gaps. float dataWithoutGaps = request.security(syminfo.tickerid, "60", close, gaps = barmerge.gaps_off) //@variable The `close` requested from the hourly timeframe with gaps. float dataWithGaps = request.security(syminfo.tickerid, "60", close, gaps = barmerge.gaps_on) // Plot the requested data. plot(dataWithoutGaps, "Data without gaps", color.blue, 3, plot.style_linebr) plot(dataWithGaps, "Data with gaps", color.purple, 15, plot.style_linebr) // Highlight the background for realtime bars. bgcolor(barstate.isrealtime ? color.new(color.aqua, 70) : na, title = "Realtime bar highlight")

Note that:

- barmerge.gaps_off is the default value for the gaps parameter in all applicable request.*() functions.
- The script plots the requested series as lines with breaks (plot.style_linebr), which do not bridge over na values as the default style (plot.style_line) does.
- When using barmerge.gaps_off, the request.security() function returns the last confirmed close from the hourly timeframe on all historical bars. When running on *realtime bars* (the bars with the color.aqua background in this example), it returns the symbol's current close value, regardless of confirmation. For more information, see the Historical and realtime behavior section of this page.

ignore_invalid_symbol

The ignore_invalid_symbol parameter of request.*() functions determines how a function handles invalid data requests, e.g.:

- Using a request.*() function with a nonexistent ticker ID as the symbol/ticker parameter.
- Using request.financial() to retrieve information that does not exist for the specified symbol or period.
- Using request.economic() to request a field that does not exist for a country_code.

A request.*() function call produces a *runtime error* and halts the execution of the script when making an erroneous request if its ignore_invalid_symbol parameter is false. When this parameter's value is true, the function returns na values in such a case instead of raising an error.

This example uses request.*() calls within a user-defined function to retrieve data for estimating an instrument's market capitalization (market cap). The user-defined calcMarketCap() function calls request.financial() to retrieve the total shares outstanding for a symbol and request.security() to retrieve a tuple containing the symbol's close and syminfo.currency values. We've included ignore_invalid_symbol = true in both of these request.*() calls to prevent runtime errors for invalid requests.

The script displays a formatted string representing the symbol's estimated market cap value and currency in a table on the chart and uses a plot() call to visualize the `marketCap` history:

image

//@version=6 indicator("`ignore_invalid_symbol` demo", "Market cap estimate", format = format.volume) //@variable The symbol to request data from. string symbol = input.symbol("TSX:SHOP", "Symbol") //@function Estimates the market capitalization of the specified `tickerID` if the data exists. calcMarketCap(simple string tickerID) => //@variable The quarterly total shares outstanding for the `tickerID`. Returns `na` when the data isn't available. float tso = request.financial(tickerID, "TOTAL_SHARES_OUTSTANDING", "FQ", ignore_invalid_symbol = true) // @variable The `close` price and currency for the `tickerID`. Returns `[na, na]` when the `tickerID` is invalid. [price, currency] = request.security(tickerID, timeframe.period, [close, syminfo.currency], ignore_invalid_symbol = true) // Return a tuple containing the market cap estimate and the quote currency. [tso * price, currency] //@variable A `table` object with a single cell that displays the `marketCap` and `quoteCurrency`. var table infoTable = table.new(position.top_right, 1, 1) // Initialize the table's cell on the first bar. if barstate.isfirst table.cell(infoTable, 0, 0, "", text_color = color.white, text_size = size.huge, bgcolor = color.teal) // Get the market cap estimate and quote currency for the `symbol`. [marketCap, quoteCurrency] = calcMarketCap(symbol) if barstate.islast //@variable The formatted text displayed inside the `infoTable`. string tableText = str.format("Market cap: \n{0} {1}", str.tostring(marketCap, format.volume), quoteCurrency) // Update the `infoTable`. table.cell_set_text(infoTable, 0, 0, tableText) // Plot the `marketCap` value. plot(marketCap, "Market cap", color.new(color.purple, 60), style = plot.style_area)

Note that:

- The `calcMarketCap()` function only returns non-na values on valid instruments with total shares outstanding data, such as the one we selected for this example. It returns na on others that do not have financial data, including forex, crypto, and derivatives.
- Not all issuing companies publish quarterly financial reports. If the issuing company of the `symbol` does not report on a quarterly basis,

change the “FQ” value in this script to the company’s minimum reporting period. See the request.financial() section for more information.

- We included format.volume in the indicator() and str.tostring() calls to specify that the y-axis of the chart pane represents volume-formatted values and the “string” representation of the `marketCap` value shows as volume-formatted text.
- For efficiency, this script creates a table and initializes its cell on the *first* chart bar, then updates the cell’s text on the *last* bar. To learn more about working with tables, see the Tables page.

`currency`

The `currency` parameter of a `request.*()` function allows users to specify the currency of the requested data. When this parameter’s value differs from the syminfo.currency of the symbol, the function converts the requested values to express them in the specified `currency`. This parameter accepts a built-in variable from the `currency.*` namespace, such as currency.JPY, or a “string” representing a valid currency code (e.g., “JPY”). By default, this parameter accepts a “series” argument that can change across executions. If dynamic requests are disabled, it accepts a value with only a “simple” or weaker qualifier.

The conversion rate between the syminfo.currency of the requested data and the specified `currency` depends on the *previous daily value* of the corresponding currency pair from the most popular exchange. If no exchange provides the rate directly, the function derives the rate using a spread symbol.

`lookahead`

The `lookahead` parameter in request.security(), request.dividends(), request.splits(), and request.earnings() specifies the lookahead behavior of the function call. Its default value is barmerge.lookahead_off.

When requesting data from a higher-timeframe (HTF) context, the `lookahead` value determines whether the `request.*()` function can return values from times *beyond* those of the historical bars it executes on. In other words, the

`lookahead` paremeter determines whether the requested data may contain *lookahead bias* on historical bars.

When requesting data from a lower-timeframe (LTF) context, the `lookahead` parameter determines whether the function requests values from the first or last *intrabar* (LTF bar) of each chart-timeframe bar.

Programmers should exercise extreme caution when using lookahead in their requests, namely when requesting data from higher timeframes. When using barmerge.lookahead_on as the `lookahead` value, ensure that it does not compromise the integrity of the script's logic by leaking *future data* into historical chart bars.

The following scenarios are cases where enabling lookahead is acceptable in a `request.*()` call:

- The `expression` in request.security() references a series with a *historical offset* (e.g., `close[1]`), which prevents the function from requesting future values that it would **not** have access to on a realtime basis.
- The specified `timeframe` in the call is the same as the chart the script executes on, i.e., timeframe.period.
- The function call requests data from an intrabar timeframe, i.e., a timeframe smaller than the timeframe.period. See this section for more information.

This example demonstrates how the `lookahead` parameter affects the behavior of higher-timeframe data requests and why enabling lookahead in request.security() without offsetting the `expression` is misleading. The script calls request.security() to get the HTF high price for the current chart's symbol in three different ways and plots the resulting series on the chart for comparison.

The first call uses barmerge.lookahead_off (default), and the others use barmerge.lookahead_on. However, the third request.security() call also *offsets* its `expression` using the history-referencing operator [] to avoid leaking future data into the past.

As we see on the chart, the plot of the series requested using barmerge.lookahead_on without an offset (fuchsia line) shows final HTF high

prices *before* they are actually available on historical bars, whereas the other two calls do not:

image

//@version=6 indicator("`lookahead` demo", overlay = true) //@variable The timeframe to request the data from. string timeframe = input.timeframe("30", "Timeframe") //@variable The requested `high` price from the current symbol on the `timeframe` without lookahead bias. // On realtime bars, it returns the current `high` of the `timeframe`. float lookaheadOff = request.security(syminfo.tickerid, timeframe, high, lookahead = barmerge.lookahead_off) //@variable The requested `high` price from the current symbol on the `timeframe` with lookahead bias. // Returns values that should NOT be accessible yet on historical bars. float lookaheadOn = request.security(syminfo.tickerid, timeframe, high, lookahead = barmerge.lookahead_on) //@variable The requested `high` price from the current symbol on the `timeframe` without lookahead bias or repainting. // Behaves the same on historical and realtime bars. float lookaheadOnOffset = request.security(syminfo.tickerid, timeframe, high[1], lookahead = barmerge.lookahead_on) // Plot the values. plot(lookaheadOff, "High, no lookahead bias", color.new(color.blue, 40), 5) plot(lookaheadOn, "High with lookahead bias", color.fuchsia, 3) plot(lookaheadOnOffset, "High, no lookahead bias or repaint", color.aqua, 3) // Highlight the background on realtime bars. bgcolor(barstate.isrealtime ? color.new(color.orange, 60) : na, title = "Realtime bar highlight")

Note that:

- The series requested using barmerge.lookahead_off has a new historical value at the *end* of each HTF period, and both series requested using barmerge.lookahead_on have new historical data at the *start* of each period.
- On realtime bars, the plot of the series without lookahead (blue) and the series with lookahead and no historical offset (fuchsia) show the *same value* (i.e., the HTF period's unconfirmed high price), as no data exists beyond those points to leak into the past. Both of these plots *repaint* their results after the user reloads the script, because the *elapsed* realtime bars from the previous run become *historical* bars in the new run.

- The series that uses lookahead and a historical offset (aqua) *does not* repaint its results, because it always uses the last *confirmed* value from the higher timeframe. See the Avoiding repainting section of this page for more information.

Dynamic requests

By default, unlike all previous Pine Script versions, all v6 script's `request.*()` functions are *dynamic*.

In contrast to non-dynamic requests, dynamic requests can:

- Access data from different data feeds using a single `request.*()` instance with "series" arguments.
- Execute within the local scopes of conditional structures, loops, and exported functions.
- Execute nested requests.

Aside from the features listed above, there are insignificant differences in the behavior of dynamic and non-dynamic requests. However, for backward compatibility, programmers can deactivate dynamic requests by specifying `dynamic_requests = false` in the indicator(), strategy(), or library() declaration statement.

"series" arguments

Scripts without dynamic requests enabled cannot use "series" arguments for most `request.*()` function parameters, which means the argument values *cannot change*. The only exception is the `expression` parameter in request.security(), request.security_lower_tf(), and request.seed(), which *always* allows "series" values.

In contrast, when a script allows dynamic requests, all `request.*()` function parameters that define parts of the ticker ID or timeframe of a request accept "series" arguments that *can change* with each script execution. In other words, with dynamic requests, it's possible for a single `request.*()` instance to fetch data from *different contexts* in different executions. Some other optional parameters, such as `ignore_invalid_symbol`, can also accept

“series” arguments, allowing additional flexibility in `request.*()` call behaviors.

The following script declares a `symbolSeries` variable that is assigned four different symbol strings in 20-bar cycles, with its value changing after every five bars. The request.security() call uses this variable as the `symbol` argument. The script plots the `requestedClose` values, which therefore represent a different symbol’s close prices for each five-bar period.

image

//@version=6 indicator("'series' arguments demo") //@variable A "series" that cycles through four different symbol strings. Its value changes every five bars. string symbolSeries = switch int(bar_index / 5) % 4 1 => "NASDAQ:MSFT" 2 => "NASDAQ:AMD" 3 => "NASDAQ:INTC" => "AMEX:SPY" //@variable The requested `close` value from one of the four `symbolSeries` values on the chart's timeframe. float requestedClose = request.security(symbolSeries, timeframe.period, close) // Plot the `requestedClose`. plot(requestedClose, "Requested close", color.purple, 3) // Draw a label displaying the requested symbol each time the `symbolSeries` changes. if symbolSeries != symbolSeries[1] label.new(bar_index, requestedClose, symbolSeries, textcolor = color.white)

Note that:

- The script draws a label every time the `symbolSeries` changes, to signify which symbol’s data the `requestedClose` currently represents.
- Pine v6 scripts enable dynamic requests by default, allowing this script to use a “series string” `symbol` argument in its request.security() call without error. If the dynamic behavior is disabled by including `dynamic_requests = false` in the indicator() declaration, then the “series” argument causes a compilation error.

An important limitation is that when using dynamic `request.*()` calls with “series” arguments or within local scopes, scripts must request all required datasets while executing on **historical bars**. All `request.*()` calls on *realtime* bars can retrieve data from the datasets that the script previously accessed on historical bars, but they **cannot** request a new context or evaluate a new expression.

To illustrate this limitation, let's revisit the above script. Notice that it requests close data for all four symbols on the chart's timeframe during its historical executions. The external datasets for those four contexts are the **only** ones that any `request.*()` call on realtime bars can access.

Below, we changed the `timeframe` argument in the script's request.security() call to specify that it requests `symbolSeries` data from the chart's timeframe on historical bars and the "240" (240 minutes = 4H) timeframe on realtime bars. This version raises a runtime error on the first realtime tick, if it is run on any timeframe other than the 4H timeframe, because it **cannot** access the 4H data feeds without requesting them on historical bars first:

//@version=6 indicator("Invalid realtime request demo") //@variable A "series" that cycles through four different symbol strings. Its value changes every five bars. string symbolSeries = switch int(bar_index / 5) % 4 1 => "NASDAQ:MSFT" 2 => "NASDAQ:AMD" 3 => "NASDAQ:INTC" => "AMEX:SPY" // Request the `close` of the `symbolSeries` from the chart's timeframe on historical bars and the "240" (4H) timeframe // on realtime bars. Causes a runtime error on the first realtime tick because the script did not previously access // data from the "240" timeframe on any historical bars. float requestedClose = request.security(symbolSeries, barstate.isrealtime ? "240" : timeframe.period, close) // Plot the `requestedClose`. plot(requestedClose, "Requested close", color.purple, 3) // Draw a label displaying the requested symbol each time the `symbolSeries` changes. if symbolSeries != symbolSeries[1] label.new(bar_index, requestedClose, symbolSeries, textcolor = color.white)

In local scopes

When scripts do not allow dynamic requests, all `request.*()` calls execute once on *every* bar or realtime tick, which prevents their use within most local scopes. The only exception is for `request.*()` calls in the scopes of *non-exported* functions and methods, because the Pine Script compiler extracts such calls into the *global scope* during translation.

Scripts that allow dynamic requests *do not* restrict the execution of `request.*()` calls to the global scope. They can call `request.*()` functions directly within the scopes of conditional structures and loops, meaning that

each `request.*()` instance in the code can activate zero, one, or several times on each script execution.

The following example uses a single request.security() instance within a loop to request data from multiple forex data feeds. The script declares an array of `symbols` on the first chart bar, which it iterates through on all bars using a for...in loop. Each loop iteration calls request.security() to retrieve the volume value for one of the symbols and pushes the result into the `requestedData` array. After the loop terminates, the script calculates the average, maximum, and minimum values from the `requestedData` array using built-in methods, then plots the results on the chart:

image

//@version=6 indicator("In local scopes demo", format = format.volume) // @variable An array of "string" values representing different symbols to request. var array<string> symbols = array.from("EURUSD", "USDJPY", "GBPUSD", "AUDUSD", "USDCAD", "USDCHF", "NZDUSD", "EURJPY", "GBPJPY", "EURGBP") //@variable An array containing the data retrieved for each requested symbol. array<float> requestedData = array.new<float>() // Retrieve `volume` data for each symbol in the `symbols` array and push the results into the `requestedData` array. for symbol in symbols float data = request.security("OANDA:" + symbol, timeframe.period, volume) requestedData.push(data) // Calculate the average, maximum, and minimum tick volume in the `requestedData`. float avgVolume = requestedData.avg() float maxVolume = requestedData.max() float minVolume = requestedData.min() // Plot the `avgVolume`, `maxVolume`, and `minVolume`. plot(avgVolume, "Average volume", color.gray, 3) plot(maxVolume, "Highest volume", color.teal, 3) plot(minVolume, "Lowest volume", color.maroon, 3)

Notice that the `expression` argument in the above example (volume) is *loop-invariant*, i.e., it does not change on any loop iteration. When using `request.*()` calls within a loop, all parameters defining parts of the requested *context* can accept arguments that depend on variables from the loop's header or mutable variables that change within the loop's local scope. However, a `request.*()` call's evaluated expression **cannot** depend on the values of those variables.

Here, we modified the above script to use the *second form* of the for...in loop statement, which creates a tuple containing the index and value of each element in the `symbols` array. The request.security() instance in this version uses the index (`i`) in its `expression` argument, resulting in a *compilation error*:

```
//@version=6 indicator("Loop-dependent expression demo", format = format.volume) //@variable An array of "string" values representing different symbols to request. var array<string> symbols = array.from( "EURUSD", "USDJPY", "GBPUSD", "AUDUSD", "USDCAD", "USDCHF", "NZDUSD", "EURJPY", "GBPJPY", "EURGBP" ) //@variable An array containing the data retrieved for each requested symbol. array<float> requestedData = array.new<float>() // Retrieve `volume` data for each symbol in the `symbols` array, weighted using the element index. // Causes a compilation error because the `expression` in `request.security()` cannot depend on loop variables // or mutable variables that change within the loop's scope. for [i, symbol] in symbols float data = request.security("OANDA:" + symbol, timeframe.period, volume * (10 - i)) requestedData.push(data) // Calculate the average, maximum, and minimum tick volume in the `requestedData`. float avgVolume = requestedData.avg() float maxVolume = requestedData.max() float minVolume = requestedData.min() // Plot the `avgVolume`, `maxVolume`, and `minVolume`. plot(avgVolume, "Average volume", color.gray, 3) plot(maxVolume, "Highest volume", color.teal, 3) plot(minVolume, "Lowest volume", color.maroon, 3)
```

In libraries

Libraries with dynamic requests enabled can *export* functions and methods that utilize `request.*()` calls within their local scopes, provided that the evaluated expressions **do not** depend on any exported function parameters.

For example, this simple library exports an `htfPrices()` function that requests a tuple of confirmed open, high, low, and close prices using a specified `tickerID` and `timeframe`. If we publish this library, another script can *import* the function to request higher-timeframe prices without explicitly calling request.security().

```
//@version=6 library("DynamicRequests") //@function Requests a tuple containing confirmed HTF OHLC data for a specified `tickerID` and
```

`timeframe`. //@param tickerID The ticker identifier to request data for. //@param timeframe The timeframe of the requested data. //@returns A tuple containing the last confirmed `open`, `high`, `low`, and `close` from the requested context. export htfPrices(string tickerID, string timeframe) => if timeframe.in_seconds() >= timeframe.in_seconds(timeframe) runtime.error("The `timeframe` argument of `getHTFPrices()` must be higher than the chart's timeframe.") request.security(tickerID, timeframe, [open[1], high[1], low[1], close[1]], lookahead = barmerge.lookahead_on)

Note that:

- The tuple that the request.security() call includes as the `expression` argument *does not* depend on the `htfPrices()` parameters.
- The `htfPrices()` function includes a runtime.error() call that raises a custom runtime error when the `timeframe` argument does not represent a higher timeframe than the chart's timeframe. See the higher timeframes section for more information.
- The request.security() call uses barmerge.lookahead_on and offsets each item in the tuple by one bar. This is the only recommended method to avoid repainting.

Nested requests

Scripts can use dynamic requests to execute *nested requests*, i.e., `request.*()` calls that dynamically evaluate other `request.*()` calls that their `expression` arguments depend on.

When a request.security() or request.security_lower_tf() call uses an empty string or syminfo.tickerid for its `symbol` argument, or if it uses an empty string or timeframe.period for the `timeframe` argument, the requested ticker ID or timeframe *depends* on the context where the call executes. This context is normally the ticker ID or timeframe of the chart that the script is running on. However, if such a request.security() or request.security_lower_tf() function call is evaluated by another `request.*()` call, the nested request *inherits* that `request.*()` call's ticker ID or timeframe information.

For example, the script below contains two request.security() calls and uses Pine Logs to display their results. The first call uses empty strings as its `symbol` and `timeframe` arguments, meaning that the requested context

depends on where the call executes. It evaluates a concatenated string containing the call's requested ticker ID and timeframe, and the script assigns its result to the `info1` variable.

The second call requests data for a specific `symbol` and `timeframe` using the `info1` variable as its `expression` argument. Since the `info1` variable depends on the first request.security() call, the second call evaluates the first call *within* its own context. Therefore, the first call adopts the second call's ticker ID and timeframe while executing within that context, resulting in a different returned value:

image

//@version=6 indicator("Nested requests demo") //@variable A concatenated string containing the current `syminfo.tickerid` and `timeframe.period`. string info1 = request.security("", "", syminfo.tickerid + "_" + timeframe.period) //@variable A concatenated string representing the `info1` value calculated within the "NASDAQ:AAPL, 240" context. // This call evaluates the call on line 5 within its context to determine its result because the script // allows dynamic requests. string info2 = request.security("NASDAQ:AAPL", "240", info1) // Log the results from both calls in the Pine Logs pane on the last historical bar. if barstate.islastconfirmedhistory log.info("First request: {0}", info1) log.info("Second request: {0}", info2)

This script allows the execution of the first request.security() call within the context of the second call because Pine v6 scripts enable dynamic `request.*()` calls by default. We can disable this behavior by including `dynamic_requests = false` in the indicator() declaration statement. Without dynamic requests enabled, the script evaluates each call *independently*, passing the first call's calculated value directly into the second call rather than executing the first call within the second context. Consequently, the second call's returned value is the *same* as the first call's value, as we see below:

image

//@version=6 indicator("Nested requests demo", dynamic_requests = false) //@variable A concatenated string containing the current `syminfo.tickerid` and `timeframe.period`. string info1 = request.security("", "",

syminfo.tickerid + "_" + timeframe.period) //@variable The same value as `info1`. This call does not evalutate the call on line 5 because dynamic requests aren't // allowed. Instead, it only uses the value of `info1`, meaning its result does not change. string info2 = request.security("NASDAQ:AAPL", "240", info1) // Log the results from both calls in the Pine Logs pane on the last historical bar. if barstate.islastconfirmedhistory log.info("First request: {0}", info1) log.info("Second request: {0}", info2)

Data feeds

TradingView's data providers supply different data feeds that scripts can access to retrieve information about an instrument, including:

- Intraday historical data (for timeframes < 1D)
- End-of-day (EOD) historical data (for timeframes >= 1D)
- Realtime data (which may be delayed, depending on your account type and extra data services)
- Extended hours data

Not all of these data feed types exist for every instrument. For example, the symbol "BNC:BLX" only has EOD data available.

For some instruments with intraday and EOD historical feeds, volume data may not be the same since some trades (block trades, OTC trades, etc.) may only be available at the *end* of the trading day. Consequently, the EOD feed will include this volume data, but the intraday feed will not. Differences between EOD and intraday volume feeds are almost nonexistent for instruments such as cryptocurrencies, but they are commonplace in stocks.

Slight price discrepancies may also occur between EOD and intraday feeds. For example, the high value on one EOD bar may not match any intraday high values supplied by the data provider for that day.

Another distinction between EOD and intraday data feeds is that EOD feeds do not contain information from *extended hours*.

When retrieving information on realtime bars with `request.*()` functions, it's important to note that historical and realtime data reported for an instrument often rely on *different* data feeds. A broker/exchange may retroactively

modify values reported on realtime bars, which the data will only reflect after refreshing the chart or restarting the script.

Another important consideration is that the chart's data feeds and feeds requested from providers by the script are managed by *independent*, concurrent processes. Consequently, in some *rare* cases, it's possible for races to occur where requested results temporarily fall out of synch with the chart on a realtime bar, which a script retroactively adjusts after restarting its executions.

These points may account for variations in the values retrieved by `request.*()` functions when requesting data from other contexts. They may also result in discrepancies between data received on realtime bars and historical bars. There are no steadfast rules about the variations one may encounter in their requested data feeds.

When using data feeds requested from other contexts, it's also crucial to consider the *time axis* differences between the chart the script executes on and the requested feeds since `request.*()` functions adapt the returned series to the chart's time axis. For example, requesting "BTCUSD" data on the "SPY" chart with request.security() will only show new values when the "SPY" chart has new data as well. Since "SPY" is not a 24-hour symbol, the "BTCUSD" data returned will contain gaps that are otherwise not present when viewing its chart directly.

request.security()

The request.security() function allows scripts to request data from other contexts than the chart the script executes on, such as:

- Other symbols, including spread symbols
- Other timeframes (see our User Manual's page on Timeframes to learn about timeframe specifications in Pine Script)
- Custom contexts, including alternative sessions, price adjustments, chart types, etc. using `ticker.*()` functions

This is the function's signature:

```
request.security(symbol, timeframe, expression, gaps, lookahead, ignor
```

The `symbol` value is the ticker identifier representing the symbol to fetch data from. This parameter accepts values in any of the following formats:

- A “string” representing a symbol (e.g., “IBM” or “EURUSD”) or an *“Exchange:Symbol” pair* (e.g., “NYSE:IBM” or “OANDA:EURUSD”). When the value does not contain an exchange prefix, the function selects the exchange automatically. We recommend specifying the exchange prefix when possible for consistent results. Users can also pass an empty string to this parameter, which prompts the function to use the current chart’s symbol.
- A “string” representing a spread symbol (e.g., “AMD/INTC”). Note that “Bar Replay” mode does not work with these symbols.
- The syminfo.ticker or syminfo.tickerid built-in variables, which return the symbol or the “Exchange:Symbol” pair that the current chart references. We recommend using syminfo.tickerid to avoid ambiguity unless the exchange information does not matter in the data request. For more information on `syminfo.*` variables, see this section of our Chart information page.
- A custom ticker identifier created using `ticker.*()` functions. Ticker IDs constructed from these functions may contain additional settings for requesting data using non-standard chart calculations, alternative sessions, and other contexts. See the Custom contexts section for more information.

The `timeframe` value specifies the timeframe of the requested data. This parameter accepts “string” values in our timeframe specification format (e.g., a value of “1D” represents the daily timeframe). To request data from the same timeframe as the chart the script executes on, use the timeframe.period variable or an empty string.

The `expression` parameter of the request.security() function determines the data it retrieves from the specified context. This versatile parameter accepts “series” values of int, float, bool, color, string, and chart.point types. It can also accept tuples, collections, user-defined types, and the outputs of function and method calls. For more details on the data one can retrieve, see the Requestable data section below.

Timeframes

The request.security() function can request data from any available timeframe, regardless of the chart the script executes on. The timeframe of the data retrieved depends on the `timeframe` argument in the function call, which may represent a higher timeframe (e.g., using “1D” as the `timeframe` value while running the script on an intraday chart) or the chart’s timeframe (i.e., using timeframe.period or an empty string as the `timeframe` argument).

Scripts can also request *limited* data from lower timeframes with request.security() (e.g., using “1” as the `timeframe` argument while running the script on a 60-minute chart). However, we don’t typically recommend using this function for LTF data requests. The request.security_lower_tf() function is more optimal for such cases.

Higher timeframes

Most use cases of request.security() involve requesting data from a timeframe higher than or the same as the chart timeframe. For example, this script retrieves the hl2 price from a requested `higherTimeframe`. It plots the resulting series on the chart alongside the current chart’s hl2 for comparison:

image

//@version=6 indicator("Higher timeframe security demo", overlay = true) // @variable The higher timeframe to request data from. string higherTimeframe = input.timeframe("240", "Higher timeframe") //@variable The `hl2` value from the `higherTimeframe`. Combines lookahead with an offset to avoid repainting. float htfPrice = request.security(syminfo.tickerid, higherTimeframe, hl2[1], lookahead = barmerge.lookahead_on) // Plot the `hl2` from the chart timeframe and the `higherTimeframe`. plot(hl2, "Current timeframe HL2", color.teal, 2) plot(htfPrice, "Higher timeframe HL2", color.purple, 3)

Note that:

- We’ve included an offset to the `expression` argument and used barmerge.lookahead_on in request.security() to ensure the series

returned behaves the same on historical and realtime bars. See the Avoiding repainting section for more information.

Notice that in the above example, it is possible to select a `higherTimeframe` value that actually represents a *lower timeframe* than the one the chart uses, as the code does not prevent it. When designing a script to work specifically with higher timeframes, we recommend including conditions to prevent it from accessing lower timeframes, especially if you intend to publish it.

Below, we've added an if structure to our previous example. If the `higherTimeframe` value represents a timeframe that is smaller than the chart's timeframe, the script calls runtime.error() within the structure's local block to raise a custom runtime error, effectively preventing the script from requesting LTF data:

image

//@version=6 indicator("Higher timeframe security demo", overlay = true) // @variable The higher timeframe to request data from. string higherTimeframe = input.timeframe("240", "Higher timeframe") // Raise a runtime error when the `higherTimeframe` is smaller than the chart's timeframe. if timeframe.in_seconds() > timeframe.in_seconds(higherTimeframe) runtime.error("The requested timeframe is smaller than the chart's timeframe. Select a higher timeframe.") //@variable The `hl2` value from the `higherTimeframe`. Combines lookahead with an offset to avoid repainting. float htfPrice = request.security(syminfo.tickerid, higherTimeframe, hl2[1], lookahead = barmerge.lookahead_on) // Plot the `hl2` from the chart timeframe and the `higherTimeframe`. plot(hl2, "Current timeframe HL2", color.teal, 2) plot(htfPrice, "Higher timeframe HL2", color.purple, 3)

Lower timeframes

Although the request.security() function is intended to operate on timeframes greater than or equal to the chart timeframe, it *can* request data from lower timeframes as well, with limitations. When calling this function to access a lower timeframe, it will evaluate the `expression` from the LTF context. However, it can only return the results from a *single* intrabar (LTF bar) on each chart bar.

The intrabar that the function returns data from on each historical chart bar depends on the `lookahead` value in the function call. When using barmerge.lookahead_on, it will return the *first* available intrabar from the chart period. When using barmerge.lookahead_off, it will return the *last* intrabar from the chart period. On realtime bars, it returns the last available value of the `expression` from the timeframe, regardless of the `lookahead` value, as the realtime intrabar information retrieved by the function is not yet sorted.

This script retrieves close data from the valid timeframe closest to a fourth of the size of the chart timeframe. It makes two calls to request.security() with different `lookahead` values. The first call uses barmerge.lookahead_on to access the first intrabar value in each chart bar. The second uses the default `lookahead` value (barmerge.lookahead_off), which requests the last intrabar value assigned to each chart bar. The script plots the outputs of both calls on the chart to compare the difference:

image

//@version=6 indicator("Lower timeframe security demo", overlay = true) // @variable The valid timeframe closest to 1/4 the size of the chart timeframe. string lowerTimeframe = timeframe.from_seconds(int(timeframe.in_seconds() / 4)) //@variable The `close` value on the `lowerTimeframe`. Represents the first intrabar value on each chart bar. float firstLTFClose = request.security(syminfo.tickerid, lowerTimeframe, close, lookahead = barmerge.lookahead_on) //@variable The `close` value on the `lowerTimeframe`. Represents the last intrabar value on each chart bar. float lastLTFClose = request.security(syminfo.tickerid, lowerTimeframe, close) // Plot the values. plot(firstLTFClose, "First intrabar close", color.teal, 3) plot(lastLTFClose, "Last intrabar close", color.purple, 3) // Highlight the background on realtime bars. bgcolor(barstate.isrealtime ? color.new(color.orange, 70) : na, title = "Realtime background highlight")

Note that:

- The script determines the value of the `lowerTimeframe` by calculating the number of seconds in the chart timeframe with

timeframe.in_seconds(), then dividing by four and converting the result to a valid timeframe string via timeframe.from_seconds().
- The plot of the series without lookahead (purple) aligns with the close value on the chart timeframe, as this is the last intrabar value in the chart bar.
- Both request.security() calls return the *same* value (the current close) on each *realtime* bar, as shown on the bars with the orange background.
- Scripts can retrieve up to 200,000 intrabars from a lower-timeframe context. The number of chart bars with available intrabar data varies with the requested lower timeframe, the `calc_bars_count` value, and the user's plan. For more information, see this section of the Limitations page.

Requestable data

The request.security() function is quite versatile, as it can retrieve values of any fundamental type (int, float, bool, color, or string). It can also request the IDs of data structures and built-in or user-defined types that reference fundamental types. The data this function requests depends on its `expression` parameter, which accepts any of the following arguments:

- Built-in variables and function calls
- Variables declared by the script
- Tuples
- Calls to user-defined functions
- Chart points
- Collections
- User-defined types

Built-in variables and functions

A frequent use case of request.security() is requesting the output of a built-in variable or function/method call from another symbol or timeframe.

For example, suppose we want to calculate the 20-bar SMA of a symbol's ohlc4 prices from the daily timeframe while on an intraday chart. We can accomplish this task with a single line of code:

```
float ma = request.security(syminfo.tickerid, "1D", ta.sma(ohlc4, 20))
```

The above line calculates the value of `ta.sma(ohlc4, 20)` on the current symbol's data from the daily timeframe.

It's important to note that newcomers to Pine might sometimes confuse the above line of code as being equivalent to the following:

float ma = ta.sma(request.security(syminfo.tickerid, "1D", ohlc4), 20)

However, this line returns an entirely *different* result. Rather than requesting a 20-bar SMA from the daily timeframe, it requests the ohlc4 price from the daily timeframe and calclates the ta.sma() of the results over 20 **chart bars**.

In essence, when the intention is to request the results of an expression from other contexts, pass the expression *directly* to the `expression` parameter in the request.security() call, as demonstrated in the initial example.

Let's expand on this concept. The script below calculates a multi-timeframe (MTF) ribbon of moving averages, where each moving average in the ribbon calculates over the same number of bars on its respective timeframe. Each request.security() call uses a ta.sma() call as its `expression` argument to return a `length`-bar SMA from the specified timeframe:

image

//@version=6 indicator("Requesting built-ins demo", "MTF Ribbon", true) // @variable The length of each moving average. int length = input.int(20, "Length", 1) //@variable The number of seconds in the chart timeframe. int chartSeconds = timeframe.in_seconds() // Calculate the higher timeframes closest to 2, 3, and 4 times the size of the chart timeframe. string htf1 = timeframe.from_seconds(chartSeconds * 2) string htf2 = timeframe.from_seconds(chartSeconds * 3) string htf3 = timeframe.from_seconds(chartSeconds * 4) // Calculate the `length`-bar moving averages from each timeframe. float chartAvg = ta.sma(ohlc4, length) float htfAvg1 = request.security(syminfo.tickerid, htf1, ta.sma(ohlc4, length)) float htfAvg2 = request.security(syminfo.tickerid, htf2, ta.sma(ohlc4, length)) float htfAvg3 = request.security(syminfo.tickerid, htf3, ta.sma(ohlc4, length)) // Plot the results. plot(chartAvg, "Chart timeframe SMA", color.red, 3) plot(htfAvg1, "Double timeframe SMA", color.orange, 3) plot(htfAvg2, "Triple timeframe SMA", color.green, 3) plot(htfAvg3, "Quadruple timeframe SMA", color.blue, 3) // Highlight the background on realtime bars.

bgcolor(barstate.isrealtime ? color.new(color.aqua, 70) : na, title = "Realtime highlight")

Note that:

- The script calculates the ribbon's higher timeframes by multiplying the chart's timeframe.in_seconds() value by 2, 3, and 4, then converting each result into a valid timeframe string using timeframe.from_seconds().
- Instead of calling ta.sma() within each request.security() call, one could use the `chartAvg` variable as the `expression` in each call to achieve the same result. See the next section for more information.
- On realtime bars, this script also tracks *unconfirmed* SMA values from each higher timeframe. See the Historical and realtime behavior section to learn more.

Declared variables

The request.security() function's `expression` parameter can accept declared variables that are accessible to the scope from which the function call executes. When using a declared variable as the `expression` argument, the function call *duplicates* all *preceding code* that determines the assigned value or reference. This duplication allows the function to evaluate necessary calculations and logic in the requested context without affecting the original variable.

For instance, this line of code declares a `priceReturn` variable that holds the current bar's arithmetic price return:

float priceReturn = (close - close[1]) / close[1]

We can evaluate the `priceReturn` variable's calculations in another context by using it as the `expression` in a request.security() call. The call below duplicates the variable's calculation and evaluates it across the data from another `symbol`, returning a *separate series* adapted to the chart's time axis:

float requestedReturn = request.security(symbol, timeframe.period, priceReturn)

This example script compares the price returns of the current chart's symbol and a user-specified symbol. It calculates the value of the `priceReturn` variable, then uses that variable as the `expression` in a request.security() call to evaluate the calculation on the input symbol's data. After the request, the script calculates the correlation between the `priceReturn` and `requestedReturn` series using ta.correlation() and plots the result on the chart:

image

//@version=6 indicator("Requesting calculated variables demo", "Price return correlation") //@variable The symbol to compare to the chart symbol. string symbol = input.symbol("SPY", "Symbol to compare") //@variable The number of bars in the calculation window. int length = input.int(60, "Length", 1) // @variable The close-to-close price return. float priceReturn = (close - close[1]) / close[1] //@variable The close-to-close price return calculated on another `symbol`. float requestedReturn = request.security(symbol, timeframe.period, priceReturn) //@variable The correlation between the `priceReturn` and `requestedReturn` over `length` bars. float correlation = ta.correlation(priceReturn, requestedReturn, length) //@variable The color of the correlation plot. color plotColor = color.from_gradient(correlation, -1, 1, color.purple, color.orange) // Plot the correlation value. plot(correlation, "Correlation", plotColor, style = plot.style_area)

Note that:

- The request.security() call executes the same calculation used in the `priceReturn` declaration, but the request's calculation operates on the close values from the specified symbol's data.
- The script uses the color.from_gradient() function to calculate the color for the plot of the `correlation` series on each bar. See this section of the Colors page to learn more about color gradients.

When using a variable as the `expression` argument of a `request.*()` call, it's important to note that the function only duplicates code that affects the variable *before* the call. It *cannot* copy any subsequent code following the call. Consequently, if the script reassigns the variable or modifies its referenced data *after* calling request.security(), the code evaluated on the requested data **does not** include those additional operations.

For example, the following script declares a `counter` variable and calls request.security() to evaluate the variable from the same context as the chart. After the call, the script increments the `counter` value by one with the addition assignment operator (+=), then uses plots and Pine Logs to display the `counter` and `requestedCounter` values for comparison.

As shown below, the plots and logs of the two variables display *different* values. The `requestedCounter` variable has a consistent value of 0 because the request.security() call evaluates only the initial variable declaration. The request cannot evaluate the addition assignment operation because the script includes that code *after* the function call:

image

//@version=6 indicator("Modifying variables after requests demo") // @variable A counter that starts at 0 and increments by 1 on each bar. var int counter = 0 //@variable Holds a consistent value of 0. // `request.security()` cannot evaluate `counter += 1` in its requested context // because that modification occurs *after* the call. int requestedCounter = request.security(syminfo.tickerid, timeframe.period, counter) // Increment the `counter` by 1. This operation is *not* included in the `requestedCounter` calculation. counter += 1 // Plot both variables for comparison. plot(counter, "Original counter", color.purple, 3) plot(requestedCounter, "Requested counter", color.red, 3) // Log the values of both variables in the Pine Logs pane. if barstate.isconfirmed log.info("counter: {0}, requestedCounter: {1}", counter, requestedCounter)

Tuples

Tuples in Pine Script are comma-separated sets of expressions enclosed in brackets that can hold multiple values of any available type. We use tuples when creating functions or other local blocks that return more than one value.

The request.security() function can accept a tuple as its `expression` argument, allowing scripts to request multiple series of different types using a single function call. The expressions within requested tuples can be of any type outlined throughout the Requestable data section of this page, excluding other tuples.

Tuples are particularly handy when a script needs to retrieve more than one value from a specific context.

For example, the following script calculates the percent rank of the close series over `length` bars and assigns the result to the `rank` variable. It then calls request.security() to request a tuple containing the values of `rank`, `ta.crossover(rank, 50)`, and `ta.crossunder(rank, 50)` from the specified `timeframe`. The script plots the `requestedRank` series in a separate pane, then uses the result of a ternary expression based on the `crossOver` and `crossUnder` values within a bgcolor() call to conditionally highlight the pane's background:

image

//@version=6 indicator("Requesting tuples demo", "Percent rank cross") // @variable The timeframe of the request. string timeframe = input.timeframe("240", "Timeframe") //@variable The number of bars in the calculation. int length = input.int(20, "Length") //@variable The previous bar's percent rank of the `close` price over `length` bars. float rank = ta.percentrank(close, length)[1] // Request the `rank` value from another `timeframe`, and two "bool" values indicating the `rank` from the `timeframe` // crossed over or under 50. [requestedRank, crossOver, crossUnder] = request.security(syminfo.tickerid, timeframe, [rank, ta.crossover(rank, 50), ta.crossunder(rank, 50)], lookahead = barmerge.lookahead_on) // Plot the `requestedRank` and create a horizontal line at 50. plot(requestedRank, "Percent Rank", linewidth = 3) hline(50, "Cross line", linewidth = 2) // Highlight the background of all bars where the `timeframe`'s `crossOver` or `crossUnder` value is `true`. bgcolor(crossOver ? color.new(color.green, 50) : crossUnder ? color.new(color.red, 50) : na)

Note that:

- We've offset the `rank` variable's expression by one bar using the history-referencing operator [] and included barmerge.lookahead_on in the request.security() call to ensure the values on realtime bars do not repaint after becoming historical bars. See the Avoiding repainting section for more information.

- The request.security() call returns a tuple, so we use a *tuple declaration* to declare the `requestedRank`, `crossOver`, and `crossUnder` variables. To learn more about using tuples, see this section of our User Manual's Type system page.

User-defined functions

User-defined functions and methods are custom functions written by users. They allow users to define sequences of operations associated with an identifier that scripts can conveniently call throughout their executions (e.g., `myUDF()`).

The request.security() function can request the results of user-defined functions and methods whose scopes consist of any types outlined throughout this page's Requestable data section.

For example, this script contains a user-defined `weightedBB()` function that calculates Bollinger Bands with the basis average weighted by a specified `weight` series. The function returns a tuple of custom band values. The script calls the `weightedBB()` as the `expression` argument in request.security() to retrieve a tuple of band values calculated on the specified `timeframe` and plots the results on the chart:

image

//@version=6 indicator("Requesting user-defined functions demo", "Weighted Bollinger Bands", true) //@variable The timeframe of the request. string timeframe = input.timeframe("480", "Timeframe") //@function Calculates Bollinger Bands with a custom weighted basis. //@param source The series of values to process. //@param length The number of bars in the calculation. //@param mult The standard deviation multiplier. //@param weight The series of weights corresponding to each `source` value. //@returns A tuple containing the basis, upper band, and lower band respectively. weightedBB(float source, int length, float mult = 2.0, float weight = 1.0) => //@variable The basis of the bands. float ma = math.sum(source * weight, length) / math.sum(weight, length) //@variable The standard deviation from the `ma`. float dev = 0.0 // Loop to accumulate squared error. for i = 0 to length - 1 difference = source[i] - ma dev += difference * difference // Divide `dev` by the `length`, take the square root, and multiply

by the `mult`. dev := math.sqrt(dev / length) * mult // Return the bands. [ma, ma + dev, ma - dev] // Request weighted bands calculated on the chart symbol's prices over 20 bars from the // last confirmed bar on the `timeframe`. [basis, highBand, lowBand] = request.security(syminfo.tickerid, timeframe, weightedBB(close[1], 20, 2.0, (high - low)[1]), lookahead = barmerge.lookahead_on) // Plot the values. basisPlot = plot(basis, "Basis", color.orange, 2) upperPlot = plot(highBand, "Upper", color.teal, 2) lowerPlot = plot(lowBand, "Lower", color.maroon, 2) fill(upperPlot, lowerPlot, color.new(color.gray, 90), "Background")

Note that:

- We offset the `source` and `weight` arguments in the `weightedBB()` call used as the `expression` in request.security() and used barmerge.lookahead_on to ensure the requested results reflect the last confirmed values from the `timeframe` on realtime bars. See this section to learn more.

Chart points

Chart points are reference types that represent coordinates on the chart. Lines, boxes, polylines, and labels use chart.point objects to set their display locations.

The request.security() function can use the ID of a chart.point instance in its `expression` argument, allowing scripts to retrieve chart coordinates from other contexts.

The example below requests a tuple of historical chart points from a higher timeframe and uses them to draw boxes on the chart. The script declares the `topLeft` and `bottomRight` variables that reference chart.point IDs from the last confirmed bar. It then uses request.security() to request a tuple containing the IDs of chart points representing the `topLeft` and `bottomRight` from a `higherTimeframe`.

When a new bar starts on the `higherTimeframe`, the script draws a new box using the `time` and `price` coordinates from the `requestedTopLeft` and `requestedBottomRight` chart points:

image

```
//@version=6 indicator("Requesting chart points demo", "HTF Boxes", true, max_boxes_count = 500) //@variable The timeframe to request data from. string higherTimeframe = input.timeframe("1D", "Timeframe") // Raise a runtime error if the `higherTimeframe` is smaller than the chart's timeframe. if timeframe.in_seconds(higherTimeframe) < timeframe.in_seconds(timeframe.period) runtime.error("The selected timeframe is too small. Choose a higher timeframe.") //@variable A `chart.point` containing top-left coordinates from the last confirmed bar. topLeft = chart.point.now(high)[1] //@variable A `chart.point` containing bottom-right coordinates from the last confirmed bar. bottomRight = chart.point.from_time(time_close, low)[1] // Request the last confirmed `topLeft` and `bottomRight` chart points from the `higherTimeframe`. [requestedTopLeft, requestedBottomRight] = request.security( syminfo.tickerid, higherTimeframe, [topLeft, bottomRight], lookahead = barmerge.lookahead_on ) // Draw a new box when a new `higherTimeframe` bar starts. // The box uses the `time` fields from the `requestedTopLeft` and `requestedBottomRight` as x-coordinates. if timeframe.change(higherTimeframe) box.new( requestedTopLeft, requestedBottomRight, color.purple, 3, xloc = xloc.bar_time, bgcolor = color.new(color.purple, 90) )
```

Note that:

- Because we designed this example to request data from higher timeframes, we've included a runtime.error() call that the script executes if the `higherTimeframe` value represents a lower timeframe than timeframe.period.

Collections

Pine Script *collections* (arrays, matrices, and maps) are data structures that contain an arbitrary number of elements with specified types. The request.security() function can retrieve the IDs of collections whose elements consist of:

- Fundamental types
- Chart points
- User-defined types that satisfy the criteria listed in the section below

This example below calculates the ratio of a confirmed bar's high-low range to the range between the highest and lowest prices over 10 bars from a from a specified `symbol` and `timeframe`. It uses maps to hold the values used in the calculations.

The script uses a `data` map with "string" keys and "float" values to store the current bar's high, low, ta.highest(), and ta.lowest() results. It passes the map as the `expression` argument in a request.security() call on each bar to retrieve another map containing the values calculated from the specified context, then assigns that map's reference to the `otherData` variable. The script uses the "float" values associated with the "High", "Low", "Highest", and "Lowest" keys of the `otherData` map to calculate the `ratio` series that it plots in the chart pane:

image

//@version=6 indicator("Requesting collections demo", "Bar range ratio") // @variable The ticker ID to request data from. string symbol = input.symbol("", "Symbol") //@variable The timeframe of the request. string timeframe = input.timeframe("30", "Timeframe") //@variable A map with "string" keys and "float" values. var map<string, float> data = map.new<string, float>() // Put key-value pairs into the `data` map. map.put(data, "High", high) map.put(data, "Low", low) map.put(data, "Highest", ta.highest(10)) map.put(data, "Lowest", ta.lowest(10)) //@variable A new `map` whose data is calculated from the last confirmed bar of the requested context. map<string, float> otherData = request.security(symbol, timeframe, data[1], lookahead = barmerge.lookahead_on) //@variable The ratio of the context's bar range to the max range over 10 bars. Returns `na` if no data is available. float ratio = na if not na(otherData) ratio := (otherData.get("High") - otherData.get("Low")) / (otherData.get("Highest") - otherData.get("Lowest")) //@variable A gradient color for the plot of the `ratio`. color ratioColor = color.from_gradient(ratio, 0, 1, color.purple, color.orange) // Plot the `ratio`. plot(ratio, "Range Ratio", ratioColor, 3, plot.style_area)

Note that:

- The request.security() call in this script can return na if no data is available from the specified context. Since one cannot call methods on a

map variable when its value is na, we've added an if structure to only calculate a new `ratio` value when `otherData` references a valid map instance.

User-defined types

User-defined types (UDTs) are *composite types* containing an arbitrary number of *fields*, which can be of any available type, including other user-defined types.

The request.security() function can retrieve the IDs of objects produced by UDTs from other contexts if their fields consist of:

- Fundamental types
- Chart points
- Collections that satisfy the criteria listed in the section above
- Other UDTs whose fields consist of any of these types

The following example requests an object ID using a specified `symbol` and displays its field values on a chart pane.

The script contains a `TickerInfo` UDT with "string" fields for `syminfo.*` values, an array field to store recent "float" price data, and an "int" field to hold the requested ticker's bar_index value. It assigns a new `TickerInfo` ID to an `info` variable on every bar and uses the variable as the `expression` in request.security() to retrieve the ID of an object representing the calculated `info` from the specified `symbol`.

The script displays the `requestedInfo` object's `description`, `tickerType`, `currency`, and `barIndex` values in a label and uses plotcandle() to display the values from its `prices` array:

image

//@version=6 indicator("Requesting user-defined types demo", "Ticker info") //@variable The symbol to request information from. string symbol = input.symbol("NASDAQ:AAPL", "Symbol") //@type A custom type containing information about a ticker. //@field description The symbol's description. //@field tickerType The type of ticker. //@field currency The symbol's currency. //@field prices An array of the symbol's current prices. //@field

barIndex The ticker's `bar_index`. type TickerInfo string description string tickerType string currency array<float> prices int barIndex //@variable A `TickerInfo` object containing current data. info = TickerInfo.new(syminfo.description, syminfo.type, syminfo.currency, array.from(open, high, low, close), bar_index) //@variable The `info` requested from the specified `symbol`. TickerInfo requestedInfo = request.security(symbol, timeframe.period, info) // Assign a new `TickerInfo` instance to `requestedInfo` if one wasn't retrieved. if na(requestedInfo) requestedInfo := TickerInfo.new(prices = array.new<float>(4)) //@variable A label displaying information from the `requestedInfo` object. var infoLabel = label.new(na, na, "", color = color.purple, style = label.style_label_left, textcolor = color.white, size = size.large) //@variable The text to display inside the `infoLabel`. string infoText = na(requestedInfo) ? "" : str.format("{0}\nType: {1}\nCurrency: {2}\nBar Index: {3}", requestedInfo.description, requestedInfo.tickerType, requestedInfo.currency, requestedInfo.barIndex) // Set the `point` and `text` of the `infoLabel`. label.set_point(infoLabel, chart.point.now(array.last(requestedInfo.prices))) label.set_text(infoLabel, infoText) // Plot candles using the values from the `prices` array of the `requestedInfo`. plotcandle(requestedInfo.prices.get(0), requestedInfo.prices.get(1), requestedInfo.prices.get(2), requestedInfo.prices.get(3), "Requested Prices")

Note that:

- The `syminfo.*` variables used in this script all return “simple string” qualified types. However, objects in Pine are *always* qualified as “series”. Consequently, all values assigned to the `info` object’s fields automatically adopt the “series” qualifier.
- It is possible for the request.security() call to return na due to differences between the data requested from the `symbol` and the main chart. This script assigns a new `TickerInfo` object to the `requestedInfo` in that case to prevent runtime errors.

request.security_lower_tf()

The request.security_lower_tf() function is an alternative to request.security() designed for reliably requesting information from lower-timeframe (LTF) contexts.

While request.security() can retrieve data from a *single* intrabar (LTF bar) in each chart bar, request.security_lower_tf() retrieves data from *all* available intrabars in each chart bar, which the script can access and use in additional calculations. Each request.security_lower_tf() call can retrieve up to 200,000 intrabars from a lower timeframe, depending on the user's plan. See this section of our Limitations page for more information.

Below is the function's signature, which is similar to the signature of request.security():

```
request.security_lower_tf(symbol, timeframe, expression, ignore_invali
```

This function requests data only from timeframes that are *lower than* or *equal to* the chart's timeframe (timeframe.period). If the `timeframe` argument of the request.security_lower_tf() call represents a higher timeframe, the function raises a runtime error or returns na results, depending on the `ignore_invalid_timeframe` parameter. The parameter's default value is `false`, meaning the function raises an error and halts the script's executions if the `timeframe` argument is invalid.

Requesting intrabar data

Intrabar data can provide a script with additional information that may not be obvious or accessible from solely analyzing data sampled on the chart's timerframe. The request.security_lower_tf() function can retrieve many data types from an intrabar context.

Before you venture further in this section, we recommend exploring the Requestable data portion of the request.security() section above, which provides foundational information about the types of data one can request. The `expression` parameter in request.security_lower_tf() accepts most of the same arguments discussed in that section, excluding direct references to collections and mutable variables. Although it accepts many of the same types of arguments, this function returns array results, which comes with some differences in interpretation and handling, as explained below.

Intrabar data arrays

Lower timeframes contain more data points than higher timeframes, as new values come in at a *higher frequency*. For example, when comparing a 1-minute chart to an hourly chart, the 1-minute chart will have up to 60 times the number of bars per hour, depending on the available data.

To address the fact that multiple intrabars exist within a chart bar, request.security_lower_tf() always returns its results as arrays. The elements in the returned arrays represent the `expression` values retrieved from the lower timeframe sorted in ascending order based on each intrabar's timestamp.

The type template assigned to the returned arrays corresponds to the data types passed in the request.security_lower_tf() call. For example, using an "int" as the `expression` will produce an `array<int>` instance, a "bool" as the `expression` will produce an `array<bool>` instance, etc.

The following script uses intrabar information to decompose the chart's close-to-close price changes into positive and negative parts. It calls request.security_lower_tf() to fetch a "float" array containing `ta.change(close)` values from a specified lower timeframe on each chart bar, then accesses all the array's elements using a for...in loop to accumulate `positiveChange` and `negativeChange` sums. The script adds the accumulated values to calculate the `netChange` value, then plots the results on the chart alongside the `priceChange` value for comparison:

image

//@version=6 indicator("Intrabar arrays demo", "Intrabar price changes") //@variable The lower timeframe of the requested data. string lowerTimeframe = input.timeframe("1", "Timeframe") //@variable The close-to-close price change. float priceChange = ta.change(close) //@variable An array of `close` values from available intrabars on the `lowerTimeframe`. array<float> intrabarChanges = request.security_lower_tf(syminfo.tickerid, lowerTimeframe, priceChange) //@variable The total positive intrabar `close` movement on the chart bar. float positiveChange = 0.0 //@variable The total negative intrabar `close` movement on the chart bar. float negativeChange = 0.0 // Loop to calculate totals, starting from the chart bar's first available

intrabar. for change in intrabarChanges // Add the `change` to `positiveChange` if its sign is 1, and add to `negativeChange` if its sign is -1. switch math.sign(change) 1 => positiveChange += change -1 => negativeChange += change //@variable The sum of `positiveChange` and `negativeChange`. Equals the `priceChange` on bars with available intrabars. float netChange = positiveChange + negativeChange // Plot the `positiveChange`, `negativeChange`, and `netChange`. plot(positiveChange, "Positive intrabar change", color.teal, style = plot.style_area) plot(negativeChange, "Negative intrabar change", color.maroon, style = plot.style_area) plot(netChange, "Net intrabar change", color.yellow, 5) // Plot the `priceChange` to compare. plot(priceChange, "Chart price change", color.orange, 2)

Note that:

- The plots based on intrabar data may not appear on all available chart bars, as request.security_lower_tf() can only access up to the most recent 200,000 intrabars available from the requested context. When executing this function on a chart bar that doesn't have accessible intrabar data, it will return an *empty array*.
- The number of intrabars per chart bar may vary depending on the data available from the context and the chart the script executes on. For example, a provider's 1-minute data feed may not include data for every minute within the 60-minute timeframe due to a lack of trading activity over some 1-minute intervals. To check the number of intrabars retrieved for a chart bar, one can use array.size() on the resulting array.
- If the `lowerTimeframe` value is greater than the chart's timeframe, the script will raise a *runtime error*, as we have not supplied an `ignore_invalid_timeframe` argument in the request.security_lower_tf() call.

Tuples of intrabar data

When passing a tuple or a function call that returns a tuple as the `expression` argument in request.security_lower_tf(), the result is a tuple of arrays with type templates corresponding to the types within the argument. For example, using a `[float, string, color]` tuple as the `expression` will result in `[array<float>, array<string>, array<color>]` data returned by

the function. Using a tuple `expression` allows a script to fetch several arrays of intrabar data with a single request.security_lower_tf() function call.

The following example requests OHLC data from a lower timeframe and visualizes the current bar's intrabars on the chart using lines and boxes. The script calls request.security_lower_tf() with the `[open, high, low, close]` tuple as its `expression` to retrieve a tuple of arrays representing OHLC information from a calculated `lowerTimeframe`. It then uses a for loop to set line coordinates with the retrieved data and current bar indices to display the results next to the current chart bar, providing a "magnified view" of the price movement within the latest candle. It also draws a box around the lines to indicate the chart region occupied by intrabar drawings:

image

//@version=6 indicator("Tuples of intrabar data demo", "Candle magnifier", max_lines_count = 500) //@variable The maximum number of intrabars to display. int maxIntrabars = input.int(20, "Max intrabars", 1, 250) //@variable The width of the drawn candle bodies. int candleWidth = input.int(20, "Candle width", 2) //@variable The largest valid timeframe closest to `maxIntrabars` times smaller than the chart timeframe. string lowerTimeframe = timeframe.from_seconds(math.ceil(timeframe.in_seconds() / maxIntrabars)) //@variable An array of lines to represent intrabar wicks. var array<line> wicks = array.new<line>() //@variable An array of lines to represent intrabar bodies. var array<line> bodies = array.new<line>() //@variable A box that surrounds the displayed intrabars. var box magnifierBox = box.new(na, na, na, na, bgcolor = na) // Fill the `wicks` and `bodies` arrays with blank lines on the first bar. if barstate.isfirst for i = 1 to maxIntrabars array.push(wicks, line.new(na, na, na, na, color = color.gray)) array.push(bodies, line.new(na, na, na, na, width = candleWidth)) //@variable A tuple of "float" arrays containing `open`, `high`, `low`, and `close` prices from the `lowerTimeframe`. [oData, hData, lData, cData] = request.security_lower_tf(syminfo.tickerid, lowerTimeframe, [open, high, low, close]) //@variable The number of intrabars retrieved from the `lowerTimeframe` on the chart bar. int numIntrabars = array.size(oData) if numIntrabars > 0 // Define the start and end bar index values for intrabar display. int startIndex = bar_index + 2 int endIndex = startIndex + numIntrabars // Loop to update lines. for i = 0 to maxIntrabars - 1 line

wickLine = array.get(wicks, i) line bodyLine = array.get(bodies, i) if i < numIntrabars //@variable The `bar_index` of the drawing. int candleIndex = startIndex + i // Update the properties of the `wickLine` and `bodyLine`. line.set_xy1(wickLine, startIndex + i, array.get(hData, i)) line.set_xy2(wickLine, startIndex + i, array.get(lData, i)) line.set_xy1(bodyLine, startIndex + i, array.get(oData, i)) line.set_xy2(bodyLine, startIndex + i, array.get(cData, i)) line.set_color(bodyLine, bodyLine.get_y2() > bodyLine.get_y1() ? color.teal : color.maroon) continue // Set the coordinates of the `wickLine` and `bodyLine` to `na` if no intrabar data is available at the index. line.set_xy1(wickLine, na, na) line.set_xy2(wickLine, na, na) line.set_xy1(bodyLine, na, na) line.set_xy2(bodyLine, na, na) // Set the coordinates of the `magnifierBox`. box.set_lefttop(magnifierBox, startIndex - 1, array.max(hData)) box.set_rightbottom(magnifierBox, endIndex, array.min(lData))

Note that:

- The script draws each candle using two lines: one to represent wicks and the other to represent the body. Since the script can display up to 500 lines on the chart, we've limited the `maxIntrabars` input to 250.
- The `lowerTimeframe` value is the result of calculating the math.ceil() of the timeframe.in_seconds() divided by the `maxIntrabars` and converting to a valid timeframe string with timeframe.from_seconds().
- The script sets the top of the box drawing using the array.max() of the requested `hData` array, and it sets the box's bottom using the array.min() of the requested `lData` array. As we see on the chart, these values correspond to the high and low of the chart bar.

Requesting collections

In some cases, a script might need to request collections from an intrabar context. However, in contrast to request.security(), scripts cannot use collection references or calls to functions that return them as the `expression` argument in a request.security_lower_tf() call, because arrays cannot directly store references to other collections.

Despite these limitations, it is possible to request collections from lower timeframes, if needed, with the help of *wrapper* types.

To make collections requestable with request.security_lower_tf(), we must create a UDT with a field to reference a collection ID. This step is necessary since arrays cannot reference other collections directly but *can* reference UDTs with collection fields:

//@type A "wrapper" type to hold an `array<float>` instance. type Wrapper array<float> collection

With our `Wrapper` UDT defined, we can now pass the IDs of objects of the UDT to the `expression` parameter in request.security_lower_tf().

A straightforward approach is to use a call to the type's built-in `*.new()` function as the `expression` argument. For example, this line of code uses a call to `Wrapper.new()` with `array.from(close)` as the `collection` argument directly within the request.security_lower_tf() call:

//@variable An array of `Wrapper` IDs requested from the 1-minute timeframe. array<Wrapper> wrappers = request.security_lower_tf(syminfo.tickerid, "1", Wrapper.new(array.from(close)))

Alternatively, we can create a user-defined function or method that returns an object of the UDT and call that function within request.security_lower_tf(). For instance, this code calls a custom `newWrapper()` function that returns a `Wrapper` ID as the `expression` argument:

//@function Creates a new `Wrapper` instance to wrap the specified `collection`. newWrapper(array<float> collection) => Wrapper.new(collection) //@variable An array of `Wrapper` IDs requested from the 1-minute timeframe. array<Wrapper> wrappers = request.security_lower_tf(syminfo.tickerid, "1", newWrapper(array.from(close)))

The result with either of the above is an array containing `Wrapper` IDs from all available intrabars in the chart bar, which the script can use to reference `Wrapper` instances from specific intrabars and use their `collection` fields in additional operations.

The script below utilizes this approach to collect arrays of intrabar data from a `lowerTimeframe` and uses them to display data from a specific intrabar. Its

custom `Prices` type contains a single `data` field to reference `array<float>` instances that hold price data, and the user-defined `newPrices()` function returns the ID of a `Prices` object.

The script calls request.security_lower_tf() with a `newPrices()` call as its `expression` argument to retrieve an array of `Prices` IDs from each intrabar in the chart bar, then uses array.get() to get the ID from a specified available intrabar, if it exists. Lastly, it uses array.get() on the `data` array assigned to that instance and calls plotcandle() to display its values on the chart:

image

//@version=6 indicator("Requesting LTF collections demo", "Intrabar viewer", true) //@variable The timeframe of the LTF data request. string lowerTimeframe = input.timeframe("1", "Timeframe") //@variable The index of the intrabar to show on each chart bar. 0 is the first available intrabar. int intrabarIndex = input.int(0, "Intrabar to show", 0) //@variable A custom type to hold an array of price `data`. type Prices array<float> data //@function Returns a new `Prices` instance containing current `open`, `high`, `low`, and `close` prices. newPrices() => Prices.new(array.from(open, high, low, close)) //@variable An array of `Prices` requested from the `lowerTimeframe`. array<Prices> requestedPrices = request.security_lower_tf(syminfo.tickerid, lowerTimeframe, newPrices()) //@variable The `Prices` ID from the `requestedPrices` array at the `intrabarIndex`, or `na` if not available. Prices intrabarPrices = array.size(requestedPrices) > intrabarIndex ? array.get(requestedPrices, intrabarIndex) : na //@variable The `data` array from the `intrabarPrices`, or an array of `na` values if `intrabarPrices` is `na`. array<float> intrabarData = na(intrabarPrices) ? array.new<float>(4, na) : intrabarPrices.data // Plot the `intrabarData` values as candles. plotcandle(intrabarData.get(0), intrabarData.get(1), intrabarData.get(2), intrabarData.get(3))

Note that:

- The `intrabarPrices` variable references a `Prices` object only if the size of the `requestedPrices` array is greater than the `intrabarIndex`, because attempting to use array.get() to retrieve an element that doesn’t exist causes an out of bounds error.

- The `intrabarData` variable references an array from the `intrabarPrices.data` field only if the `intrabarPrices` variable references a `Prices` object. If `intrabarPrices` holds na because intrabar data is not available for a bar, the `intrabarData` variable references an array of na values.
- The process used in this example is *not* necessary to achieve the intended result. Instead of using UDTs, we can use the tuple `[open, high, low, close]` as the `expression` argument in the request to retrieve a tuple of arrays for further operations. See the Tuples of intrabar data section above for more information.

Custom contexts

Pine Script includes multiple `ticker.*()` functions that allow scripts to construct *custom* ticker IDs that specify additional settings for data requests when used as a `symbol` argument in request.security() and request.security_lower_tf():

- ticker.new() constructs a custom ticker ID from a specified `prefix` and `ticker` with additional `session` and `adjustment` settings.
- ticker.modify() constructs a modified form of a specified `tickerid` with additional `session` and `adjustment` settings.
- ticker.heikinashi(), ticker.renko(), ticker.pointfigure(), ticker.kagi(), and ticker.linebreak() construct a modified form a `symbol` with non-standard chart settings.
- ticker.inherit() constructs a new ticker ID for a `symbol` with additional parameters inherited from the `from_tickerid` specified in the function call, allowing scripts to request the `symbol` data with the same modifiers as the `from_tickerid`, including session, dividend adjustment, currency conversion, non-standard chart type, back-adjustment, settlement-as-close, etc.
- ticker.standard() constructs a standard ticker ID representing the `symbol` *without* additional modifiers.

Let's explore some practical examples of applying `ticker.*()` functions to request data from custom contexts.

Suppose we want to include dividend adjustment in a stock symbol's prices without enabling the "Adjust data for dividends" option in the "Symbol" section of the chart's settings. We can achieve this in a script by constructing a custom ticker ID for the instrument using ticker.new() or ticker.modify() with an `adjustment` value of adjustment.dividends.

This script creates an `adjustedTickerID` using ticker.modify(), uses that ticker ID as the `symbol` in request.security() to retrieve a tuple of adjusted price values, then uses plotcandle() to plot the result as candles on the chart. It also highlights the background of bars where the requested prices differ from the prices without dividend adjustment.

As we see on the "NYSE:XOM" chart below, enabling dividend adjustment results in different historical values before the date of the latest dividend:

image

//@version=6 indicator("Custom contexts demo 1", "Adjusted prices", true) // @variable A custom ticker ID representing the chart's symbol with the dividend adjustment modifier. string adjustedTickerID = ticker.modify(syminfo.tickerid, adjustment = adjustment.dividends) // Request the adjusted prices for the chart's symbol. [o, h, l, c] = request.security(adjustedTickerID, timeframe.period, [open, high, low, close]) //@variable The color of the candles on the chart. color candleColor = c > o ? color.teal : color.maroon // Plot the adjusted prices. plotcandle(o, h, l, c, "Adjusted Prices", candleColor) // Highlight the background when `c` is different from `close`. bgcolor(c != close ? color.new(color.orange, 80) : na)

Note that:

- If a modifier included in a constructed ticker ID does not apply to the symbol, the script will *ignore* that modifier when requesting data. For instance, this script will display the same values as the main chart on forex symbols such as "EURUSD".

While the example above demonstrates a simple way to modify the chart's symbol, a more frequent use case for `ticker.*()` functions is applying custom modifiers to another symbol while requesting data. If a ticker ID referenced in a script already has the modifiers one would like to apply (e.g.,

adjustment settings, session type, etc.), they can use ticker.inherit() to quickly and efficiently add those modifiers to another symbol.

In the example below, we've edited the previous script to request data for a `symbolInput` using modifiers inherited from the `adjustedTickerID`. This script calls ticker.inherit() to construct an `inheritedTickerID` and uses that ticker ID in a request.security() call. It also requests data for the `symbolInput` without additional modifiers and plots candles for both ticker IDs in a separate chart pane to compare the difference.

As shown on the chart, the data requested using the `inheritedTickerID` includes dividend adjustment, whereas the data requested using the `symbolInput` directly does not:

image

//@version=6 indicator("Custom contexts demo 2", "Inherited adjustment") // @variable The symbol to request data from. string symbolInput = input.symbol("NYSE:PFE", "Symbol") //@variable A custom ticker ID representing the chart's symbol with the dividend adjustment modifier. string adjustedTickerID = ticker.modify(syminfo.tickerid, adjustment = adjustment.dividends) //@variable A custom ticker ID representing the `symbolInput` with modifiers inherited from the `adjustedTickerID`. string inheritedTickerID = ticker.inherit(adjustedTickerID, symbolInput) // Request prices using the `symbolInput`. [o1, h1, l1, c1] = request.security(symbolInput, timeframe.period, [open, high, low, close]) // Request prices using the `inheritedTickerID`. [o2, h2, l2, c2] = request.security(inheritedTickerID, timeframe.period, [open, high, low, close]) //@variable The color of the candles that use the `inheritedTickerID` prices. color candleColor = c2 > o2 ? color.teal : color.maroon // Plot the `symbol` prices. plotcandle(o1, h1, l1, c1, "Symbol", color.gray, color.gray, bordercolor = color.gray) // Plot the `inheritedTickerID` prices. plotcandle(o2, h2, l2, c2, "Symbol With Modifiers", candleColor) // Highlight the background when `c1` is different from `c2`. bgcolor(c1 != c2 ? color.new(color.orange, 80) : na)

Note that:

- Since the `adjustedTickerID` represents a modified form of the syminfo.tickerid, if we modify the chart's context in other ways, such as

changing the chart type or enabling extended trading hours in the chart's settings, those modifiers will also apply to the `adjustedTickerID` and `inheritedTickerID`. However, they will *not* apply to the `symbolInput` since it represents a *standard* ticker ID.

Another frequent use case for requesting custom contexts is retrieving data that uses non-standard chart calculations. For example, suppose we want to use Renko price values to calculate trade signals in a strategy() script. If we simply change the chart type to "Renko" to get the prices, the strategy will also simulate its trades based on those synthetic prices, producing misleading results:

image

//@version=6 strategy("Custom contexts demo 3", "Renko strategy", true, default_qty_type = strategy.percent_of_equity, default_qty_value = 2, initial_capital = 50000, slippage = 2, commission_type = strategy.commission.cash_per_contract, commission_value = 1, margin_long = 100, margin_short = 100) //@variable When `true`, the strategy places a long market order. bool longEntry = ta.crossover(close, open) //@variable When `true`, the strategy places a short market order. bool shortEntry = ta.crossunder(close, open) if longEntry strategy.entry("Long Entry", strategy.long) if shortEntry strategy.entry("Short Entry", strategy.short)

To ensure our strategy shows results based on *actual* prices, we can create a Renko ticker ID using ticker.renko() while keeping the chart on a *standard type*, allowing the script to request and use Renko prices to calculate its signals without calculating the strategy results on them:

image

//@version=6 strategy("Custom contexts demo 3", "Renko strategy", true, default_qty_type = strategy.percent_of_equity, default_qty_value = 2, initial_capital = 50000, slippage = 1, commission_type = strategy.commission.cash_per_contract, commission_value = 1, margin_long = 100, margin_short = 100) //@variable A Renko ticker ID. string renkoTickerID = ticker.renko(syminfo.tickerid, "ATR", 14) // Request the `open` and `close` prices using the `renkoTickerID`. [renkoOpen, renkoClose] = request.security(renkoTickerID, timeframe.period, [open, close]) // @variable When `true`, the strategy places a long market order. bool

longEntry = ta.crossover(renkoClose, renkoOpen) //@variable When `true`, the strategy places a short market order. bool shortEntry = ta.crossunder(renkoClose, renkoOpen) if longEntry strategy.entry("Long Entry", strategy.long) if shortEntry strategy.entry("Short Entry", strategy.short) plot(renkoOpen) plot(renkoClose)

Historical and realtime behavior

Functions in the `request.*()` namespace can behave differently on historical and realtime bars. This behavior is closely related to Pine's Execution model.

Consider how a script behaves within the main context. Throughout the chart's history, the script calculates its required values once and *commits* them to that bar so their states are accessible on subsequent executions. On an unconfirmed bar, however, the script recalculates its values on *each update* to the bar's data to align with realtime changes. Before recalculating the values on that bar, it reverts calculated values to their last committed states, otherwise known as *rollback*, and it only commits values to that bar once the bar closes.

Now consider the behavior of data requests from other contexts with request.security(). As when evaluating historical bars in the main context, request.security() only returns new historical values when it confirms a bar in its specified context. When executing on realtime bars, it returns recalculated values on each chart bar, similar to how a script recalculates values in the main context on the open chart bar.

However, the function only *confirms* the requested values when a bar from its context closes. When the script restarts, what were previously *realtime* bars become *historical* bars. Therefore, request.security() only returns the values it confirmed on those bars. In essence, this behavior means that requested data may *repaint* when its values fluctuate on realtime bars without confirmation from the context.

In most circumstances where a script requests data from a broader context, one will typically require confirmed, stable values that *do not* fluctuate on realtime bars. The section below explains how to achieve such a result and avoid repainting data requests.

Avoiding Repainting

Higher-timeframe data

When requesting values from a higher timeframe, they are subject to repainting since realtime bars can contain *unconfirmed* information from developing HTF bars, and the script may adjust the times that new values come in on historical bars. To avoid repainting HTF data, one must ensure that the function only returns confirmed values with consistent timing on all bars, regardless of bar state.

The most reliable approach to achieve non-repainting results is to use an `expression` argument that only references past bars (e.g., `close[1]`) while using barmerge.lookahead_on as the `lookahead` value.

Using barmerge.lookahead_on with non-offset HTF data requests is discouraged since it prompts request.security() to "look ahead" to the final values of an HTF bar, retrieving confirmed values *before* they're actually available in the script's history. However, if the values used in the `expression` are offset by at least one bar, the "future" data the function retrieves is no longer from the future. Instead, the data represents confirmed values from established, *available* HTF bars. In other words, applying an offset to the `expression` effectively prevents the requested data from repainting when the script restarts its executions and eliminates lookahead bias in the historical series.

The following example demonstrates a repainting HTF data request. The script uses request.security() without offset modifications or additional arguments to retrieve the results of a ta.wma() call from a higher timeframe. It also highlights the background to indicate which bars were in a realtime state during its calculations.

As shown on the chart below, the plot of the requested WMA only changes on historical bars when HTF bars close, whereas it fluctuates on all realtime bars since the data includes unconfirmed values from the higher timeframe:

image

//@version=6 indicator("Avoiding HTF repainting demo", overlay = true) // @variable The multiplier applied to the chart's timeframe. int tfMultiplier = input.int(10, "Timeframe multiplier", 1) //@variable The number of bars in the moving average. int length = input.int(5, "WMA smoothing length") // @variable The valid timeframe string closest to `tfMultiplier` times larger than the chart timeframe. string timeframe = timeframe.from_seconds(timeframe.in_seconds() * tfMultiplier) //@variable The weighted MA of `close` prices over `length` bars on the `timeframe`. // This request repaints because it includes unconfirmed HTF data on realtime bars and it may offset the // times of its historical results. float requestedWMA = request.security(syminfo.tickerid, timeframe, ta.wma(close, length)) // Plot the requested series. plot(requestedWMA, "HTF WMA", color.purple, 3) // Highlight the background on realtime bars. bgcolor(barstate.isrealtime ? color.new(color.orange, 70) : na, title = "Realtime bar highlight")

To avoid repainting in this script, we can add `lookahead = barmerge.lookahead_on` to the request.security() call and offset the call history of ta.wma() by one bar with the history-referencing operator [], ensuring the request always retrieves the last confirmed HTF bar's WMA at the start of each new `timeframe`. Unlike the previous script, this version has consistent behavior on historical and realtime bar states, as we see below:

image

//@version=6 indicator("Avoiding HTF repainting demo", overlay = true) // @variable The multiplier applied to the chart's timeframe. int tfMultiplier = input.int(10, "Timeframe multiplier", 1) //@variable The number of bars in the moving average. int length = input.int(5, "WMA smoothing length") // @variable The valid timeframe string closest to `tfMultiplier` times larger than the chart timeframe. string timeframe = timeframe.from_seconds(timeframe.in_seconds() * tfMultiplier) //@variable The weighted MA of `close` prices over `length` bars on the `timeframe`. // This request does not repaint, as it always references the last confirmed WMA value on all bars. float requestedWMA = request.security(syminfo.tickerid, timeframe, ta.wma(close, length)[1], lookahead = barmerge.lookahead_on) // Plot the requested value. plot(requestedWMA, "HTF WMA", color.purple, 3) // Highlight the background on realtime bars.

bgcolor(barstate.isrealtime ? color.new(color.orange, 70) : na, title = "Realtime bar highlight")

Lower-timeframe data

The request.security() and request.security_lower_tf() functions can retrieve data from lower-timeframe contexts. The request.security() function can only retrieve data from a *single* intrabar in each chart bar, and request.security_lower_tf() retrieves data from *all* available intrabars.

When using these functions to retrieve intrabar data, it's important to note that such requests are **not** immune to repainting behavior. Historical and realtime series often rely on *separate* data feeds. Data providers may retroactively modify realtime data, and it's possible for races to occur in realtime data feeds, as explained in the Data feeds section of this page. Either case may result in intrabar data retrieved on realtime bars repainting after the script restarts its executions.

Additionally, a particular case that *will* cause repainting LTF requests is using request.security() with barmerge.lookahead_on to retrieve data from the first intrabar in each chart bar. While it will generally work as expected on historical bars, it will track only the most recent intrabar on realtime bars, as request.security() does not retain all intrabar information, and the intrabars the function retrieves on realtime bars are unsorted until restarting the script:

image

//@version=6 indicator("Avoiding LTF repainting demo", overlay = true) // @variable The lower timeframe of the requested data. string lowerTimeframe = input.timeframe("1", "Timeframe") //@variable The first intrabar `close` requested from the `lowerTimeframe` on each bar. // Only works as intended on historical bars. float requestedClose = request.security(syminfo.tickerid, lowerTimeframe, close, lookahead = barmerge.lookahead_on) // Plot the `requestedClose`. plot(requestedClose, "First intrabar close", linewidth = 3) // Highlight the background on realtime bars. bgcolor(barstate.isrealtime ? color.new(color.orange, 60) : na, title = "Realtime bar Highlight")

One can mitigate this behavior and track the values from the first intrabar, or any available intrabar in the chart bar, by using request.security_lower_tf() since it maintains an array of intrabar values ordered by the times they come in. Here, we call array.first() on a requested array of intrabar data to retrieve the close price from the first available intrabar in each chart bar:

image

//@version=6 indicator("Avoiding LTF repainting demo", overlay = true) // @variable The lower timeframe of the requested data. string lowerTimeframe = input.timeframe("1", "Timeframe") //@variable An array of intrabar `close` values requested from the `lowerTimeframe` on each bar. array<float> requestedCloses = request.security_lower_tf(syminfo.tickerid, lowerTimeframe, close) //@variable The first intrabar `close` on each bar with available data. float firstClose = requestedCloses.size() > 0 ? requestedCloses.first() : na // Plot the `firstClose`. plot(firstClose, "First intrabar close", linewidth = 3) // Highlight the background on realtime bars. bgcolor(barstate.isrealtime ? color.new(color.orange, 60) : na, title = "Realtime bar Highlight")

Note that:

- While request.security_lower_tf() is more optimized for handling historical and realtime intrabars, it's still possible in some cases for minor repainting to occur due to data differences from the provider, as outlined above.
- This code may not show intrabar data on all available chart bars, depending on how many intrabars each chart bar contains, as `request.*()` functions can retrieve up to 200,000 intrabars from an LTF context. The maximum number of requestable intrabars depends on the user's plan. See this section of the Limitations page for more information.

request.currency_rate()

When a script needs to convert values expressed in one currency to another, one can use request.currency_rate(). This function requests a *daily rate* for currency conversion calculations based on currency pair or spread data from

the most popular exchanges, providing a simpler alternative to fetching specific pairs or spreads with request.security().

While one can use request.security() to retrieve daily currency rates, its use case is more involved than request.currency_rate(), as one needs to supply a valid *ticker ID* for a currency pair or spread to request the rate. Additionally, a historical offset and barmerge.lookahead_on are necessary to prevent the results from repainting, as explained in this section.

The request.currency_rate() function, on the other hand, only requires *currency codes*. No ticker ID is needed when requesting rates with this function, and it ensures non-repainting results without requiring additional specification.

The function's signature is as follows:

```
request.currency_rate(from, to, ignore_invalid_currency) → series floa
```

The `from` parameter specifies the currency to convert, and the `to` parameter specifies the target currency. Both parameters accept "string" values representing valid currency codes (e.g., "USD") or any built-in `currency.*` variable (e.g., currency.USD).

When the function cannot calculate a valid conversion rate between the specified `from` and `to` currencies, programmers can decide whether it raises a runtime error or returns na via the `ignore_invalid_currency` parameter. The default value is `false`, meaning the function raises a runtime error and halts the script's executions.

The following example demonstrates a simple use case for request.currency_rate(). Suppose we want to convert values expressed in Turkish lira (currency.TRY) to South Korean won (currency.KRW) using a daily conversion rate. If we use request.security() to retrieve the rate, we must supply a valid ticker ID and request the last confirmed close from the previous day.

In this case, no valid symbol exists that would allow us to retrieve a conversion rate directly with request.security(). Therefore, we first need a ticker ID for a spread that converts TRY to an intermediate currency, such as USD, then converts the intermediate currency to KRW. We can then use that

ticker ID within request.security() with `close[1]` as the `expression` and barmerge.lookahead_on as the `lookahead` value to request a non-repainting daily rate.

Alternatively, we can achieve the same result more simply by calling request.currency_rate(). This function does all the heavy lifting for us, only requiring `from` and `to` currency arguments to perform its calculation.

As we see below, both approaches return the same daily rate:

image

//@version=6 indicator("Requesting currency rates demo") //@variable The currency to convert. simple string fromCurrency = currency.TRY //@variable The resulting currency. simple string toCurrency = currency.KRW //@variable The spread symbol to request. Required in `request.security()` because no direct symbol exists. simple string spreadSymbol = str.format("{0}{2} * {2}{1}", fromCurrency, toCurrency, currency.USD) //@variable The non-repainting conversion rate from `request.security()` using the `spreadSymbol`. float securityRequestedRate = request.security(spreadSymbol, "1D", close[1], lookahead = barmerge.lookahead_on) //@variable The non-repainting conversion rate from `request.currency_rate()`. float nonSecurityRequestedRate = request.currency_rate(fromCurrency, toCurrency) // Plot the requested rates. We can multiply TRY values by these rates to convert them to KRW. plot(securityRequestedRate, "`request.security()` value", color.purple, 5) plot(nonSecurityRequestedRate, "`request.currency_rate()` value", color.yellow, 2)

request.dividends(), request.splits(), and request.earnings()

Analyzing a stock's earnings data and corporate actions provides helpful insights into its underlying financial strength. Pine Script provides the ability to retrieve essential information about applicable stocks via request.dividends(), request.splits(), and request.earnings().

These are the functions' signatures:

```
request.dividends(ticker, field, gaps, lookahead, ignore_invalid_symbo

request.splits(ticker, field, gaps, lookahead, ignore_invalid_symbol)

request.earnings(ticker, field, gaps, lookahead, ignore_invalid_symbol
```

Each function has the same parameters in its signature, with the exception of request.splits(), which doesn't have a `currency` parameter.

Note that unlike the `symbol` parameter in other `request.*()` functions, the `ticker` parameter in these functions only accepts an *"Exchange:Symbol" pair*, such as "NASDAQ:AAPL". The built-in syminfo.ticker variable does not work with these functions since it does not contain exchange information. Instead, one must use syminfo.tickerid for such cases.

The `field` parameter determines the data the function will retrieve. Each of these functions accepts different built-in variables as the `field` argument since each requests different information about a stock:

- The request.dividends() function retrieves current dividend information for a stock, i.e., the amount per share the issuing company paid out to investors who purchased shares before the ex-dividend date. Passing the built-in dividends.gross or dividends.net variables to the `field` parameter specifies whether the returned value represents dividends before or after factoring in expenses the company deducts from its payouts.
- The request.splits() function retrieves current split and reverse split information for a stock. A split occurs when a company increases its outstanding shares to promote liquidity. A reverse split occurs when a company consolidates its shares and offers them at a higher price to attract specific investors or maintain their listing on a market that has a minimum per-share price. Companies express their split information as *ratios*. For example, a 5:1 split means the company issued additional shares to its shareholders so that they have five times the number of shares they had before the split, and the raw price of each share becomes one-fifth of the previous price. Passing splits.numerator or splits.denominator to the `field` parameter of request.splits() determines whether it returns the numerator or denominator of the split ratio.

- The request.earnings() function retrieves the earnings per share (EPS) information for a stock `ticker`'s issuing company. The EPS value is the ratio of a company's net income to the number of outstanding stock shares, which investors consider an indicator of the company's profitability. Passing earnings.actual, earnings.estimate, or earnings.standardized as the `field` argument in request.earnings() respectively determines whether the function requests the actual, estimated, or standardized EPS value.

For a detailed explanation of the `gaps`, `lookahead`, and `ignore_invalid_symbol` parameters of these functions, see the Common characteristics section at the top of this page.

It's important to note that the values returned by these functions reflect the data available as it comes in. This behavior differs from financial data originating from a request.financial() call in that the underlying data from such calls becomes available according to a company's fiscal reporting period.

Here, we've included an example that displays a handy table containing the most recent dividend, split, and EPS data. The script calls the `request.*()` functions discussed in this section to retrieve the data, then converts the values to "strings" with `str.*()` functions and displays the results in the `infoTable` with table.cell():

image

//@version=6 indicator("Dividends, splits, and earnings demo", overlay = true) //@variable The size of the table's text. string tableSize = input.string(size.large, "Table size", [size.auto, size.tiny, size.small, size.normal, size.large, size.huge]) //@variable The color of the table's text and frame. var color tableColor = chart.fg_color //@variable A `table` displaying the latest dividend, split, and EPS information. var table infoTable = table.new(position.top_right, 3, 4, frame_color = tableColor, frame_width = 1) // Add header cells on the first bar. if barstate.isfirst table.cell(infoTable, 0, 0, "Field", text_color = tableColor, text_size = tableSize) table.cell(infoTable, 1, 0, "Value", text_color = tableColor, text_size = tableSize) table.cell(infoTable, 2, 0, "Date", text_color = tableColor, text_size = tableSize) table.cell(infoTable, 0, 1, "Dividend", text_color = tableColor,

text_size = tableSize) table.cell(infoTable, 0, 2, "Split", text_color = tableColor, text_size = tableSize) table.cell(infoTable, 0, 3, "EPS", text_color = tableColor, text_size = tableSize) //@variable The amount of the last reported dividend as of the current bar. float latestDividend = request.dividends(syminfo.tickerid, dividends.gross, barmerge.gaps_on) //@variable The numerator of that last reported split ratio as of the current bar. float latestSplitNum = request.splits(syminfo.tickerid, splits.numerator, barmerge.gaps_on) //@variable The denominator of the last reported split ratio as of the current bar. float latestSplitDen = request.splits(syminfo.tickerid, splits.denominator, barmerge.gaps_on) //@variable The last reported earnings per share as of the current bar. float latestEPS = request.earnings(syminfo.tickerid, earnings.actual, barmerge.gaps_on) // Update the "Value" and "Date" columns when new values come in. if not na(latestDividend) table.cell(infoTable, 1, 1, str.tostring(math.round(latestDividend, 3)), text_color = tableColor, text_size = tableSize) table.cell(infoTable, 2, 1, str.format_time(time, "yyyy-MM-dd"), text_color = tableColor, text_size = tableSize) if not na(latestSplitNum) table.cell(infoTable, 1, 2, str.format("{0}-for-{1}", latestSplitNum, latestSplitDen), text_color = tableColor, text_size = tableSize) table.cell(infoTable, 2, 2, str.format_time(time, "yyyy-MM-dd"), text_color = tableColor, text_size = tableSize) if not na(latestEPS) table.cell(infoTable, 1, 3, str.tostring(latestEPS), text_color = tableColor, text_size = tableSize) table.cell(infoTable, 2, 3, str.format_time(time, "yyyy-MM-dd"), text_color = tableColor, text_size = tableSize)

Note that:

- We've included barmerge.gaps_on in the `request.*()` calls, so they only return values when new data is available. Otherwise, they return na.
- The script assigns a table ID to the `infoTable` variable on the first chart bar. On subsequent bars, it updates necessary cells with new information whenever data is available.
- If no information is available from any of the `request.*()` calls throughout the chart's history (e.g., if the `ticker` has no dividend information), the script does not initialize the corresponding cells since it's unnecessary.

request.financial()

Financial metrics provide investors with insights about a company's economic and financial health that are not tangible from solely analyzing its stock prices. TradingView offers a wide variety of financial metrics from FactSet that traders can access via the "Financials" tab in the "Indicators" menu of the chart. Scripts can access available metrics for an instrument directly via the request.financial() function.

This is the function's signature:

```
request.financial(symbol, financial_id, period, gaps, ignore_invalid_s
```

As with the first parameter in request.dividends(), request.splits(), and request.earnings(), the `symbol` parameter in request.financial() requires an *"Exchange:Symbol" pair*. To request financial information for the chart's ticker ID, use syminfo.tickerid, as syminfo.ticker will not work.

The `financial_id` parameter accepts a "string" value representing the ID of the requested financial metric. TradingView has numerous financial metrics to choose from. See the Financial IDs section below for an overview of all accessible metrics and their "string" identifiers.

The `period` parameter specifies the fiscal period for which new requested data comes in. It accepts one of the following "string" arguments: **"FQ" (quarterly), "FH" (semiannual), "FY" (annual), or "TTM" (trailing twelve months)**. Not all fiscal periods are available for all metrics or instruments. To confirm which periods are available for specific metrics, see the second column of the tables in the Financial IDs section.

See this page's Common characteristics section for a detailed explanation of this function's `gaps`, `ignore_invalid_symbol`, and `currency` parameters.

It's important to note that the data retrieved from this function comes in at a *fixed frequency*, independent of the precise date on which the data is made available within a fiscal period. For a company's dividends, splits, and earnings per share (EPS) information, one can request data reported on exact dates via request.dividends(), request.splits(), and request.earnings().

This script uses request.financial() to retrieve information about the income and expenses of a stock's issuing company and visualize the profitability of its typical business operations. It requests the "OPER_INCOME", "TOTAL_REVENUE", and "TOTAL_OPER_EXPENSE" financial IDs for the syminfo.tickerid over the latest `fiscalPeriod`, then plots the results on the chart:

image

//@version=6 indicator("Requesting financial data demo", format = format.volume) //@variable The size of the fiscal reporting period. Some options may not be available, depending on the instrument. string fiscalPeriod = input.string("FQ", "Period", ["FQ", "FH", "FY", "TTM"]) // @variable The operating income after expenses reported for the stock's issuing company. float operatingIncome = request.financial(syminfo.tickerid, "OPER_INCOME", fiscalPeriod) //@variable The total revenue reported for the stock's issuing company. float totalRevenue = request.financial(syminfo.tickerid, "TOTAL_REVENUE", fiscalPeriod) // @variable The total operating expenses reported for the stock's issuing company. float totalExpenses = request.financial(syminfo.tickerid, "TOTAL_OPER_EXPENSE", fiscalPeriod) //@variable Is aqua when the `totalRevenue` exceeds the `totalExpenses`, fuchsia otherwise. color incomeColor = operatingIncome > 0 ? color.new(color.aqua, 50) : color.new(color.fuchsia, 50) // Display the requested data. plot(operatingIncome, "Operating income", incomeColor, 1, plot.style_area) plot(totalRevenue, "Total revenue", color.green, 3) plot(totalExpenses, "Total operating expenses", color.red, 3)

Note that:

- Not all `fiscalPeriod` options are available for every ticker ID. For example, companies in the US typically publish *quarterly* reports, whereas many European companies publish *semiannual* reports. See this page in our Help Center for more information.

Calculating financial metrics

The request.financial() function can provide scripts with numerous useful financial metrics that don't require additional calculations. However, some

commonly used financial estimates require combining an instrument's current market price with requested financial data. Such is the case for:

- Market Capitalization (market price * total shares outstanding)
- Earnings Yield (12-month EPS / market price)
- Price-to-Book Ratio (market price / BVPS)
- Price-to-Earnings Ratio (market price / EPS)
- Price-to-Sales Ratio (market cap / 12-month total revenue)

The following script contains user-defined functions that calculate the above financial metrics for the syminfo.tickerid. We've created these functions so users can easily copy them into their scripts. This example uses them within a str.format() call to construct a `tooltipText`, which it displays in tooltips on the chart using labels. Hovering over any bar's label will expose the tooltip containing the metrics calculated on that bar:

image

//@version=6 indicator("Calculating financial metrics demo", overlay = true, max_labels_count = 500) //@function Calculates the market capitalization (market cap) for the chart's symbol. marketCap() => //@variable The most recent number of outstanding shares reported for the symbol. float totalSharesOutstanding = request.financial(syminfo.tickerid, "TOTAL_SHARES_OUTSTANDING", "FQ") // Return the market cap value. totalSharesOutstanding * close //@function Calculates the Earnings Yield for the chart's symbol. earningsYield() => //@variable The most recent 12-month earnings per share reported for the symbol. float eps = request.financial(syminfo.tickerid, "EARNINGS_PER_SHARE", "TTM") //Return the Earnings Yield percentage. 100.0 * eps / close //@function Calculates the Price-to-Book (P/B) ratio for the chart's symbol. priceBookRatio() => //@variable The most recent Book Value Per Share (BVPS) reported for the symbol. float bookValuePerShare = request.financial(syminfo.tickerid, "BOOK_VALUE_PER_SHARE", "FQ") // Return the P/B ratio. close / bookValuePerShare //@function Calculates the Price-to-Earnings (P/E) ratio for the chart's symbol. priceEarningsRatio() => //@variable The most recent 12-month earnings per share reported for the symbol. float eps = request.financial(syminfo.tickerid, "EARNINGS_PER_SHARE", "TTM") // Return the P/E ratio. close / eps //@function Calculates the Price-to-Sales (P/S) ratio for the chart's symbol. priceSalesRatio() => //@variable The most recent

number of outstanding shares reported for the symbol. float totalSharesOutstanding = request.financial(syminfo.tickerid, "TOTAL_SHARES_OUTSTANDING", "FQ") //@variable The most recent 12-month total revenue reported for the symbol. float totalRevenue = request.financial(syminfo.tickerid, "TOTAL_REVENUE", "TTM") // Return the P/S ratio. totalSharesOutstanding * close / totalRevenue //@variable The text to display in label tooltips. string tooltipText = str.format("Market Cap: {0} {1}\nEarnings Yield: {2}%\nP/B Ratio: {3}\nP/E Ratio: {4}\nP/S Ratio: {5}", str.tostring(marketCap(), format.volume), syminfo.currency, earningsYield(), priceBookRatio(), priceEarningsRatio(), priceSalesRatio()) //@variable Displays a blank label with a tooltip containing the `tooltipText`. label info = label.new(chart.point.now(high), tooltip = tooltipText)

Note that:

- Since not all companies publish quarterly financial reports, one may need to change the “FQ” in these functions to match the minimum reporting period for a specific company, as the request.financial() calls will return na when “FQ” data isn’t available.

Financial IDs

Below is an overview of all financial metrics one can request via request.financial(), along with the periods in which reports may be available. We’ve divided this information into four tables corresponding to the categories displayed in the “Financials” section of the “Indicators” menu:

- Income statements
- Balance sheet
- Cash flow
- Statistics

Each table has the following three columns:

- The first column contains descriptions of each metric with links to Help Center pages for additional information.
- The second column lists the possible `period` arguments allowed for the metric. Note that all available values may not be compatible with

specific ticker IDs, e.g., while “FQ” may be a possible argument, it will not work if the issuing company does not publish quarterly data.
- The third column lists the “string” IDs for the `financial_id` argument in request.financial().

Income statements

This table lists the available metrics that provide information about a company’s income, costs, profits and losses.

Click to show/hide

Financial	period	financial_id
After tax other income/ expense	FQ, FH, FY, TTM	AFTER_TAX_OTHER_INCOME
Average basic shares outstanding	FQ, FH, FY	BASIC_SHARES_OUTSTANDING
Basic earnings per share (Basic EPS)	FQ, FH, FY, TTM	EARNINGS_PER_SHARE_BASIC
Cost of goods sold	FQ, FH, FY, TTM	COST_OF_GOODS
Deprecation and amortization	FQ, FH, FY, TTM	DEP_AMORT_EXP_INCOME_S
Diluted earnings per share (Diluted EPS)	FQ, FH, FY, TTM	EARNINGS_PER_SHARE_DILUTED
Diluted net income available to common stockholders	FQ, FH, FY, TTM	DILUTED_NET_INCOME
Diluted shares outstanding	FQ, FH, FY	DILUTED_SHARES_OUTSTANDING
Dilution adjustment	FQ, FH, FY, TTM	DILUTION_ADJUSTMENT
Discontinued operations	FQ, FH, FY, TTM	DISCONTINUED_OPERATIONS
EBIT	FQ, FH, FY, TTM	EBIT
EBITDA	FQ, FH, FY, TTM	EBITDA
Equity in earnings		EQUITY_IN_EARNINGS

Financial	period	financial_id
	FQ, FH, FY, TTM	
Gross profit	FQ, FH, FY, TTM	GROSS_PROFIT
Interest capitalized	FQ, FH, FY, TTM	INTEREST_CAPITALIZED
Interest expense on debt	FQ, FH, FY, TTM	INTEREST_EXPENSE_ON_DEBT
Interest expense, net of interest capitalized	FQ, FH, FY, TTM	NON_OPER_INTEREST_EXP
Miscellaneous non-operating expense	FQ, FH, FY, TTM	OTHER_INCOME
Net income	FQ, FH, FY, TTM	NET_INCOME
Net income before discontinued operations	FQ, FH, FY, TTM	NET_INCOME_BEF_DISC_OPER
Non-controlling/minority interest	FQ, FH, FY, TTM	MINORITY_INTEREST_EXP
Non-operating income, excl. interest expenses	FQ, FH, FY, TTM	NON_OPER_INCOME
Non-operating income, total	FQ, FH, FY, TTM	TOTAL_NON_OPER_INCOME
Non-operating interest income	FQ, FH, FY, TTM	NON_OPER_INTEREST_INCOME
Operating expenses (excl. COGS)	FQ, FH, FY, TTM	OPERATING_EXPENSES
Operating income	FQ, FH, FY, TTM	OPER_INCOME
Other cost of goods sold	FQ, FH, FY, TTM	COST_OF_GOODS_EXCL_DEP_AMORT
Other operating expenses, total	FQ, FH, FY, TTM	OTHER_OPER_EXPENSE_TOTAL
Preferred dividends	FQ, FH, FY, TTM	PREFERRED_DIVIDENDS

Financial	period	financial_id
Pretax equity in earnings	FQ, FH, FY, TTM	PRETAX_EQUITY_IN_EARNINGS
Pretax income	FQ, FH, FY, TTM	PRETAX_INCOME
Research & development	FQ, FH, FY, TTM	RESEARCH_AND_DEV
Selling/general/admin expenses, other	FQ, FH, FY, TTM	SELL_GEN_ADMIN_EXP_OTHER
Selling/general/admin expenses, total	FQ, FH, FY, TTM	SELL_GEN_ADMIN_EXP_TOTAL
Taxes	FQ, FH, FY, TTM	INCOME_TAX
Total operating expenses	FQ, FH, FY, TTM	TOTAL_OPER_EXPENSE
Total revenue	FQ, FH, FY, TTM	TOTAL_REVENUE
Unusual income/expense	FQ, FH, FY, TTM	UNUSUAL_EXPENSE_INC

Balance sheet

This table lists the metrics that provide information about a company’s capital structure.

Click to show/hide

Financial	period	financial_id
Accounts payable	FQ, FH, FY	ACCOUNTS_PAYABLE
Accounts receivable - trade, net	FQ, FH, FY	ACCOUNTS_RECEIVABLES_NET
Accrued payroll	FQ, FH, FY	ACCRUED_PAYROLL
Accumulated depreciation, total	FQ, FH, FY	ACCUM_DEPREC_TOTAL

Financial	period	financial_id
Additional paid-in capital/ Capital surplus	FQ, FH, FY	ADDITIONAL_PAID_IN_CAPITAL
Book value per share	FQ, FH, FY	BOOK_VALUE_PER_SHARE
Capital and operating lease obligations	FQ, FH, FY	CAPITAL_OPERATING_LEASE_OBLIGATIONS
Capitalized lease obligations	FQ, FH, FY	CAPITAL_LEASE_OBLIGATIONS
Cash & equivalents	FQ, FH, FY	CASH_N_EQUIVALENTS
Cash and short term investments	FQ, FH, FY	CASH_N_SHORT_TERM_INVEST
Common equity, total	FQ, FH, FY	COMMON_EQUITY_TOTAL
Common stock par/ Carrying value	FQ, FH, FY	COMMON_STOCK_PAR
Current portion of LT debt and capital leases	FQ, FH, FY	CURRENT_PORT_DEBT_CAPITAL_LEASES
Deferred income, current	FQ, FH, FY	DEFERRED_INCOME_CURRENT
Deferred income, non-current	FQ, FH, FY	DEFERRED_INCOME_NON_CURRENT
Deferred tax assets	FQ, FH, FY	DEFERRED_TAX_ASSESTS
Deferred tax liabilities	FQ, FH, FY	DEFERRED_TAX_LIABILITIES
Dividends payable	FY	DIVIDENDS_PAYABLE
Goodwill, net	FQ, FH, FY	GOODWILL
Gross property/plant/ equipment	FQ, FH, FY	PPE_TOTAL_GROSS
Income tax payable	FQ, FH, FY	INCOME_TAX_PAYABLE
		INVENTORY_FINISHED_GOODS

Financial	period	financial_id
Inventories - finished goods	FQ, FH, FY	
Inventories - progress payments & other	FQ, FH, FY	INVENTORY_PROGRESS_PAYMENTS
Inventories - raw materials	FQ, FH, FY	INVENTORY_RAW_MATERIALS
Inventories - work in progress	FQ, FH, FY	INVENTORY_WORK_IN_PROGRESS
Investments in unconsolidated subsidiaries	FQ, FH, FY	INVESTMENTS_IN_UNCONCSOLIDATE
Long term debt	FQ, FH, FY	LONG_TERM_DEBT
Long term debt excl. lease liabilities	FQ, FH, FY	LONG_TERM_DEBT_EXCL_CAPITAL_LEASE
Long term investments	FQ, FH, FY	LONG_TERM_INVESTMENTS
Minority interest	FQ, FH, FY	MINORITY_INTEREST
Net debt	FQ, FH, FY	NET_DEBT
Net intangible assets	FQ, FH, FY	INTANGIBLES_NET
Net property/plant/ equipment	FQ, FH, FY	PPE_TOTAL_NET
Note receivable - long term	FQ, FH, FY	LONG_TERM_NOTE_RECEIVABLE
Notes payable	FY	NOTES_PAYABLE_SHORT_TERM_DEBT
Operating lease liabilities	FQ, FH, FY	OPERATING_LEASE_LIABILITIES
Other common equity	FQ, FH, FY	OTHER_COMMON_EQUITY
Other current assets, total	FQ, FH, FY	OTHER_CURRENT_ASSETS_TOTAL

Financial	period	financial_id
Other current liabilities	FQ, FH, FY	OTHER_CURRENT_LIABILITIES
Other intangibles, net	FQ, FH, FY	OTHER_INTANGIBLES_NET
Other investments	FQ, FH, FY	OTHER_INVESTMENTS
Other long term assets, total	FQ, FH, FY	LONG_TERM_OTHER_ASSETS_TOTAL
Other non-current liabilities, total	FQ, FH, FY	OTHER_LIABILITIES_TOTAL
Other receivables	FQ, FH, FY	OTHER_RECEIVABLES
Other short term debt	FY	OTHER_SHORT_TERM_DEBT
Paid in capital	FQ, FH, FY	PAID_IN_CAPITAL
Preferred stock, carrying value	FQ, FH, FY	PREFERRED_STOCK_CARRYING_VALUE
Prepaid expenses	FQ, FH, FY	PREPAID_EXPENSES
Provision for risks & charge	FQ, FH, FY	PROVISION_F_RISKS
Retained earnings	FQ, FH, FY	RETAINED_EARNINGS
Shareholders’ equity	FQ, FH, FY	SHRHLDRS_EQUITY
Short term debt	FQ, FH, FY	SHORT_TERM_DEBT
Short term debt excl. current portion of LT debt	FQ, FH, FY	SHORT_TERM_DEBT_EXCL_CURRENT_PORT
Short term investments	FQ, FH, FY	SHORT_TERM_INVEST
Tangible book value per share	FQ, FH, FY	BOOK_TANGIBLE_PER_SHARE
Total assets		TOTAL_ASSETS

Financial	period	financial_id
	FQ, FH, FY	
Total current assets	FQ, FH, FY	TOTAL_CURRENT_ASSETS
Total current liabilities	FQ, FH, FY	TOTAL_CURRENT_LIABILITIES
Total debt	FQ, FH, FY	TOTAL_DEBT
Total equity	FQ, FH, FY	TOTAL_EQUITY
Total inventory	FQ, FH, FY	TOTAL_INVENTORY
Total liabilities	FQ, FH, FY	TOTAL_LIABILITIES
Total liabilities & shareholders' equities	FQ, FH, FY	TOTAL_LIABILITIES_SHRHLDRS_EQUITY
Total non-current assets	FQ, FH, FY	TOTAL_NON_CURRENT_ASSETS
Total non-current liabilities	FQ, FH, FY	TOTAL_NON_CURRENT_LIABILITIES
Total receivables, net	FQ, FH, FY	TOTAL_RECEIVABLES_NET
Treasury stock - common	FQ, FH, FY	TREASURY_STOCK_COMMON

Cash flow

This table lists the available metrics that provide information about how cash flows through a company.

Click to show/hide

Financial	period	financial_id
Amortization	FQ, FH, FY, TTM	AMORTIZATION

Financial	period	financial_id
Capital expenditures	FQ, FH, FY, TTM	CAPITAL_EXPENDITURES
Capital expenditures - fixed assets	FQ, FH, FY, TTM	CAPITAL_EXPENDITURES_FIXED_ASSETS
Capital expenditures - other assets	FQ, FH, FY, TTM	CAPITAL_EXPENDITURES_OTHER_ASSETS
Cash from financing activities	FQ, FH, FY, TTM	CASH_F_FINANCING_ACTIVITIES
Cash from investing activities	FQ, FH, FY, TTM	CASH_F_INVESTING_ACTIVITIES
Cash from operating activities	FQ, FH, FY, TTM	CASH_F_OPERATING_ACTIVITIES
Change in accounts payable	FQ, FH, FY, TTM	CHANGE_IN_ACCOUNTS_PAYABLE
Change in accounts receivable	FQ, FH, FY, TTM	CHANGE_IN_ACCOUNTS_RECEIVABLE
Change in accrued expenses	FQ, FH, FY, TTM	CHANGE_IN_ACCRUED_EXPENSES
Change in inventories	FQ, FH, FY, TTM	CHANGE_IN_INVENTORIES
Change in other assets/liabilities	FQ, FH, FY, TTM	CHANGE_IN_OTHER_ASSETS
Change in taxes payable	FQ, FH, FY, TTM	CHANGE_IN_TAXES_PAYABLE
Changes in working capital	FQ, FH, FY, TTM	CHANGES_IN_WORKING_CAPITAL
Common dividends paid	FQ, FH, FY, TTM	COMMON_DIVIDENDS_CASH_FLOW
Deferred taxes (cash flow)	FQ, FH, FY, TTM	CASH_FLOW_DEFERRED_TAXES
	FQ, FH, FY, TTM	CASH_FLOW_DEPRECATION_N_AMORTIZATION

Financial	period	financial_id
Depreciation & amortization (cash flow)		
Depreciation/ depletion	FQ, FH, FY, TTM	DEPRECIATION_DEPLETION
Financing activities - other sources	FQ, FH, FY, TTM	OTHER_FINANCING_CASH_FLOW_SOURCES
Financing activities - other uses	FQ, FH, FY, TTM	OTHER_FINANCING_CASH_FLOW_USES
Free cash flow	FQ, FH, FY, TTM	FREE_CASH_FLOW
Funds from operations	FQ, FH, FY, TTM	FUNDS_F_OPERATIONS
Investing activities - other sources	FQ, FH, FY, TTM	OTHER_INVESTING_CASH_FLOW_SOURCES
Investing activities - other uses	FQ, FH, FY	OTHER_INVESTING_CASH_FLOW_USES
Issuance of long term debt	FQ, FH, FY, TTM	SUPPLYING_OF_LONG_TERM_DEBT
Issuance/retirement of debt, net	FQ, FH, FY, TTM	ISSUANCE_OF_DEBT_NET
Issuance/retirement of long term debt	FQ, FH, FY, TTM	ISSUANCE_OF_LONG_TERM_DEBT
Issuance/retirement of other debt	FQ, FH, FY, TTM	ISSUANCE_OF_OTHER_DEBT
Issuance/retirement of short term debt	FQ, FH, FY, TTM	ISSUANCE_OF_SHORT_TERM_DEBT
Issuance/retirement of stock, net	FQ, FH, FY, TTM	ISSUANCE_OF_STOCK_NET
Net income (cash flow)	FQ, FH, FY, TTM	NET_INCOME_STARTING_LINE
Non-cash items	FQ, FH, FY, TTM	NON_CASH_ITEMS
		OTHER_FINANCING_CASH_FLOW_ITEMS_TOTAL

Financial	period	financial_id
Other financing cash flow items, total	FQ, FH, FY, TTM	
Other investing cash flow items, total	FQ, FH, FY	OTHER_INVESTING_CASH_FLOW_ITEMS_TOTAL
Preferred dividends paid	FQ, FH, FY	PREFERRED_DIVIDENDS_CASH_FLOW
Purchase of investments	FQ, FH, FY, TTM	PURCHASE_OF_INVESTMENTS
Purchase/ acquisition of business	FQ, FH, FY, TTM	PURCHASE_OF_BUSINESS
Purchase/sale of business, net	FQ, FH, FY	PURCHASE_SALE_BUSINESS
Purchase/sale of investments, net	FQ, FH, FY, TTM	PURCHASE_SALE_INVESTMENTS
Reduction of long term debt	FQ, FH, FY, TTM	REDUCTION_OF_LONG_TERM_DEBT
Repurchase of common & preferred stock	FQ, FH, FY, TTM	PURCHASE_OF_STOCK
Sale of common & preferred stock	FQ, FH, FY, TTM	SALE_OF_STOCK
Sale of fixed assets & businesses	FQ, FH, FY, TTM	SALES_OF_BUSINESS
Sale/maturity of investments	FQ, FH, FY	SALES_OF_INVESTMENTS
Total cash dividends paid	FQ, FH, FY, TTM	TOTAL_CASH_DIVIDENDS_PAID

Statistics

This table contains a variety of statistical metrics, including commonly used financial ratios.

Click to show/hide

Financial	period	financial_id
Accruals	FQ, FH, FY	ACCRUALS_RATIO
Altman Z-score	FQ, FH, FY	ALTMAN_Z_SCORE
Asset turnover	FQ, FH, FY	ASSET_TURNOVER
Beneish M-score	FQ, FH, FY	BENEISH_M_SCORE
Buyback yield %	FQ, FH, FY	BUYBACK_YIELD
COGS to revenue ratio	FQ, FH, FY	COGS_TO_REVENUE
Cash conversion cycle	FQ, FY	CASH_CONVERSION_CYCLE
Cash to debt ratio	FQ, FH, FY	CASH_TO_DEBT
Current ratio	FQ, FH, FY	CURRENT_RATIO
Days inventory	FQ, FY	DAYS_INVENT
Days payable	FQ, FY	DAYS_PAY
Days sales outstanding	FQ, FY	DAY_SALES_OUT
Debt to EBITDA ratio	FQ, FH, FY	DEBT_TO_EBITDA
Debt to assets ratio	FQ, FH, FY	DEBT_TO_ASSET

Financial	period	financial_id
Debt to equity ratio	FQ, FH, FY	DEBT_TO_EQUITY
Debt to revenue ratio	FQ, FH, FY	DEBT_TO_REVENUE
Dividend payout ratio %	FQ, FH, FY, TTM	DIVIDEND_PAYOUT_RATIO
Dividend yield %	FQ, FH, FY	DIVIDENDS_YIELD
Dividends per share - common stock primary issue	FQ, FH, FY, TTM	DPS_COMMON_STOCK_PRIM_ISSUE
EBITDA margin %	FQ, FH, FY, TTM	EBITDA_MARGIN
EPS basic one year growth	FQ, FH, FY, TTM	EARNINGS_PER_SHARE_BASIC_ONE_YEAR_GROWTH
EPS diluted one year growth	FQ, FH, FY	EARNINGS_PER_SHARE_DILUTED_ONE_YEAR_GROWTH
EPS estimates	FQ, FH, FY	EARNINGS_ESTIMATE
Effective interest rate on debt %	FQ, FH, FY	EFFECTIVE_INTEREST_RATE_ON_DEBT
Enterprise value	FQ, FH, FY	ENTERPRISE_VALUE
Enterprise value to EBIT ratio	FQ, FH, FY	EV_EBIT
	FQ, FH, FY	ENTERPRISE_VALUE_EBITDA

Financial	period	financial_id
Enterprise value to EBITDA ratio		
Enterprise value to revenue ratio	FQ, FH, FY	EV_REVENUE
Equity to assets ratio	FQ, FH, FY	EQUITY_TO_ASSET
Float shares outstanding	FY	FLOAT_SHARES_OUTSTANDING
Free cash flow margin %	FQ, FH, FY	FREE_CASH_FLOW_MARGIN
Fulmer H factor	FQ, FY	FULMER_H_FACTOR
Goodwill to assets ratio	FQ, FH, FY	GOODWILL_TO_ASSET
Graham’s number	FQ, FY	GRAHAM_NUMBERS
Gross margin %	FQ, FH, FY, TTM	GROSS_MARGIN
Gross profit to assets ratio	FQ, FY	GROSS_PROFIT_TO_ASSET
Interest coverage	FQ, FH, FY	INTERST_COVER
Inventory to revenue ratio	FQ, FH, FY	INVENT_TO_REVENUE
Inventory turnover	FQ, FH, FY	INVENT_TURNOVER
KZ index	FY	KZ_INDEX
Long term debt to total assets ratio	FQ, FH, FY	LONG_TERM_DEBT_TO_ASSETS
	FQ, FY	NCAVPS_RATIO

Financial	period	financial_id
Net current asset value per share		
Net income per employee	FY	NET_INCOME_PER_EMPLOYEE
Net margin %	FQ, FH, FY, TTM	NET_MARGIN
Number of employees	FY	NUMBER_OF_EMPLOYEES
Operating earnings yield %	FQ, FH, FY	OPERATING_EARNINGS_YIELD
Operating margin %	FQ, FH, FY	OPERATING_MARGIN
PEG ratio	FQ, FY	PEG_RATIO
Piotroski F-score	FQ, FH, FY	PIOTROSKI_F_SCORE
Price earnings ratio forward	FQ, FY	PRICE_EARNINGS_FORWARD
Price sales ratio forward	FQ, FY	PRICE_SALES_FORWARD
Quality ratio	FQ, FH, FY	QUALITY_RATIO
Quick ratio	FQ, FH, FY	QUICK_RATIO
Research & development to revenue ratio	FQ, FH, FY	RESEARCH_AND_DEVELOP_TO_REVENUE
Return on assets %	FQ, FH, FY	RETURN_ON_ASSETS
	FQ, FH, FY	RETURN_ON_COMMON_EQUITY

Financial	period	financial_id
Return on common equity %		
Return on equity %	FQ, FH, FY	RETURN_ON_EQUITY
Return on equity adjusted to book value %	FQ, FH, FY	RETURN_ON_EQUITY_ADJUST_TO_BOOK
Return on invested capital %	FQ, FH, FY	RETURN_ON_INVESTED_CAPITAL
Return on tangible assets %	FQ, FH, FY	RETURN_ON_TANG_ASSETS
Return on tangible equity %	FQ, FH, FY	RETURN_ON_TANG_EQUITY
Revenue estimates	FQ, FH, FY	SALES_ESTIMATES
Revenue one year growth	FQ, FH, FY, TTM	REVENUE_ONE_YEAR_GROWTH
Revenue per employee	FY	REVENUE_PER_EMPLOYEE
Shares buyback ratio %	FQ, FH, FY	SHARE_BUYBACK_RATIO
Sloan ratio %	FQ, FH, FY	SLOAN_RATIO
Springate score	FQ, FY	SPRINGATE_SCORE
Sustainable growth rate	FQ, FY	SUSTAINABLE_GROWTH_RATE

Financial	period	financial_id
Tangible common equity ratio	FQ, FH, FY	TANGIBLE_COMMON_EQUITY_RATIO
Tobin’s Q (approximate)	FQ, FH, FY	TOBIN_Q_RATIO
Total common shares outstanding	FQ, FH, FY	TOTAL_SHARES_OUTSTANDING
Zmijewski score	FQ, FY	ZMIJEWSKI_SCORE

request.economic()

The request.economic() function provides scripts with the ability to retrieve economic data for a specified country or region, including information about the state of the economy (GDP, inflation rate, etc.) or of a particular industry (steel production, ICU beds, etc.).

Below is the signature for this function:

```
request.economic(country_code, field, gaps, ignore_invalid_symbol) → s
```

The `country_code` parameter accepts a “string” value representing the identifier of the country or region to request economic data for (e.g., “US”, “EU”, etc.). See the Country/region codes section for a complete list of codes this function supports. Note that the economic metrics available depend on the country or region specified in the function call.

The `field` parameter accepts a “string” specifying the metric that the function requests. The Field codes section covers all accessible metrics and the countries/regions they’re available for.

For a detailed explanation on the last two parameters of this function, see the Common characteristics section at the top of this page.

This simple example requests the growth rate of the Gross Domestic Product (“GDPQQ”) for the United States (“US”) using request.economic(), then plots its value on the chart with a gradient color:

image

//@version=6 indicator("Requesting economic data demo") //@variable The GDP growth rate for the US economy. float gdpqq = request.economic("US", "GDPQQ") //@variable The all-time maximum growth rate. float maxRate = ta.max(gdpqq) //@variable The all-time minimum growth rate. float minRate = ta.min(gdpqq) //@variable The color of the `gdpqq` plot. color rateColor = switch gdpqq >= 0 => color.from_gradient(gdpqq, 0, maxRate, color.purple, color.blue) => color.from_gradient(gdpqq, minRate, 0, color.red, color.purple) // Plot the results. plot(gdpqq, "US GDP Growth Rate", rateColor, style = plot.style_area)

Note that:

- This example does not include a gaps argument in the request.economic() call, so the function uses the default barmerge.gaps_off. In other words, it returns the last retrieved value when new data isn't yet available.

Country/region codes

The table in this section lists all country/region codes available for use with request.economic(). The first column of the table contains the "string" values that represent the country or region code, and the second column contains the corresponding country/region names.

It's important to note that the value used as the `country_code` argument determines which field codes are accessible to the function.

Click to show/hide

`country_code`	Country/region name
AF	Afghanistan
AL	Albania
DZ	Algeria
AD	Andorra
AO	Angola
AG	Antigua and Barbuda
AR	Argentina

country_code	Country/region name
AM	Armenia
AW	Aruba
AU	Australia
AT	Austria
AZ	Azerbaijan
BS	Bahamas
BH	Bahrain
BD	Bangladesh
BB	Barbados
BY	Belarus
BE	Belgium
BZ	Belize
BJ	Benin
BM	Bermuda
BT	Bhutan
BO	Bolivia
BA	Bosnia and Herzegovina
BW	Botswana
BR	Brazil
BN	Brunei
BG	Bulgaria
BF	Burkina Faso
BI	Burundi
KH	Cambodia
CM	Cameroon
CA	Canada
CV	Cape Verde
KY	Cayman Islands
CF	Central African Republic
TD	Chad
CL	Chile

country_code	Country/region name
CN	China
CO	Colombia
KM	Comoros
CG	Congo
CR	Costa Rica
HR	Croatia
CU	Cuba
CY	Cyprus
CZ	Czech Republic
DK	Denmark
DJ	Djibouti
DM	Dominica
DO	Dominican Republic
TL	East Timor
EC	Ecuador
EG	Egypt
SV	El Salvador
GQ	Equatorial Guinea
ER	Eritrea
EE	Estonia
ET	Ethiopia
EU	Euro area
FO	Faroe Islands
FJ	Fiji
FI	Finland
FR	France
GA	Gabon
GM	Gambia
GE	Georgia
DE	Germany
GH	Ghana

country_code	Country/region name
GR	Greece
GL	Greenland
GD	Grenada
GT	Guatemala
GN	Guinea
GW	Guinea Bissau
GY	Guyana
HT	Haiti
HN	Honduras
HK	Hong Kong
HU	Hungary
IS	Iceland
IN	India
ID	Indonesia
IR	Iran
IQ	Iraq
IE	Ireland
IM	Isle of Man
IL	Israel
IT	Italy
CI	Ivory Coast
JM	Jamaica
JP	Japan
JO	Jordan
KZ	Kazakhstan
KE	Kenya
KI	Kiribati
XK	Kosovo
KW	Kuwait
KG	Kyrgyzstan
LA	Laos

country_code	Country/region name
LV	Latvia
LB	Lebanon
LS	Lesotho
LR	Liberia
LY	Libya
LI	Liechtenstein
LT	Lithuania
LU	Luxembourg
MO	Macau
MK	Macedonia
MG	Madagascar
MW	Malawi
MY	Malaysia
MV	Maldives
ML	Mali
MT	Malta
MR	Mauritania
MU	Mauritius
MX	Mexico
MD	Moldova
MC	Monaco
MN	Mongolia
ME	Montenegro
MA	Morocco
MZ	Mozambique
MM	Myanmar
NA	Namibia
NP	Nepal
NL	Netherlands
NC	New Caledonia
NZ	New Zealand

country_code	Country/region name
NI	Nicaragua
NE	Niger
NG	Nigeria
KP	North Korea
NO	Norway
OM	Oman
PK	Pakistan
PS	Palestine
PA	Panama
PG	Papua New Guinea
PY	Paraguay
PE	Peru
PH	Philippines
PL	Poland
PT	Portugal
PR	Puerto Rico
QA	Qatar
CD	Republic of the Congo
RO	Romania
RU	Russia
RW	Rwanda
WS	Samoa
SM	San Marino
ST	Sao Tome and Principe
SA	Saudi Arabia
SN	Senegal
RS	Serbia
SC	Seychelles
SL	Sierra Leone
SG	Singapore
SK	Slovakia

country_code	Country/region name
SI	Slovenia
SB	Solomon Islands
SO	Somalia
ZA	South Africa
KR	South Korea
SS	South Sudan
ES	Spain
LK	Sri Lanka
LC	St Lucia
VC	St Vincent and the Grenadines
SD	Sudan
SR	Suriname
SZ	Swaziland
SE	Sweden
CH	Switzerland
SY	Syria
TW	Taiwan
TJ	Tajikistan
TZ	Tanzania
TH	Thailand
TG	Togo
TO	Tonga
TT	Trinidad and Tobago
TN	Tunisia
TR	Turkey
TM	Turkmenistan
UG	Uganda
UA	Ukraine
AE	United Arab Emirates
GB	United Kingdom
US	United States

country_code	Country/region name
UY	Uruguay
UZ	Uzbekistan
VU	Vanuatu
VE	Venezuela
VN	Vietnam
YE	Yemen
ZM	Zambia
ZW	Zimbabwe

Field codes

The table in this section lists the field codes available for use with request.economic(). The first column contains the “string” values used as the `field` argument, and the second column contains names of each metric and links to our Help Center with additional information, including the countries/regions they’re available for.

Click to show/hide

field	Metric
AA	Asylum Applications
ACR	API Crude Runs
AE	Auto Exports
AHE	Average Hourly Earnings
AHO	API Heating Oil
AWH	Average Weekly Hours
BBS	Banks Balance Sheet
BCLI	Business Climate Indicator
BCOI	Business Confidence Index
BI	Business Inventories
BLR	Bank Lending Rate
BOI	NFIB Business Optimism Index
BOT	Balance Of Trade
BP	Building Permits

field	Metric
BR	Bankruptcies
CA	Current Account
CAG	Current Account To GDP
CAP	Car Production
CAR	Car Registrations
CBBS	Central Bank Balance Sheet
CCC	Claimant Count Change
CCI	Consumer Confidence Index
CCOS	Cushing Crude Oil Stocks
CCP	Core Consumer Prices
CCPI	Core CPI
CCPT	Consumer Confidence Price Trends
CCR	Consumer Credit
CCS	Credit Card Spending
CEP	Cement Production
CF	Capital Flows
CFNAI	Chicago Fed National Activity Index
CI	API Crude Imports
CIND	Coincident Index
CIR	Core Inflation Rate, YoY
CJC	Continuing Jobless Claims
CN	API Cushing Number
COI	Crude Oil Imports
COIR	Crude Oil Imports from Russia
CONSTS	Construction Spending
COP	Crude Oil Production
COR	Crude Oil Rigs
CORD	Construction Orders, YoY
CORPI	Corruption Index
CORR	Corruption Rank
COSC	Crude Oil Stocks Change

field	Metric
COUT	Construction Output, YoY
CP	Copper Production
CPCEPI	Core PCE Price Index
CPI	Consumer Price Index
CPIHU	CPI Housing Utilities
CPIM	CPI Median
CPIT	CPI Transportation
CPITM	CPI Trimmed Mean
CPMI	Chicago PMI
CPPI	Core Producer Price Index
CPR	Corporate Profits
CRLPI	Cereals Price Index
CRR	Cash Reserve Ratio
CS	Consumer Spending
CSC	API Crude Oil Stock Change
CSHPI	Case Shiller Home Price Index
CSHPIMM	Case Shiller Home Price Index, MoM
CSHPIYY	Case Shiller Home Price Index, YoY
CSS	Chain Store Sales
CTR	Corporate Tax Rate
CU	Capacity Utilization
DFMI	Dallas Fed Manufacturing Index
DFP	Distillate Fuel Production
DFS	Distillate Stocks
DFSI	Dallas Fed Services Index
DFSRI	Dallas Fed Services Revenues Index
DG	Deposit Growth
DGO	Durable Goods Orders
DGOED	Durable Goods Orders Excluding Defense
DGOET	Durable Goods Orders Excluding Transportation
DIR	Deposit Interest Rate

field	Metric
DPI	Disposable Personal Income
DRPI	Dairy Price Index
DS	API Distillate Stocks
DT	CBI Distributive Trades
EC	ADP Employment Change
ED	External Debt
EDBR	Ease Of Doing Business Ranking
EHS	Existing Home Sales
ELP	Electricity Production
EMC	Employment Change
EMCI	Employment Cost Index
EMP	Employed Persons
EMR	Employment Rate
EOI	Economic Optimism Index
EP	Export Prices
ESI	ZEW Economic Sentiment Index
EWS	Economy Watchers Survey
EXP	Exports
EXPYY	Exports, YoY
FAI	Fixed Asset Investment
FBI	Foreign Bond Investment
FDI	Foreign Direct Investment
FE	Fiscal Expenditure
FER	Foreign Exchange Reserves
FI	Food Inflation, YoY
FO	Factory Orders
FOET	Factory Orders Excluding Transportation
FPI	Food Price Index
FSI	Foreign Stock Investment
FTE	Full Time Employment
FYGDPG	Full Year GDP Growth

field	Metric
GASP	Gasoline Prices
GBP	Government Budget
GBV	Government Budget Value
GCI	Competitiveness Index
GCR	Competitiveness Rank
GD	Government Debt
GDG	Government Debt To GDP
GDP	Gross Domestic Product
GDPA	GDP From Agriculture
GDPC	GDP From Construction
GDPCP	GDP Constant Prices
GDPD	GDP Deflator
GDPGA	GDP Growth Annualized
GDPMAN	GDP From Manufacturing
GDPMIN	GDP From Mining
GDPPA	GDP From Public Administration
GDPPC	GDP Per Capita
GDPPCP	GDP Per Capita, PPP
GDPQQ	GDP Growth Rate
GDPS	GDP From Services
GDPSA	GDP Sales
GDPT	GDP From Transport
GDPU	GDP From Utilities
GDPYY	GDP, YoY
GDTPI	Global Dairy Trade Price Index
GFCF	Gross Fixed Capital Formation
GNP	Gross National Product
GP	Gold Production
GPA	Government Payrolls
GPRO	Gasoline Production
GR	Government Revenues

field	Metric
GRES	Gold Reserves
GS	API Gasoline Stocks
GSC	Grain Stocks Corn
GSCH	Gasoline Stocks Change
GSG	Government Spending To GDP
GSP	Government Spending
GSS	Grain Stocks Soy
GSW	Grain Stocks Wheat
GTB	Goods Trade Balance
HB	Hospital Beds
HDG	Households Debt To GDP
HDI	Households Debt To Income
HICP	Harmonised Index of Consumer Prices
HIRMM	Harmonised Inflation Rate, MoM
HIRYY	Harmonised Inflation Rate, YoY
HMI	NAHB Housing Market Index
HOR	Home Ownership Rate
HOS	Heating Oil Stocks
HOSP	Hospitals
HPI	House Price Index
HPIMM	House Price Index, MoM
HPIYY	House Price Index, YoY
HS	Home Loans
HSP	Household Spending
HST	Housing Starts
IC	Changes In Inventories
ICUB	ICU Beds
IE	Inflation Expectations
IFOCC	IFO Assessment Of The Business Situation
IFOE	IFO Business Developments Expectations
IJC	Initial Jobless Claims

field	Metric
IMP	Imports
IMPYY	Imports, YoY
INBR	Interbank Rate
INTR	Interest Rate
IPA	IP Addresses
IPMM	Industrial Production, MoM
IPRI	Import Prices
IPYY	Industrial Production, YoY
IRMM	Inflation Rate, MoM
IRYY	Inflation Rate, YoY
IS	Industrial Sentiment
ISP	Internet Speed
JA	Job Advertisements
JAR	Jobs To Applications Ratio
JC	Challenger Job Cuts
JC4W	Jobless Claims, 4-Week Average
JO	Job Offers
JV	Job Vacancies
KFMI	Kansas Fed Manufacturing Index
LB	Loans To Banks
LC	Labor Costs
LEI	Leading Economic Index
LFPR	Labor Force Participation Rate
LG	Loan Growth, YoY
LIVRR	Liquidity Injections Via Reverse Repo
LMIC	LMI Logistics Managers Index Current
LMICI	LMI Inventory Costs
LMIF	LMI Logistics Managers Index Future
LMITP	LMI Transportation Prices
LMIWP	LMI Warehouse Prices
LPS	Loans To Private Sector

field	Metric
LR	Central Bank Lending Rate
LTUR	Long Term Unemployment Rate
LWF	Living Wage Family
LWI	Living Wage Individual
M0	Money Supply M0
M1	Money Supply M1
M2	Money Supply M2
M3	Money Supply M3
MA	Mortgage Approvals
MAPL	Mortgage Applications
MCE	Michigan Consumer Expectations
MCEC	Michigan Current Economic Conditions
MD	Medical Doctors
ME	Military Expenditure
MGDPYY	Monthly GDP, YoY
MIE1Y	Michigan Inflation Expectations
MIE5Y	Michigan 5 Year Inflation Expectations
MIP	Mining Production, YoY
MMI	MBA Mortgage Market Index
MO	Machinery Orders
MP	Manufacturing Payrolls
MPI	Meat Price Index
MPRMM	Manufacturing Production, MoM
MPRYY	Manufacturing Production, YoY
MR	Mortgage Rate
MRI	MBA Mortgage Refinance Index
MS	Manufacturing Sales
MTO	Machine Tool Orders
MW	Minimum Wages
NDCGOEA	Orders For Non-defense Capital Goods Excluding Aircraft
NEGTB	Goods Trade Deficit With Non-EU Countries

field	Metric
NFP	Nonfarm Payrolls
NGI	Natural Gas Imports
NGIR	Natural Gas Imports from Russia
NGSC	Natural Gas Stocks Change
NHPI	Nationwide House Price Index
NHS	New Home Sales
NHSMM	New Home Sales, MoM
NMPMI	Non-Manufacturing PMI
NO	New Orders
NODXMM	Non-Oil Domestic Exports, MoM
NODXYY	Non-Oil Domestic Exports, YoY
NOE	Non-Oil Exports
NPP	Nonfarm Payrolls Private
NURS	Nurses
NYESMI	NY Empire State Manufacturing Index
OE	Oil Exports
OPI	Oils Price Index
PCEPI	PCE Price Index
PDG	Private Debt To GDP
PFMI	Philadelphia Fed Manufacturing Index
PHSIMM	Pending Home Sales Index, MoM
PHSIYY	Pending Home Sales Index, YoY
PI	Personal Income
PIN	Private Investment
PIND	MBA Purchase Index
PITR	Personal Income Tax Rate
POP	Population
PPI	Producer Price Index
PPII	Producer Price Index Input
PPIMM	Producer Price Inflation, MoM
PPIYY	Producer Prices Index, YoY

field	Metric
PRI	API Product Imports
PROD	Productivity
PS	Personal Savings
PSC	Private Sector Credit
PSP	Personal Spending
PTE	Part Time Employment
PUAC	Pandemic Unemployment Assistance Claims
RAM	Retirement Age Men
RAW	Retirement Age Women
RCR	Refinery Crude Runs
REM	Remittances
RFMI	Richmond Fed Manufacturing Index
RFMSI	Richmond Fed Manufacturing Shipments Index
RFSI	Richmond Fed Services Index
RI	Redbook Index
RIEA	Retail Inventories Excluding Autos
RPI	Retail Price Index
RR	Repo Rate
RRR	Reverse Repo Rate
RSEA	Retail Sales Excluding Autos
RSEF	Retail Sales Excluding Fuel
RSMM	Retail Sales, MoM
RSYY	Retail Sales, YoY
RTI	Reuters Tankan Index
SBSI	Small Business Sentiment Index
SFHP	Single Family Home Prices
SP	Steel Production
SPI	Sugar Price Index
SS	Services Sentiment
SSR	Social Security Rate
SSRC	Social Security Rate For Companies

field	Metric
SSRE	Social Security Rate For Employees
STR	Sales Tax Rate
TA	Tourist Arrivals
TAXR	Tax Revenue
TCB	Treasury Cash Balance
TCPI	Tokyo CPI
TI	Terrorism Index
TII	Tertiary Industry Index
TOT	Terms Of Trade
TR	Tourism Revenues
TVS	Total Vehicle Sales
UC	Unemployment Change
UP	Unemployed Persons
UR	Unemployment Rate
WAG	Wages
WES	Weapons Sales
WG	Wage Growth, YoY
WHS	Wages High Skilled
WI	Wholesale Inventories
WLS	Wages Low Skilled
WM	Wages In Manufacturing
WPI	Wholesale Price Index
WS	Wholesale Sales
YUR	Youth Unemployment Rate
ZCC	ZEW Current Conditions

request.seed()

TradingView aggregates a vast amount of data from its many providers, including price and volume information on tradable instruments, financials, economic data, and more, which users can retrieve in Pine Script using the

functions discussed in the sections above, as well as multiple built-in variables.

To further expand the horizons of possible data one can analyze on TradingView, we have Pine Seeds, which allows users to supply custom *user-maintained* EOD data feeds via GitHub for use on TradingView charts and within Pine Script code.

To retrieve data from a Pine Seeds data feed within a script, use the request.seed() function. Below is the function's signature:

```
request.seed(source, symbol, expression, ignore_invalid_symbol, calc_b
```

The `source` parameter specifies the unique name of the user-maintained GitHub repository that contains the data feed.

The `symbol` parameter represents the file name from the "data/" directory of the `source` repository, excluding the ".csv" file extension. See this page for information about the structure of the data stored in repositories.

The `expression` parameter is the series to evaluate using data extracted from the requested context. It is similar to the equivalent in request.security() and request.security_lower_tf(). Data feeds stored in user-maintained repos contain time, open, high, low, close, and volume information, meaning the `expression` argument can use the corresponding built-in variables, including variables derived from them (e.g., bar_index, ohlc4, etc.) to request their values from the context of the custom data.

The script below visualizes sample data from the seed_crypto_santiment demo repository. It uses two calls to request.seed() to retrieve the close values from the repository's BTC_SENTIMENT_POSITIVE_TOTAL and BTC_SENTIMENT_NEGATIVE_TOTAL data feeds and plots the results on the chart as step lines:

image

//@version=6 indicator("Pine Seeds demo", format=format.volume) // @variable The total positive sentiment for BTC extracted from the "seed_crypto_santiment" repository. float positiveTotal = request.seed("seed_crypto_santiment", "BTC_SENTIMENT_POSITIVE_TOTAL", close) //@variable The total negative sentiment for BTC extracted from the

"seed_crypto_santiment" repository. float negativeTotal = request.seed("seed_crypto_santiment", "BTC_SENTIMENT_NEGATIVE_TOTAL", close) // Plot the data. plot(positiveTotal, "Positive sentiment", color.teal, 2, plot.style_stepline) plot(negativeTotal, "Negative sentiment", color.maroon, 2, plot.style_stepline)

Note that:

- This example requests data from the repository highlighted in the Pine Seeds documentation. It exists solely for example purposes, and its data *does not* update on a regular basis.
- Unlike most other `request.*()` functions, request.seed() does not have a gaps parameter. It always returns na values when no new data exists.
- Pine Seeds data is searchable from the chart's symbol search bar. To load a data feed on the chart, enter the *"Repo:File" pair*, similar to searching for an "Exchange:Symbol" pair.

Previous

Next

Plots

Pine Script® v6 Documentation

Plots

Introduction

The plot() function is the most frequently used function used to display information calculated using Pine scripts. It is versatile and can plot different styles of lines, histograms, areas, columns (like volume columns), fills, circles or crosses.

The use of plot() to create fills is explained in the page on Fills.

This script showcases a few different uses of plot() in an overlay script:

image

//@version=6 indicator("`plot()`", "", true) plot(high, "Blue `high` line") plot(math.avg(close, open), "Crosses in body center", close > open ? color.lime : color.purple, 6, plot.style_cross) plot(math.min(open, close), "Navy step line on body low point", color.navy, 3, plot.style_stepline) plot(low, "Gray dot on `low`", color.gray, 3, plot.style_circles) color VIOLET = #AA00FF color GOLD = #CCCC00 ma = ta.alma(hl2, 40, 0.85, 6) var almaColor = color.silver almaColor := ma > ma[2] ? GOLD : ma < ma[2] ? VIOLET : almaColor plot(ma, "Two-color ALMA", almaColor, 2)

Note that:

- The first plot() call plots a 1-pixel blue line across the bar highs.
- The second plots crosses at the mid-point of bodies. The crosses are colored lime when the bar is up and purple when it is down. The argument used for `linewidth` is 6 but it is not a pixel value; just a relative size.

- The third call plots a 3-pixel wide step line following the low point of bodies.
- The fourth call plot a gray circle at the bars' low.
- The last plot requires some preparation. We first define our bull/bear colors, calculate an Arnaud Legoux Moving Average, then make our color calculations. We initialize our color variable on bar zero only, using var. We initialize it to color.silver, so on the dataset's first bars, until one of our conditions causes the color to change, the line will be silver. The conditions that change the color of the line require it to be higher/lower than its value two bars ago. This makes for less noisy color transitions than if we merely looked for a higher/lower value than the previous one.

This script shows other uses of plot() in a pane:

image

//@version=6 indicator("Volume change", format = format.volume) color GREEN = #008000 color GREEN_LIGHT = color.new(GREEN, 50) color GREEN_LIGHTER = color.new(GREEN, 85) color PINK = #FF0080 color PINK_LIGHT = color.new(PINK, 50) color PINK_LIGHTER = color.new(PINK, 90) bool barUp = ta.rising(close, 1) bool barDn = ta.falling(close, 1) float volumeChange = ta.change(volume) volumeColor = barUp ? GREEN_LIGHTER : barDn ? PINK_LIGHTER : color.gray plot(volume, "Volume columns", volumeColor, style = plot.style_columns) volumeChangeColor = barUp ? volumeChange > 0 ? GREEN : GREEN_LIGHT : volumeChange > 0 ? PINK : PINK_LIGHT plot(volumeChange, "Volume change columns", volumeChangeColor, 12, plot.style_histogram) plot(0, "Zero line", color.gray)

Note that:

- We are plotting normal volume values as wide columns above the zero line (see the `style = plot.style_columns` in our plot() call).
- Before plotting the columns we calculate our `volumeColor` by using the values of the `barUp` and `barDn` boolean variables. They become respectively `true` when the current bar's close is higher/lower than the previous one. Note that the "Volume" built-in does not use the same condition; it identifies an up bar with `close > open`. We use the `GREEN_LIGHTER` and `PINK_LIGHTER` colors for the volume columns.

- Because the first plot plots columns, we do not use the `linewidth` parameter, as it has no effect on columns.
- Our script's second plot is the **change** in volume, which we have calculated earlier using `ta.change(volume)`. This value is plotted as a histogram, for which the `linewidth` parameter controls the width of the column. We make this width 12 so that histogram elements are thinner than the columns of the first plot. Positive/negative `volumeChange` values plot above/below the zero line; no manipulation is required to achieve this effect.
- Before plotting the histogram of `volumeChange` values, we calculate its color value, which can be one of four different colors. We use the bright `GREEN` or `PINK` colors when the bar is up/down AND the volume has increased since the last bar (`volumeChange > 0`). Because `volumeChange` is positive in this case, the histogram's element will be plotted above the zero line. We use the bright `GREEN_LIGHT` or `PINK_LIGHT` colors when the bar is up/down AND the volume has NOT increased since the last bar. Because `volumeChange` is negative in this case, the histogram's element will be plotted below the zero line.
- Finally, we plot a zero line. We could just as well have used `hline(0)` there.
- We use `format = format.volume` in our indicator() call so that large values displayed for this script are abbreviated like those of the built-in "Volume" indicator.

plot() calls must always be placed in a line's first position, which entails they are always in the script's global scope. They can't be placed in user-defined functions or structures like if, for, etc. Calls to plot() **can**, however, be designed to plot conditionally in two ways, which we cover in the Plotting conditionally section of this page.

A script can only plot in its own visual space, whether it is in a pane or on the chart as an overlay. Scripts running in a pane can only color bars in the chart area.

plot() parameters

The plot() function has the following signature:

```
plot(series, title, color, linewidth, style, trackprice, histbase, off
```

The parameters of plot() are:

`series`

It is the only mandatory parameter. Its argument must be of “series int/float” type. Note that because the auto-casting rules in Pine Script® convert in the int float bool direction, a “bool” type variable cannot be used as is; it must be converted to an “int” or a “float” for use as an argument. For example, if newDay is of “bool” type, then `newDay ? 1 : 0` can be used to plot 1 when the variable is `true`, and zero when it is `false`.

`title`

Requires a “const string” argument, so it must be known at compile time. The string appears:

- In the script’s scale when the “Chart settings/Scales/Indicator Name Label” field is checked.
- In the Data Window.
- In the “Settings/Style” tab.
- In the dropdown of input.source() fields.
- In the “Condition” field of the “Create Alert” dialog box, when the script is selected.
- As the column header when exporting chart data to a CSV file.

`color`

Accepts “series color”, so can be calculated on the fly, bar by bar. Plotting with na as the color, or any color with a transparency of 100, is one way to hide plots when they are not needed.

`linewidth`

Is the plotted element’s size, but it does not apply to all styles. When a line is plotted, the unit is pixels. It has no impact when plot.style_columns is used.

`style`

The available arguments are:

- plot.style_line (the default): It plots a continous line using the `linewidth` argument in pixels for its width. na values will not plot as a line, but they will be bridged when a value that is not na comes in. Non-na values are only bridged if they are visible on the chart.
- plot.style_linebr: Allows the plotting of discontinuous lines by not plotting on na values, and not joining gaps, i.e., bridging over na values.
- plot.style_stepline: Plots using a staircase effect. Transitions between changes in values are done using a vertical line drawn in middle of bars, as opposed to a point-to-point diagonal joining the midpoints of bars. Can also be used to achieve an effect similar to that of plot.style_linebr, but only if care is taken to plot no color on na values.
- plot.style_area: plots a line of `linewidth` width, filling the area between the line and the `histbase`. The `color` argument is used for both the line and the fill. You can make the line a different color by using another plot() call. Positive values are plotted above the `histbase`, negative values below it.
- plot.style_areabr: This is similar to plot.style_area but it doesn't bridge over na values. Another difference is how the indicator's scale is calculated. Only the plotted values serve in the calculation of the *y* range of the script's visual space. If only high values situated far away from the `histbase` are plotted, for example, those values will be used to calculate the *y* scale of the script's visual space. Positive values are plotted above the `histbase`, negative values below it.
- plot.style_columns: Plots columns similar to those of the "Volume" built-in indicator. The `linewidth` value does **not** affect the width of the columns. Positive values are plotted above the `histbase`, negative values below it. Always includes the value of `histbase` in the *y* scale of the script's visual space.
- plot.style_histogram: Plots columns similar to those of the "Volume" built-in indicator, except that the `linewidth` value is used to determine the width of the histogram's bars in pixels. Note that since `linewidth` requires an "input int" value, the width of the histogram's bars cannot vary bar to bar. Positive values are plotted above the `histbase`, negative values below it. Always includes the value of `histbase` in the *y* scale of the script's visual space.

- plot.style_circles and plot.style_cross: These plot a shape that is not joined across bars unless `join = true` is also used. For these styles, the `linewidth` argument becomes a relative sizing measure — its units are not pixels.

`trackprice`

The default value of this is `false`. When it is `true`, a dotted line made up of small squares will be plotted the full width of the script's visual space. It is often used in conjuction with `show_last = 1, offset = -99999` to hide the actual plot and only leave the residual dotted line.

`histbase`

It is the reference point used with plot.style_area, plot.style_columns and plot.style_histogram. It determines the level separating positive and negative values of the `series` argument. It cannot be calculated dynamically, as an "input int/float" is required.

`offset`

This allows shifting the plot in the past/future using a negative/positive offset in bars. The value cannot change during the script's execution.

`join`

This only affect styles plot.style_circles or plot.style_cross. When `true`, the shapes are joined by a one-pixel line.

`editable`

This boolean parameter controls whether or not the plot's properties can be edited in the "Settings/Style" tab. Its default value is `true`.

`show_last`

Allows control over how many of the last bars the plotted values are visible. An "input int" argument is required, so it cannot be calculated dynamically.

`display`

The default is display.all. When it is set to display.none, plotted values will not affect the scale of the script's visual space. The plot will be invisible and will

not appear in indicator values or the Data Window. It can be useful in plots destined for use as external inputs for other scripts, or for plots used with the `{{plot("[plot_title]")}}` placeholder in alertcondition() calls, e.g.:

//@version=6 indicator("") r = ta.rsi(close, 14) xUp = ta.crossover(r, 50) plot(r, "RSI", display = display.none) alertcondition(xUp, "xUp alert", message = 'RSI is bullish at: {{plot("RSI")}}')

`force_overlay`

If `true`, the plotted results will display on the main chart pane, even when the script occupies a separate pane. Optional. The default is `false`.

Plotting conditionally

plot() calls cannot be used in conditional structures such as if, but they can be controlled by varying their plotted values, or their color. When no plot is required, you can either plot na values, or plot values using na color or any color with 100 transparency (which also makes it invisible).

Value control

One way to control the display of plots is to plot na values when no plot is needed. Sometimes, values returned by functions such as request.security() will return na values, when `gaps = barmerge.gaps_on` is used, for example. In both these cases it is sometimes useful to plot discontinuous lines. This script shows a few ways to do it:

image

//@version=6 indicator("Discontinuous plots", "", true) bool plotValues = bar_index % 3 == 0 plot(plotValues ? high : na, color = color.fuchsia, linewidth = 6, style = plot.style_linebr) plot(plotValues ? high : na) plot(plotValues ? math.max(open, close) : na, color = color.navy, linewidth = 6, style = plot.style_cross) plot(plotValues ? math.min(open, close) : na, color = color.navy, linewidth = 6, style = plot.style_circles) plot(plotValues ? low : na, color = plotValues ? color.green : na, linewidth = 6, style = plot.style_stepline)

Note that:

- We define the condition determining when we plot using `bar_index % 3 == 0`, which becomes `true` when the remainder of the division of the bar index by 3 is zero. This will happen every three bars.
- In the first plot, we use plot.style_linebr, which plots the fuchsia line on highs. It is centered on the bar's horizontal midpoint.
- The second plot shows the result of plotting the same values, but without using special care to break the line. What's happening here is that the thin blue line of the plain plot() call is automatically bridged over na values (or *gaps*), so the plot does not interrupt.
- We then plot navy blue crosses and circles on the body tops and bottoms. The plot.style_circles and plot.style_cross style are a simple way to plot discontinuous values, e.g., for stop or take profit levels, or support & resistance levels.
- The last plot in green on the bar lows is done using plot.style_stepline. Note how its segments are wider than the fuchsia line segments plotted with plot.style_linebr. Also note how on the last bar, it only plots halfway until the next bar comes in.
- The plotting order of each plot is controlled by their order of appearance in the script. See

This script shows how you can restrict plotting to bars after a user-defined date. We use the input.time() function to create an input widget allowing script users to select a date and time, using Jan 1st 2021 as its default value:

//@version=6 indicator("", "", true) startInput = input.time(timestamp("2021-01-01")) plot(time > startInput ? close : na)

Color control

The Conditional coloring section of the page on colors discusses color control for plots. We'll look here at a few examples.

The value of the `color` parameter in plot() can be a constant, such as one of the built-in constant colors or a color literal. In Pine Script, the qualified type of such colors is called **"const color"** (see the Type system page). They are known at compile time:

//@version=6 indicator("", "", true) plot(close, color = color.gray)

The color of a plot can also be determined using information that is only known when the script begins execution on the first historical bar of a chart (bar zero, i.e., `bar_index == 0` or `barstate.isfirst == true`), as will be the case when the information needed to determine a color depends on the chart the script is running on. Here, we calculate a plot color using the syminfo.type built-in variable, which returns the type of the chart's symbol. The qualified type of `plotColor` in this case will be **"simple color"**:

//@version=6 indicator("", "", true) plotColor = switch syminfo.type "stock" => color.purple "futures" => color.red "index" => color.gray "forex" => color.fuchsia "crypto" => color.lime "fund" => color.orange "dr" => color.aqua "cfd" => color.blue plot(close, color = plotColor) printTable(txt) => var table t = table.new(position.middle_right, 1, 1), table.cell(t, 0, 0, txt, bgcolor = color.yellow) printTable(syminfo.type)

Plot colors can also be chosen through a script's inputs. In this case, the `lineColorInput` variable is of the **"input color"** type:

//@version=6 indicator("", "", true) color lineColorInput = input(#1848CC, "Line color") plot(close, color = lineColorInput)

Finally, plot colors can also be *dynamic* values, i.e., calculated values that can change on each bar. These values are of the **"series color"** type:

//@version=6 indicator("", "", true) plotColor = close >= open ? color.lime : color.red plot(close, color = plotColor)

When plotting pivot levels, one common requirement is to avoid plotting level transitions. Using lines is one alternative, but you can also use plot() like this:

image

//@version=6 indicator("Pivot plots", "", true) pivotHigh = fixnan(ta.pivothigh(3,3)) plot(pivotHigh, "High pivot", ta.change(pivotHigh) != 0 ? na : color.olive, 3) plotchar(ta.change(pivotHigh), "ta.change(pivotHigh)", "•", location.top, size = size.small)

Note that:

- We use `pivotHigh = fixnan(ta.pivothigh(3,3))` to hold our pivot values. Because ta.pivothigh() only returns a value when a new pivot is found, we use fixnan() to fill the gaps with the last pivot value returned. The gaps here refer to the na values ta.pivothigh() returns when no new pivot is found.
- Our pivots are detected three bars after they occur because we use the argument 3 for both the `leftbars` and `rightbars` parameters in our ta.pivothigh() call.
- The last plot is plotting a continuous value, but it is setting the plot's color to na when the pivot's value changes, so the plot isn't visible then. Because of this, a visible plot will only appear on the bar following the one where we plotted using na color.
- The blue dot indicates when a new high pivot is detected and no plot is drawn between the preceding bar and that one. Note how the pivot on the bar indicated by the arrow has just been detected in the realtime bar, three bars later, and how no plot is drawn. The plot will only appear on the next bar, making the plot visible **four bars** after the actual pivot.

Levels

Pine Script has an hline() function to plot horizontal lines (see the page on Levels). hline() is useful because it has some line styles unavailable with plot(), but it also has some limitations, namely that it does not accept "series color", and that its `price` parameter requires an "input int/float", so cannot vary during the script's execution.

You can plot levels with plot() in a few different ways. This shows a CCI indicator with levels plotted using plot():

image

//@version=6 indicator("CCI levels with `plot()`") plot(ta.cci(close, 20)) plot(0, "Zero", color.gray, 1, plot.style_circles) plot(bar_index % 2 == 0 ? 100 : na, "100", color.lime, 1, plot.style_linebr) plot(bar_index % 2 == 0 ? -100 : na, "-100", color.fuchsia, 1, plot.style_linebr) plot(200, "200", color.green, 2, trackprice = true, show_last = 1, offset = -99999) plot(-200, "-200", color.red,

2, trackprice = true, show_last = 1, offset = -99999) plot(300, "300", color.new(color.green, 50), 1) plot(-300, "-300", color.new(color.red, 50), 1)

Note that:

- The zero level is plotted using plot.style_circles.
- The 100 levels are plotted using a conditional value that only plots every second bar. In order to prevent the na values from being bridged, we use the plot.style_linebr line style.
- The 200 levels are plotted using `trackprice = true` to plot a distinct pattern of small squares that extends the full width of the script's visual space. The `show_last = 1` in there displays only the last plotted value, which would appear as a one-bar straight line if the next trick wasn't also used: the `offset = -99999` pushes that one-bar segment far away in the past so that it is never visible.
- The 300 levels are plotted using a continuous line, but a lighter transparency is used to make them less prominent.

Offsets

The `offset` parameter specifies the shift used when the line is plotted (negative values shift in the past, positive values shift into the future). For example:

//@version=6 indicator("", "", true) plot(close, color = color.red, offset = -5) plot(close, color = color.lime, offset = 5)

image

As can be seen in the screenshot, the *red* series has been shifted to the left (since the argument's value is negative), while the *green* series has been shifted to the right (its value is positive).

Plot count limit

Each script is limited to a maximum plot count of 64. All `plot*()` calls and alertcondition() calls count in the plot count of a script. Some types of calls count for more than one in the total plot count.

plot() calls count for one in the total plot count if they use a “const color” argument for the `color` parameter, which means it is known at compile time, e.g.:

plot(close, color = color.green)

When they use another qualified type, such as any one of these, they will count for two in the total plot count:

plot(close, color = syminfo.mintick > 0.0001 ? color.green : color.red) // "simple color" plot(close, color = input.color(color.purple)) // "input color" plot(close, color = close > open ? color.green : color.red) // "series color" plot(close, color = color.new(color.silver, close > open ? 40 : 0)) // "series color"

Scale

Not all values can be plotted everywhere. Your script’s visual space is always bound by upper and lower limits that are dynamically adjusted with the values plotted. An RSI indicator will plot values between 0 and 100, which is why it is usually displayed in a distinct *pane* — or area — above or below the chart. If RSI values were plotted as an overlay on the chart, the effect would be to distort the symbol’s normal price scale, unless it just hapenned to be close to RSI’s 0 to 100 range. This shows an RSI signal line and a centerline at the 50 level, with the script running in a separate pane:

image

//@version=6 indicator("RSI") myRSI = ta.rsi(close, 20) bullColor = color.from_gradient(myRSI, 50, 80, color.new(color.lime, 70), color.new(color.lime, 0)) bearColor = color.from_gradient(myRSI, 20, 50, color.new(color.red, 0), color.new(color.red, 70)) myRSIColor = myRSI > 50 ? bullColor : bearColor plot(myRSI, "RSI", myRSIColor, 3) hline(50)

Note that the *y* axis of our script’s visual space is automatically sized using the range of values plotted, i.e., the values of RSI. See the page on Colors for more information on the color.from_gradient() function used in the script.

If we try to plot the symbol’s close values in the same space by adding the following line to our script:

plot(close)

This is what happens:

image

The chart is on the BTCUSD symbol, whose close prices are around 40000 during this period. Plotting values in the 40000 range makes our RSI plots in the 0 to 100 range indiscernible. The same distorted plots would occur if we placed the RSI indicator on the chart as an overlay.

Merging two indicators

If you are planning to merge two signals in one script, first consider the scale of each. It is impossible, for example, to correctly plot an RSI and a MACD in the same script's visual space because RSI has a fixed range (0 to 100) while MACD doesn't, as it plots moving averages calculated on price._

If both your indicators used fixed ranges, you can shift the values of one of them so they do not overlap. We could, for example, plot both RSI (0 to 100) and the True Strength Indicator (TSI) (-100 to +100) by displacing one of them. Our strategy here will be to compress and shift the TSI values so they plot over RSI:

image

//@version=6 indicator("RSI and TSI") myRSI = ta.rsi(close, 20) bullColor = color.from_gradient(myRSI, 50, 80, color.new(color.lime, 70), color.new(color.lime, 0)) bearColor = color.from_gradient(myRSI, 20, 50, color.new(color.red, 0), color.new(color.red, 70)) myRSIColor = myRSI > 50 ? bullColor : bearColor plot(myRSI, "RSI", myRSIColor, 3) hline(100) hline(50) hline(0) // 1. Compress TSI's range from -100/100 to -50/50. // 2. Shift it higher by 150, so its -50 min value becomes 100. myTSI = 150 + (100 * ta.tsi(close, 13, 25) / 2) plot(myTSI, "TSI", color.blue, 2) plot(ta.ema(myTSI, 13), "TSI EMA", #FF006E) hline(200) hline(150)

Note that:

- We have added levels using hline to situate both signals.

- In order for both signal lines to oscillate on the same range of 100, we divide the TSI value by 2 because it has a 200 range (-100 to +100). We then shift this value up by 150 so it oscillates between 100 and 200, making 150 its centerline.
- The manipulations we make here are typical of the compromises required to bring two indicators with different scales in the same visual space, even when their values, contrary to MACD, are bounded in a fixed range.

Previous

Next

Repainting

Pine Script® v6 Documentation

Repainting

Introduction

We define repainting as: **script behavior causing historical vs realtime calculations or plots to behave differently**.

Repainting behavior is widespread and many factors can cause it. Following our definition, our estimate is that more than 95% of indicators in existence exhibit some form of repainting behavior. Commony used indicators such as MACD and RSI, for example, show confirmed values on historical bars, but will fluctuate on a realtime, unconfirmed chart bar until it closes. Therefore, they behave *differently* in historical and realtime states.

Not all repainting behavior is inherently useless or misleading, nor does such behavior prevent knowledgeable traders from using indicators with such behavior. For example, who would think of discrediting a volume profile indicator solely because it updates its values on realtime bars?

One may encounter any of the following forms of repainting in the scripts they use, depending on what a script's calculations entail:

- **Widespread but often acceptable**: A script may use values that update with realtime price changes on the unconfirmed bar. For example, if one uses the close variable in calculations performed on an open chart bar, its values will reflect the most recent price in the bar. However, the script will only commit a new data point to its historical series once the bar closes. Another common case is using request.security() to fetch higher-timeframe data on realtime bars, as explained in the Historical and realtime behavior section of the Other timeframes and data page. As with the unconfirmed chart bar in the

chart's timeframe, request.security() can track unconfirmed values from a higher-timeframe context on realtime bars, which can lead to repainting after the script restarts its execution. There is often nothing wrong with using such scripts, provided you understand how they work. When electing to use such scripts to issue alerts or trade orders, however, it's important to understand the difference between their realtime and historical behavior and decide for yourself whether it provides utility for your needs.

- **Potentially misleading**: Scripts that plot values into the past, calculate results on realtime bars that one cannot replicate on historical bars, or relocate past events are potentially misleading. For example, Ichimoku, most scripts based on pivots, most strategies using `calc_on_every_tick = true`, scripts using request.security() when it behaves differently on realtime bars, many scripts using varip, many scripts using timenow, and some scripts that use `barstate.*` variables can exhibit misleading repainting behavior.
- **Unacceptable**: Scripts that leak future information into the past, strategies that execute on non-standard charts, and scripts using realtime intrabars to generate alerts or orders, are examples that can produce heavily misleading repainting behavior.
- **Unavoidable**: Revisions of the data feed from a provider and variations in the starting bar of the chart's history can cause repainting behavior that may be unavoidable in a script.

The first two types of repainting can be perfectly acceptable if:

1. You are aware of the behavior.
2. You can live with it, or
3. You can circumvent it.

It should now be clear that not **all** repainting behavior is wrong and requires avoiding at all costs. In many situations, some forms of repainting may be exactly what a script needs. What's important is to know when repainting behavior is **not** acceptable for one's needs. To avoid repainting that's not acceptable, it's important to understand how a tool works or how you should design the tools you build. If you publish scripts, ensure you mention any potentially misleading behavior along with the other limitations of your script in the publication's description.

For script users

One can decide to use repainting indicators if they understand the behavior, and whether that behavior meets their analysis requirements. Don't be one of those newcomers who slap "repaint" sentences on published scripts in an attempt to discredit them, as doing so reveals a lack of foundational knowledge on the subject.

Simply asking whether a script repaints is relatively meaningless, given that there are forms of repainting behavior that are perfectly acceptable in a script. Therefore, such a question will not beget a meaningful answer. One should instead ask *specific* questions about a script's potential repainting behavior, such as:

- Does the script calculate/display in the same way on historical and realtime bars?
- Do alerts from the script wait for the end of a realtime bar before triggering?
- Do signal markers shown by the script wait for the end of a realtime bar before showing?
- Does the script plot/draw values into the past?
- Does the strategy use `calc_on_every_tick = true`?
- Do the script's request.security() calls leak future information into the past on historical bars?

What's important is that you understand how the tools you use work, and whether their behavior is compatible with your objectives, repainting or not. As you will learn if you read this page, repainting is a complex matter. It has many faces and many causes. Even if you don't program in Pine Script®, this page will help you understand the array of causes that can lead to repainting, and hopefully enable more meaningful discussions with script authors.

For Pine Script programmers

As discussed above, not all forms of repainting behavior must be avoided at all costs, nor is all potential repainting behavior necessarily avoidable. We hope this page helps you better understand the dynamics at play so that you can design your trading tools with these behaviors in mind. This page's

content should help make you aware of common coding mistakes that produce misleading repainting results.

Whatever your design decisions are, if you publish your script, explain the script to traders so they can understand how it behaves.

This page covers three broad categories of repainting causes:

- Historical vs realtime calculations
- Plotting in the past
- Dataset variations

Historical vs realtime calculations

Fluid data values

Historical data does not include records of intermediary price movements on bars; only open, high, low and close values (OHLC).

On realtime bars (bars running when the instrument's market is open), however, the high, low and close values are not fixed; they can change values many times before the realtime bar closes and its HLC values are fixed. They are *fluid*. This leads to a script sometimes working differently on historical data and in real time, where only the open price will not change during the bar.

Any script using values like high, low and close in realtime is subject to producing calculations that may not be repeatable on historical bars — thus repaint.

Let's look at this simple script. It detects crosses of the close value (in the realtime bar, this corresponds to the current price of the instrument) over and under an EMA:

image

//@version=6 indicator("Repainting", "", true) ma = ta.ema(close, 5) xUp = ta.crossover(close, ma) xDn = ta.crossunder(close, ma) plot(ma, "MA", color.black, 2) bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)

Note that:

- The script uses bgcolor() to color the background green when close crosses over the EMA, and red on crosses under the EMA.
- The screen snapshot shows the script in realtime on a 30sec chart. A cross over the EMA has been detected, thus the background of the realtime bar is green.
- The problem here is that nothing guarantees this condition will hold true until the end of the realtime bar. The arrow points to the timer showing that 21 seconds remain in the realtime bar, and anything could happen until then.
- We are witnessing a repainting script.

To prevent this repainting, we must rewrite our script so that it does not use values that fluctuate during the realtime bar. This will require using values from a bar that has elapsed (typically the preceding bar), or the open price, which does not vary in realtime.

We can achieve this in many ways. This method adds a and `barstate.isconfirmed` condition to our cross detections, which requires the script to be executing on the bar's last iteration, when it closes and prices are confirmed. It is a simple way to avoid repainting:

//@version=6 indicator("Repainting", "", true) ma = ta.ema(close, 5) xUp = ta.crossover(close, ma) and barstate.isconfirmed xDn = ta.crossunder(close, ma) and barstate.isconfirmed plot(ma, "MA", color.black, 2) bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)

This uses the crosses detected on the previous bar:

//@version=6 indicator("Repainting", "", true) ma = ta.ema(close, 5) xUp = ta.crossover(close, ma)[1] xDn = ta.crossunder(close, ma)[1] plot(ma, "MA", color.black, 2) bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)

This uses only confirmed close and EMA values for its calculations:

//@version=6 indicator("Repainting", "", true) ma = ta.ema(close[1], 5) xUp = ta.crossover(close[1], ma) xDn = ta.crossunder(close[1], ma) plot(ma,

"MA", color.black, 2) bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)

This detects crosses between the realtime bar's open and the value of the EMA from the previous bars. Notice that the EMA is calculated using close, so it repaints. We must ensure we use a confirmed value to detect crosses, thus `ma[1]` in the cross detection logic:

//@version=6 indicator("Repainting", "", true) ma = ta.ema(close, 5) xUp = ta.crossover(open, ma[1]) xDn = ta.crossunder(open, ma[1]) plot(ma, "MA", color.black, 2) bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)

All these methods have one thing in common: while they prevent repainting, they will also trigger signals later than repainting scripts. This is an inevitable compromise if one wants to avoid repainting. You can't have your cake and eat it too.

Repainting `request.security()` calls

The request.security() function behaves differently on historical and realtime bars. On historical bars, it only returns *confirmed* values from its requested context, wheras it can return *unconfirmed* values on realtime bars. When the script restarts its execution, the bars that had a realtime state become historical bars, and will therefore only contain the values it confirmed on those bars. If the values returned by request.security() fluctuate on realtime bars without confirmation from the context, the script will repaint them when it restarts its execution. See the Historical and realtime behavior section of the Other timeframes and data page for a detailed explanation.

One can ensure higher-timeframe data requests only return confirmed values on all bars, regardless of bar state, by offsetting the `expression` argument by at least one bar with the history-referencing operator [] and using barmerge.lookahead_on for the `lookahead` argument in the request.security() call, as explained here.

The script below demonstrates the difference between repainting and non-repainting HTF data requests. It contains two request.security() calls. The first function call requests close data from the `higherTimeframe` without

additional specification, and the second call requests the same series with an offset and barmerge.lookahead_on.

As we see on all realtime bars (the ones with an orange background), the `repaintingClose` series contains values that fluctuate without confirmation from the `higherTimeframe` context, meaning the results will *repaint* after the script restarts its executions. The `nonRepaintingClose`, on the other hand, behaves the same on realtime and historical bars, i.e., it only changes its value when new, confirmed data is available:

image

//@version=6 indicator("Repainting vs non-repainting `request.security()` demo", overlay = true) //@variable The timeframe to request data from. string higherTimeframe = input.timeframe("30", "Timeframe") if timeframe.in_seconds() > timeframe.in_seconds(higherTimeframe) runtime.error("The 'Timeframe' input is smaller than the chart's timeframe. Choose a higher timeframe.") //@variable The current `close` requested from the `higherTimeframe`. Fluctuates without confirmation on realtime bars. float repaintingClose = request.security(syminfo.tickerid, higherTimeframe, close) //@variable The last confirmed `close` requested from the `higherTimeframe`. // Behaves the same on historical and realtime bars. float nonRepaintingClose = request.security(syminfo.tickerid, higherTimeframe, close[1], lookahead = barmerge.lookahead_on) // Plot the values. plot(repaintingClose, "Repainting close", color.new(color.purple, 50), 8) plot(nonRepaintingClose, "Non-repainting close", color.teal, 3) // Plot a shape when a new `higherTimeframe` starts. plotshape(timeframe.change(higherTimeframe), "Timeframe change marker", shape.square, location.top, size = size.small) // Color the background on realtime bars. bgcolor(barstate.isrealtime ? color.new(color.orange, 60) : na, title = "Realtime bar highlight")

Note that:

- The script uses the plotshape() function to mark the chart when a new bar starts on the specified higher timeframe.
- This script calls runtime.error() to raise a custom runtime error if the `higherTimeframe` value represents a timeframe that is *lower* than the chart’s timeframe.

- On historical bars, the `repaintingClose` series has a new value at the *end* of each timeframe, and the `nonRepaintingClose` has a new value at the *start* of each timeframe.

For the sake of easy reusability, below is a simple a `noRepaintSecurity()` function that one can apply in their scripts to request non-repainting higher-timeframe values:

//@function Requests non-repainting `expression` values from the context of the `symbol` and `timeframe`. noRepaintSecurity(symbol, timeframe, expression) => request.security(symbol, timeframe, expression[1], lookahead = barmerge.lookahead_on)

Note that:

- The `[1]` offset to the series and the use of `lookahead = barmerge.lookahead_on` are interdependent. *Neither* can be removed without compromising the integrity of the function.
- Unlike a plain request.security() call, a call to this wrapper function cannot use a tuple as the `expression` argument. For multi-element requests, programmers can pass an object of a user-defined type whose *fields* contain the desired elements.

Using `request.security()` at lower timeframes

Some scripts use request.security() to request data from a timeframe **lower** than the chart's timeframe. This can be useful when functions specifically designed to handle intrabars at lower timeframes are sent down the timeframe. When this type of user-defined function requires the detection of the intrabars' first bar, as most do, the technique will only work on historical bars. This is due to the fact that realtime intrabars are not yet sorted. The impact of this is that such scripts cannot reproduce in real time their behavior on historical bars. Any logic generating alerts, for example, will be flawed, and constant refreshing will be required to recalculate elapsed realtime bars as historical bars.

When used at lower timeframes than the chart's without specialized functions able to distinguish between intrabars, request.security() will only return the value of the **last** intrabar in the dilation of the chart's bar, which is

usually not useful, and will also not reproduce in real time, so lead to repainting.

For all these reasons, unless you understand the subtleties of using request.security() at lower timeframes than the chart's, it is best to avoid using the function at those timeframes. Higher-quality scripts will have logic to detect such anomalies and prevent the display of results which would be invalid when a lower timeframe is used.

For more reliable lower-timeframe data requests, use request.security_lower_tf(), as explained in this section of the Other timeframes and data page.

Future leak with `request.security()`

When request.security() is used with `lookahead = barmerge.lookahead_on` to fetch prices without offsetting the series by `[1]`, it will return data from the future on historical bars, which is dangerously misleading.

While historical bars will magically display future prices before they should be known, no lookahead is possible in realtime because the future there is unknown, as it should, so no future bars exist.

This is an example:

image

// FUTURE LEAK! DO NOT USE! //@version=6 indicator("Future leak", "", true) futureHigh = request.security(syminfo.tickerid, "1D", high, lookahead = barmerge.lookahead_on) plot(futureHigh)

Note how the higher timeframe line is showing the timeframe's high value before it occurs. The solution to avoid this effect is to use the function as demonstrated in this previous section.

Using lookahead to produce misleading results is not allowed in script publications, as explained in the lookahead section of the Other timeframes and data page. Script publications that use this misleading technique **will be moderated**.

varip

Scripts using the varip declaration mode for variables (see our section on varip for more information) save information across realtime updates, which cannot be reproduced on historical bars where only OHLC information is available. Such scripts may be useful in realtime, including to generate alerts, but their logic cannot be backtested, nor can their plots on historical bars reflect calculations that will be done in realtime.

Bar state built-ins

Scripts using bar states may or may not repaint. As we have seen in the previous section, using barstate.isconfirmed is actually one way to **avoid** repainting that **will** reproduce on historical bars, which are always "confirmed". Uses of other bar states such as barstate.isnew, however, will lead to repainting. The reason is that on historical bars, barstate.isnew is `true` on the bar's close, yet in realtime, it is `true` on the bar's open. Using the other bar state variables will usually cause some type of behavioral discrepancy between historical and realtime bars.

timenow

The timenow built-in returns the current time. Scripts using this variable cannot show consistent historical and realtime behavior, so they necessarily repaint.

Strategies

Strategies using `calc_on_every_tick = true` execute on each realtime update, while strategies run on the close of historical bars. They will most probably not generate the same order executions, and so repaint. Note that when this happens, it also invalidates backtesting results, as they are not representative of the strategy's behavior in realtime.

Plotting in the past

Scripts detecting pivots after 5 bars have elapsed will often go back in the past to plot pivot levels or values on the actual pivot, 5 bars in the past. This

will often cause unsuspecting traders looking at plots on historical bars to infer that when the pivot happens in realtime, the same plots will apppear on the pivot when it occurs, as opposed to when it is detected.

Let’s look at a script showing the price of high pivots by placing the price in the past, 5 bars after the pivot was detected:

//@version=6 indicator("Plotting in the past", "", true) pHi = ta.pivothigh(5, 5) if not na(pHi) label.new(bar_index[5], na, str.tostring(pHi, format.mintick) + "\n ", yloc = yloc.abovebar, style = label.style_none, textcolor = color.black, size = size.normal)

image

Note that:

- This script repaints because an elapsed realtime bar showing no price may get a price placed on it if it is identified as a pivot, 5 bars after the actual pivot occurs.
- The display looks great, but it can be misleading.

The best solution to this problem when developing script for others is to plot **without** an offset by default, but give the option for script users to turn on plotting in the past through inputs, so they are necessarily aware of what the script is doing, e.g.:

//@version=6 indicator("Plotting in the past", "", true) plotInThePast = input(false, "Plot in the past") pHi = ta.pivothigh(5, 5) if not na(pHi) label.new(bar_index[plotInThePast ? 5 : 0], na, str.tostring(pHi, format.mintick) + "\n ", yloc = yloc.abovebar, style = label.style_none, textcolor = color.black, size = size.normal)

Dataset variations

Starting points

Scripts begin executing on the chart’s first historical bar, and then execute on each bar sequentially, as is explained in this manual’s page on Pine Script’s execution model. If the first bar changes, then the script will often not

calculate the same way it did when the dataset began at a different point in time.

The following factors have an impact on the quantity of bars you see on your charts, and their *starting point*:

- The type of account you hold
- The historical data available from the data supplier
- The alignment requirements of the dataset, which determine its *starting point*

These are the account-specific bar limits:

- 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.

Starting points are determined using the following rules, which depend on the chart's timeframe:

- **Tick-based timeframes**: return the exact number of bars based on the plan.
- **Second-based timeframes**: aligns to the beginning of a day.
- **1 - 14 minutes**: aligns to the beginning of a week.
- **15 - 29 minutes**: aligns to the beginning of a month.
- **30 - 1439 minutes**: aligns to the beginning of a year.
- **1440 minutes and higher**: aligns to the first available historical data point.

As time goes by, these factors cause your chart's history to start at different points in time. This often has an impact on your scripts calculations, because changes in calculation results in early bars can ripple through all the other bars in the dataset. Using functions like ta.valuewhen(), ta.barssince() or ta.ema(), for example, will yield results that vary with early history.

Revision of historical data

Historical and realtime bars are built using two different data feeds supplied by exchanges/brokers: historical data, and realtime data. When realtime bars elapse, exchanges/brokers sometimes make what are usually small adjustments to bar prices, which are then written to their historical data. When the chart is refreshed or the script is re-executed on those elapsed realtime bars, they will then be built and calculated using the historical data, which will contain those usually small price revisions, if any have been made.

Historical data may also be revised for other reasons, e.g., for stock splits.

Previous

Next

Sessions

Sessions

Introduction

Session information is usable in three different ways in Pine Script®:

1. **Session strings** containing from-to start times and day information that can be used in functions such as time() and time_close() to detect when bars are in a particular time period, with the option of limiting valid sessions to specific days. The input.session() function provides a way to allow script users to define session values through a script's "Inputs" tab (see the Session input section for more information).
2. **Session states** built-in variables such as session.ismarket can identify which session a bar belongs to.
3. When fetching data with request.security() you can also choose to return data from *regular* sessions only or *extended* sessions. In this case, the definition of **regular and extended sessions** is that of the exchange. It is part of the instrument's properties — not user-defined, as in point #1. This notion of *regular* and *extended* sessions is the same one used in the chart's interface, in the "Chart Settings/Symbol/Session" field, for example.

The following sections cover both methods of using session information in Pine Script.

Note that:

- Not all user accounts on TradingView have access to extended session information.
- There is no special "session" type in Pine Script. Instead, session strings are of "string" type but must conform to the session string syntax.

Session strings

Session string specifications

Session strings used with time() and time_close() must have a specific format. Their syntax is:

```
<time_period>:<days>
```

Where:

- <time_period> uses times in “hhmm” format, with “hh” in 24-hour format, so 1700 for 5PM.

The time periods are in the “hhmm-hhmm” format, and a comma can separate multiple time periods to specify combinations of discrete periods.

For example, - <days> is a set of digits from 1 to 7 that specifies on which days the session is valid.

1 is Sunday, 7 is Saturday.

These are examples of session strings:

`"24x7"`

A 7-day, 24-hour session beginning at midnight.

`"0000-0000:1234567"`

Equivalent to the previous example.

`"0000-0000"`

Equivalent to the previous two examples because the default days are 1234567.

`"0000-0000:23456"`

The same as the previous example, but only Monday to Friday.

`"2000-1630:1234567"`

An overnight session that begins at 20:00 and ends at 16:30 the next day. It is valid on all days of the week.

`"0930-1700:146"`

A session that begins at 9:30 and ends at 17:00 on Sundays (1), Wednesdays (4), and Fridays (6).

`"1700-1700:23456"`

An *overnight session*. The Monday session starts Sunday at 17:00 and ends Monday at 17:00. It is valid Monday through Friday.

`"1000-1001:26"`

A weird session that lasts only one minute on Mondays (2) and Fridays (6).

`"0900-1600,1700-2000"`

A session that begins at 9:00, breaks from 16:00 to 17:00, and continues until 20:00. Applies to every day of the week.

Using session strings

Session properties defined with session strings are independent of the exchange-defined sessions determining when an instrument can be traded. Programmers have complete liberty in creating whatever session definitions suit their purpose, which is usually to detect when bars belong to specific time periods. This is accomplished in Pine Script by using one of the following two signatures of the time() function:

```
text time(timeframe, session, timezone) → series int
time(timeframe, session) → series int
```

Here, we use time() with a `session` argument to display the market's opening high and low values on an intraday chart:

image

//@version=6 indicator("Opening high/low", overlay = true) sessionInput = input.session("0930-0935") sessionBegins(sess) => t = time("", sess) timeframe.isintraday and (not barstate.isfirst) and na(t[1]) and not na(t) var

float hi = na var float lo = na if sessionBegins(sessionInput) hi := high lo := low plot(lo, "lo", color.fuchsia, 2, plot.style_circles) plot(hi, "hi", color.lime, 2, plot.style_circles)

Note that:

- We use a session input to allow users to specify the time they want to detect. We are only looking for the session's beginning time on bars, so we use a five-minute gap between the beginning and end time of our `"0930-0935"` default value.
- We create a `sessionBegins()` function to detect the beginning of a session. Its `time("", sess)` call uses an empty string for the function's `timeframe` parameter, which means it uses the chart's timeframe, whatever that is. The function returns `true` when:
  - The chart uses an intraday timeframe (seconds or minutes).
  - The script isn't on the chart's first bar, which we ensure with `(not barstate.isfirst)`. This check prevents the code from always detecting a session beginning on the first bar because `na(t[1])` `and not na(t)` is always `true` there.
  - The time() call has returned na on the previous bar because it wasn't in the session's time period, and it has returned a value that is not na on the current bar, which means the bar is **in** the session's time period.

Session states

Three built-in variables allow you to distinguish the type of session the current bar belongs to. They are only helpful on intraday timeframes:

- session.ismarket returns `true` when the bar belongs to regular trading hours.
- session.ispremarket returns `true` when the bar belongs to the extended session preceding regular trading hours.
- session.ispostmarket returns `true` when the bar belongs to the session following regular trading hours.

Using sessions with request.security()

When your TradingView account provides access to extended sessions, you can choose to see their bars with the “Settings/Symbol/Session” field. There are two types of sessions:

- **regular** (which does not include pre- and post-market data), and
- **extended** (which includes pre- and post-market data).

Scripts using the request.security() function to access data can return extended session data or not. This is an example where only regular session data is fetched:

image

//@version=6 indicator("Example 1: Regular Session Data") regularSessionData = request.security("NASDAQ:AAPL", timeframe.period, close, barmerge.gaps_on) plot(regularSessionData, style = plot.style_linebr)

If you want the request.security() call to return extended session data, you must first use the ticker.new() function to build the first argument of the request.security() call:

image

//@version=6 indicator("Example 2: Extended Session Data") t = ticker.new("NASDAQ", "AAPL", session.extended) extendedSessionData = request.security(t, timeframe.period, close, barmerge.gaps_on) plot(extendedSessionData, style = plot.style_linebr)

Note that the previous chart’s gaps in the script’s plot are now filled. Also, keep in mind that our example scripts do not produce the background coloring on the chart; it is due to the chart’s settings showing extended hours.

The ticker.new() function has the following signature:

```
ticker.new(prefix, ticker, session, adjustment) → simple string
```

Where:

- `prefix` is the exchange prefix, e.g., `"NASDAQ"`

- `ticker` is a symbol name, e.g., `"AAPL"`
- `session` can be `session.extended` or `session.regular`. Note that this is **not** a session string.
- `adjustment` adjusts prices using different criteria: `adjustment.none`, `adjustment.splits`, `adjustment.dividends`.

Our first example could be rewritten as:

//@version=6 indicator("Example 1: Regular Session Data") t = ticker.new("NASDAQ", "AAPL", session.regular) regularSessionData = request.security(t, timeframe.period, close, barmerge.gaps_on) plot(regularSessionData, style = plot.style_linebr)

If you want to use the same session specifications used for the chart's main symbol, omit the third argument in ticker.new(); it is optional. If you want your code to declare your intention explicitly, use the syminfo.session built-in variable. It holds the session type of the chart's main symbol:

//@version=6 indicator("Example 1: Regular Session Data") t = ticker.new("NASDAQ", "AAPL", syminfo.session) regularSessionData = request.security(t, timeframe.period, close, barmerge.gaps_on) plot(regularSessionData, style = plot.style_linebr)

Previous

Next

Strategies

Pine Script® v6 Documentation

Strategies

Introduction

Pine Script® Strategies are specialized scripts that simulate trades across historical and realtime bars, allowing users to backtest and forward test their trading systems. Strategy scripts have many of the same capabilities as indicator scripts, and they provide the ability to place, modify, and cancel hypothetical orders and analyze performance results.

When a script uses the strategy() function as its declaration statement, it gains access to the `strategy.*` namespace, which features numerous functions and variables for simulating orders and retrieving essential strategy information. It also displays relevant information and simulated performance results in the dedicated Strategy Tester tab.

A simple strategy example

The following script is a simple strategy that simulates entering a long or short position when two moving averages cross. When the `fastMA` crosses above the `slowMA`, it places a “buy” market order to enter a long position. When the `fastMA` crosses below the `slowMA`, it places a “sell” market order to enter a short position:

//@version=6 strategy("Simple strategy demo", overlay = true, margin_long = 100, margin_short = 100) //@variable The length of the `fastMA` and half the length of the `slowMA`. int lengthInput = input.int(14, "Base length", 2) // Calculate two moving averages with different lengths. float fastMA = ta.sma(close, lengthInput) float slowMA = ta.sma(close, lengthInput * 2) // Place an order to enter a long position when `fastMA` crosses over `slowMA`.

if ta.crossover(fastMA, slowMA) strategy.entry("buy", strategy.long) // Place an order to enter a short position when `fastMA` crosses under `slowMA`. if ta.crossunder(fastMA, slowMA) strategy.entry("sell", strategy.short) // Plot the moving averages. plot(fastMA, "Fast MA", color.aqua) plot(slowMA, "Slow MA", color.orange)

Note that:

- The strategy() function call declares that the script is a strategy named “Simple strategy demo” that displays visuals on the main chart pane.
- The `margin_long` and `margin_short` arguments in the strategy() call specify that the strategy must have 100% of a long or short trade’s amount available to allow the trade. See this section for more information.
- The strategy.entry() function is the command that the script uses to create entry orders and reverse positions. The “buy” entry order closes any short position and opens a new long position. The “sell” entry order closes any long position and opens a new short position.

Applying a strategy to a chart

To test a strategy, add it to the chart. Select a built-in or published strategy from the “Indicators, Metrics & Strategies” menu, or write a custom strategy in the Pine Editor and click the “Add to chart” option in the top-right corner:

image

The script plots trade markers on the main chart pane and displays simulated performance results inside the Strategy Tester tab:

image

Strategy Tester

The *Strategy Tester* visualizes the hypothetical performance of a strategy script and displays its properties. To use it, add a script declared with the strategy() function to the chart, then open the “Strategy Tester” tab. If two or more strategies are on the chart, specify which one to analyze by selecting its name in the top-left corner.

After the selected script executes across the chart's data, the Strategy Tester populates the following four tabs with relevant strategy information:

- Overview
- Performance Summary
- List of Trades
- Properties

Overview

The Overview tab provides a quick look into a strategy's performance over a sequence of simulated trades. This tab displays essential performance metrics and a chart with three helpful plots:

- The Equity baseline plot visualizes the strategy's simulated equity across closed trades.
- The Drawdown column plot shows how far the strategy's equity fell below its peak across trades.
- The Buy & hold equity plot shows the equity growth of a strategy that enters a single long position and holds that position throughout the testing range.

image

Note that:

- The chart has two separate vertical scales. The "Equity" and "Buy & hold equity" plots use the scale on the left, and the "Drawdown" plot uses the scale on the right. Users can toggle the plots and choose between absolute or percentage scales using the options at the bottom.
- When a user clicks on a point in this chart, the main chart scrolls to the corresponding bar where the trade closed and displays a tooltip containing the closing time.

Performance Summary

The Performance Summary tab presents an in-depth summary of a strategy's key performance metrics, organized into separate columns. The "All" column shows performance information for all simulated trades, and the "Long" and "Short" columns show relevant metrics separately for long and short trades.

This view provides more detailed insights into a strategy's overall and directional trading performance:

image

List of Trades

The List of Trades tab chronologically lists a strategy's simulated trades. Each item in the list displays vital information about a trade, including the dates and times of entry and exit orders, the names of the orders, the order prices, and the number of contracts/shares/lots/units. In addition, each item shows the trade's profit or loss and the strategy's cumulative profit, run-up, and drawdown:

image

Note that:

- Hovering the mouse over a list item's entry or exit information reveals a "Scroll to bar" button. Clicking that button navigates the main chart to the bar where the entry or exit occurred.
- The list shows each trade in *descending* order by default, with the latest trade at the top. Users can reverse this order by clicking the "Trade #" button above the list.

Properties

The "Properties" tab provides detailed information about a strategy's configuration and the dataset that it executes across, organized into four collapsible sections:

- The "Date Range" section shows the range of dates that had simulated trades, and the overall available backtesting range.
- The "Symbol Info" section displays the chart's symbol, timeframe, type, point value, currency, and tick size. It also includes the chart's specified precision setting.
- The "Strategy Inputs" section lists the names and values of all the inputs available in the strategy's "Settings/Inputs" tab. This section only appears if the script includes `input*()` calls or specifies a nonzero `calc_bars_count` argument in the strategy() declaration statement.

- The “Strategy Properties” section provides an overview of the strategy’s properties, including the initial capital, account currency, order size, margin, pyramiding, commission, slippage, and other settings.

image

Broker emulator

TradingView uses a *broker emulator* to simulate trades while running a strategy script. Unlike in real-world trading, the emulator fills a strategy’s orders exclusively using available *chart data* by default. Consequently, it executes orders on historical bars *after a bar closes*. Similarly, the earliest point that it can fill orders on realtime bars is after a new price tick. For more information about this behavior, see the Execution model page.

Because the broker emulator only uses price data from the chart by default, it makes *assumptions* about intrabar price movement when filling orders. The emulator analyzes the opening, high, low, and closing prices of chart bars to infer intrabar activity using the following logic:

- If the opening price of a bar is closer to the high than the low, the emulator assumes that the market price moved in this order: **open → high → low → close**.
- If the opening price of a bar is closer to the low than the high, the emulator assumes that the market price moved in this order: **open → low → high → close**.
- The emulator assumes *no gaps* exist between intrabars inside each chart bar, meaning it considers *any* value within a bar’s high-low range as a valid price for order execution.
- When filling *price-based orders* (all orders except market orders), the emulator assumes intrabars **do not** exist within the gap between the previous bar’s close and the current bar’s open. If the market price crosses an order’s price during the gap between two bars, the emulator fills the order at the current bar’s *open* and not at the specified price.

image

Bar magnifier

Users with Premium and higher-tier plans can override the broker emulator's default assumptions about intrabar prices by enabling the Bar Magnifier backtesting mode. In this mode, the emulator uses data from *lower timeframes* to obtain more granular information about price action within bars, allowing more precise order fills in the strategy's simulation.

To enable the Bar Magnifier mode, include `use_bar_magnifier = true` in the strategy() declaration statement, or select the "Using bar magnifier" option in the "Fill orders" section of the strategy's "Settings/Properties" tab.

The following example script illustrates how the Bar Magnifier can enhance order-fill behavior. When the time value crosses the defined `orderTime`, it creates "Buy" and "Exit" limit orders at the calculated `entryPrice` and `exitPrice`. For visual reference, the script colors the background orange when it places the orders, and it draws two horizontal lines at the order prices:

image

//@version=6 strategy("Bar Magnifier Demo", overlay = true, use_bar_magnifier = false) //@variable The UNIX timestamp to place the order at. int orderTime = timestamp("UTC", 2023, 3, 22, 18) //@variable Is `color.orange` when `time` crosses the `orderTime`, false otherwise. color orderColor = na // Entry and exit prices. float entryPrice = hl2 - (high - low) float exitPrice = entryPrice + (high - low) * 0.25 // Entry and exit lines. var line entryLine = na var line exitLine = na if ta.cross(time, orderTime) // Draw new entry and exit lines. entryLine := line.new(bar_index, entryPrice, bar_index + 1, entryPrice, color = color.green, width = 2) exitLine := line.new(bar_index, exitPrice, bar_index + 1, exitPrice, color = color.red, width = 2) // Update order highlight color. orderColor := color.new(color.orange, 80) // Place limit orders at the `entryPrice` and `exitPrice`. strategy.entry("Buy", strategy.long, limit = entryPrice) strategy.exit("Exit", "Buy", limit = exitPrice) // Update lines while the position is open. else if strategy.position_size > 0.0 entryLine.set_x2(bar_index + 1) exitLine.set_x2(bar_index + 1) bgcolor(orderColor)

Because the script does not include a `use_bar_magnifier` argument in the strategy() function, the broker emulator uses the default assumptions when filling the orders: that the bar's price moved from open to high, high to low, and then low to close. Therefore, after filling the "Buy" order at the price indicated by the green line, the broker emulator inferred that the market price did not go back up to touch the red line and trigger the "Exit" order. In other words, the strategy *could not* enter and exit the position on the same bar according to the broker emulator's assumptions.

After we enable the Bar Magnifier mode, the broker emulator can access *10-minute* data on the 60-minute chart instead of relying on its assumptions about hourly bars. On this timeframe, the market price *did* move back up to the "Exit" order's price after reaching the "Buy" order's price in the same hour. Therefore, with the Bar Magnifier enabled in this scenario, both orders execute on the same hourly bar:

image

Orders and trades

Pine Script strategies use orders to make trades and manage positions, similar to real-world trading. In this context, an *order* is an instruction that a strategy sends to the broker emulator to perform a market action, and a *trade* is the resulting transaction after the emulator fills an order.

Let's take a closer look at how strategy orders work and how they become trades. Every 20 bars, the following script creates a long market order with strategy.entry() and draws a label. It calls strategy.close_all() on each bar from the global scope to generate a market order to close any open position:

image

//@version=6 strategy("Order execution demo", "My strategy", true, margin_long = 100, margin_short = 100) //@function Displays the specified `txt` in a label at the `high` of the current bar. debugLabel(string txt) => label.new(bar_index, high, text = txt, color=color.lime, style = label.style_label_lower_right, textcolor = color.black, size = size.large) // @variable Is `true` on every 20th bar, `false` otherwise. bool longCondition = bar_index % 20 == 0 // Draw a label and place a long market order when

`longCondition` occurs. if longCondition debugLabel("Long entry order created") strategy.entry("My Long Entry Id", strategy.long) // Place a closing market order whenever there is an open position. strategy.close_all()

Note that:

- Although the script calls strategy.close_all() on every bar, the function only creates a new exit order when the strategy has an *open position*. If there is no open position, the function call has no effect.

The blue arrows on the above chart show where the strategy entered a long position, and the purple arrows mark the bars where the strategy closed the position. Notice that the label drawings appear one bar *before* the entry markers, and the entry markers appear one bar *before* the closing markers. This sequence illustrates order creation and execution in action.

By default, the earliest point the broker emulator fills an order is on the next available price tick, because creating and filling an order on the same tick is unrealistic. Since strategies recalculate after each bar closes by default, the next available tick where the emulator fills a generated order is at the *open* of the *following bar*. For example, when the `longCondition` occurs on bar 20, the script places an entry order to fill on the next tick, which is at the open of bar 21. When the strategy recalculates its values after bar 21 closes, it places an order to close the current position on the next tick, which is at the open of bar 22.

Order types

Pine Script strategies can simulate different order types to suit specific trading system needs. The main notable order types include market, limit, stop, and stop-limit.

Market orders

A *market order* is the simplest type of order, which most order placement commands generate by default. A market order is an instruction to buy or sell a security as soon as possible, irrespective of the price. As such, the broker emulator always executes a market order on the next available tick.

The example below alternates between placing a long and short market order once every `lengthInput` bars. When the bar_index is divisible by 2 * `lengthInput`, the strategy generates a long market order. Otherwise, it places a short market order when the bar_index is divisible by the `lengthInput`:

image

//@version=6 strategy("Market order demo", overlay = true, margin_long = 100, margin_short = 100) //@variable Number of bars between long and short entries. int lengthInput = input.int(10, "Cycle length", 1) //@function Displays the specified `txt` in a label on the current bar. debugLabel(string txt, color lblColor) => label.new(bar_index, high, text = txt, color = lblColor, textcolor = color.white, style = label.style_label_lower_right, size = size.large) //@variable Is `true` every `2 * lengthInput` bars, `false` otherwise. longCondition = bar_index % (2 * lengthInput) == 0 //@variable Is `true` every `lengthInput` bars, `false` otherwise. shortCondition = bar_index % lengthInput == 0 // Generate a long market order with a `color.green` label on `longCondition`. if longCondition debugLabel("Long market order created", color.green) strategy.entry("My Long Entry Id", strategy.long) // Otherwise, generate a short market order with a `color.red` label on `shortCondition`. else if shortCondition debugLabel("Short market order created", color.red) strategy.entry("My Short Entry Id", strategy.short)

Note that:

- The labels indicate the bars where the script generates the market orders. The broker emulator fills each order at the open of the following bar.
- The strategy.entry() command can automatically *reverse* an open position in the opposite direction. See this section below for more information.

Limit orders

A *limit order* is an instruction to buy or sell a security at a specific price or better (lower than specified for long orders, and higher than specified for short orders), irrespective of the time. To simulate a limit order in a strategy

script, pass a *price* value to the `limit` parameter of an applicable order placement command.

When the market price reaches a limit order's value, or crosses it in the favorable direction, the broker emulator fills the order at that value or a better price. When a strategy generates a limit order at a *worse* value than the current market price (higher for long orders and lower for short orders), the emulator fills the order without waiting for the market price to reach that value.

For example, the following script generates a long limit order 800 ticks below the close of the bar 100 bars before the last chart bar using the strategy.entry() command. It draws a label to signify the bar where the strategy created the order and a line to visualize the order's price:

image

//@version=6 strategy("Limit order demo", overlay = true, margin_long = 100, margin_short = 100) //@function Displays text passed to `txt` and a horizontal line at `price` when called. debugLabel(float price, string txt) => label.new(bar_index, price, text = txt, color = color.teal, textcolor = color.white, style = label.style_label_lower_right, size = size.large) line.new(bar_index, price, bar_index + 1, price, color = color.teal, extend = extend.right, style = line.style_dashed) // Generate a long limit order with a label and line 100 bars before the `last_bar_index`. if last_bar_index - bar_index == 100 limitPrice = close - syminfo.mintick * 800 debugLabel(limitPrice, "Long Limit order created") strategy.entry("Long", strategy.long, limit = limitPrice)

Notice that in the chart above, the label and the start of the line occurred several bars before the "Long" entry marker. The broker emulator could not fill the order while the market price remained *above* the `limitPrice` because such a price is a *worse* value for the long trade. After the price fell and reached the `limitPrice`, the emulator filled the order mid-bar at that value.

If we set the `limitPrice` to a value *above* the bar's close rather than *below*, the broker emulator fills the order at the open of the following bar because the closing price is already a more *favorable* value for the long trade. Here, we set the `limitPrice` in the script to 800 ticks above the bar's close to demonstrate this effect:

image

//@version=6 strategy("Limit order demo", overlay = true, margin_long = 100, margin_short = 100) //@function Displays text passed to `txt` and a horizontal line at `price` when called. debugLabel(float price, string txt) => label.new(bar_index, price, text = txt, color = color.teal, textcolor = color.white, style = label.style_label_lower_right, size = size.large) line.new(bar_index, price, bar_index + 1, price, color = color.teal, extend = extend.right, style = line.style_dashed) // Generate a long limit order with a label and line 100 bars before the `last_bar_index`. if last_bar_index - bar_index == 100 limitPrice = close + syminfo.mintick * 800 debugLabel(limitPrice, "Long Limit order created") strategy.entry("Long", strategy.long, limit = limitPrice)

Stop and stop-limit orders

A *stop order* is an instruction to activate a new market or limit order when the market price reaches a specific price or a worse value (higher than specified for long orders and lower than specified for short orders). To simulate a stop order, pass a price value to the `stop` parameter of an applicable order placement command.

When a strategy generates a stop order at a *better* value than the current market price, it activates the subsequent order without waiting for the market price to reach that value.

The following example calls strategy.entry() to place a stop order 800 ticks above the close 100 bars before the last historical chart bar. It also draws a label on the bar where it created the order and a line to display the stop price. As we see in the chart below, the strategy entered a long position immediately after the price crossed the stop level:

image

//@version=6 strategy("Stop order demo", overlay = true, margin_long = 100, margin_short = 100) //@function Displays text passed to `txt` when called and shows the `price` level on the chart. debugLabel(price, txt) => label.new(bar_index, high, text = txt, color = color.teal, textcolor = color.white, style = label.style_label_lower_right, size = size.large)

line.new(bar_index, high, bar_index, price, style = line.style_dotted, color = color.teal) line.new(bar_index, price, bar_index + 1, price, color = color.teal, extend = extend.right, style = line.style_dashed) // Generate a long stop order with a label and lines 100 bars before the last bar. if last_bar_index - bar_index == 100 stopPrice = close + syminfo.mintick * 800 debugLabel(stopPrice, "Long Stop order created") strategy.entry("Long", strategy.long, stop = stopPrice)

Note that:

- A basic stop order is essentially the opposite of a limit order in terms of its execution based on the market price. If we use a limit order instead of a stop order in this scenario, the order executes immediately on the next bar. See the previous section for an example.

When a strategy.entry() or strategy.order() call includes a `stop` *and* `limit` argument, it creates a *stop-limit order*. Unlike a basic stop order, which triggers a market order when the current price is at the `stop` level or a worse value, a stop-limit order creates a subsequent limit order to fill at the specified `limit` price.

Below, we modified the previous script to simulate and visualize a stop-limit order. This script version includes the bar's low as the `limit` price in the strategy.entry() command. It also includes additional drawings to show where the strategy activated the subsequent limit order and to visualize the limit price.

In this example chart, notice how the market price reached the limit level on the next bar after the stop-limit order was created, but the strategy did not enter a position because the limit order was not yet active. After price later reached the stop level, the strategy placed the limit order, and then the broker emulator filled it after the market price dropped back down to the limit level:

image

//@version=6 strategy("Stop-Limit order demo", overlay = true, margin_long = 100, margin_short = 100) //@function Displays text passed to `txt` when called and shows the `price` level on the chart. debugLabel(price, txt, lblColor, lineWidth = 1) => label.new(bar_index, high, text = txt, color =

lblColor, textcolor = color.white, style = label.style_label_lower_right, size = size.large) line.new(bar_index, close, bar_index, price, style = line.style_dotted, color = lblColor, width = lineWidth) line.new(bar_index, price, bar_index + 1, price, color = lblColor, extend = extend.right, style = line.style_dashed, width = lineWidth) var float stopPrice = na var float limitPrice = na // Generate a long stop-limit order with a label and lines 100 bars before the last bar. if last_bar_index - bar_index == 100 stopPrice := close + syminfo.mintick * 800 limitPrice := low debugLabel(limitPrice, "", color.gray) debugLabel(stopPrice, "Long Stop-Limit order created", color.teal) strategy.entry("Long", strategy.long, stop = stopPrice, limit = limitPrice) // Draw a line and label when the strategy activates the limit order. if high >= stopPrice debugLabel(limitPrice, "Limit order activated", color.green, 2) stopPrice := na

Order placement and cancellation

The `strategy.*` namespace features the following five functions that simulate the placement of orders, known as *order placement commands*: strategy.entry(), strategy.order(), strategy.exit(), strategy.close(), and strategy.close_all().

Additionally, the namespace includes the following two functions that cancel pending orders, known as *order cancellation commands*: strategy.cancel() and strategy.cancel_all().

The segments below explain these commands, their unique characteristics, and how to use them.

`strategy.entry()`

The strategy.entry() command generates *entry orders*. Its unique features help simplify opening and managing positions. This order placement command generates market orders by default. It can also create limit, stop, and stop-limit orders with the `limit` and `stop` parameters, as explained in the Order types section above.

Reversing positions

One of the strategy.entry() command's unique features is its ability to *reverse* an open position automatically. By default, when an order from strategy.entry() executes while there is an open position in the opposite direction, the command automatically *adds* the position's size to the new order's size. The added quantity allows the order to close the current position and open a new position for the specified number of contracts/lots/shares/units in the new direction.

For instance, if a strategy has an open position of 15 shares in the strategy.long direction and calls strategy.entry() to place a new market order in the strategy.short direction, the size of the resulting transaction is the specified entry size **plus** 15 shares.

The example below demonstrates this behavior in action. When the `buyCondition` occurs once every 100 bars, the script calls strategy.entry() with `qty = 15` to open a long position of 15 shares. Otherwise, when the `sellCondition` occurs on every 50th bar, the script calls strategy.entry() with `qty = 5` to enter a new short position of five shares. The script also highlights the chart's background on the bars where the `buyCondition` and `sellCondition` occurs:

image

//@version=6 strategy("Reversing positions demo", overlay = true) // @variable Is `true` on every 100th bar, `false` otherwise. bool buyCondition = bar_index % 100 == 0 //@variable Is `true` on every 50th bar, `false` otherwise. bool sellCondition = bar_index % 50 == 0 if buyCondition // Place a "buy" market order to close the short position and enter a long position of 15 shares. strategy.entry("buy", strategy.long, qty = 15) else if sellCondition // Place a "sell" market order to close the long position and enter a short position of 5 shares. strategy.entry("sell", strategy.short, qty = 5) // Highlight the background when the `buyCondition` or `sellCondition` occurs. bgcolor(buyCondition ? color.new(color.blue, 90) : sellCondition ? color.new(color.red, 90) : na)

The trade markers on the chart show the *transaction size*, not the size of the resulting position. The markers above show that the transaction size was *20*

shares on each order fill rather than 15 for long orders and five for short orders. Since strategy.entry() reverses a position in the opposite direction by default, each call *adds* the open position's size (e.g., 15 for long entries) to the new order's size (e.g., 5 for short entries), resulting in a quantity of 20 shares on each entry after the first. Although each of these *transactions* is 20 shares in size, the resulting positions are 5 shares for each short entry and 15 for each long entry.

Note that:

- The strategy.risk.allow_entry_in() function *overrides* the allowed direction for the strategy.entry() command. When a script specifies a trade direction with this risk management command, orders from strategy.entry() in the opposite direction *close* the open position without allowing a reversal.

Pyramiding

Another unique characteristic of the strategy.entry() command is its connection to a strategy's *pyramiding* property. Pyramiding specifies the maximum number of *successive entries* a strategy allows in the same direction. Users can set this property by including a `pyramiding` argument in the strategy() declaration statement or by adjusting the "Pyramiding" input in the script's "Settings/Properties" tab. The default value is 1, meaning the strategy can open new positions but cannot add to them using orders from strategy.entry() calls.

The following example uses strategy.entry() to place a market order when the `entryCondition` occurs on every 25th bar. The direction of the orders changes once every 100 bars, meaning every 100-bar cycle includes *four* strategy.entry() calls with the same direction. For visual reference of the conditions, the script highlights the chart's background based on the current direction each time the `entryCondition` occurs:

image

//@version=6 strategy("Pyramiding demo", overlay = true) //@variable Represents the direction of the entry orders. A value of 1 means long, and -1 means short. var int direction = 1 //@variable Is `true` once every 25 bars,

`false` otherwise. bool entryCondition = bar_index % 25 == 0 // Change the `direction` on every 100th bar. if bar_index % 100 == 0 direction *= -1 // Place a market order based on the current `direction` when the `entryCondition` occurs. if entryCondition strategy.entry("Entry", direction == 1 ? strategy.long : strategy.short) //@variable When the `entryCondition` occurs, is a blue color if the `direction` is 1 and a red color otherwise. color bgColor = entryCondition ? (direction == 1 ? color.new(color.blue, 80) : color.new(color.red, 80)) : na // Highlight the chart's background using the `bgColor`. bgcolor(bgColor, title = "Background highlight")

Notice that although the script calls strategy.entry() with the same direction four times within each 100-bar cycle, the strategy *does not* execute an order after every call. It cannot open more than one trade per position with strategy.entry() because it uses the default pyramiding value of 1.

Below, we modified the script by including `pyramiding = 4` in the strategy() declaration statement to allow up to four successive trades in the same direction. Now, an order fill occurs after every strategy.entry() call:

image

//@version=6 strategy("Pyramiding demo", overlay = true, pyramiding = 4) //@variable Represents the direction of the entry orders. A value of 1 means long, and -1 means short. var int direction = 1 //@variable Is `true` once every 25 bars, `false` otherwise. bool entryCondition = bar_index % 25 == 0 // Change the `direction` on every 100th bar. if bar_index % 100 == 0 direction *= -1 // Place a market order based on the current `direction` when the `entryCondition` occurs. if entryCondition strategy.entry("Entry", direction == 1 ? strategy.long : strategy.short) //@variable When the `entryCondition` occurs, is a blue color if the `direction` is 1 and a red color otherwise. color bgColor = entryCondition ? (direction == 1 ? color.new(color.blue, 80) : color.new(color.red, 80)) : na // Highlight the chart's background using the `bgColor`. bgcolor(bgColor, title = "Background highlight")

strategy.order()

The strategy.order() command generates a *basic order*. Unlike other order placement commands, which can behave differently based on a strategy's

properties and open trades, this command *ignores* most properties, such as pyramiding, and simply creates orders with the specified parameters. This command generates market orders by default. It can also create limit, stop, and stop-limit orders with the `limit` and `stop` parameters. Orders from strategy.order() can open new positions and modify or close existing ones. When a strategy executes an order from this command, the resulting market position is the *net sum* of the open position and the filled order quantity.

The following script uses strategy.order() calls to enter and exit positions. The strategy places a long market order for 15 units once every 100 bars. On every 25th bar that is not a multiple of 100, it places a short market order for five units. The script highlights the background to signify where the strategy places a “buy” or “sell” order:

image

//@version=6 strategy("`strategy.order()` demo", overlay = true) //@variable Is `true` on every 100th bar, `false` otherwise. bool buyCondition = bar_index % 100 == 0 //@variable Is `true` on every 25th bar, `false` otherwise. bool sellCondition = bar_index % 25 == 0 if buyCondition // Place a "buy" market order to trade 15 units in the long direction. strategy.order("buy", strategy.long, qty = 15) else if sellCondition // Place a "sell" market order to trade 5 units in the short direction. strategy.order("sell", strategy.short, qty = 5) // Highlight the background when the `buyCondition` or `sellCondition` occurs. bgcolor(buyCondition ? color.new(color.blue, 90) : sellCondition ? color.new(color.red, 90) : na)

This particular strategy never simulates a *short position*. Unlike the strategy.entry() command, strategy.order() *does not* automatically reverse open positions. After filling a “buy” order, the strategy has an open long position of 15 units. The three subsequent “sell” orders *reduce* the position by five units each, and 15 - 5 * 3 = 0. In other words, the strategy opens a long position on every 100th bar and gradually reduces the size to 0 using three successive short orders. If we used strategy.entry() instead of the strategy.order() command in this example, the strategy would alternate between entering long and short positions of 15 and five units, respectively.

`strategy.exit()`

The strategy.exit() command generates *exit orders*. It features several unique behaviors that link to open trades, helping to simplify closing market positions and creating multi-level exits with *take-profit*, *stop-loss*, and *trailing stop* orders.

Unlike other order placement commands, which can generate a *single order* per call, each call to strategy.exit() can produce *more than one* type of exit order, depending on its arguments. Additionally, a single call to this command can generate exit orders for *multiple entries*, depending on the specified `from_entry` value and the strategy's open trades.

Take-profit and stop-loss

The most basic use of the strategy.exit() command is the placement of limit orders to trigger exits after earning enough money (take-profit), stop orders to trigger exits after losing too much money (stop-loss), or both (bracket).

Four parameters determine the prices of the command's take-profit and stop-loss orders:

- The `profit` and `loss` parameters accept *relative* values representing the number of *ticks* the market price must move away from the entry price to trigger an exit.
- The `limit` and `stop` parameters accept *absolute* values representing the specific *prices* that trigger an exit when the market price reaches them.

When a strategy.exit() call includes arguments for the relative *and* absolute parameters defining take-profit or stop-loss levels (`profit` and `limit` or `loss` and `stop`), it creates orders only at the levels expected to trigger exits *first*.

For instance, if the `profit` distance is 19 ticks and the `limit` level is 20 ticks past the entry price in the favorable direction, the strategy.exit() command places a take-profit order `profit` ticks past the entry price because the market price will move that distance before reaching the `limit` value. In contrast, if the `profit` distance is 20 ticks and the `limit` level is 19 ticks

past the entry price in the favorable direction, the command places a take-profit order at the `limit` level because the price will reach that value first.

The following example creates exit bracket (take-profit and stop-loss) orders with the strategy.exit() command. When the `buyCondition` occurs, the script calls strategy.entry() to place a “buy” market order. It also calls strategy.exit() with `limit` and `stop` arguments to create a take-profit order at the `limitPrice` and a stop-loss order at the `stopPrice`. The script plots the `limitPrice` and `stopPrice` values on the chart to visualize the exit order prices:

image

//@version=6 strategy("Take-profit and stop-loss demo", overlay = true) // @variable Is `true` on every 100th bar. bool buyCondition = bar_index % 100 == 0 //@variable The current take-profit order price. var float takeProfit = na //@variable The current stop-loss order price. var float stopLoss = na if buyCondition // Update the `takeProfit` and `stopLoss` values. if strategy.opentrades == 0 takeProfit := close * 1.01 stopLoss := close * 0.99 // Place a long market order. strategy.entry("buy", strategy.long) // Place a take-profit order at the `takeProfit` price and a stop-loss order at the `stopLoss` price. strategy.exit("exit", "buy", limit = takeProfit, stop = stopLoss) // Set `takeProfit` and `stopLoss` to `na` when the position closes. if ta.change(strategy.closedtrades) > 0 takeProfit := na stopLoss := na // Plot the `takeProfit` and `stopLoss` values. plot(takeProfit, "TP", color.green, style = plot.style_circles) plot(stopLoss, "SL", color.red, style = plot.style_circles)

Note that:

- We did not specify a `qty` or `qty_percent` argument in the strategy.exit() call, meaning it creates orders to exit 100% of the “buy” order’s size.
- The strategy.exit() command’s exit orders *do not* necessarily execute at the specified prices. Strategies can fill limit orders at *better* prices and stop orders at *worse* prices, depending on the range of values available to the broker emulator.

When a strategy.exit() call includes a `from_entry` argument, the resulting exit orders only apply to existing entry orders that have a matching ID. If the specified `from_entry` value does not match the ID of any entry in the current position, the command *does not* create any exit orders.

Below, we changed the `from_entry` argument of the strategy.exit() call in our previous script to “buy2”, which means it creates exit orders only for open trades with the “buy2” entry ID. This version does not place *any* exit orders because it does not create any entry orders with the “buy2” ID:

//@version=6 strategy("Invalid `from_entry` ID demo", overlay = true) // @variable Is `true` on every 100th bar. bool buyCondition = bar_index % 100 == 0 //@variable The current take-profit order price. var float takeProfit = na //@variable The current stop-loss order price. var float stopLoss = na if buyCondition // Update the `takeProfit` and `stopLoss` values before entering the trade. if strategy.opentrades == 0 takeProfit := close * 1.01 stopLoss := close * 0.99 // Place a long market order. strategy.entry("buy", strategy.long) // Attempt to place an exit bracket for "buy2" entries. // This call has no effect because the strategy does not create entry orders with the "buy2" ID. strategy.exit("exit", "buy2", limit = takeProfit, stop = stopLoss) // Set `takeProfit` and `stopLoss` to `na` when the position closes. if ta.change(strategy.closedtrades) > 0 takeProfit := na stopLoss := na // Plot the `takeProfit` and `stopLoss` values. plot(takeProfit, "TP", color.green, style = plot.style_circles) plot(stopLoss, "SL", color.red, style = plot.style_circles)

Note that:

- When a strategy.exit() call *does not* include a `from_entry` argument, it creates exit orders for *all* the position’s open trades, regardless of their entry IDs. See the Exits for multiple entries section below to learn more.

Partial and multi-level exits

Strategies can use more than one call to strategy.exit() to create successive *partial* exit orders for the same entry ID, helping to simplify the formation of multi-level exit strategies. To use multiple strategy.exit() calls to exit from an open trade, include a `qty` or `qty_percent` argument in each call to specify how much of the traded quantity to close. If the sum of the exit order sizes exceeds the open position, the strategy automatically *reduces* their sizes to match the position.

Note that:

- When a strategy.exit() call includes *both* `qty` and `qty_percent` arguments, the command uses the `qty` value to size the order and ignores the `qty_percent` value.

This example demonstrates a simple strategy that creates two partial exit order brackets for an entry ID. When the `buyCondition` occurs, the script places a “buy” market order for two shares with strategy.entry(), and it creates “exit1” and “exit2” brackets using two calls to strategy.exit(). The first call uses a `qty` of 1, and the second uses a `qty` of 3:

image

//@version=6 strategy("Multi-level exit demo", "test", overlay = true) // @variable Is `true` on every 100th bar. bool buyCondition = bar_index % 100 == 0 //@variable The take-profit price for "exit1" orders. var float takeProfit1 = na //@variable The take-profit price for "exit2" orders. var float takeProfit2 = na //@variable The stop-loss price for "exit1" orders. var float stopLoss1 = na //@variable The stop-loss price for "exit2" orders. var float stopLoss2 = na if buyCondition // Update the `takeProfit*` and `stopLoss*` values before entering the trade. if strategy.opentrades == 0 takeProfit1 := close * 1.01 takeProfit2 := close * 1.02 stopLoss1 := close * 0.99 stopLoss2 := close * 0.98 // Place a long market order with a `qty` of 2. strategy.entry("buy", strategy.long, qty = 2) // Place an "exit1" bracket with a `qty` of 1 at the `takeProfit1` and `stopLoss1` prices. strategy.exit("exit1", "buy", limit = takeProfit1, stop = stopLoss1, qty = 1) // Place an "exit2" bracket with a `qty` of 3 at the `takeProfit1` and `stopLoss1` prices. // The size of the resulting orders decreases to match the open position. strategy.exit("exit2", "buy", limit = takeProfit2, stop = stopLoss2, qty = 3) // Set `takeProfit1` and `stopLoss1` to `na` when the price touches either value. if high >= takeProfit1 or low <= stopLoss1 takeProfit1 := na stopLoss1 := na // Set `takeProfit2` and `stopLoss2` to `na` when the price touches either value. if high >= takeProfit2 or low <= stopLoss2 takeProfit2 := na stopLoss2 := na // Plot the `takeProfit*` and `stopLoss*` values. plot(takeProfit1, "TP1", color.green, style = plot.style_circles) plot(takeProfit2, "TP2", color.green, style = plot.style_circles) plot(stopLoss1, "SL1", color.red, style = plot.style_circles) plot(stopLoss2, "SL2", color.red, style = plot.style_circles)

As we can see from the trade markers on the chart above, the strategy first executes the “exit1” take-profit or stop-loss order to reduce the open position by one share, leaving one remaining share in the position. However, we specified a size of *three shares* for the “exit2” order bracket, which exceeds the remaining position. Rather than using this specified quantity, the strategy automatically *reduces* the “exit2” orders to one share, allowing it to close the position successfully.

Note that:

- This strategy only fills **one** exit order from the “exit1” bracket, **not both**. When a strategy.exit() call generates more than one exit order type for an entry ID, the strategy fills the only the *first* triggered one and automatically cancels the others.
- The strategy reduced the “exit2” orders because all orders from the strategy.exit() calls automatically belong to the same strategy.oca.reduce group by default. Learn more about OCA groups below.

When creating multiple exit orders with *different* strategy.exit() calls, it’s crucial to note that the orders from each call *reserve* a portion of the open position. The orders from one strategy.exit() call *cannot* exit the portion of a position that a previous call already reserved.

For example, this script generates a “buy” entry order for 20 shares with a strategy.entry() call and “limit” and “stop” exit orders with two separate calls to strategy.exit() 100 bars before the last chart bar. We specified a quantity of 19 shares for the “limit” order and 20 for the “stop” order:

//@version=6 strategy("Reserved exit demo", "test", overlay = true) // @variable The price of the "limit" exit order. var float limitPrice = na // @variable The price of the "stop" exit order. var float stopPrice = na // @variable Is `true` 100 bars before the last chart bar. bool longCondition = last_bar_index - bar_index == 100 if longCondition // Update the `limitPrice` and `stopPrice`. limitPrice := close * 1.01 stopPrice := close * 0.99 // Place a long market order for 20 shares. strategy.entry("buy", strategy.long, 20) // Create a take-profit order for 19 shares at the `limitPrice`. strategy.exit("limit", limit = limitPrice, qty = 19) // Create a stop-loss order at the `stopPrice`. Although this call specifies a `qty` of 20, the previous //

`strategy.exit()` call reserved 19, meaning this call creates an exit order for only 1 share. strategy.exit("stop", stop = stopPrice, qty = 20) //@variable Is `true` when the strategy has an open position, `false` otherwise. bool showPlot = strategy.opentrades == 1 // Plot the `limitPrice` and `stopPrice` when `showPlot` is `true`. plot(showPlot ? limitPrice : na, "Limit (take-profit) price", color.green, 2, plot.style_linebr) plot(showPlot ? stopPrice : na, "Stop (stop-loss) price", color.red, 2, plot.style_linebr)

Users unfamiliar with the strategy.exit() command's unique behaviors might expect this strategy to close the entire market position if it fills the "stop" order before the "limit" order. However, the trade markers in the chart below show that the "stop" order only reduces the position by **one share**. The strategy.exit() call for the "limit" order executes first in the code, reserving 19 shares of the open position for closure with that order. This reservation leaves only one share available for the "stop" order to close, regardless of when the strategy fills it:

image

Trailing stops

One of the strategy.exit() command's key features is its ability to create *trailing stops*, i.e., stop-loss orders that trail behind the market price by a specified amount whenever it moves to a better value in the favorable direction (upward for long positions and downward for short positions).

This type of exit order has two components: an *activation level* and a *trail offset*. The activation level is the value the market price must cross to activate the trailing stop calculation, and the trail offset is the distance the activated stop follows behind the price as it reaches successively better values.

Three strategy.exit() parameters determine the activation level and trail offset of a trailing stop order:

- The `trail_price` parameter accepts an *absolute price value* for the trailing stop's activation level.

- The `trail_points` parameter is an alternative way to specify the activation level. Its value represents the *tick distance* from the entry price required to activate the trailing stop.
- The `trail_offset` parameter accepts a value representing the order's trail offset as a specified number of ticks.

To create and activate a trailing stop order, a strategy.exit() call must specify a `trail_offset` argument and either a `trail_price` or `trail_points` argument. If the call contains both `trail_price` and `trail_points` arguments, the command uses the level expected to activate the stop *first*. For instance, if the `trail_points` distance is 50 ticks and the `trail_price` value is 51 ticks past the entry price in the favorable direction, the strategy.exit() command uses the `trail_points` value to set the activation level because the market price will move that distance *before* reaching the `trail_price` level.

The example below demonstrates how a trailing stop order works in detail. The strategy places a "Long" market order with the strategy.entry() command 100 bars before the last chart bar, and it calls strategy.exit() with `trail_price` and `trail_offset` arguments on the following bar to create a trailing stop. The script uses lines, labels, and a plot to visualize the trailing stop's behavior.

The green line on the chart shows the level the market price must reach to activate the trailing stop order. After the price reaches this level from below, the script uses a blue plot to display the trailing stop's price. Each time the market price reaches a new high after activating the trailing stop, the stop's price *increases* to maintain a distance of `trailOffsetInput` ticks from the best value. The exit order *does not* change its price level when the price decreases or does not reach a new high. Eventually, the market price crosses below the trailing stop, triggering an exit:

image

//@version=6 strategy("Trailing stop order demo", overlay = true, margin_long = 100, margin_short = 100) //@variable The distance from the entry price required to activate the trailing stop. int activationOffsetInput = input.int(1000, "Activation level offset (in ticks)", 0) //@variable The distance the stop follows behind the highest `high` after activation. int trailOffsetInput

= input.int(2000, "Trailing stop offset (in ticks)", 0) //@variable Draws a label and an optional line at the specified `price`. debugDrawings(float price, string txt, color drawingColor, bool drawLine = false) => // Draw a label showing the `txt` at the `price` on the current bar. label.new(bar_index, price, text = txt, color = drawingColor, textcolor = color.white, style = label.style_label_lower_right, size = size.large) // Draw a horizontal line at the `price` starting from the current bar when `drawLine` is `true`. line.new(bar_index, price, bar_index + 1, price, color = drawingColor, extend = extend.right, style = line.style_dashed) //@variable The level required to activate the trailing stop. var float activationLevel = na //@variable The price of the trailing stop. var float trailingStop = na //@variable The value that the trailing stop would have if it was currently active. float theoreticalStopPrice = high - trailOffsetInput * syminfo.mintick // Place a long market order 100 bars before the last historical bar. if last_bar_index - bar_index == 100 strategy.entry("Long", strategy.long) // Create and visualize the exit order on the next bar. if last_bar_index - bar_index == 99 // Update the `activationLevel`. activationLevel := open + syminfo.mintick * activationOffsetInput // Create the trailing stop order that activates at the `activationLevel` and trails behind the `high` by // `trailOffsetInput` ticks. strategy.exit("Trailing Stop", from_entry = "Long", trail_price = activationLevel, trail_offset = trailOffsetInput) // Create drawings to signify the activation level. debugDrawings(activationLevel, "Trailing Stop Activation Level", color.green, true) // Visualize the trailing stop's levels while the position is open. if strategy.opentrades == 1 // Create drawings when the `high` is above the `activationLevel` for the first time to show when the // stop activates. if na(trailingStop) and high >= activationLevel debugDrawings(activationLevel, "Activation level crossed", color.green) trailingStop := theoreticalStopPrice debugDrawings(trailingStop, "Trailing Stop Activated", color.blue) // Otherwise, update the `trailingStop` value when the `theoreticalStopPrice` reaches a new high. else if theoreticalStopPrice > trailingStop trailingStop := theoreticalStopPrice // Plot the `trailingStop` value to visualize the trailing price movement. plot(trailingStop, "Trailing Stop")

Exits for multiple entries

A single call to the strategy.exit() command can generate exit orders for *more than one* entry in an open position, depending on the call's `from_entry` value.

If an open position consists of two or more entries with the same ID, a single call to strategy.exit() with that ID as the `from_entry` argument places exit orders for each corresponding entry created before or on the bar where the call occurs.

For example, this script periodically calls strategy.entry() on two consecutive bars to enter and add to a long position. Both calls use "buy" as the `id` argument. After creating the second entry, the script calls strategy.exit() once with "buy" as its `from_entry` argument to generate separate exit orders for each entry with that ID. When the market price reaches the `takeProfit` or `stopLoss` value, the broker emulator fills *two* exit orders and closes the position:

image

//@version=6 strategy("Exits for entries with the same ID demo", overlay = true, pyramiding = 2) //@variable Take-profit price for exit commands. var float takeProfit = na //@variable Stop-loss price for exit commands. var float stopLoss = na //@variable Is `true` on two consecutive bars in 100-bar cycles. bool buyCondition = math.min(bar_index % 100, math.max(bar_index - 1, 0) % 100) == 0 if buyCondition // Place a "buy" market order to enter a trade. strategy.entry("buy", strategy.long) // Calculate exits on the second order. if strategy.opentrades == 1 // Update the `takeProfit` and `stopLoss`. takeProfit := close * 1.01 stopLoss := close * 0.99 // Place exit orders for both "buy" entries. strategy.exit("exit", "buy", limit = takeProfit, stop = stopLoss) // Set `takeProfit` and `stopLoss` to `na` when both trades close. if ta.change(strategy.closedtrades) == 2 takeProfit := na stopLoss := na // Plot the `takeProfit` and `stopLoss` values. plot(takeProfit, "TP", color.green, style = plot.style_circles) plot(stopLoss, "SL", color.red, style = plot.style_circles)

A single strategy.exit() call can also generate exit orders for *all* entries in an open position, irrespective of entry ID, when it does not include a `from_entry` argument.

Here, we changed the strategy.entry() instance in the above script to create an entry order with a distinct ID on each call, and we removed the `from_entry` argument from the strategy.exit() call. Since this version does not specify which entries the exit orders apply to, the strategy.exit() call creates orders for *every* entry in the position:

image

//@version=6 strategy("Exits for entries with different IDs demo", overlay = true, pyramiding = 2) //@variable Take-profit price for exit commands. var float takeProfit = na //@variable Stop-loss price for exit commands. var float stopLoss = na //@variable Is `true` on two consecutive bars in 100-bar cycles. bool buyCondition = math.min(bar_index % 100, math.max(bar_index - 1, 0) % 100) == 0 if buyCondition // Place a long market order with a unique ID. strategy.entry("buy" + str.tostring(strategy.opentrades + strategy.closedtrades), strategy.long) // Calculate exits on the second order. if strategy.opentrades == 1 // Update the `takeProfit` and `stopLoss`. takeProfit := close * 1.01 stopLoss := close * 0.99 // Place exit orders for ALL entries in the position, irrespective of ID. strategy.exit("exit", limit = takeProfit, stop = stopLoss) // Set `takeProfit` and `stopLoss` to `na` when both trades close. if ta.change(strategy.closedtrades) == 2 takeProfit := na stopLoss := na // Plot the `takeProfit` and `stopLoss` values. plot(takeProfit, "TP", color.green, style = plot.style_circles) plot(stopLoss, "SL", color.red, style = plot.style_circles)

It's crucial to note that a call to strategy.exit() without a `from_entry` argument *persists* and creates exit orders for all open trades in a position, regardless of *when* the entries occur. This behavior can affect strategies that manage positions with multiple entries or exits. When a strategy has an open position and calls strategy.exit() on any bar without specifying a `from_entry` ID, it generates exit orders for each entry created *before* or on that bar, and it continues to generate exit orders for subsequent entries *after* that bar until the position closes.

Let's explore this behavior and how it works. The script below creates a long entry order with strategy.entry() on each bar within a user-specified time range, and it calls strategy.exit() without a `from_entry` argument on *one bar* within that range to generate exit orders for *every* entry in the open position.

The exit command uses a `loss` value of 0, which means an exit order fills each time the market price is not above an entry order's price.

The script prompts users to select three points before it starts its calculations. The first point specifies when order creation begins, the second determines when the single strategy.exit() call occurs, and the third specifies when order creation stops:

image

//@version=6 strategy("Exit persist demo", overlay = true, margin_long = 100, margin_short = 100, pyramiding = 100) //@variable The time when order creation starts. int entryStartTime = input.time(0, "Start time for entries", confirm = true) //@variable The time when the `strategy.exit()` call occurs. int exitCallTime = input.time(0, "Exit call time", confirm = true) // @variable The time when order creation stops. int entryEndTime = input.time(0, "End time for entries", confirm = true) // Raise a runtime error if incorrect timestamps are chosen. if exitCallTime <= entryStartTime or entryEndTime <= exitCallTime or entryEndTime <= entryStartTime runtime.error("The input timestamps must follow this condition: entryStartTime < exitCallTime < entryEndTime.") // Create variables to track entry and exit conditions. bool entriesStart = time == entryStartTime bool callExit = time == exitCallTime bool entriesEnd = time == entryEndTime bool callEntry = time >= entryStartTime and time < entryEndTime // Place a long entry order when `callEntry` is `true`. if callEntry strategy.entry("Entry", strategy.long) // Call `strategy.exit()` when `callExit` is `true`, which occurs only once. // This single call persists and creates exit orders for EVERY entry in the position because it does not // specify a `from_entry` ID. if callExit strategy.exit("Exit", loss = 0) // Draw labels to signify when entries start, when the `strategy.exit()` call occurs, and when order placement stops. switch entriesStart => label.new(bar_index, high, "Start placing entry orders.", color = color.green, textcolor = color.white, style = label.style_label_lower_right, size = size.large) callExit => label.new(bar_index, high, "Call `strategy.exit()` once.", color = color.blue, textcolor = color.white, style = label.style_label_lower_right, size = size.large) entriesEnd => label.new(bar_index, high, "Stop placing orders.", color = color.red, textcolor = color.white, style = label.style_label_lower_left, size = size.large) // Create a line and label to visualize the lowest entry price, i.e., the price required to close the position. var line lowestLine =

line.new(entryStartTime + 1000, na, entryEndTime, na, xloc.bar_time, extend.right, color.orange, width = 2) var lowestLabel = label.new(entryStartTime + 1000, na, "Lowest entry price", color = color.orange, style = label.style_label_upper_right, xloc = xloc.bar_time) // Update the price values of the `lowestLine` and `lowestLabel` after each new entry. if callEntry[1] var float lowestPrice = strategy.opentrades.entry_price(0) float entryPrice = strategy.opentrades.entry_price(strategy.opentrades - 1) if not na(entryPrice) lowestPrice := math.min(lowestPrice, entryPrice) lowestLine.set_y1(lowestPrice) lowestLine.set_y2(lowestPrice) lowestLabel.set_y(lowestPrice) // Highlight the background when `entriesStart`, `callExit`, and `entriesEnd` occurs. bgcolor(entriesStart ? color.new(color.green, 80) : na, title = "Entries start highlight") bgcolor(callExit ? color.new(color.blue, 80) : na, title = "Exit call highlight") bgcolor(entriesEnd ? color.new(color.red, 80) : na, title = "Entries end highlight")

Note that:

- We included `pyramiding = 100` in the strategy() declaration statement, which allows the position to have up to 100 open entries from strategy.entry().
- The script uses labels and bgcolor() to signify when order placement starts and stops and when the strategy.exit() call occurs.
- The script draws a line and a label at the lowest entry price to show the value the market price must reach to close the position.

We can observe the unique strategy.exit() behavior in this example by comparing the code itself with the script's chart outputs. The script calls strategy.exit() *one time*, only on the bar with the blue label. However, this single call placed exit orders for every entry **before** or on that bar and continued placing exit orders for all entries **after** that bar. This behavior occurs because strategy.exit() has no way to determine when to stop placing orders if it does not link to entries with a specific ID. In this case, the command only ceases to create new exit orders after the position fully closes.

The above script would exhibit different behavior if we included a `from_entry` argument in the strategy.exit() call. When a call to this

command specifies a `from_entry` ID, it only applies to entries with that ID which the strategy created *before* or *on* the bar of the call. The command does not place exit orders for subsequent entries created *after* that bar in that case, even ones with the same ID.

Here, we added `from_entry = "Entry"` to our script's strategy.exit() call, meaning it only produces exit orders for entries with the "Entry" ID. Only 17 exits occur this time, each corresponding to an entry order created before or on the bar with the blue label. The call does not affect any entries that the strategy creates *after* that bar:

image

//@version=6 strategy("Exit persist demo", overlay = true, margin_long = 100, margin_short = 100, pyramiding = 100) //@variable The time when order creation starts. int entryStartTime = input.time(0, "Start time for entries", confirm = true) //@variable The time when the `strategy.exit()` call occurs. int exitCallTime = input.time(0, "Exit call time", confirm = true) // @variable The time when order creation stops. int entryEndTime = input.time(0, "End time for entries", confirm = true) // Raise a runtime error if incorrect timestamps are chosen. if exitCallTime <= entryStartTime or entryEndTime <= exitCallTime or entryEndTime <= entryStartTime runtime.error("The input timestamps must follow this condition: entryStartTime < exitCallTime < entryEndTime.") // Create variables to track entry and exit conditions. bool entriesStart = time == entryStartTime bool callExit = time == exitCallTime bool entriesEnd = time == entryEndTime bool callEntry = time >= entryStartTime and time < entryEndTime // Place a long entry order when `callEntry` is `true`. if callEntry strategy.entry("Entry", strategy.long) // Call `strategy.exit()` when `callExit` is `true`, which occurs only once. // This single call only places exit orders for all entries with the "Entry" ID created before or on the bar where // `callExit` occurs. It DOES NOT affect any subsequent entries created after that bar. if callExit strategy.exit("Exit", from_entry = "Entry", loss = 0) // Draw labels to signify when entries start, when the `strategy.exit()` call occurs, and when order placement stops. switch entriesStart => label.new(bar_index, high, "Start placing entry orders.", color = color.green, textcolor = color.white, style = label.style_label_lower_right, size = size.large) callExit => label.new(bar_index, high, "Call `strategy.exit()` once.", color = color.blue, textcolor = color.white, style = label.style_label_lower_right, size =

size.large) entriesEnd => label.new(bar_index, high, "Stop placing orders.", color = color.red, textcolor = color.white, style = label.style_label_lower_left, size = size.large) // Create a line and label to visualize the lowest entry price, i.e., the price required to close the position. var line lowestLine = line.new(entryStartTime + 1000, na, entryEndTime, na, xloc.bar_time, extend.right, color.orange, width = 2) var lowestLabel = label.new(entryStartTime + 1000, na, "Lowest entry price", color = color.orange, style = label.style_label_upper_right, xloc = xloc.bar_time) // Update the price values of the `lowestLine` and `lowestLabel` after each new entry. if callEntry[1] var float lowestPrice = strategy.opentrades.entry_price(0) float entryPrice = strategy.opentrades.entry_price(strategy.opentrades - 1) if not na(entryPrice) lowestPrice := math.min(lowestPrice, entryPrice) lowestLine.set_y1(lowestPrice) lowestLine.set_y2(lowestPrice) lowestLabel.set_y(lowestPrice) // Highlight the background when `entriesStart`, `callExit`, and `entriesEnd` occurs. bgcolor(entriesStart ? color.new(color.green, 80) : na, title = "Entries start highlight") bgcolor(callExit ? color.new(color.blue, 80) : na, title = "Exit call highlight") bgcolor(entriesEnd ? color.new(color.red, 80) : na, title = "Entries end highlight")

strategy.close() and strategy.close_all()

The strategy.close() and strategy.close_all() commands generate orders to exit from an open position. Unlike strategy.exit(), which creates *price-based* exit orders (e.g., stop-loss), these commands generate market orders that the broker emulator fills on the next available tick, irrespective of the price.

The example below demonstrates a simple strategy that places a “buy” entry order with strategy.entry() once every 50 bars and a market order to close the long position with strategy.close() 25 bars afterward:

image

//@version=6 strategy("Close demo", "test", overlay = true) //@variable Is `true` on every 50th bar. buyCond = bar_index % 50 == 0 //@variable Is `true` on every 25th bar except for those that are divisible by 50. sellCond = bar_index % 25 == 0 and not buyCond if buyCond strategy.entry("buy", strategy.long) if sellCond strategy.close("buy") bgcolor(buyCond ?

color.new(color.blue, 90) : na) bgcolor(sellCond ? color.new(color.red, 90) : na)

Notice that the strategy.close() call in this script uses “buy” as its required `id` argument. Unlike strategy.exit(), this command’s `id` parameter specifies the *entry ID* of an open trade. It **does not** represent the ID of the resulting exit order. If a market position consists of multiple open trades with the same entry ID, a single strategy.close() call with that ID as its `id` argument generates a single market order to exit from all of them.

The following script creates a “buy” order with strategy.entry() once every 25 bars, and it calls strategy.close() with “buy” as its `id` argument to close all open trades with that entry ID once every 100 bars. The market order from strategy.close() closes the entire position in this case because every open trade has the same “buy” entry ID:

image

//@version=6 strategy("Multiple close demo", "test", overlay = true, pyramiding = 3) //@variable Is `true` on every 100th bar. sellCond = bar_index % 100 == 0 //@variable Is `true` on every 25th bar except for those that are divisible by 100. buyCond = bar_index % 25 == 0 and not sellCond if buyCond strategy.entry("buy", strategy.long) if sellCond strategy.close("buy") bgcolor(buyCond ? color.new(color.blue, 90) : na) bgcolor(sellCond ? color.new(color.red, 90) : na)

Note that:

- We included `pyramiding = 3` in the strategy() declaration statement, allowing the script to generate up to three entries per position with strategy.entry() calls.

The strategy.close_all() command generates a market order to exit from the open position that *does not* link to any specific entry ID. This command is helpful when a strategy needs to exit as soon as possible from a position consisting of multiple open trades with different entry IDs.

The script below places “A”, “B”, and “C” entry orders sequentially based on the number of open trades as tracked by the strategy.opentrades variable,

and then it calls strategy.close_all() to create a single order that closes the entire position on the following bar:

image

//@version=6 strategy("Close multiple ID demo", "test", overlay = true, pyramiding = 3) switch strategy.opentrades 0 => strategy.entry("A", strategy.long) 1 => strategy.entry("B", strategy.long) 2 => strategy.entry("C", strategy.long) 3 => strategy.close_all()

`strategy.cancel()` and `strategy.cancel_all()`

The strategy.cancel() and strategy.cancel_all() commands allow strategies to cancel *unfilled* orders before the broker emulator processes them. These order cancellation commands are most helpful when working with *price-based orders*, including all orders from strategy.exit() calls and the orders from strategy.entry() and strategy.order() calls that use `limit` or `stop` arguments.

The strategy.cancel() command has a required `id` parameter, which specifies the ID of the entry or exit orders to cancel. The strategy.cancel_all() command does not have such a parameter because it cancels *all* unfilled orders, regardless of ID.

The following strategy places a “buy” limit order 500 ticks below the closing price 100 bars before the last chart bar with strategy.entry(), and it cancels the order on the next bar with strategy.cancel(). The script highlights the chart’s background to signify when it places and cancels the “buy” order, and it draws a horizontal line at the order’s price. As we see below, our example chart shows no entry marker when the market price crosses the horizontal line because the strategy already cancels the order (when the chart’s background is orange) before it reaches that level:

image

//@version=6 strategy("Cancel demo", "test", overlay = true) //@variable Draws a horizontal line at the `limit` price of the "buy" order. var line limitLine = na //@variable Is `color.green` when the strategy places the "buy" order, `color.orange` when it cancels the order. color bgColor = na if last_bar_index - bar_index == 100 float limitPrice = close - syminfo.mintick *

500 strategy.entry("buy", strategy.long, limit = limitPrice) limitLine := line.new(bar_index, limitPrice, bar_index + 1, limitPrice, extend = extend.right) bgColor := color.new(color.green, 50) if last_bar_index - bar_index == 99 strategy.cancel("buy") bgColor := color.new(color.orange, 50) bgcolor(bgColor)

The strategy.cancel() command affects *all* unfilled orders with a specified ID. It does nothing if the specified `id` represents the ID of an order that does not exist. When there is more than one unfilled order with the specified ID, the command cancels *all* of them at once.

Below, we've modified the previous script to place a "buy" limit order on three consecutive bars, starting 100 bars before the last chart bar. After placing all three orders, the strategy cancels them using strategy.cancel() with "buy" as the `id` argument, resulting in nothing happening when the market price reaches any of the order prices (horizontal lines):

image

//@version=6 strategy("Multiple cancel demo", "test", overlay = true, pyramiding = 3) //@variable Draws a horizontal line at the `limit` price of the "buy" order. var line limitLine = na //@variable Is `color.green` when the strategy places the "buy" order, `color.orange` when it cancels the order. color bgColor = na if last_bar_index - bar_index <= 100 and last_bar_index - bar_index >= 98 float limitPrice = close - syminfo.mintick * 500 strategy.entry("buy", strategy.long, limit = limitPrice) limitLine := line.new(bar_index, limitPrice, bar_index + 1, limitPrice, extend = extend.right) bgColor := color.new(color.green, 50) if last_bar_index - bar_index == 97 strategy.cancel("buy") bgColor := color.new(color.orange, 50) bgcolor(bgColor)

Note that:

- We included `pyramiding = 3` in the strategy() declaration statement, allowing three successive entries from strategy.entry() per position. The script would also achieve the same result without this setting if it called strategy.order() instead because pyramiding *does not* affect orders from that command.

The strategy.cancel() and strategy.cancel_all() commands can cancel orders of any type, including market orders. However, it is important to note that either command can cancel a market order only if its call occurs on the *same* script execution as the order placement command. If the call happens after that point, it has *no effect* because the broker emulator fills market orders on the *next available tick*.

This example places a “buy” market order 100 bars before the last chart bar with strategy.entry(), then it attempts to cancel the order on the next bar with strategy.cancel_all(). The cancellation command *does not* affect the “buy” order because the broker emulator fills the order on the next bar’s *opening tick*, which occurs *before* the script evaluates the strategy.cancel_all() call:

image

//@version=6 strategy("Cancel market demo", "test", overlay = true) // @variable Is `color.green` when the strategy places the "buy" order, `color.orange` when it tries to cancel the order. color bgColor = na if last_bar_index - bar_index == 100 strategy.entry("buy", strategy.long) bgColor := color.new(color.green, 50) if last_bar_index - bar_index == 99 strategy.cancel_all() bgColor := color.new(color.orange, 50) bgcolor(bgColor)

Position sizing

Pine Script strategies feature two ways to control the sizes of the orders that open and manage positions:

- Set a default *fixed* quantity type and value for the orders. Programmers can specify defaults for these properties by including `default_qty_type` and `default_qty_value` arguments in the strategy() declaration statement. Script users can adjust these values with the “Order size” inputs in the “Settings/Properties” tab.
- Include a *non-na* `qty` argument in the strategy.entry() or strategy.order() call. When a call to either of these commands specifies a non-na `qty` value, that call ignores the strategy’s default quantity type and value and places an order for `qty` contracts/shares/lots/units instead.

The following example uses strategy.entry() calls with different `qty` values for long and short trades. When the current bar's low equals the `lowest` value, the script places a "Buy" order to enter a long position of `longAmount` units. Otherwise, when the high equals the `highest` value, it places a "Sell" order to enter a short position of `shortAmount` units:

image

```
//@version=6 strategy("Buy low, sell high", overlay = true, default_qty_type = strategy.cash, default_qty_value = 5000) int length = input.int(20, "Length", 1) float longAmount = input.float(4.0, "Long Amount", 0.0) float shortAmount = input.float(2.0, "Short Amount", 0.0) float highest = ta.highest(length) float lowest = ta.lowest(length) switch low == lowest => strategy.entry("Buy", strategy.long, longAmount) high == highest => strategy.entry("Sell", strategy.short, shortAmount)
```

Notice that although we've included `default_qty_type` and `default_qty_value` arguments in the strategy() declaration statement, the strategy *does not* use this default setting to size its orders because the specified `qty` in the entry commands takes precedence. If we want to use the default size, we must *remove* the `qty` arguments from the strategy.entry() calls or set their values to na.

Here, we edited the previous script by including ternary expressions for the `qty` arguments in both strategy.entry() calls that replace input values of 0 with na. If the specified `longAmount` or `shortAmount` is 0, which is what we set as the new default, the corresponding entry orders use the strategy's default order size instead, as we see below:

image

```
//@version=6 strategy("Buy low, sell high", overlay = true, default_qty_type = strategy.cash, default_qty_value = 5000) int length = input.int(20, "Length", 1) float longAmount = input.float(0.0, "Long Amount", 0.0) float shortAmount = input.float(0.0, "Short Amount", 0.0) float highest = ta.highest(length) float lowest = ta.lowest(length) switch low == lowest => strategy.entry("Buy", strategy.long, longAmount == 0.0 ? na : longAmount) high == highest => strategy.entry("Sell", strategy.short, shortAmount == 0.0 ? na : shortAmount)
```

Closing a market position

By default, strategies close a market position using the *First In, First Out (FIFO)* method, which means that any exit order closes or reduces the position starting with the *first* open trade, even if the exit command specifies the entry ID of a *different* open trade. To override this default behavior, include `close_entries_rule = "ANY"` in the strategy() declaration statement.

The following example places “Buy1” and “Buy2” entry orders sequentially, starting 100 bars before the latest chart bar. When the position size is 0, it calls strategy.entry() to place the “Buy1” order for five units. After the strategy’s position size matches the size of that order, it uses strategy.entry() to place the “Buy2” order for ten units. The strategy then creates “bracket” exit orders for both entries using a single strategy.exit() call without a `from_entry` argument. For visual reference, the script plots the strategy.position_size value in a separate pane:

image

//@version=6 strategy("Exit Demo", pyramiding = 2) float positionSize = strategy.position_size if positionSize == 0 and last_bar_index - bar_index <= 100 strategy.entry("Buy1", strategy.long, 5) else if positionSize == 5 strategy.entry("Buy2", strategy.long, 10) else if positionSize == 15 strategy.exit("bracket", loss = 10, profit = 10) plot(positionSize == 0 ? na : positionSize, "Position Size", color.lime, 4, plot.style_histogram)

Note that:

- We included `pyramiding = 2` in the strategy() declaration statement, allowing two successive entries from strategy.entry() per position.

Each time the market price triggers an exit order, the above script exits from the open position, starting with the *oldest* open trade. This FIFO behavior applies even if we explicitly specify an exit from “Buy2” before “Buy1” in the code.

The script version below calls strategy.close() with “Buy2” as its `id` argument, and it includes “Buy1” as the `from_entry` argument in the strategy.exit() call. The market order from strategy.close() executes on the next available tick,

meaning the broker emulator fills it *before* the take-profit and stop-loss orders from strategy.exit():

//@version=6 strategy("Exit Demo", pyramiding = 2) float positionSize = strategy.position_size if positionSize == 0 and last_bar_index - bar_index <= 100 strategy.entry("Buy1", strategy.long, 5) else if positionSize == 5 strategy.entry("Buy2", strategy.long, 10) else if positionSize == 15 strategy.close("Buy2") strategy.exit("bracket", "Buy1", loss = 10, profit = 10) plot(positionSize == 0 ? na : positionSize, "Position Size", color.lime, 4, plot.style_histogram)

The market order from the script's strategy.close() call is for 10 units because it links to the open trade with the "Buy2" entry ID. A user might expect this strategy to close that trade completely when the order executes. However, the "List of Trades" tab shows that five units of the order go toward closing the "Buy1" trade *first* because it is the oldest, and the remaining five units close *half* of the "Buy2" trade. After that, the "bracket" orders from the strategy.exit() call close the rest of the position:

image

Note that:

- If we included `close_entries_rule = "ANY"` in the strategy() declaration statement, the market order from strategy.close() would close the open trade with the "Buy2" entry ID *first*, and then the "bracket" orders from strategy.exit() would close the trade with the "Buy1" entry ID.

OCA groups

One-Cancels-All (OCA) groups allow a strategy to fully or partially *cancel* specific orders when the broker emulator executes another order from the same group. To assign an order to an OCA group, include an `oca_name` argument in the call to the order placement command. The strategy.entry() and strategy.order() commands also allow programmers to specify an *OCA type*, which defines whether a strategy cancels, reduces, or does not modify the order after executing other orders.

strategy.oca.cancel

When an order placement command uses strategy.oca.cancel as its `oca_type` argument, the strategy completely *cancels* the resulting order if another order from the same OCA group executes first.

To demonstrate how this OCA type impacts a strategy's orders, consider the following script, which places orders when the `ma1` value crosses the `ma2` value. If the strategy.position_size is 0 when the cross occurs, the strategy places two stop orders with strategy.order() calls. The first is a long order at the bar's high, and the second is a short order at the bar's low. If the strategy already has an open position during the cross, it calls strategy.close_all() to close the position with a market order:

//@version=6 strategy("OCA Cancel Demo", overlay=true) float ma1 = ta.sma(close, 5) float ma2 = ta.sma(close, 9) if ta.cross(ma1, ma2) if strategy.position_size == 0 strategy.order("Long", strategy.long, stop = high) strategy.order("Short", strategy.short, stop = low) else strategy.close_all() plot(ma1, "Fast MA", color.aqua) plot(ma2, "Slow MA", color.orange)

Depending on the price action, the strategy might fill *both* stop orders before creating the closing market order. In that case, the strategy exits the position without evaluating strategy.close_all() because both orders have the same size. We see this behavior in the chart below, where the strategy alternated between executing "Long" and "Short" orders a few times without executing an order from strategy.close_all():

image

To eliminate scenarios where the strategy fills the "Long" and "Short" orders before evaluating the strategy.close_all() call, we can instruct it to *cancel* one of the orders after it executes the other. Below, we included "Entry" as the `oca_name` argument and strategy.oca.cancel as the `oca_type` argument in both strategy.order() calls. Now, after the strategy executes either the "Long" or "Short" order, it cancels the other order and waits for strategy.close_all() to close the position:

image

//@version=6 strategy("OCA Cancel Demo", overlay=true) float ma1 = ta.sma(close, 5) float ma2 = ta.sma(close, 9) if ta.cross(ma1, ma2) if strategy.position_size == 0 strategy.order("Long", strategy.long, stop = high, oca_name = "Entry", oca_type = strategy.oca.cancel) strategy.order("Short", strategy.short, stop = low, oca_name = "Entry", oca_type = strategy.oca.cancel) else strategy.close_all() plot(ma1, "Fast MA", color.aqua) plot(ma2, "Slow MA", color.orange)

strategy.oca.reduce

When an order placement command uses strategy.oca.reduce as its OCA type, the strategy *does not* cancel the resulting order entirely if another order with the same OCA name executes first. Instead, it *reduces* the order's size by the filled number of contracts/shares/lots/units, which is particularly useful for custom exit strategies.

The following example demonstrates a *long-only* strategy that generates a single stop-loss order and two take-profit orders for each new entry. When a faster moving average crosses over a slower one, the script calls strategy.entry() with `qty = 6` to create an entry order, and then it uses three strategy.order() calls to create a stop order at the `stop` price and two limit orders at the `limit1` and `limit2` prices. The strategy.order() call for the "Stop" order uses `qty = 6`, and the two calls for the "Limit 1" and "Limit 2" orders both use `qty = 3`:

//@version=6 strategy("Multiple TP Demo", overlay = true) var float stop = na var float limit1 = na var float limit2 = na bool longCondition = ta.crossover(ta.sma(close, 5), ta.sma(close, 9)) if longCondition and strategy.position_size == 0 stop := close * 0.99 limit1 := close * 1.01 limit2 := close * 1.02 strategy.entry("Long", strategy.long, 6) strategy.order("Stop", strategy.short, stop = stop, qty = 6) strategy.order("Limit 1", strategy.short, limit = limit1, qty = 3) strategy.order("Limit 2", strategy.short, limit = limit2, qty = 3) bool showPlot = strategy.position_size != 0 plot(showPlot ? stop : na, "Stop", color.red, style = plot.style_linebr) plot(showPlot ? limit1 : na, "Limit 1", color.green, style = plot.style_linebr) plot(showPlot ? limit2 : na, "Limit 2", color.green, style = plot.style_linebr)

After adding this strategy to the chart, we see it does not work as initially intended. The problem with this script is that the orders from strategy.order() **do not** belong to an OCA group by default (unlike strategy.exit(), whose orders automatically belong to a strategy.oca.reduce OCA group). Since the strategy does not assign the strategy.order() calls to any OCA group, it does not reduce any unfilled stop or limit orders after executing an order. Consequently, if the broker emulator fills the stop order and at least one of the limit orders, the traded quantity **exceeds** the open long position, resulting in an open *short* position:

image

For our long-only strategy to work as we intended, we must instruct it to *reduce* the sizes of the unfilled stop/limit orders after one of them executes to prevent selling a larger quantity than the open long position.

Below, we specified “Bracket” as the `oca_name` and strategy.oca.reduce as the `oca_type` in all the script’s strategy.order() calls. These changes tell the strategy to reduce the sizes of the orders in the “Bracket” group each time the broker emulator fills one of them. This version of the strategy never simulates a short position because the total size of its filled stop and limit orders never *exceeds* the long position’s size:

image

//@version=6 strategy("Multiple TP Demo", overlay = true) var float stop = na var float limit1 = na var float limit2 = na bool longCondition = ta.crossover(ta.sma(close, 5), ta.sma(close, 9)) if longCondition and strategy.position_size == 0 stop := close * 0.99 limit1 := close * 1.01 limit2 := close * 1.02 strategy.entry("Long", strategy.long, 6) strategy.order("Stop", strategy.short, stop = stop, qty = 6, oca_name = "Bracket", oca_type = strategy.oca.reduce) strategy.order("Limit 1", strategy.short, limit = limit1, qty = 3, oca_name = "Bracket", oca_type = strategy.oca.reduce) strategy.order("Limit 2", strategy.short, limit = limit2, qty = 6, oca_name = "Bracket", oca_type = strategy.oca.reduce) bool showPlot = strategy.position_size != 0 plot(showPlot ? stop : na, "Stop", color.red, style = plot.style_linebr) plot(showPlot ? limit1 : na, "Limit 1", color.green, style = plot.style_linebr) plot(showPlot ? limit2 : na, "Limit 2", color.green, style = plot.style_linebr)

Note that:

- We also changed the `qty` value of the “Limit 2” order to 6 instead of 3 because the strategy reduces its amount by three units when it executes the “Limit 1” order. Keeping the `qty` value of 3 would cause the second limit order’s size to drop to 0 after the strategy fills the first limit order, meaning it would never execute.

`strategy.oca.none`

When an order placement command uses strategy.oca.none as its `oca_type` value, all orders from that command execute *independently* of any OCA group. This value is the default `oca_type` for the strategy.order() and strategy.entry() commands.

Currency

Pine Script strategies can use different currencies in their calculations than the instruments they simulate trades on. Programmers can specify a strategy’s account currency by including a `currency.*` variable as the `currency` argument in the strategy() declaration statement. The default value is currency.NONE, meaning the strategy uses the same currency as the current chart (syminfo.currency). Script users can change the account currency using the “Base currency” input in the script’s “Settings/Properties” tab.

When a strategy script uses an account currency that differs from the chart’s currency, it uses the *previous daily value* of a corresponding currency pair from the most popular exchange to determine the conversion rate. If no exchange provides the rate directly, it derives the rate using a spread symbol. The strategy multiplies all monetary values, including simulated profits/losses, by the determined cross rate to express them in the account currency. To retrieve the rate that a strategy uses to convert monetary values, call request.currency_rate() with syminfo.currency as the `from` argument and strategy.account_currency as the `to` argument.

Note that:

- Programmers can directly convert values expressed in a strategy's account currency to the chart's currency and vice versa via the strategy.convert_to_symbol() and strategy.convert_to_account() functions.

The following example demonstrates how currency conversion affects a strategy's monetary values and how a strategy's cross-rate calculations match those that `request.*()` functions use.

On each of the latest 500 bars, the strategy places an entry order with strategy.entry(), and it places a take-profit and stop-loss order one tick away from the entry price with strategy.exit(). The size of each entry order is `1.0 / syminfo.mintick`, rounded to the nearest tick, which means that the profit/loss of each closed trade is equal to *one point* in the chart's *quote currency*. We specified currency.EUR as the account currency in the strategy() declaration statement, meaning the strategy multiplies all monetary values by a cross rate to express them in Euros.

The script calculates the absolute change in the ratio of the strategy's net profit (strategy.netprofit) to the symbol's point value (syminfo.pointvalue) to determine the value of *one unit* of the chart's currency in Euros. It plots this value alongside the result from a request.currency_rate() call that uses syminfo.currency and strategy.account_currency as the `from` and `to` arguments. As we see below, both plots align, confirming that strategies and `request.*()` functions use the *same* daily cross-rate calculations:

image

//@version=6 strategy("Currency Test", currency = currency.EUR) if last_bar_index - bar_index < 500 // Place an entry order with a size that results in a P/L of `syminfo.pointvalue` units of chart currency per tick. strategy.entry("LE", strategy.long, math.round_to_mintick(1.0 / syminfo.mintick)) // Place exit orders one tick above and below the "LE" entry price, // meaning each trade closes with one point of profit or loss in the chart's currency. strategy.exit("LX", "LE", profit = 1, loss = 1) // Plot the absolute change in `strategy.netprofit / syminfo.pointvalue`, which represents 1 chart unit of profit/loss.
plot(math.abs(ta.change(strategy.netprofit / syminfo.pointvalue)), "1 chart

unit of profit/loss in EUR", color = color.fuchsia, linewidth = 4) // Plot the requested currency rate. plot(request.currency_rate(syminfo.currency, strategy.account_currency), "Requested conversion rate", color.lime)

Note that:

- When a strategy executes on a chart with a timeframe higher than “1D”, it uses the data from *one day before* each *historical* bar’s closing time for its cross-rate calculations. For example, on a “1W” chart, the strategy bases its cross rate on the previous Thursday’s closing values. However, it still uses the latest confirmed daily rate on *realtime* bars.

Altering calculation behavior

Strategy scripts execute across all available historical chart bars and continue to execute on realtime bars as new data comes in. However, by default, strategies only recalculate their values after a bar *closes*, even on realtime bars, and the earliest point that the broker emulator fills the orders a strategy places on the close one bar is at the *open* of the following bar.

Users can change these behaviors with the `calc_on_every_tick`, `calc_on_order_fills`, and `process_orders_on_close` parameters of the strategy() declaration statement or the corresponding inputs in the “Recalculate” and “Fill orders” sections of the script’s “Settings/Properties” tab. The sections below explain how these settings affect a strategy’s calculations.

`calc_on_every_tick`

The `calc_on_every_tick` parameter of the strategy() function determines the frequency of a strategy’s calculations on *realtime bars*. When this parameter’s value is `true`, the script recalculates on each *new tick* in the realtime data feed. Its default value is `false`, meaning the script only executes on a realtime bar after it closes. Users can also toggle this recalculation behavior with the “On every tick” input in the script’s “Settings/Properties” tab.

Enabling this setting can be useful in forward testing because it allows a strategy to use realtime price updates in its calculations. However, it *does*

not affect the calculations on historical bars because historical data feeds *do not* contain complete tick data: the broker emulator considers each historical bar to have only four ticks (open, high, low, and close). Therefore, users should exercise caution and understand the limitations of this setting. If enabling calculation on every tick causes a strategy to behave *differently* on historical and realtime bars, the strategy will **repaint** after the user reloads it.

The following example demonstrates how recalculation on every tick can cause strategy repainting. The script uses strategy.entry() calls to place a long entry order each time the close reaches its `highest` value and a short entry order each time the close reaches its `lowest` value. The strategy() declaration statement includes `calc_on_every_tick = true`, meaning that on realtime bars, it can recalculate and place orders on new price updates *before* a bar closes:

//@version=6 strategy("Donchian Channel Break", overlay = true, calc_on_every_tick = true, pyramiding = 20) int length = input.int(15, "Length") float highest = ta.highest(close, length) float lowest = ta.lowest(close, length) if close == highest strategy.entry("Buy", strategy.long) if close == lowest strategy.entry("Sell", strategy.short) // Highlight the background of realtime bars. bgcolor(barstate.isrealtime ? color.new(color.orange, 80) : na) plot(highest, "Highest", color = color.lime) plot(lowest, "Lowest", color = color.red)

Note that:

- The script uses a pyramiding value of 20, allowing it to simulate up to 20 entries per position with the strategy.entry() command.
- The script highlights the chart's background orange when barstate.isrealtime is `true` to indicate realtime bars.

After applying the script to our chart and letting it run on several realtime bars, we see the following output:

image

The script placed a "Buy" order on *each tick* where the close was at the `highest` value, which happened *more than once* on each realtime bar.

Additionally, the broker emulator filled each market order at the current realtime price rather than strictly at the open of the following chart bar.

After we reload the chart, we see that the strategy *changed* its behavior and *repainted* its results on those bars. This time, the strategy placed only *one* "Buy" order for each *closed bar* where the condition was valid, and the broker emulator filled each order at the open of the following bar. It did not generate multiple entries per bar because what were previously realtime bars became *historical* bars, which **do not** hold complete tick data:

image

`calc_on_order_fills`

The `calc_on_order_fills` parameter of the strategy() function enables a strategy to recalculate immediately after an *order fills*, allowing it to use more granular information and place additional orders without waiting for a bar to close. Its default value is `false`, meaning the strategy does not allow recalculation immediately after every order fill. Users can also toggle this behavior with the "After order is filled" input in the script's "Settings/ Properties" tab.

Enabling this setting can provide a strategy script with additional data that would otherwise not be available until after a bar closes, such as the current average price of a simulated position on an open bar.

The example below shows a simple strategy that creates a "Buy" order with strategy.entry() whenever the strategy.position_size is 0. The script uses strategy.position_avg_price to calculate price levels for the strategy.exit() call's stop-loss and take-profit orders that close the position.

We've included `calc_on_order_fills = true` in the strategy() declaration statement, meaning that the strategy recalculates each time the broker emulator fills a "Buy" or "Exit" order. Each time an "Exit" order fills, the strategy.position_size reverts to 0, triggering a new "Buy" order. The broker emulator fills the "Buy" order on the next tick at one of the bar's OHLC values, and then the strategy uses the recalculated strategy.position_avg_price value to determine new "Exit" order prices:

image

//@version=6 strategy("Intrabar exit", overlay = true, calc_on_order_fills = true) float stopSize = input.float(5.0, "SL %", minval = 0.0) / 100.0 float profitSize = input.float(5.0, "TP %", minval = 0.0) / 100.0 if strategy.position_size == 0.0 strategy.entry("Buy", strategy.long) float stopLoss = strategy.position_avg_price * (1.0 - stopSize) float takeProfit = strategy.position_avg_price * (1.0 + profitSize) strategy.exit("Exit", stop = stopLoss, limit = takeProfit)

Note that:

- Without enabling recalculation on order fills, this strategy would not place new orders *before* a bar closes. After an exit, the strategy would wait for the bar to close before placing a new “Buy” order, which the broker emulator would fill on the *next tick* after that, i.e., the open of the following bar.

It’s important to note that enabling `calc_on_order_fills` can produce unrealistic strategy results in some cases because the broker emulator may assume order-fill prices that are *not* obtainable in real-world trading. Therefore, users should exercise caution and carefully examine their strategy logic when allowing recalculation on order fills.

For example, the following script places a “Buy” order after each new order fill and bar close over the most recent 25 historical bars. The strategy simulates *four* entries per bar because the broker emulator considers each historical bar to have *four ticks* (open, high, low, and close). This behavior is unrealistic because it is not typically possible to fill an order at a bar’s *exact* high or low price:

image

//@version=6 strategy("buy on every fill", overlay = true, calc_on_order_fills = true, pyramiding = 100) if last_bar_index - bar_index <= 25 strategy.entry("Buy", strategy.long)

process_orders_on_close

By default, strategies simulate orders at the close of each bar, meaning that the earliest opportunity to fill the orders and execute strategy calculations and alerts is on the opening of the following bar. Programmers can change

this behavior to process orders on the *closing tick* of each bar by setting `process_orders_on_close` to `true` in the strategy() declaration statement. Users can set this behavior by changing the "Fill Orders/On Bar Close" setting in the "Settings/Properties" tab.

This behavior is most useful when backtesting manual strategies in which traders exit from a position before a bar closes, or in scenarios where algorithmic traders in non-24x7 markets set up after-hours trading capability so that alerts sent after close still have hope of filling before the following day.

Note that:

- Using strategies with `process_orders_on_close` enabled to send alerts to a third-party service might cause unintended results. Alerts on the close of a bar still occur after the market closes, and real-world orders based on such alerts might not fill until after the market opens again.
- The strategy.close() and strategy.close_all() commands feature an `immediately` parameter that, if `true`, allows the resulting market order to fill on the same tick where the strategy created it. This parameter provides an alternative way for programmers to selectively apply `process_orders_on_close` behavior to closing market orders without affecting the behavior of other order placement commands.

Simulating trading costs

Strategy performance reports are more relevant and meaningful when they include potential real-world trading costs. Without modeling the potential costs associated with their trades, traders may overestimate a strategy's historical profitability, potentially leading to suboptimal decisions in live trading. Pine Script strategies include inputs and parameters for simulating trading costs in performance results.

Commission

Commission is the fee a broker/exchange charges when executing trades. Commission can be a flat fee per trade or contract/share/lot/unit, or a percentage of the total transaction value. Users can set the commission properties of their strategies by including `commission_type` and

`commission_value` arguments in the strategy() function, or by setting the “Commission” inputs in the “Properties” tab of the strategy settings.

The following script is a simple strategy that simulates a “Long” position of 2% of equity when `close` equals the `highest` value over the `length`, and closes the trade when it equals the `lowest` value:

image

//@version=6 strategy("Commission Demo", overlay=true, default_qty_value = 2, default_qty_type = strategy.percent_of_equity) length = input.int(10, "Length") float highest = ta.highest(close, length) float lowest = ta.lowest(close, length) switch close highest => strategy.entry("Long", strategy.long) lowest => strategy.close("Long") plot(highest, color = color.new(color.lime, 50)) plot(lowest, color = color.new(color.red, 50))

The results in the Strategy Tester show that the strategy had a positive equity growth of 17.61% over the testing range. However, the backtest results do not account for fees the broker/exchange may charge. Let’s see what happens to these results when we include a small commission on every trade in the strategy simulation. In this example, we’ve included `commission_type = strategy.commission.percent` and `commission_value = 1` in the strategy() declaration, meaning it will simulate a commission of 1% on all executed orders:

image

//@version=6 strategy("Commission Demo", overlay=true, default_qty_value = 2, default_qty_type = strategy.percent_of_equity, commission_type = strategy.commission.percent, commission_value = 1) length = input.int(10, "Length") float highest = ta.highest(close, length) float lowest = ta.lowest(close, length) switch close highest => strategy.entry("Long", strategy.long) lowest => strategy.close("Long") plot(highest, color = color.new(color.lime, 50)) plot(lowest, color = color.new(color.red, 50))

As we can see in the example above, after applying a 1% commission to the backtest, the strategy simulated a significantly reduced net profit of only 1.42% and a more volatile equity curve with an elevated max drawdown. These results highlight the impact that commission can have on a strategy’s hypothetical performance.

Slippage and unfilled limits

In real-life trading, a broker/exchange may fill orders at slightly different prices than a trader intended, due to volatility, liquidity, order size, and other market factors, which can profoundly impact a strategy's performance. The disparity between expected prices and the actual prices at which the broker/exchange executes trades is what we refer to as *slippage*. Slippage is dynamic and unpredictable, making it impossible to simulate precisely. However, factoring in a small amount of slippage on each trade during a backtest or forward test might help the results better align with reality. Users can model slippage in their strategy results, sized as a fixed number of *ticks*, by including a `slippage` argument in the strategy() declaration statement or by setting the "Slippage" input in the "Settings/Properties" tab.

The following example demonstrates how simulating slippage affects the fill prices of market orders in a strategy test. The script below places a "Buy" market order of 2% equity when the market price is above a rising EMA and closes the position when the price dips below the EMA while it's falling. We've included `slippage = 20` in the strategy() function, which declares that the price of each simulated order will slip 20 ticks in the direction of the trade.

The script uses strategy.opentrades.entry_bar_index() and strategy.closedtrades.exit_bar_index() to get the `entryIndex` and `exitIndex`, which it uses to obtain the `fillPrice` of the order. When the bar index is at the `entryIndex`, the `fillPrice` is the first strategy.opentrades.entry_price() value. At the `exitIndex`, `fillPrice` is the strategy.closedtrades.exit_price() value from the last closed trade. The script plots the expected fill price along with the simulated fill price after slippage to visually compare the difference:

image

//@version=6 strategy("Slippage Demo", overlay = true, slippage = 20, default_qty_value = 2, default_qty_type = strategy.percent_of_equity) int length = input.int(5, "Length") //@variable Exponential moving average with an input `length`. float ma = ta.ema(close, length) //@variable Is `true` when `ma` has increased and `close` is above it, `false` otherwise. bool longCondition = close > ma and ma > ma[1] //@variable Is `true` when `ma` has decreased and `close` is below it, `false` otherwise. bool shortCondition = close < ma and ma < ma[1] // Enter a long market position on

`longCondition` and close the position on `shortCondition`. if longCondition strategy.entry("Buy", strategy.long) if shortCondition strategy.close("Buy") // @variable The `bar_index` of the position's entry order fill. int entryIndex = strategy.opentrades.entry_bar_index(0) //@variable The `bar_index` of the position's close order fill. int exitIndex = strategy.closedtrades.exit_bar_index(strategy.closedtrades - 1) //@variable The fill price simulated by the strategy. float fillPrice = switch bar_index entryIndex => strategy.opentrades.entry_price(0) exitIndex => strategy.closedtrades.exit_price(strategy.closedtrades - 1) //@variable The expected fill price of the open market position. float expectedPrice = not na(fillPrice) ? open : na color expectedColor = na color filledColor = na if bar_index == entryIndex expectedColor := color.green filledColor := color.blue else if bar_index == exitIndex expectedColor := color.red filledColor := color.fuchsia plot(ma, color = color.new(color.orange, 50)) plotchar(not na(fillPrice) ? open : na, "Expected fill price", "—", location.absolute, expectedColor) plotchar(fillPrice, "Fill price after slippage", "—", location.absolute, filledColor)

Note that:

- Since the strategy applies constant slippage to all order fills, some orders can fill *outside* the candle range in the simulation. Exercise caution with this setting, as adding excessive simulated slippage can produce unrealistically worse testing results.

Some traders might assume that they can avoid the adverse effects of slippage by using limit orders, as unlike market orders, they cannot execute at a worse price than the specified value. However, even if the market price reaches an order's price, there's a chance that a limit order might not fill, depending on the state of the real-life market, because limit orders can only fill if a security has sufficient liquidity and price action around their values. To account for the possibility of *unfilled* orders in a backtest, users can specify the `backtest_fill_limits_assumption` value in the declaration statement or use the "Verify price for limit orders" input in the "Settings/Properties" tab. This setting instructs the strategy to fill limit orders only after the market price moves a defined number of ticks past the order prices.

The following example places a limit order of 2% equity at a bar's hlcc4 price when the high is the `highest` value over the past `length` bars and there are

no pending entries. The strategy closes the market position and cancels all orders after the low is the `lowest` value. Each time the strategy triggers an order, it draws a horizontal line at the `limitPrice`, which it updates on each bar until closing the position or canceling the order:

image

//@version=6 strategy("Verify price for limits example", overlay = true, default_qty_type = strategy.percent_of_equity, default_qty_value = 2) int length = input.int(25, title = "Length") //@variable Draws a line at the limit price of the most recent entry order. var line limitLine = na // Highest high and lowest low highest = ta.highest(length) lowest = ta.lowest(length) // Place an entry order and draw a new line when the the `high` equals the `highest` value and `limitLine` is `na`. if high == highest and na(limitLine) float limitPrice = hlcc4 strategy.entry("Long", strategy.long, limit = limitPrice) limitLine := line.new(bar_index, limitPrice, bar_index + 1, limitPrice) // Close the open market position, cancel orders, and set `limitLine` to `na` when the `low` equals the `lowest` value. if low == lowest strategy.cancel_all() limitLine := na strategy.close_all() // Update the `x2` value of `limitLine` if it isn't `na`. if not na(limitLine) limitLine.set_x2(bar_index + 1) plot(highest, "Highest High", color = color.new(color.green, 50)) plot(lowest, "Lowest Low", color = color.new(color.red, 50))

By default, the script assumes that all limit orders are guaranteed to fill when the market price reaches their values, which is often not the case in real-life trading. Let’s add price verification to our limit orders to account for potentially unfilled ones. In this example, we’ve included `backtest_fill_limits_assumption = 3` in the strategy() function call. As we can see, using limit verification omits some simulated order fills and changes the times of others, because the entry orders can now only fill after the price exceeds the limit price by *three ticks*:

image

Risk management

Designing a strategy that performs well, especially in a broad class of markets, is a challenging task. Most strategies are designed for specific market patterns/conditions and can produce uncontrolled losses when

applied to other data. Therefore, a strategy's risk management behavior can be critical to its performance. Programmers can set risk management criteria in their strategy scripts using the `strategy.risk.*()` commands.

Strategies can incorporate any number of risk management criteria in any combination. All risk management commands execute *on every tick and order execution event*, regardless of any changes to the strategy's calculation behavior. There is no way to deactivate any of these commands on specific script executions. Irrespective of a risk management command's location, it *always* applies to the strategy unless the programmer removes the call from the code.

strategy.risk.allow_entry_in()

This command overrides the market direction allowed for all strategy.entry() commands in the script. When a user specifies the trade direction with the strategy.risk.allow_entry_in() function (e.g., strategy.direction.long), the strategy enters trades only in that direction. If a script calls an entry command in the opposite direction while there's an open market position, the strategy simulates a market order to *close* the position.

strategy.risk.max_cons_loss_days()

This command cancels all pending orders, closes any open market position, and stops all additional trade actions after the strategy simulates a defined number of trading days with consecutive losses.

strategy.risk.max_drawdown()

This command cancels all pending orders, closes any open market position, and stops all additional trade actions after the strategy's drawdown reaches the amount specified in the function call.

strategy.risk.max_intraday_filled_orders()

This command specifies the maximum number of filled orders per trading day (or per chart bar if the timeframe is higher than daily). If the strategy creates more orders than the maximum, the command cancels all pending orders, closes any open market position, and halts trading activity until the end of the current session.

strategy.risk.max_intraday_loss()

This command controls the maximum loss the strategy tolerates per trading day (or per chart bar if the timeframe is higher than daily). When the strategy's losses reach this threshold, it cancels all pending orders, closes the open market position, and stops all trading activity until the end of the current session.

strategy.risk.max_position_size()

This command specifies the maximum possible position size when using strategy.entry() commands. If the quantity of an entry command results in a market position that exceeds this threshold, the strategy reduces the order quantity so that the resulting position does not exceed the limit.

Margin

Margin is the minimum percentage of a market position that a trader must hold in their account as collateral to receive and sustain a loan from their broker to achieve their desired *leverage*. The `margin_long` and `margin_short` parameters of the strategy() declaration statement and the "Margin for long/short positions" inputs in the "Properties" tab of the script settings specify margin percentages for long and short positions. For example, if a trader sets the margin for long positions to 25%, they must have enough funds to cover 25% of an open long position. This margin percentage also means the trader can potentially spend up to 400% of their equity on their trades.

If a strategy's simulated funds cannot cover the losses from a margin trade, the broker emulator triggers a *margin call*, which forcibly liquidates all or part of the open position. The exact number of contracts/shares/lots/units that the emulator liquidates is *four times* the amount required to cover the loss, which helps prevent constant margin calls on subsequent bars. The emulator determines liquidated quantity using the following algorithm:

1. Calculate the amount of capital spent on the position: `Money Spent = Quantity * Entry Price`
2. Calculate the Market Value of Security (MVS): `MVS = Position Size * Current Price`

3. Calculate the Open Profit as the difference between `MVS` and `Money Spent`. If the position is short, multiply this value by -1.
4. Calculate the strategy's equity value: `Equity = Initial Capital + Net Profit + Open Profit`
5. Calculate the margin ratio: `Margin Ratio = Margin Percent / 100`
6. Calculate the margin value, which is the cash required to cover the hypothetical account's portion of the position: `Margin = MVS * Margin Ratio`
7. Calculate the strategy's available funds: `Available Funds = Equity - Margin`
8. Calculate the total amount of money lost: `Loss = Available Funds / Margin Ratio`
9. Calculate the number of contracts/shares/lots/units the account must liquidate to cover the loss, truncated to the same decimal precision as the minimum position size for the current symbol: `Cover Amount = TRUNCATE(Loss / Current Price).`
10. Multiply the quantity required to cover the loss by four to determine the margin call size: `Margin Call Size = Cover Amount * 4`

To examine this calculation in detail, let's add the built-in Supertrend Strategy to the NASDAQ:TSLA chart on the "1D" timeframe and set the "Order size" to 300% of equity and the "Margin for long positions" to 25% in the "Properties" tab of the strategy settings:

image

The first entry happened at the bar's opening price on 16 Sep 2010. The strategy bought 682,438 shares (Position Size) at 4.43 USD (Entry Price). Then, on 23 Sep 2010, when the price dipped to 3.9 (Current Price), the emulator forcibly liquidated 111,052 shares with a margin call. The calculations below show how the broker emulator determined this amount for the margin call event:

```
Money spent: 682438 * 4.43 = 3023200.34
MVS: 682438 * 3.9 = 2661508.2
Open Profit: −361692.14
Equity: 1000000 + 0 − 361692.14 = 638307.86
Margin Ratio: 25 / 100 = 0.25
Margin: 2661508.2 * 0.25 = 665377.05
```

```
Available Funds: 638307.86 - 665377.05 = -27069.19
Money Lost: -27069.19 / 0.25 = -108276.76
Cover Amount: TRUNCATE(-108276.76 / 3.9) = TRUNCATE(-27763.27) = -2776
Margin Call Size: -27763 * 4 = - 111052
```

Note that:

- The strategy.margin_liquidation_price variable's value represents the price level that will cause a margin call if the market price reaches it. For more information about how margin works and the formula for calculating a position's margin call price, see this page in our Help Center.

Using strategy information in scripts

Numerous built-ins within the `strategy.*` namespace and its *sub-namespaces* provide convenient solutions for programmers to use a strategy's trade and performance information, including data shown in the Strategy Tester, directly within their code's logic and calculations.

Several `strategy.*` variables hold fundamental information about a strategy, including its starting capital, equity, profits and losses, run-up and drawdown, and open position:

- strategy.account_currency
- strategy.initial_capital
- strategy.equity
- strategy.netprofit and strategy.netprofit_percent
- strategy.grossprofit and strategy.grossprofit_percent
- strategy.grossloss and strategy.grossloss_percent
- strategy.openprofit and strategy.openprofit_percent
- strategy.max_runup and strategy.max_runup_percent
- strategy.max_drawdown and strategy.max_drawdown_percent
- strategy.position_size
- strategy.position_avg_price
- strategy.position_entry_name

Additionally, the namespace features multiple variables that hold general trade information, such as the number of open and closed trades, the

number of winning and losing trades, average trade profits, and maximum trade sizes:

- strategy.opentrades
- strategy.closedtrades
- strategy.wintrades
- strategy.losstrades
- strategy.eventrades
- strategy.avg_trade and strategy.avg_trade_percent
- strategy.avg_winning_trade and strategy.avg_winning_trade_percent
- strategy.avg_losing_trade and strategy.avg_losing_trade_percent
- strategy.max_contracts_held_all
- strategy.max_contracts_held_long
- strategy.max_contracts_held_short

Programmers can use these variables to display relevant strategy information on their charts, create customized trading logic based on strategy data, calculate custom performance metrics, and more.

The following example demonstrates a few simple use cases for these `strategy.*` variables. The script uses them in its order placement and display calculations. When the calculated `rank` crosses above 10 and the strategy.opentrades value is 0, the script calls strategy.entry() to place a "Buy" market order. On the following bar, where that order fills, it calls strategy.exit() to create a stop-loss order at a user-specified percentage below the strategy.position_avg_price value. If the `rank` crosses above 80 during the open trade, the script uses strategy.close() to exit the position on the next bar.

The script creates a table to display formatted strings representing information from several of the above `strategy.*` variables on the main chart pane. The text in the table shows the strategy's net profit and net profit percentage, the account currency, the number of winning trades and the win percentage, the ratio of the average winning trade to the average losing trade, and the profit factor (the ratio of the gross profit to the gross loss). The script also plots the strategy.equity series in a separate pane and highlights the pane's background based on the value of strategy.openprofit:

image

```
//@version=6 strategy( "Using strategy information demo", default_qty_type = strategy.percent_of_equity, default_qty_value = 5, margin_long = 100, margin_short = 100 ) //@variable The number of bars in the `rank` calculation. int lengthInput = input.int(50, "Length", 1) //@variable The stop-loss percentage. float slPercentInput = input.float(4.0, "SL %", 0.0, 100.0) / 100.0 //@variable The percent rank of `close` prices over `lengthInput` bars. float rank = ta.percentrank(close, lengthInput) // Entry and exit signals. bool entrySignal = ta.crossover(rank, 10) and strategy.opentrades == 0 bool exitSignal = ta.crossover(rank, 80) and strategy.opentrades == 1 // Place orders based on the `entrySignal` and `exitSignal` occurrences. switch entrySignal => strategy.entry("Buy", strategy.long) entrySignal[1] => strategy.exit("SL", "Buy", stop = strategy.position_avg_price * (1.0 - slPercentInput)) exitSignal => strategy.close("Buy") if barstate.islastconfirmedhistory or barstate.isrealtime //@variable A table displaying strategy information on the main chart pane. var table dashboard = table.new( position.top_right, 2, 10, border_color = chart.fg_color, border_width = 1, force_overlay = true ) //@variable The strategy's currency. string currency = strategy.account_currency // Display the net profit as a currency amount and percentage. dashboard.cell(0, 1, "Net P/L") dashboard.cell( 1, 1, str.format("{0, number, 0.00} {1} ({2}%)", strategy.netprofit, currency, strategy.netprofit_percent), text_color = chart.fg_color, bgcolor = strategy.netprofit > 0 ? color.lime : color.red ) // Display the number of winning trades as an absolute value and percentage of all completed trades. dashboard.cell(0, 2, "Winning trades") dashboard.cell( 1, 2, str.format("{0} ({1, number, #.##%})", strategy.wintrades, strategy.wintrades / strategy.closedtrades), text_color = chart.fg_color, bgcolor = strategy.wintrades > strategy.losstrades ? color.lime : color.red ) // Display the ratio of average trade profit to average trade loss. dashboard.cell(0, 3, "Avg. win / Avg. loss") dashboard.cell( 1, 3, str.format("{0, number, #.###}", strategy.avg_winning_trade / strategy.avg_losing_trade), text_color = chart.fg_color, bgcolor = strategy.avg_winning_trade > strategy.avg_losing_trade ? color.lime : color.red ) // Display the profit factor, i.e., the ratio of gross profit to gross loss. dashboard.cell(0, 4, "Profit factor") dashboard.cell( 1, 4, str.format("{0, number, #.###}", strategy.grossprofit / strategy.grossloss), text_color = chart.fg_color, bgcolor = strategy.grossprofit > strategy.grossloss ? color.lime : color.red ) // Plot the current equity in a separate pane and
```

highlight the pane's background while there is an open position. plot(strategy.equity, "Total equity", strategy.equity > strategy.initial_capital ? color.teal : color.maroon, 3) bgcolor(strategy.openprofit > 0 ? color.new(color.teal, 80) : strategy.openprofit < 0 ? color.new(color.maroon, 80) : na, title = "Open position highlight")

Note that:

- This script creates a stop-loss order one bar after the entry order because it uses strategy.position_avg_price to determine the price level. This variable has a non-na value only when the strategy has an *open position*.
- The script draws the table only on the last historical bar and all realtime bars because the historical states of tables are **never visible**. See the Reducing drawing updates section of the Profiling and optimization page for more information.
- The table.new() call includes `force_overlay = true` to display the table on the main chart pane.

Individual trade information

The `strategy.*` namespace features two sub-namespaces that provide access to *individual trade* information: `strategy.opentrades.*` and `strategy.closedtrades.*`. The `strategy.opentrades.*` built-ins return data for *incomplete* (open) trades, and the `strategy.closedtrades.*` built-ins return data for *completed* (closed) trades. With these built-ins, programmers can use granular trade data in their scripts, allowing for more detailed strategy analysis and advanced calculations.

Both sub-namespaces contain several similar functions that return information about a trade's orders, simulated costs, and profit/loss, including:

- strategy.opentrades.entry_id() / strategy.closedtrades.entry_id()
- strategy.opentrades.entry_price() / strategy.closedtrades.entry_price()
- strategy.opentrades.entry_bar_index() / strategy.closedtrades.entry_bar_index()
- strategy.opentrades.entry_time() / strategy.closedtrades.entry_time()
- strategy.opentrades.entry_comment() / strategy.closedtrades.entry_comment()

- strategy.opentrades.size() / strategy.closedtrades.size()
- strategy.opentrades.profit() / strategy.closedtrades.profit()
- strategy.opentrades.profit_percent() / strategy.closedtrades.profit_percent()
- strategy.opentrades.commission() / strategy.closedtrades.commission()
- strategy.opentrades.max_runup() / strategy.closedtrades.max_runup()
- strategy.opentrades.max_runup_percent() / strategy.closedtrades.max_runup_percent()
- strategy.opentrades.max_drawdown() / strategy.closedtrades.max_drawdown()
- strategy.opentrades.max_drawdown_percent() / strategy.closedtrades.max_drawdown_percent()
- strategy.closedtrades.exit_id()
- strategy.closedtrades.exit_price()
- strategy.closedtrades.exit_time()
- strategy.closedtrades.exit_bar_index()
- strategy.closedtrades.exit_comment()

Note that:

- Most built-ins within these namespaces are *functions*. However, the `strategy.opentrades.*` namespace also features a unique *variable*: strategy.opentrades.capital_held. Its value represents the amount of capital reserved by *all* open trades.
- Only the `strategy.closedtrades.*` namespace has `.exit_*()` functions that return information about *exit orders*.

All `strategy.opentrades.*()` and `strategy.closedtrades.*()` functions have a `trade_num` parameter, which accepts an “int” value representing the index of the open or closed trade. The index of the first open/closed trade is 0, and the last trade’s index is *one less* than the value of the strategy.opentrades/strategy.closedtrades variable.

The following example places up to five long entry orders per position, each with a unique ID, and it calculates metrics for specific closed trades.

The strategy places a new entry order when the close crosses above the `median` value without reaching the `highest` value, but only if the number of open trades is less than five. It exits each position using stop-loss orders from

strategy.exit() or a market order from strategy.close_all(). Each successive entry order's ID depends on the number of open trades. The first entry ID in each position is "Buy0", and the last possible entry ID is "Buy4".

The script calls `strategy.closedtrades.*()` functions within a for loop to access closed trade entry IDs, profits, entry bar indices, and exit bar indices. It uses this information to calculate the total number of closed trades with the specified entry ID, the number of winning trades, the average number of bars per trade, and the total profit from all the trades. The script then organizes this information in a formatted string and displays the result using a single-cell table:

image

//@version=6 strategy("Individual trade information demo", pyramiding = 5, default_qty_type = strategy.percent_of_equity, default_qty_value = 1, margin_long = 100, margin_short = 100) //@variable The number of bars in the `highest` and `lowest` calculation. int lengthInput = input.int(50, "Length", 1) string idInput = input.string("Buy0", "Entry ID to analyze", ["Buy0", "Buy1", "Buy2", "Buy3", "Buy4"]) // Calculate the highest, lowest, and median `close` values over `lengthInput` bars. float highest = ta.highest(close, lengthInput) float lowest = ta.lowest(close, lengthInput) float median = 0.5 * (highest + lowest) // Define entry and stop-loss orders when the `close` crosses above the `median` without touching the `highest` value. if ta.crossover(close, median) and close != highest and strategy.opentrades < 5 strategy.entry("Buy" + str.tostring(strategy.opentrades), strategy.long) if strategy.opentrades == 0 strategy.exit("SL", stop = lowest) // Close the entire position when the `close` reaches the `lowest` value. if close == lowest strategy.close_all() // The total number of closed trades with the `idInput` entry, the number of wins, the average number of bars, // and the total profit. int trades = 0 int wins = 0 float avgBars = 0 float totalPL = 0.0 if barstate.islastconfirmedhistory or barstate.isrealtime //@variable A single-cell table displaying information about closed trades with the `idInput` entry ID. var table infoTable = table.new(position.middle_center, 1, 1, color.purple) // Iterate over closed trade indices. for tradeNum = 0 to strategy.closedtrades - 1 // Skip the rest of the current iteration if the `tradeNum` closed trade didn't open with an `idInput` entry. if strategy.closedtrades.entry_id(tradeNum) != idInput continue // Accumulate `trades`, `wins`, `avgBars`, and `totalPL` values.

float profit = strategy.closedtrades.profit(tradeNum) trades += 1 wins += profit > 0 ? 1 : 0 avgBars += strategy.closedtrades.exit_bar_index(tradeNum) - strategy.closedtrades.entry_bar_index(tradeNum) + 1 totalPL += profit avgBars /= trades //@variable A formatted string containing the calculated closed trade information. string displayText = str.format("ID: {0}\n\nTotal trades: {1}\nWin trades: {2}\nAvg. bars: {3}\nTotal P/L: {4} {5}", idInput, trades, wins, avgBars, totalPL, strategy.account_currency) // Populate the table's cell with `displayText`. infoTable.cell(0, 0, displayText, text_color = color.white, text_halign = text.align_left, text_size = size.large) // Plot the highest, median, and lowest values on the main chart pane. plot(highest, "Highest close", force_overlay = true) plot(median, "Median close", force_overlay = true) plot(lowest, "Lowest close", force_overlay = true)

Note that:

- This strategy can open up to five long trades per position because we included `pyramiding = 5` in the strategy() declaration statement. See the pyramiding section for more information.
- The strategy.exit() instance in this script persists and generates exit orders for every entry in the open position because we did not specify a `from_entry` ID. See the Exits for multiple entries section to learn more about this behavior.

Strategy alerts

Pine Script indicators (not strategies) have two different mechanisms to set up custom alert conditions: the alertcondition() function, which tracks one specific condition per function call, and the alert() function, which tracks all its calls simultaneously, but provides greater flexibility in the number of calls, alert messages, etc.

Pine Script strategies cannot create alert triggers using the alertcondition() function, but they can create triggers with the alert() function. Additionally, each order placement command comes with its own built-in alert functionality that does not require any additional code to implement. As such, any strategy that uses an order placement command can issue alerts upon order execution. The precise mechanics of such built-in strategy alerts are described in the Order Fill events section of the Alerts page.

When a strategy uses both the alert() function and functions that create orders in the same script, the “Create Alert” dialog box provides a choice between the conditions to use as a trigger: alert() events, order fill events, or both.

For many trading strategies, the delay between a triggered alert and a live trade can be a critical performance factor. By default, strategy scripts can only execute alert() function calls on the close of realtime bars, as if they used alert.freq_once_per_bar_close, regardless of the `freq` argument in the call. Users can change the alert frequency by including `calc_on_every_tick = true` in the strategy() call or selecting the “Recalculate/On every tick” option in the “Settings/Properties” tab before creating the alert. However, depending on the script, this setting can adversely impact the strategy’s behavior. See the `calc_on_every_tick` section for more information.

Order fill alert triggers do not suffer the same limitations as the triggers from alert() calls, which makes them more suitable for sending alerts to third parties for automation. Alerts from order fill events execute *immediately*, unaffected by a script’s `calc_on_every_tick` setting. Users can set the default message for order fill alerts via the `//@strategy_alert_message` compiler annotation. The text provided with this annotation populates the “Message” field in the “Create Alert” dialog box.

The following script shows a simple example of a default order fill alert message. Above the strategy() declaration statement, the script includes `@strategy_alert_message` with *placeholders* for the trade action, current position size, ticker name, and fill price values in the message text:

//@version=6 //@strategy_alert_message {{strategy.order.action}} {{strategy.position_size}} {{ticker}} @ {{strategy.order.price}} strategy("Alert Message Demo", overlay = true) float fastMa = ta.sma(close, 5) float slowMa = ta.sma(close, 10) if ta.crossover(fastMa, slowMa) strategy.entry("buy", strategy.long) if ta.crossunder(fastMa, slowMa) strategy.entry("sell", strategy.short) plot(fastMa, "Fast MA", color.aqua) plot(slowMa, "Slow MA", color.orange)

This script populates the “Create Alert” dialog box with its default message when the user selects its name from the “Condition” dropdown tab:

image

When the alert fires, the strategy populates the placeholders in the alert message with their corresponding values. For example:

image

Notes on testing strategies

Testing and tuning strategies in historical and live market conditions can provide insight into a strategy's characteristics, potential weaknesses, and *possibly* its future potential. However, traders should always be aware of the biases and limitations of simulated strategy results, especially when using the results to support live trading decisions. This section outlines some caveats associated with strategy validation and tuning and possible solutions to mitigate their effects.

Backtesting and forward testing

Backtesting is a technique to evaluate the historical performance of a trading strategy or model by simulating and analyzing its past results on historical market data. This technique assumes that a strategy's results on past data can provide insight into its strengths and weaknesses. When backtesting, many traders adjust the parameters of a strategy in an attempt to optimize its results. Analysis and optimization of historical results can help traders to gain a deeper understanding of a strategy. However, traders should always understand the risks and limitations when basing their decisions on optimized backtest results.

It is prudent to also use realtime analysis as a tool for evaluating a trading system on a forward-looking basis. *Forward testing* aims to gauge the performance of a strategy in live market conditions, where factors such as trading costs, slippage, and liquidity can meaningfully affect its performance. While forward testing has the distinct advantage of not being affected by certain types of biases (e.g., lookahead bias or "future data leakage"), it does carry the disadvantage of being limited in the quantity of data to test. Therefore, although it can provide helpful insights into a strategy's performance in current market conditions, forward testing is not typically used on its own.

Lookahead bias

One typical issue in backtesting strategies that request alternate timeframe data, use repainting variables such as timenow, or alter calculation behavior for intrabar order fills, is the leakage of future data into the past during evaluation, which is known as *lookahead bias*. Not only is this bias a common cause of unrealistic strategy results, since the future is never actually knowable beforehand, but it is also one of the typical causes of strategy repainting.

Traders can often confirm whether a strategy has lookahead bias by forward testing it on realtime data, where no known data exists beyond the latest bar. Since there is no future data to leak into the past on realtime bars, the strategy will behave differently on historical and realtime bars if its results have lookahead bias.

To eliminate lookahead bias in a strategy:

- Do not use repainting variables that leak future values into the past in the order placement or cancellation logic.
- Do not include barmerge.lookahead_on in `request.*()` calls without offsetting the data series, as described in this section of the Repainting page.
- Use realistic strategy calculation behavior.

Selection bias

Selection bias occurs when a trader analyzes only results on specific instruments or timeframes while ignoring others. This bias can distort the perspective of the strategy's robustness, which can impact trading decisions and performance optimizations. Traders can reduce the effects of selection bias by evaluating their strategies on multiple, ideally diverse, symbols and timeframes, and ensuring not to ignore poor performance results or "cherry-pick" testing ranges.

Overfitting

A common problem when optimizing a strategy based on backtest results is overfitting ("curve fitting"), which means tailoring the strategy for specific

data. An overfitted strategy often fails to generalize well on new, unseen data. One widely-used approach to help reduce the potential for overfitting and promote better generalization is to split an instrument's data into two or more parts to test the strategy outside the sample used for optimization, otherwise known as "in-sample" (IS) and "out-of-sample" (OOS) backtesting.

In this approach, traders optimize strategy parameters on the IS data, and they test the optimized configuration on the OOS data without additional fine-tuning. Although this and other, more robust approaches might provide a glimpse into how a strategy might fare after optimization, traders should still exercise caution. No trading strategy can guarantee future performance, regardless of the data used for optimization and testing, because the future is inherently unknowable.

Order limit

Outside of Deep Backtesting, a strategy can keep track of up to 9000 orders. If a strategy creates more than 9000 orders, the earliest orders are *trimmed* so that the strategy stores the information for only the most recent orders.

Trimmed orders do **not** appear in the Strategy Tester. Referencing the trimmed order IDs using `strategy.closedtrades.*` functions returns na.

The strategy.closedtrades.first_index variable holds the index of the oldest *untrimmed* trade, which corresponds to the first trade listed in the List of Trades. If the strategy creates less than 9000 orders, there are no trimmed orders, and this variable's value is 0.

Previous

Next

Strings

Strings

Introduction

Pine Script® strings are immutable values containing sequences of up to 4096 encoded characters, such as letters, digits, symbols, spaces, control characters, or other Unicode characters and code points. Strings allow scripts to represent a wide range of data as character patterns and human-readable text.

Pine scripts use strings for several purposes, such as defining titles, expressing symbol and timeframe information, setting the contexts of data requests, creating alert and debug messages, and displaying text on the chart. The specialized functions in the `str.*` namespace provide convenient ways to construct strings, create modified copies of other strings, and inspect or extract substrings.

This page explains how Pine strings work, and how to construct, inspect, and modify strings using the available `str.*()` functions.

Literal strings

Literal strings in Pine are character sequences enclosed by two *ASCII* quotation marks (`"`) or apostrophes (`'`). For example, this code snippet declares two variables with equivalent literal strings containing the text `Hello world!`:

//@variable A literal string containing `Hello world!`. Uses the `"` character as the enclosing delimiter. string hello1 = "Hello world!" //@variable A literal

string containing `Hello world!`. Uses the `'` character as the enclosing delimiter. string hello2 = 'Hello world!'

The " or ' enclosing delimiters in a literal string definition are *not* parts of the specified character sequence. They only mark the sequence's start and end boundaries in the code. These characters *do not* appear in outputs of Pine Logs or drawing objects that display "string" values.

This example calls the log.info() function on the first bar to display the contents of the literal value `"Hello world!"` in the Pine Logs pane. The message in the pane displays the `Hello world!` text only, without the " characters:

image

//@version=6 indicator("Literal strings demo") // The script's displayed title does not include the quotation marks. if barstate.isfirst // Log "Hello world!" on the first bar. The logged text does not include `"` characters. log.info("Hello world!")

Note that:

- The script also uses a literal string to define the `title` argument of the indicator() declaration statement.
- Only the " and ' ASCII characters are valid enclosing delimiters for literal strings. Other Unicode characters, such as U+FF02 (Fullwidth Quotation Mark), are not allowed as enclosing delimiters.
- The timestamp in square brackets (`[` and `]`) at the start of the logged message is an *automatic prefix* showing the log's time in the chart's time zone. For more information, refer to the Pine Logs section of the Debugging page.

Programmers can define literal strings on a single line of code or use line wrapping to define them across *multiple* lines. When using line wrapping, each new code line containing parts of the "string" value must be indented by any number of spaces that is *not* a multiple of four.

This script demonstrates line wrapping in a "string" variable declaration. The character sequence wraps across three code lines, with each new line

indented by five spaces. The script displays the value's contents using a log.info() call:

image

//@version=6 indicator("Line wrapping demo") //@variable A single string defined on three wrapped lines. Each line after the first is indented by five spaces. // Every subsequent code line also inserts a leading space character. string displayText = "Single-line text defined across three wrapped code lines" if barstate.isfirst log.info(displayText)

Note that:

- Although the declaration wraps across multiple code lines, the displayed character sequence occupies a *single line* of text. To create strings with multiline text, include *line terminators* with the \n escape sequence.
- Scripts automatically *add* one space character to the start of each subsequent code line in a line-wrapped "string" value. Programmers can verify this behavior by checking the number of characters in the string with the str.length() function. See the Counting characters and substrings section below for an example.

Escape sequences

The backslash character (\), also known as the Reverse Solidus in Unicode (U+005C), is an *escape character* in Pine strings. This character forms an *escape sequence* when it precedes another character, signaling that the following character has a potentially *different* meaning than usual.

Characters with a special meaning in "string" value definitions, such as quotation marks and backslashes, become *literal* characters when prefixed by a backslash (e.g., \\ includes a single \ in the character sequence).

This simple script declares a variable with an assigned literal "string" value enclosed in apostrophes (') and displays the value's contents in the Pine Logs pane. It uses the \ character to escape an extra apostrophe and another backslash, making them literal characters in the displayed text:

image

//@version=6 indicator("Escaping special characters demo") //@variable A string containing escaped `\` and `'` characters. string displayText = 'The backslash (\\) can change another character\'s meaning in Pine strings.' if barstate.isfirst log.info(displayText)

Note that:

- This example must prefix the ' character with a backslash in the string's sequence because it also uses that character to mark its start and end boundaries. Without the backslash, it causes a *compilation error*. The script does not need to escape the apostrophe if we change the literal string's enclosing characters to quotation marks (").

The ASCII characters n and t usually have a literal meaning in Pine strings. However, when prefixed by the backslash character, they form escape sequences representing *control characters*. The \n sequence represents the newline character (U+000A), a line terminator for multiline text. The \t sequence represents the horizontal tab character (U+0009), which is helpful for indentation.

The script below creates a "string" value with multiline text on a single line of code, which it displays in a label on the last historical bar. The defined value contains several \n and \t escape sequences to include line terminators and tab spaces in the displayed text:

image

//@version=6 indicator("Control characters demo", overlay = true) if barstate.islastconfirmedhistory //@variable A string containing `\n` and `\t` escape sequences. Renders as multiline text with indentation. string displayText = "\This\n\tis\n\t\tmultiline\n\t\t\ttext\n\t\t\t\twith\n\t\t\t\t\ttab spaces." // Draw a label showing the `displayText` at the bar's `high`. label.new(bar_index, high, displayText, style = label.style_label_left, size = 24, textalign = text.align_left)

Note that:

- The "string" value also includes \ before the T character. However, that character still appears *literally* in the displayed text. If a backslash

applied to a character does not form a supported escape sequence, the character's meaning *does not change*.

Concatenation

The + and += operators signify *concatenation* when the operands are strings. A concatenation operation appends the second operand's character sequence to the first operand's sequence to form a new, *combined* "string" value.

For example, this script declares a `concatString` variable that holds the result of a concatenation operation. After declaring the variable, it uses the += operator to concatenate additional strings and reassign the variable's value. Then, the script calls log.info to show the result in the Pine Logs pane:

image

//@version=6 indicator("Concatenation demo", overlay = true) if barstate.isfirst // Declare two variables that hold concatenated literal strings. string value1 = "\n\nThis 'string' is " + "the result " string value2 = "of multiple " + "concatenation operations." //@variable A string produced by concatenating `value1`, `value2`, and three literal strings. string displayText = value1 + value2 + "\n" // Use two concatenation assignments to create new strings from `displayText` and a literal string. displayText += "\nEach operation creates a new 'string' in memory that combines the character sequences of the operands" displayText += " without modifying the original values." // Log the `displayText` in the Pine Logs pane. log.info(displayText)

Note that:

- Strings are immutable and cannot change. Therefore, every concatenation operation creates a *new* "string" value in memory. The operation does **not** modify either "string" operand directly.
- Another, more advanced way to combine strings is to collect them inside an array and use the array.join() function. For more information, see the Joining section of the Arrays page.
- In many cases, programmers can efficiently create *formatted strings* with str.format() instead of combining individual strings with

concatenation or joining. See the Formatting strings section to learn more.

String conversion and formatting

Programmers can use strings to represent data of virtually any type as human-readable character sequences. Converting data to strings allows scripts to perform many helpful tasks, including:

- Displaying dynamic prices and calculations as text inside labels, tables, or boxes.
- Creating alert messages containing realtime market and indicator information.
- Logging debug messages containing calculated script information in the Pine Logs pane.
- Performing custom calculations and logic, such as constructing symbol or timeframe strings for data requests.

Converting values to strings

The simplest way to convert data to strings is to call the str.tostring() function. The function can represent values of several types as strings, based on predefined or custom formats. It has the following two signatures:

```
str.tostring(value) → string
str.tostring(value, format) → string
```

The function's `value` parameter accepts any "int", "float", "bool", or "string" value; the reference of an array or matrix containing values of these types; or a member of an enum type.

For example, this line of code creates a string representing the "float" value `123.456`, with default formatting. The result is usable in "string" operations and any script outputs that display *dynamic* text, such as labels and tables:

//@variable Holds the "string" value `"123.456"`. string numString = str.tostring(123.456)

The str.tostring() function's `format` parameter determines the *numeric format* for converted "int" and "float" values, arrays, and matrices. It can use

one of the following `format.*` constants: format.mintick, format.percent, or format.volume. Alternatively, programmers can use strings containing # (number sign), `0` (zero), `.` (period), `,` (comma), and `%` (percent sign) tokens for customized *formatting patterns* with specific decimal precision. The default numeric format is `"#.########"`, which rounds fractional digits to eight decimal places without trailing zeros.

The script below uses the str.tostring() function to convert numeric values, a “bool” value, arrays, and a matrix into strings and displays the results in a table on the last bar. The str.tostring() calls that convert numeric values and collections contain different `format` arguments to demonstrate how various formatting patterns affect the results:

image

//@version=6 indicator("String conversion demo") //@variable A 2-row, 15-column table showing "string" representations of numbers, "bool" values, and collections. var table displayTable = table.new(position.middle_center, 2, 15, frame_color = chart.fg_color, frame_width = 1) //@function Initializes a row to show two specified strings in the display table. makeRow(table tableName, int row, string str0, string str1) => color bgColor = row == 0 ? chart.fg_color : chart.bg_color color textColor = row == 0 ? chart.bg_color : chart.fg_color displayTable.cell(0, row, str0, text_color = textColor, text_halign = text.align_left, bgcolor = bgColor) displayTable.cell(1, row, str1, text_color = textColor, text_halign = text.align_left, bgcolor = bgColor) // Initialize the header row for the `displayTable` on the first bar. if barstate.isfirst makeRow(0, "Variable", "'string' value") // Compute several "string" conversions and populate the `displayTable` with the results on the last bar. if barstate.islast //@variable Represents the `ta.vwap` with the default numeric format (`#.########`). string numberRepr1 = str.tostring(ta.vwap) //@variable Represents the `ta.vwap` rounded to the minimum tick without trailing zeros. string numberRepr2 = str.tostring(ta.vwap, format.mintick) //@variable Represents the `ta.vwap` rounded to three fractional digits without trailing zeros. string numberRepr3 = str.tostring(ta.vwap, "#.###") //@variable Represents the `ta.vwap` rounded to five fractional digits with trailing zeros. string numberRepr4 = str.tostring(ta.vwap, "0.00000") //@variable Represents the `ta.vwap` as an integer. string numberRepr5 = str.tostring(ta.vwap, "#") //@variable Represents the `ta.vwap` as an integer with leading zeros. string

numberRepr6 = str.tostring(ta.vwap, "000000") //@variable Represents the `ta.vwap` with commas for each third whole digit from the decimal point. // Values less than 1000000 include leading zeros. The fractional part includes up to two digits. string numberRepr7 = str.tostring(ta.vwap, "0000,000.##") //@variable Represents `100 * ta.tr / close` rounded to two fractional digits with `%` at the end. string numberRepr8 = str.tostring(100 * ta.tr / close, format.percent) //@variable Represents `ta.tr / close` as a percentage rounded to four fractional digits. // With `%` at the end of the `format`, the function multiplies the represented number by 100. string numberRepr9 = str.tostring(ta.tr / close, "#.####%") //@variable Represents the `volume` with fixed precision and letter characters for large figures. // `K` means "thousand", `M` means "million", `B` means "billion", and `T` means "trillion". string numberRepr10 = str.tostring(volume, format.volume) // @variable Represents a "bool" value. Is `"true"` when `ta.tr` exceeds the bar's range, `"false"` otherwise. string boolRepr = str.tostring(ta.tr > high - low) // Create an array and matrix of price values to convert into strings. array<float> pricesArray = array.from(open, close, low, high) //@variable Represents the `pricesArray` with up to two decimal places and no trailing zeros for each element. // Contains `[` and `]` characters to mark the start and end of the array's contents. string numberArrayRepr = str.tostring(pricesArray, "#.##") //@variable Represents the `pricesMatrix` with four decimal places and trailing zeros for each element. // Contains `[` and `]` characters to mark the start and end of each row. string numberMatrixRepr = str.tostring(pricesMatrix, "#.0000") //@variable Represents a "string" array containing the symbol's type and currency. string stringArrayRepr = str.tostring(array.from(syminfo.type, syminfo.currency)) // Populate the `displayTable` rows with each of the above variable names and their assigned "string" values. makeRow(1, "numberRepr1", numberRepr1) makeRow(2, "numberRepr2", numberRepr2) makeRow(3, "numberRepr3", numberRepr3) makeRow(4, "numberRepr4", numberRepr4) makeRow(5, "numberRepr5", numberRepr5) makeRow(6, "numberRepr6", numberRepr6) makeRow(7, "numberRepr7", numberRepr7) makeRow(8, "numberRepr8", numberRepr8) makeRow(9, "numberRepr9", numberRepr9) makeRow(10, "numberRepr10", numberRepr10) makeRow(11, "boolRepr", boolRepr) makeRow(12, "numberArrayRepr", numberArrayRepr) makeRow(13, "numberMatrixRepr", numberMatrixRepr) makeRow(14, "stringArrayRepr", stringArrayRepr)

Note that:

- The # and `0` tokens control the digits in the represented numbers in a similar way, but with different behaviors for leading and trailing *zeros*, as shown above. The `0` token *always* includes a digit at the specified decimal place, even for a leading or trailing zero, whereas the # token allows a leading or trailing digit only if it is *nonzero*.
- The `format` argument requires a # or `0` token for each *fractional* digit in a converted number. These tokens are optional for extra *whole* digits, because str.tostring() includes the necessary digits automatically.
- A single `,` token adds *repeated* comma separation to whole digits. In the str.tostring() call with the format `"0000,000.##"`, the token specifies that the result includes a dividing comma for every set of *three digits* to the left of the decimal point.
- When the `%` token is at the *end* of the formatting string, the representation multiplies numbers by 100 to express them as percentages, as shown by the example that uses `"#.####%"`.

Formatting strings

The str.format() function can combine multiple “int”, “float”, “bool”, “string”, or array arguments into one output string in a specified format. Using this function is a simpler alternative to creating multiple separate strings and combining them with repeated concatenation operations. Below is the function’s signature:

```
str.format(formatString, arg0, arg1, ...) → string
```

The `formatString` parameter accepts a “string” value that defines the *format* of the returned string, where the *placeholders* in curly brackets ({}) refer to the function call’s *additional arguments*. The placeholder `"{0}"` represents the first additional argument `arg0`, `"{1}"` represents `arg1`, and so on. The function *replaces* each placeholder in the `formatString` with a string representation of the corresponding argument. For instance, the call `str.format("The timeframe multiplier is {0}", timeframe.multiplier)` on a 1D chart returns `"The timeframe multiplier is 1"`.

The following example constructs a formatted string containing various bar information, then displays the result in a label at the bar's high. The str.format() call's `formatString` argument includes placeholders for 10 values, where each placeholder's *number* corresponds to one of the additional "string", "bool", "int", or "float" arguments:

image

//@version=6 indicator("Formatting strings demo", overlay = true) // @variable Counts script executions in a bar. Does not roll back because it is declared with `varip`. varip int exCount = 0 exCount += 1 //@variable Is `true` when the `close` exceeds the current `open` and the previous `close`; `false` otherwise. bool rising = close > math.max(nz(close[1]), open) // @variable Is `"realtime"` for all new bars after the script's historical executions; `"historical"` otherwise. string barState = barstate.isrealtime ? "realtime" : "historical" //@variable A formatting string containing placeholders for 10 values. string formatString = "Bar: {1} ({9}){0}{0} Executions: {2}{0}O: {3}{0}H: {4}{0}L: {5}{0}C: {6}{0}V: {7}{0}Rising: {8}" //@variable A multiline string containing formatted information for the current bar. string formattedString = str.format(formatString, "\n\t", bar_index, exCount - nz(exCount[1]), open, high, low, close, volume, rising, barState) // Draw a label displaying the `formattedString` at the bar's `high`. label.new(bar_index, high, formattedString, size = 18, textalign = text.align_left)

Note that:

- The `formatString` argument can use placeholders in *any order* and can repeat specific placeholders *more than once*. The format in this example uses `{0}` multiple times to insert the first argument (`"\n\t"`) to create multiline text with indentation.
- If a placeholder refers to a nonexistent argument, the formatted result treats that placeholder as a *literal* character sequence. For instance, a placeholder such as `{20}` in the `formatString` argument above includes those characters literally in the formatted result, because the str.format() call *does not* contain 21 additional arguments.
- *Non-quoted* left curly brackets (`{`) must have corresponding right curly brackets (`}`) inside formatting strings. If a `formatString` contains unbalanced curly brackets, it causes a *runtime error*.

It’s important to note that the apostrophe (') acts as a *quote character* inside formatting strings. When a formatting string contains a character sequence between two apostrophes, the formatted result includes that sequence directly, without treating the characters as placeholders or formatting tokens. This behavior applies even if the formatting string prefixes apostrophes with the backslash character (\). The enclosing apostrophes for a non-empty quoted sequence are **not** part of the formatted string. To include literal apostrophes in a str.format() call’s result, pass a “string” value containing the character as an *extra argument*, then use that argument’s *placeholder* in the specified `formatString`. Alternatively, use pairs of apostrophes with no characters between them directly in the `formatString` (e.g., `''` adds a single `'` character in the result).

The example below demonstrates how using apostrophes directly in formatting strings differs from inserting them via placeholders. The script uses the `'` character directly in the str.format() call’s `formatString` to define a *quoted sequence*, and it uses the {1} placeholder to insert the character from an extra argument without creating a quoted sequence. The script displays the resulting `formattedString` value in a single-cell table on the first bar:

image

//@version=6 indicator("Quotes in formatting strings demo") if barstate.isfirst //@variable A formatted string showing how using `'` directly differs from insertion via arguments. string formattedString = str.format("Apostrophes in formatting strings signify quoted character sequences, which do not have special meaning. \n\nQuoting a placeholder includes its characters literally: '{0}' \n\nInserting the apostrophe character from arguments does *not* create quoted sequences: {1}{0}{1}", ticker.standard(), "'") //@variable A single-cell table to display the `formattedString`. table display = table.new(position.middle_center, 1, 1, color.purple) // Initialize the cell with white, left-aligned text. display.cell(0, 0, formattedString, text_color = color.white, text_halign = text.align_left, text_size = 20)

When a str.format() call contains “int” or “float” arguments, the placeholders for those arguments in the `formatString` can include the `number` *modifier*

followed by a *formatting pattern* for customized numeric formats (e.g., `"{0,number,#.000}"`).

The possible numeric formatting patterns are similar to those for the `format` parameter of str.tostring(). They can contain `#`, `0`, and `.` tokens to specify decimal precision; use the `,` token for comma separation; and include `%` at the end for percentage conversion. Alternatively, a placeholder can use one of the following *keywords* that specify *predefined* formatting patterns: `integer`, `currency`, or `percent`.

The script below demonstrates how different numeric formats in a `formatString` placeholder affect the formatted representation of a "float" value. On the last bar, the script generates a pseudorandom value between 0 and 10000 with math.random(), uses several str.format() calls to format the value in different ways, and displays the results in a table:

image

//@version=6 indicator("Numeric formatting demo") //@variable A two-row, eight-column table showing a random number formatted with different format patterns. var table display = table.new(position.middle_center, 2, 8, frame_color = chart.fg_color, frame_width = 1) //@variable Initializes a row to show two specified strings in the `display` table. makeRow(int row, string str0, string str1) => color bgColor = row == 0 ? chart.fg_color : chart.bg_color color textColor = row == 0 ? chart.bg_color : chart.fg_color display.cell(0, row, str0, text_color = textColor, text_halign = text.align_left, text_size = 30, bgcolor = bgColor) display.cell(1, row, str1, text_color = textColor, text_halign = text.align_left, text_size = 30, bgcolor = bgColor) // Initialize the header row for the `display` table on the first bar. if barstate.isfirst makeRow(0, "Format pattern", "Result") if barstate.islast // @variable A random value between 0 and 10000. float value = math.random(0, 10000) //@variable Represents the `value` using the `str.format()` function's default numeric format (`#,###.###`). // This format uses comma-separated sets of three whole digits, and allows up to three fractional digits. // This default differs from the default for `str.tostring()` (`#.########`). string default = str.format("{0}", value) // @variable Represents the `value` using the `integer` preset (`#,###`). // This format rounds the `value` to the nearest whole and adds comma separation for three-digit sets. string integerPreset =

str.format("{0,number,integer}", value) //@variable Represents the `value` as an integer without comma-separated digits (`#`). string integerNoComma = str.format("{0,number,#}", value) //@variable Represents the `value` using the `currency` preset (`'$'#,###.00`). // This format prefixes the result with the `$` symbol, adds comma separation for sets of three whole // digits, and includes two fractional digits. string currencyPreset = str.format("{0,number,currency}", value) //@variable Represents the `value` in dollars with comma-separated whole digits and four fractional digits (`'$',###.0000`). string currencyCustom = str.format("{0,number,'$',###.0000}", value) //@variable Represents the `value` using the `percent` preset (`#,###%`). // This format multiplies the `value` by 100, rounds the result to the nearest whole number, adds comma // separation for three-digit sets, and includes the `%` symbol at the end. string percentPreset = str.format("{0,number,percent}", value) // @variable Represents the `value` as a percentage with comma-separated whole digits and up to three fractional digits. string percentCustom = str.format("{0,number,#,###.###%}", value) // Initialize rows showing each numeric format and the resulting representation of the `value`. makeRow(1, "Default (#,###.###)", default) makeRow(2, "integer (#,###)", integerPreset) makeRow(3, "integer without commas (#)", integerNoComma) makeRow(4, "currency ('$'#,###.00)", currencyPreset) makeRow(5, "currency with 4 fractional digits ('$',###.0000)", currencyCustom) makeRow(6, "percent (#,###%)", percentPreset) makeRow(7, "percent with up to 3 fractional digits (#,###.###%)", percentCustom)

Note that:

- In contrast to str.tostring(), the str.format() function does not directly support the preset formats defined by the `format.*` constants (format.mintick, format.percent, and format.volume). To use those formats on numeric values in a formatted string, convert the values with str.tostring() first, then use the resulting strings in the str.format() call.

The str.format() function's `formatString` also supports placeholders with the `date` or `time` modifier, which can format an "int" UNIX timestamp into a *UTC* date or time. For example, this line of code creates a string representing the current bar's opening timestamp as a date and time in the ISO 8601 standard format:

string formattedTime = str.format("{0,date,yyyy-MM-dd}
T{0,time,HH:mm:ssZ}", time)

However, str.format() **cannot** express dates and times in other time zones. It uses *UTC+0* exclusively. The specialized str.format_time() function is more optimal for constructing date-time strings, because it can express dates and times in *any time zone*. See the Formatting dates and times section of the Time page to learn more about this function and the available formatting tokens.

Custom representations

All built-in functions that create “string” values to represent data support a limited subset of built-in types. They *do not* support “color” values or objects of most *reference types* (e.g., labels). Programmers can, however, use custom logic and formatting to create “string” representations of data that the str.tostring() or str.format() functions cannot express as strings directly.

For example, this script demonstrates two ways to represent a “color” value as a string based on its red, green, blue, and transparency components. The first method formats the color components directly. The second calculates and formats each component’s *hexadecimal* form. The script displays the results of both custom formats in a label on the last historical bar.

After creating the label object, the script also uses log.info() to create formatted text containing the label’s `x`, `y`, and `text` *properties* and display the result in the Pine Logs pane:

image

//@version=6 indicator("Custom representations demo", overlay = true) // @variable The "color" value to color the label and convert to a string. color colorInput = input.color(#00897b) //@function Constructs a formatted string containing the color's R, G, B, and T components. rgbtString(color value) => str.format("color (R: {0}, G: {1}, B: {2}, T: {3})", color.r(value), color.g(value), color.b(value), color.t(value)) //@variable Constructs a string containing the color's hexadecimal RGBA representation. hexString(color value) => //@variable An array of hexadecimal characters formed by splitting a string. var array<string> chars = str.split("0123456789abcdef", "") // Get

the R, G, and B channel values from the `value`. int r = int(color.r(value)) int g = int(color.g(value)) int b = int(color.b(value)) //@variable The A (alpha) channel value (opposite of transparency), scaled to the range [0, 255]. int a = int((100 - color.t(value)) * 255 / 100) //@variable A formatted string combining hex codes for the R, G, B, and A channels. string result = str.format("#{0}{1}{2}{3}{4}{5}{6}{7}", chars.get(int(r / 16)), chars.get(r % 16), chars.get(int(g / 16)), chars.get(g % 16), chars.get(int(b / 16)), chars.get(b % 16), chars.get(int(a / 16)), chars.get(a % 16)) if barstate.islastconfirmedhistory //@variable A formatted string containing the results of `rgbtString()` and `hexString()`. string labelText = str.format("{0}\n{1}", rgbtString(colorInput), hexString(colorInput)) //@variable A label displaying the `labelText`. The script displays this object's properties in the Pine Logs pane. label displayLabel = label.new(bar_index, high, labelText, color = colorInput, textcolor = color.white, size = 36) // Log a custom representation of the `displayLabel` in the Pine Logs pane. log.info("\nlabel object\nx: {0,number,#}\ny: {1,number,#.#####}\ntext: {2}", displayLabel.get_x(), displayLabel.get_y(), displayLabel.get_text())

Note that:

- Not all special types have retrievable properties. For instance, scripts cannot retrieve information from polylines or tables. To create strings for these types, track the data used in their creation with separate variables, then format the values of those variables into strings.
- For an example of creating strings from the field values of user-defined types, see the Debugging objects of UDTs section of the Debugging page.

Modifying strings

Several `str.*()` functions provide simplified ways to modify the character sequence from a “string” value, including str.replace(), str.replace_all(), str.upper(), str.lower(), str.trim(), and str.repeat().

Programmers can use these functions to create copies of strings with replaced character sequences, modified letter cases, trimmed whitespaces, or repeated character patterns.

Replacing substrings

The str.replace() function searches a specified `source` string for the nth *non-overlapping* occurrence of a given substring, then returns a copy of the original string containing a specified replacement at that substring's position.

The str.replace_all() function searches the `source` for *every* non-overlapping occurrence of the substring and replaces each one in its returned value.

Below are the functions' signatures:

```
str.replace(source, target, replacement, occurrence) → string
str.replace_all(source, target, replacement) → string
```

Where:

- `source` is the "string" value containing the substrings to replace with a specified `replacement`.
- `target` is the substring replaced by the `replacement` in the returned copy. If the `source` value does not contain the substring, the function returns a copy of the value without modification.
- `replacement` is the substring inserted in place of the required `target` occurrences in the result.
- The `occurrence` parameter for str.replace() specifies which non-overlapping occurrence of the `target` is swapped for the `replacement` in the result. The default value is 0, meaning the function replaces the *first* occurrence of the `target`. If the specified occurrence does not exist in the `source` value, the function returns a copy of the value without modification.

The following script demonstrates the effects of str.replace() and str.replace_all() calls on a string containing the sequence `Hello world!`. Additionally, it calls these functions to define the `formatString` value for a str.format() call, which formats all the replacement results into a single "string" value. The script displays the formatted text inside a label anchored to the latest bar's opening time:

image

```
//@version=6 indicator("Replacing substrings demo") if barstate.isfirst // @variable A string containing the sequence `Hello world!`. string originalString = "Hello world!" //@variable A copy of the `originalString` with `!` replaced by `!!!`. string changePunctuation = str.replace(originalString, "!", "!!!") //@variable A copy of the `originalString` with the second `o` replaced by `0`. string changeLetter = str.replace(originalString, "o", "0", 1) // @variable A copy of the `originalString` with all `l` characters replaced by 1. string changeLetters = str.replace_all(originalString, "l", "1") //@variable A copy of the `originalString` with all zero-width boundaries replaced by `\n-`. string insertNewlines = str.replace_all(originalString, "", "\n") //@variable A copy of the `originalString` without changes, as `H` does not occur two times. string unchanged = str.replace(originalString, "H ", "_", 1) //@variable A formatting string with the following initial structure: `{0}\n{1}\n{2}\n{3}\n{4}\n{5}`. // The script creates the string by replacing zero-width boundaries in the sequence `012345` with // `}\n{` using `str.replace_all()`, then removing the extra `}\n` and `\n{` from the start and end of the // call's result with two additional `str.replace()` calls. string formatString = str.replace(str.replace(str.replace_all("012345", "", "}\n{"), "}\n", ""), "\n{", "", 5) // Create a copy of the `formatString` with the first `}` replaced by `}\n----------`, then reassign // the variable to use that copy. formatString := str.replace(formatString, "}", "}\n----------") //@variable A formatted string containing the original and modified "Hello world!" strings. string displayText = str.format( formatString, originalString, changePunctuation, changeLetter, changeLetters, insertNewlines, unchanged ) // Draw a label anchored to the latest bar's opening time to show the `displayText`. label.new( math.max(last_bar_time, chart.right_visible_bar_time), 0, displayText, xloc.bar_time, style = label.style_label_center, size = 20 )
```

Note that:

- Each `str.replace*()` call creates an independent, modified copy of the specified `source` value. Because each modification of the `originalString` is assigned to a separate variable, each value *does not* contain changes from previous `str.replace*()` calls.
- The `str.replace*()` functions can replace zero-width *boundaries* when the `target` is an empty string, as shown by the `formatString`

declaration. The str.replace_all() call inserts }\n{ around every character in the literal string "012345".

Changing case

The str.upper() and str.lower() functions create a copy of a source string with all ASCII letter characters converted to *uppercase* or *lowercase* variants, providing a convenient alternative to replacing specific characters with several str.replace*() calls. The str.upper() function replaces all lowercase characters with uppercase characters, and str.lower() does the opposite. These are the functions' signatures:

```
str.upper(source) → string
str.lower(source) → string
```

This simple example demonstrates how these functions affect strings with *standard* letter characters. The script declares an originalString variable to hold a literal string, uses str.upper() on that variable to create a copied string with all letters converted to uppercase, then calls str.lower() to make a copy with only lowercase characters. It logs all three strings in the Pine Logs pane on the first bar:

image

//@version=6 indicator("Changing case demo", overlay = true) if barstate.isfirst //@variable A literal string containing the sequence `Hello World!` string originalString = "Hello World!" //@variable A copy of the `originalString` with all lowercase ASCII characters changed to uppercase characters. string uppercaseString = str.upper(originalString) //@variable A copy of the `uppercaseString` with all uppercase ASCII characters changed to lowercase characters. string lowercaseString = str.lower(uppercaseString) // Log a formatted message containing all three values. log.info("\n\nOriginal: {0}\nUppercase: {1}\nLowercase: {2}", originalString, uppercaseString, lowercaseString)

Note that these functions can only change the cases of *ASCII* letter characters. They cannot convert other Unicode letters outside the ASCII range. For example, this script attempts to create uppercase and lowercase versions of a "string" value containing "Mathematical Sans-Serif" Unicode

characters using str.upper() and str.lower(). As shown below, both function calls return *identical* copies of the value:

image

//@version=6 indicator("Non-ASCII case demo", overlay = true) if barstate.isfirst //@variable A literal string that uses Unicode characters in the "Mathematical Sans-Serif" family. string originalString = "This string is unaffected by the case-swapping functions!" // Call `str.upper()` and `str.lower()` to change the case of letters. // Although the characters in the `originalString` have the "Letter" property, they are not part of the standard // ASCII set. Consequently, these calls return unmodified strings. string uppercaseString = str.upper(originalString) string lowercaseString = str.lower(uppercaseString) // Log a formatted message containing all three values. log.info("\n\nOriginal: {0}\nUppercase: {1}\nLowercase: {2}", originalString, uppercaseString, lowercaseString)

Trimming whitespaces

The str.trim() function copies a `source` string and removes leading and trailing whitespace characters, including the standard space ( ), newline (`\n`), and tab space (`\t`). Below is the function's signature:

```
str.trim(source) → string
```

This simple example demonstrates the str.trim() function's behavior. The script creates a literal string containing different types of whitespaces at the start and end of the character sequence. Then, it uses str.trim() to create a new "string" value with those characters removed. The script formats both values into a single string, then displays the result in a label on the last historical bar:

image

//@version=6 indicator("Trimming whitespaces demo") if barstate.islastconfirmedhistory //@variable A literal string containing space, newline, and tab characters. string originalString = "\n\n\t\tABC DEF\t \n\n " // @variable A copy of the `originalString` that contains only `ABC DEF` without the other whitespaces. string trimmedString = str.trim(originalString) // @variable A formatted string containing the `originalString` and

`trimmedString` values. string displayText = str.format("Original: \"{0}\"\n---\n\nTrimmed: \"{1}\"", originalString, trimmedString) // Draw a label to show the `displayText`. label.new(bar_index, 0, displayText, style = label.style_label_center, size = 30, textalign = text.align_left)

Note that:

- The str.trim() function removes only the ASCII whitespaces to the left of the *first* non-whitespace character and the right of the *last* non-whitespace character. It does **not** remove whitespaces between other characters.
- The formatting string in the str.format() call uses `\"` to include quotation marks around the `originalString` and `trimmedString` values in the displayed text. See the Escape sequences section above for more information.

The str.trim() function is particularly helpful when supplying calculated or input strings to built-in functions that process “string” arguments, because some function parameters require values *without* leading or trailing whitespaces.

The following example creates an array of timeframe strings by splitting the value of a text area input based on its comma characters. Within a loop, the script uses each element from the array in a time() call to retrieve an opening time, then concatenates a formatted date and time with the `displayText`.

Although each item listed in the default string represents a valid timeframe, the time() call causes a runtime error. The script splits the value only by its commas, resulting in a *leading space* in each `timeframes` element after the first, and the time() function *does not* allow whitespaces in its `timeframe` argument.

If the user enables the input to trim the input string (which is off by default), the script uses str.trim() to remove surrounding whitespaces from the time() call’s argument and prevent the formatting issue and the runtime error.

image

//@version=6 indicator("Invalid arguments with whitespaces demo", overlay = true) //@variable A string containing a comma-separated list of

timeframes. string timeframesInput = input.text_area("1, 5, 15, 30, 60, 240, 720, 1D", "Timeframes") //@variable A boolean variable that defines whether to trim the input string. bool doTrimInput = input.bool(false, "Trim the \"Timeframes\" input string") //@variable An array of timeframe substrings. var array<string> timeframes = str.split(timeframesInput, ",") //@variable A concatenated string containing formatted dates and times for each value in the `timeframes` array. string displayText = "Opening times:" for timeframe in timeframes //@variable A copy of the `timeframe` string with surrounding whitespaces removed. string trimmedTimeframe = str.trim(timeframe) // @variable The UNIX timestamp of the bar's opening time on the timeframe. // This call causes a runtime error if the input string is not trimmed, because the argument contains // a leading whitespace (e.g., " 5"), which is an *unsupported format* for timeframe strings. int timestamp = time(doTrimInput ? trimmedTimeframe : timeframe) // Add a new text line containing the `timestamp` formatted as a date and time in the exchange time zone. displayText += "\n" + timeframe + str.format_time(timestamp, " yyyy-MM-dd, HH:mm:ss") // Draw a label at the bar's `high` with a tooltip showing the `displayText`. label.new(bar_index, high, style = label.style_label_down, tooltip = displayText)

Repeating sequences

The str.repeat() function creates a “string” value that *repeats* a `source` string’s character sequence a specified number of times, providing a convenient way to construct strings with repetitive character patterns. Below is the function’s signature:

```
str.repeat(source, repeat, separator) → string
```

Where:

- `source` is the “string” value containing the character sequence to repeat in the result.
- `repeat` is an “int” value specifying the number of times the function repeats the `source` sequence. If the value is 0, the function returns an empty string.
- `separator` is an optional “string” value containing a character sequence to *insert* between each repeated instance of the `source` sequence. The

default value is an empty string, meaning the function repeats the `source` sequence *without* inserting additional characters.

The following script formats two numbers — the OHLC4 and Simple Moving Average of price — with a variable number of fractional digits. The minimum and maximum number of fractional digits are set by user inputs. The script uses a str.repeat() call to repeat `0` characters to create a pattern for the *required* digits, and another call that repeats # characters to create a pattern for the *optional* digits, which are displayed only if they are nonzero. The script then concatenates these patterns into one pattern and uses that in a str.format() call to format the two numbers.

The script calls log.info() to log the constructed `formatString` on the first bar, and it displays the formatted results for each bar using labels:

image

//@version=6 indicator("Repeating sequences demo", overlay = true) // @variable The minimum number of fractional digits in the numeric strings. int minPrecisionInput = input.int(2, "Min precision", 0, 16) //@variable The maximum number of fractional digits in the numeric strings. int maxPrecisionInput = input.int(8, "Max precision", 0, 16) // Raise an error if the `maxPrecisionInput` is less than the `minPrecisionInput`. if maxPrecisionInput < minPrecisionInput runtime.error("The 'Max precision' cannot be less than the 'Min precision.") //@variable A string containing the `0` character repeated `minPrecisionInput` times. var string requiredDigits = str.repeat("0", minPrecisionInput) //@variable A string containing the `#` character repeated `maxPrecisionInput - minPrecisionInput` times. var string extraDigits = str.repeat("#", maxPrecisionInput - minPrecisionInput) // @variable A string representing a formatting pattern for numeric strings. // With default inputs, the value is `"0.00######"`, meaning two required digits and six nonzero extra digits. var string formatPattern = "0." + requiredDigits + extraDigits //@variable A formatting string that contains two placeholders for the `formatPattern` for numeric values. var string formatString = str.format("OHLC4: '{'0,number,{0}'}', MA: '{'1,number, {0}'}'", formatPattern) // Log the `formatString` value on the first bar. if barstate.isfirst log.info(formatString) // Draw a label to show the bar's formatted result. label.new(bar_index, high, str.format(formatString, ohlc4, ta.sma(ohlc4, 20)), size = 18)

Note that:

- The apostrophe (') in the str.format() call serves as a *quote character*, **not** a literal character. The `formatString` uses the apostrophe to quote curly brackets ({ and }), treating them as literal characters instead of direct placeholder markers.

The example below demonstrates a more creative use of str.repeat(). This script generates an ASCII art representation of the Pine Script logo using alternating sequences of repeated `.` (period) and `@` (at) characters. The user-defined `makeLine()` function calls str.repeat() seven times to create the repeated sequences, then formats their results into a single "string" value with a str.format() call. On the first bar, the script formats the results of several `makeLine()` calls into a multiline string and displays the result in a single-cell table in the chart's top-right corner:

image

//@version=6 indicator("ASCII art from repeated sequences demo", overlay = true) //@function Creates a string containing alternating sequences of repeated `.` and `@` characters with `\n` at the end. // Each `dot*` argument defines a sequence of repeated `.` characters, and each `at*` argument defines a // sequence of repeated `@` characters. The function formats the repeated sequences from `str.repeat()` calls // into a single string, in the order of the parameters. // For example, `makeLine(6, 5, 4, 3, 2)` returns `"......@@@@@....@@@..\n"`. makeLine(int dot1 = 0, int at1 = 0, int dot2 = 0, int at2 = 0, int dot3 = 0, int at3 = 0, int dot4 = 0) => string result = str.format("{0}{1}{2}{3}{4}{5}{6}\n", str.repeat(".", dot1), str.repeat("@", at1), str.repeat(".", dot2), str.repeat("@", at2), str.repeat(".", dot3), str.repeat("@", at3), str.repeat(".", dot4)) result if barstate.isfirst // @variable A string representing the Pine logo using several lines of different repeated `.` and `@` sequences. string asciiArt = str.format("{0}{1}{2}{3}{4}{5}{6}{7}{8}{9}{10}{11}{12}{13}{14}{15}{16}{17}" + "{18}{19}{20}{21}{22}{23}{24}{25}{26}{27}{28}{29}{30}{31}{32}{33}", makeLine(80), makeLine(80), makeLine(38, 6, 36), makeLine(37, 8, 35), makeLine(35, 12, 33), makeLine(34, 15, 31), makeLine(33, 7, 2, 7, 31), makeLine(31, 8, 4, 8, 29), makeLine(29, 8, 8, 8, 27), makeLine(28, 8, 10, 9, 25), makeLine(27, 7, 14, 8, 24), makeLine(25, 8, 17, 8, 22), makeLine(24, 8, 19, 8, 21), makeLine(23, 7, 22, 9, 19), makeLine(21, 17, 16, 9, 17),

makeLine(20, 20, 16, 8, 16), makeLine(18, 25, 14, 9, 14), makeLine(18, 14, 3, 10, 13, 8, 14), makeLine(18, 8, 11, 10, 9, 10, 14), makeLine(39, 10, 4, 12, 15), makeLine(29, 6, 7, 20, 18), makeLine(24, 14, 6, 15, 21), makeLine(20, 21, 5, 10, 24), makeLine(18, 25, 5, 5, 5, 6, 16), makeLine(17, 28, 10, 10, 15), makeLine(16, 32, 4, 14, 14), makeLine(14, 54, 12), makeLine(13, 56, 11), makeLine(12, 58, 10), makeLine(10, 62, 8), makeLine(9, 64, 7), makeLine(9, 64, 7), makeLine(80), makeLine(80)) //@variable A single-cell table to display the `asciiArt` value. table t = table.new(position.top_right, 1, 1, frame_color = chart.fg_color, frame_width = 1) // Initialize a left-aligned cell with monospace font for proper alignment. t.cell(0, 0, asciiArt, text_color = chart.fg_color, text_halign = text.align_left, text_size = 6, text_font_family = font.family_monospace)

Note that:

- The table.cell() call uses text.align_left as the `text_halign` argument and font.family_monospace as the `text_font_family` argument to align the text lines to the *left* with relatively uniform character width.
- The formatted string from each `makeLine()` call uses the `\n` escape sequence at the end to add a line terminator.

String inspection and extraction

Several built-in `str.*()` functions allow scripts to measure a “string” value, check for substrings and retrieve their positions, split a string into several substrings, and extract substrings based on positions or match patterns. These functions include str.length(), str.contains(), str.startswith(), str.endswith(), str.split(), str.pos(), str.substring(), and str.match().

The sections below explain these functions and some helpful techniques to use them effectively.

Counting characters and substrings

The str.length() function measures the length of a specified “string” value, returning an “int” value representing the number of characters in the argument’s character sequence. It has the following signature:

```
str.length(string) → int
```

This function detects *every* character within a “string” value’s sequence, even those that are hard to see, such as leading or repeated spaces, line terminators, and invisible characters like U+200B (Zero Width Space).

For example, this simple script declares two variables with assigned literal strings and measures their length. The script creates the first “string” value on a single line, and it defines the second using *line wrapping*. It measures the length of both strings with str.length(), creates modified strings with the "__" parts replaced by the length values, then concatenates the results for display in a single-cell table.

Although the two initial literal strings are similar, their lengths differ because *line-wrapped* strings include an *extra space* for each additional code line between the enclosing characters:

image

//@version=6 indicator("Counting characters demo") if barstate.isfirst // @variable A single-cell table to display text. var table display = table.new(position.middle_center, 1, 1, color.teal) //@variable A literal string defined on a single line. string testString1 = "This 'string' contains\n__ characters." //@variable A literal string defined across multiple lines. Although this value looks identical to `testString1`, // it contains three extra characters, because each new code line between the quotation marks // automatically adds a new space to the value. string testString2 = " This 'string' contains \n__ characters. " // Count the number of characters in `testString1` and `testString2`. int length1 = str.length(testString1) int length2 = str.length(testString2) // Replace the "__" in `testString1` and `testString2` with string representations of `length1` and `length2`. testString1 := str.replace(testString1, "__", str.tostring(length1)) testString2 := str.replace(testString2, "__", str.tostring(length2)) // Concatenate both strings and two newline characters, then display the result in the table's cell. string displayString = testString1 + "\n\n" + testString2 display.cell(0, 0, displayString, text_color = color.white, text_size = 50, text_halign = text.align_left)

Note that:

- The `testString2` value contains *three* extra characters, but not all the additional characters are apparent from viewing the table’s cell. The first

is at the start of the sequence, the second is before the line terminator (\n), and the third is at the end.
- A simple way to verify the added characters is to *split* the `testString2` value into an array of substrings with str.split() and inspect the array's elements. See the Splitting strings section to learn more about this function.

The str.length() function is also useful for counting the number of *substrings* of any size contained within a string's sequence, which is helpful information when replacing substrings or performing custom routines that depend on recurring characters.

The following example defines a `countSubstrings()` function, which uses str.replace_all() and str.length() to count the number of times a `target` substring occurs within a specified `source` value. The function creates a modified copy of the `source` with all instances of the `target` removed, then calls str.length() to measure the length of each separate string. It calculates the number of `target` occurrences by dividing the length difference in the original and reduced strings by the length of the substring.

The script uses str.repeat() to generate a "string" value that repeats the sequence aba a random number of times with baab inserted between each instance, then counts all occurrences of the substring ab in the result with a `countSubstrings()` call. It then displays a formatted message containing the repeated sequence and the total number of ab substrings in the Pine Logs pane:

image

//@version=6 indicator("Counting substrings demo", overlay = true) // @function Counts the number of times a `target` substring occurs within the specified `source`. countSubstrings(string source, string target) => // @variable A copy of the `source` string with all instances of the `target` removed. string reduced = str.replace_all(source, target, "") // Count the characters in the `source`, `target`, and `reduced` strings. int sourceLength = str.length(source) int targetLength = str.length(target) int reducedLength = str.length(reduced) // Calculate the difference between `sourceLength` and `reducedLength` relative to the `targetLength`. // This value represents the number of `target` substrings inside the original `source`. (sourceLength -

reducedLength) / targetLength //@variable A string containing `aba` repeated a random number of times with `baab` inserted between each instance. string randString = str.repeat("aba", int(math.random(1, 8)), "baab") //@variable The number of times the `ab` sequence occurs in the `randString`. int count = countSubstrings(randString, "ab") // Log a formatted message containing the `randString` and `count` values in the Pine Logs pane. log.info("randString: {0}, count: {1,number,#}", randString, count)

Note that:

- This script uses the *second overload* of log.info(), which shares the same signature as str.format() but logs a formatted message instead of returning a value. See the Pine Logs section of the Debugging page to learn more about the `log*()` functions.

Checking for substrings

The str.contains() function searches a `source` string for a specified substring, returning a "bool" value representing whether it found the substring. Two similar functions, str.startswith() and str.endswith(), check whether the `source` *starts* and *ends* with a specified substring.

These functions have the following signatures:

```
str.contains(source, str) → bool
str.startswith(source, str) → bool
str.endswith(source, str) → bool
```

Where:

- `source` is the "string" value that the function searches to find the substring.
- `str` is a "string" value containing the substring to find in the `source`. The str.contains() function returns `true` if the `source` contains at least one instance of the substring. The str.startswith() function returns `true` only if the `source` starts with the substring, even if the substring exists elsewhere in the character sequence. Likewise, str.endswith() returns `true` only if the `source` ends with the substring.

These functions are convenient when a script needs to check whether a substring exists but does not require the substring in additional calculations. Programmers often use these functions in conditional logic to control script behaviors based on a “string” value’s contents.

The following script creates a spread symbol string from two symbol inputs and requests price information from that spread symbol using a request.security() call. Before executing the request, the script calls str.startswith() to check whether the `spreadInput` value starts with a leading space and forward slash (`/`), indicating that the first input is *empty*. If the call returns `true`, the script replaces the missing symbol in the “string” value with the chart’s symbol to prevent errors.

The script then plots the retrieved data as candles in a separate pane. The colors of the candles change if the chart is in Bar Replay mode. The script tests for Bar Replay mode by searching for the `replay` substring in the chart’s ticker identifier (syminfo.tickerid) using a str.contains() call:

image

//@version=6 indicator("Checking for substrings demo") //@variable A spread symbol created from two symbol inputs. The first input is an empty string by default. var string spreadInput = input.symbol("", "Symbol 1") + " / " + input.symbol("BATS:SPY", "Symbol 2") //@variable Is `true` if the `spreadInput` starts with ` /`, meaning the first input is empty. var bool missingNumerator = str.startswith(spreadInput, " /") //@variable A copy of the spreadInput` that inserts the chart's standard ticker ID when `missingNumerator` is `true`. var string spread = missingNumerator ? str.replace(spreadInput, " /", ticker.standard() + " /") : spreadInput // Request a tuple of OHLC data from the `spread` context. [o, h, l, c] = request.security(" " + spread, "", [open, high, low, close]) //@variable Is `true` if the chart's ticker ID contains `replay`, meaning Bar Replay mode is active. var bool isReplay = str.contains(syminfo.tickerid, "replay") // Define variable to hold colors for up and down candles. Their values depend on whether `isReplay` is `true`. var color upColor = isReplay ? color.blue : color.teal var color dnColor = isReplay ? color.maroon : color.red // Plot the candles in a separate pane. plotcandle(o, h, l, c, "Spread candles", c > o ? upColor : dnColor) // Log the original `spreadInput`, the final `spread` value, and the `isReplay` value` in the Pine Logs pane. if barstate.isfirst

log.info("\n\nOriginal input: {0}\nFinal spread: {1}\nReplay chart: {2}", spreadInput, spread, isReplay)

Note that:

- We used `" /"` as the substring value in the str.startswith() call because an empty string does *not* detect the empty input value. When the substring specified in a str.contains(), str.startswith(), or str.endswith() call is empty, the function always returns `true` because the argument can match *any position* in a string's sequence.

Splitting strings

The str.split() function splits a single "string" value into one or more substrings based on a `separator` substring in the value's character sequence, then collects the results in an array. Below is the function's signature:

```
str.split(string, separator) → array<string>
```

Where:

- The specified `string` is the value to divide into substrings.
- The `separator` is a "string" value containing the characters that divide each substring. The resulting array does *not* include the separator in its elements. If the value is empty, the function splits the string into single-character substrings.

The str.split() function returns an array of strings, unlike the other `str.*()` functions. Scripts can use `array.*()` functions on these arrays, or iterate through them directly with for...in loops. Programmers often use str.split() to process "string" inputs and parameters that represent *lists* of arguments for dynamic requests and other calculations.

The following script requests data from several contexts based on a text area input containing a comma-separated list of symbols. First, the script splits the input value based on its commas with str.split() to construct an array of symbol strings. Then, it uses a for...in loop to iterate over the array's contents, request data for each symbol, and populate a table with the results.

Additionally, the table’s first row contains a “string” representation of the array of symbols:

image

//@version=6 indicator("Splitting strings demo") //@variable A string containing a list of symbols separated by commas and optional spaces. string symbolListInput = input.text_area("AAPL, NVDA, MSFT, AMZN, SPY") // @variable An array of symbols created by splitting the `symbolListInput` by its commas. var array<string> symbolsArray = str.split(symbolListInput, ",") if barstate.islast //@variable A two-column table with a row for each `symbolsArray` item. var table display = table.new(position.middle_center, 2, symbolsArray.size() + 1) // Initialize a merged cell to show the `symbolsArray`. display.cell(0, 0, "Symbols:\n" + str.tostring(symbolsArray), bgcolor = color.blue, text_color = color.white, text_size = 30) display.merge_cells(0, 0, 1, 0) // Loop through the `symbolsArray`. for [i, symbol] in symbolsArray //@variable The `close` value requested for the `symbol` on the chart's timeframe. float requestedValue = request.security(symbol, timeframe.period, close) // Initialize a cell for the `symbol` and a "string" representing the `requestedValue`. display.cell(0, i + 1, symbol, text_color = chart.fg_color, text_size = 20) display.cell(1, i + 1, str.tostring(requestedValue), text_color = chart.fg_color, text_size = 20)

Note that:

- The symbol strings in the array contain *extra whitespaces*, which are not visible in the table. However, in contrast to some other function parameters, the `symbol` parameter of request.security() ignores leading and trailing whitespaces in its argument.
- The script uses str.tostring() and concatenation to create the strings used for the table’s text.
- This script can fetch data from other contexts within a loop using “series string” `symbol` values because scripts allow dynamic requests by default. See the Dynamic requests section of the Other timeframes and data page for more information.

Locating and retrieving substrings

The str.pos() function searches a `source` string for the *first* occurrence of a specified substring and returns an “int” value representing the *position* of its initial character boundary. The function’s signature is as follows:

```
str.pos(source, str) → int
```

Where:

- `source` is the “string” value to search for the first occurrence of the `str` substring.
- `str` is a “string” value representing the substring to locate in the `source`. If the argument is na or an empty “string” value, the function returns 0 (the first possible position).

The str.substring() function retrieves a substring from a `source` value at specified character positions. This function has the following signatures:

```
str.substring(source, begin_pos) → string
str.substring(source, begin_pos, end_pos) → string
```

Where:

- `source` is the “string” value containing the substring.
- `begin_pos` is an “int” value representing the position of the substring’s first character in the `source`, where the numbering starts from zero. If the value is na, the function sets the initial position to 0. The script raises an error if the specified position is invalid.
- The `end_pos` is an “int” value representing the position *after* the substring’s last character in the `source`. This position is *exclusive*, meaning the returned value does **not** contain this position’s character. If the value is not specified or represents a position outside the string’s length, the substring includes *all* characters from the `begin_pos` onward. If the value is less than the `begin_pos`, it causes a runtime error.

For example, the `begin_pos` value of the substring `"Trading"` in the string `"TradingView"` is 0, because the substring starts at the source string’s *first* character position. The `end_pos` value is 7, because the substring’s last

character (`g`) is at position 6, and `end_pos` represents the position *after* that character. To retrieve only the first character of a string as a substring, use a call such as `str.substring("TradingView", 0, 1)`.

Programmers often use these functions together by retrieving positional values with str.pos() and then using those values to extract substrings with str.substring() for additional calculations. This technique is an efficient alternative to matching patterns for substrings at specific positions that have unique characters.

The following simple script uses these functions to extract the “area” and “location” parts of the syminfo.timezone variable’s *IANA identifier*. The script calls str.pos() to get the position of the `/` character in the time zone string, which it assigns to the `dividerPos` variable. Then, it uses that variable in two str.substring() calls. The first call retrieves the substring from position 0 to `dividerPos`, and the second retrieves the substring from the position at `dividerPos + 1` to the end of the string.

The script displays the IANA identifier, the retrieved substrings, and the formatted date and time of the latest execution in a single-cell table on the last bar:

image

//@version=6 indicator("Locating and retrieving substrings demo", overlay = true) if barstate.islast //@variable A single-cell table to show the `displayText`. var table display = table.new(position.bottom_right, 1, 1, frame_color = chart.fg_color, frame_width = 1) //@variable The position of the `/` character in the exchange time zone's IANA identifier. var int dividerPos = str.pos(syminfo.timezone, "/") //@variable The `syminfo.timezone` substring from position 0 to `dividerPos`. // This substring epresents the "area" part of the time zone identifier. // The character at `dividerPos` is not included. var string areaString = str.substring(syminfo.timezone, 0, dividerPos) //@variable The `syminfo.timezone` substring from `dividerPos + 1` to the end of the string, without low lines. // This substring represents the "location" part of the time zone identifier. var string locationString = str.replace_all(str.substring(syminfo.timezone, dividerPos + 1), "_", " ") // @variable A string representing the latest execution's date and time in the

chart's time zone. string formattedTime = str.format_time(timenow, "HH:mm:ss 'on' MMM d, yyyy") //@variable A formatted string containing `syminfo.timezone`, `areaString`, `locationString`, and `formattedTime`. string displayText = str.format("IANA identifier: {0}\n\nArea: {1}\nLocation: {2}\n\nTime of latest tick: {3}", syminfo.timezone, areaString, locationString, formattedTime) // Initialize the table cell with the `displayText`. display.cell(0, 0, displayText, text_size = 24, text_color = chart.fg_color)

Note that:

- The first str.substring() call does *not* include the character at the position specified by `dividerPos`. Its result contains only the characters from position 0 to `dividerPos - 1`.

It's important to emphasize that the str.pos() function only finds the *first* occurrence of a specified substring. However, in some cases, programmers might require the positions of the substring's *other* occurrences. One way to achieve this result is by repeatedly *reducing* a "string" value with str.substring() and locating the substring in the new value with str.pos().

The advanced example script below contains a `getPositions()` function that returns an array containing every `substring` position within a specified `source`. The function first uses str.pos() to get the position of the first `substring` and creates an array containing that value with array.from(). If the initial position is not na, the function removes all characters up to the substring's end position with str.substring(). Then, it executes a for...in loop that repeatedly locates the substring, pushes the calculated position into the array, and reduces the character sequence. The loop stops only after the array contains the position of every substring in the `source` value.

On the first bar, the script uses the function to analyze substring occurrences in four arbitrarily selected strings, then logs formatted messages containing the results in the Pine Logs pane:

image

//@version=6 indicator("Locating multiple substrings demo", overlay = true) getPositions(string source, string substring) => //@variable The position of the first occurrence of the `substring`. int firstPos = str.pos(source, substring) //@variable An array containing the starting position of each

`substring` occurrence in the `source`. array<int> positions = array.from(firstPos) // Search for extra positions if `firstPos` is not `na`. if not na(firstPos) //@variable The length of the `substring`. int substringLength = str.length(substring) //@variable A substring of `source` from `firstPos + substringLength` onward. string reduced = str.substring(source, firstPos + substringLength) // Loop through the `positions` array. for pos in positions // @variable The end boundary position of the first `substring` occurrence in the `reduced` string. int newPos = str.pos(reduced, substring) + substringLength // Add a new element to the `positions` array and reduce the `reduced` string if `newPos` is not `na`. if not na(newPos) // Push `pos + newPos` into the `positions` array, allowing another iteration. // The `newPos` is added to the latest `pos` to get the actual position in the original `source`. positions.push(pos + newPos) // Assign the substring from `newPos` onward to the `reduced` variable. reduced := str.substring(reduced, newPos) positions if barstate.isfirst // Define four arbitrary strings. string testStr1 = "NASDAQ:AAPL" string testStr2 = "1234321234321" string testStr3 = str.repeat(str.repeat("abc", 3, ", "), 2, " a") string testStr4 = str.format_time(time) // Get arrays containing the positions of various substrings in the four test strings. array<int> positions1 = getPositions(testStr1, "A") array<int> positions2 = getPositions(testStr2, "12") array<int> positions3 = getPositions(testStr3, ", ") array<int> positions4 = getPositions(testStr4, "-") //@variable The formatting string for all `log.info()` calls. The `{0}` placeholder is for apostrophes. string formatString = "Positions of {0}{1}{0} in {0}{2}{0}: {3}" // Log formatted results in the Pine Logs pane. log.info(formatString, "'", "A", testStr1, str.tostring(positions1)) log.info(formatString, "'", "12", testStr2, str.tostring(positions2)) log.info(formatString, "'", ", ", testStr3, str.tostring(positions3)) log.info(formatString, "'", "-", testStr4, str.tostring(positions4))

Note that:

- Although the `positions` array starts with *one* element, the for...in loop performs *more than one* iteration because Pine loops can have dynamic boundaries. After each execution of the array.push() call, the `positions` array's size increases, allowing a *new iteration*. Refer to the Loops page for more information.

- Each reduced version of the string starts at the position *after* the last character of the detected substring. The script identifies the end position by adding the substring's str.length() value to its starting position.

Matching patterns

Pine scripts can dynamically match and retrieve substrings using the str.match() function. In contrast to the other `str.*()` functions, which only match sequences of literal characters, the str.match() function uses regular expressions (regex) to match variable *character patterns*. The function's signature is as follows:

```
str.match(source, regex) → string
```

Where:

- `source` is the "string" value containing the sequence to match using the regular expression.
- `regex` is a "string" value representing the regular expression that specifies the *pattern* to match in the `source`. The function returns the *first substring* that follows the match pattern. If the regex does not match any substring in the `source`, the function returns an empty string.

Because the str.match() function matches *patterns* in a string's character sequence rather than strictly literal characters, a single call to this function can perform a wide range of text-matching tasks that would otherwise require multiple calls to other `str.*()` functions or custom operations.

For example, this script requests data from a FINRA Short Sale Volume series for a specified symbol. It uses separate str.startswith() calls to check whether the symbol string has one of the supported exchange prefixes. It locates and removes the exchange prefix with str.pos() and str.substring(), constructs a FINRA ticker ID with str.format() and logs its value, then executes the request.security() call only if one of the str.startswith() calls returns `true`. The script plots the retrieved data on the chart as columns:

image

//@version=6 indicator("Detecting substrings with other functions demo") // @variable The symbol for which to request Short Sale Volume data. string symbolInput = input.symbol("NASDAQ:AAPL", "Symbol") //@variable `true` if the `symbolInput` starts with `BATS:`, `NASDAQ:`, `NYSE:`, or `AMEX:`. Otherwise, `false`. var bool supportedSymbol = str.startswith(symbolInput, "BATS:") or str.startswith(symbolInput, "NASDAQ:") or str.startswith(symbolInput, "NYSE:") or str.startswith(symbolInput, "AMEX:") // @variable The requested FINRA data if `supportedSymbol` is `true`, `na` otherwise. float requestedData = if supportedSymbol //@variable The `symbolInput` value without the exchange prefix. var string noPrefix = str.substring(symbolInput, str.pos(symbolInput, ":") + 1) //@variable A formatted string representing the ticker ID of a FINRA Short Sale Volume dataset. var string finraTickerID = str.format("FINRA:{0}_SHORT_VOLUME", noPrefix) // Log the `finraTickerID` in the Pine Logs pane on the first bar. if barstate.isfirst log.info(finraTickerID) // Retrieve the data. request.security(finraTickerID, timeframe.isintraday ? "1D" : "", close) // Plot the `requestedData` plot(requestedData, "Short Sale Volume", color.teal, 1, plot.style_columns)

In the script version below, we replaced the multiple str.startswith() calls with an expression containing a str.match() call. The call matches one of the supported exchange prefixes at the start of the string using the following regular expression:

```
^(?:BATS|NASDAQ|NYSE|AMEX):
```

We also replaced the str.pos() and str.substring() calls with a str.match() call. The call calculates the `noPrefix` value with a regex that matches all characters after the input value's colon (`:`):

```
(?<=:).+
```

These changes achieve the same results as the previous script, but with more concise function calls:

//@version=6 indicator("Matching substrings with regex demo") //@variable The symbol for which to request Short Sale Volume data. string symbolInput = input.symbol("NASDAQ:AAPL", "Symbol") //@variable `true` if the `symbolInput` starts with `BATS:`, `NASDAQ:`, `NYSE:`, or `AMEX:`. Otherwise, `false`. // - `^` at the beginning of the regex matches the start of

the string. // - `(?:...)` defines a non-capturing group. // - `|` is an OR operator that allows the group to match one of the listed options. var bool supportedSymbol = str.match(symbolInput, "^(?:BATS|NASDAQ|NYSE|AMEX):") != "" //@variable The requested FINRA data if `supportedSymbol` is `true`, `na` otherwise. float requestedData = if supportedSymbol //@variable The `symbolInput` value without the exchange prefix. // - `(?<=:)` in the regex is a lookbehind assertion that checks if `:` precedes the match. // - `.` matches any character except for line terminators in this context. // - `+` is a quantifier that specifies one or more consecutive `.` matches. var string noPrefix = str.match(symbolInput, "(?<=:).+") //@variable A formatted string representing the ticker ID of a FINRA Short Sale Volume dataset. var string finraTickerID = str.format("FINRA:{0}_SHORT_VOLUME", noPrefix) // Log the `finraTickerID` in the Pine Logs pane on the first bar. if barstate.isfirst log.info(finraTickerID) // Retrieve the data. request.security(finraTickerID, timeframe.isintraday ? "1D" : "", close) // Plot the `requestedData` plot(requestedData, "Short Sale Volume", color.teal, 1, plot.style_columns)

Note that:

- The caret (^) at the beginning of the regex string matches the *beginning* of the `symbolInput`.
- The `(?:...)` syntax in the regex string creates a *non-capturing group*.
- The pipe character (`|`) in the regex acts as an *OR operator*. The group matches only *one* of the character sequences separated by the character (`BATS`, `NASDAQ`, `NYSE`, or `AMEX`).
- The `(?<=...)` syntax defines a *lookbehind assertion*, which checks if the specified pattern precedes the match.
- The `.` (period) character has a special meaning in regex strings. It matches *any* character, excluding line terminators by default.
- The + (plus sign) character is a *quantifier*. It specifies that the previous token (`.`) must match *one or more* times.

The flexibility of regular expressions also allows str.match() to perform advanced matching tasks that are impractical or infeasible with other `str.*()` functions.

For instance, suppose we want to create a script that executes dynamic requests for a list of symbols specified in a text area input, and we require a specific input format consisting of only valid ticker patterns, comma

separators with optional space characters, and no empty items. This validation is difficult to achieve with the other `str.*()` functions because they rely on *literal* character sequences. However, with str.match(), we can define a single regular expression that matches the input only if it meets our required formatting criteria.

The script below demonstrates a single str.match() function call that validates the format of an input list of symbols. The user-defined `processList()` function combines strings to form the following regex for matching the `list` value:

```
^ *(?:(?:\w+:)?\w+(?:\.\w+){0,2}!? *, *)*(?:\w+:)?\w+(?:\.\w+){0,2}!?
```

If the str.match() call returns a *non-empty* string, meaning the constructed pattern matches the `list` argument, the `processList()` function uses str.replace_all() to remove all space characters, then calls str.split() to split the string based on its commas to create an array of symbol substrings. Otherwise, it raises a runtime error with the runtime.error() function.

The script loops through the returned array of substrings to request data for each specified symbol and populate a table with the results:

image

//@version=6 indicator("Processing inputs with regex demo") //@variable A string containing a list of symbols separated by commas and optional spaces. string listInput = input.text_area("AMEX:SPY, AMEX:GLD, NASDAQ:TLT, CME:BTC1!, NYMEX:CL1!, TVC:US10Y", "Symbol list") //@function Checks a string value for a pattern of symbols separated by commas and optional space characters. // If the specified `list` does not match this format, the function raises a runtime error. // Otherwise, it returns an array of substrings representing each listed symbol. processList(string list) => //@variable A pattern for an optional group of one or more word characters (`\w` class) followed by a colon (`:`). // This pattern matches a symbol's exchange prefix, if listed // (e.g., "NASDAQ:" for "NASDAQ:AAPL" or "" for "AAPL"). // - `(?:...)` defines a non-capturing group. // - `\\w` is an ASCII digit, letter, or low line. // - The `+` means the regex checks for one or more consecutive matches of the previous token (`\w`). // - The `?` at the end makes the group optional. var string exchange = "(?:\\w+:)?" //@variable A pattern for 1+ word characters, with 0-2 extra word sequences divided by `.`, then an optional `!

`. // This pattern matches symbols without the exchange prefix // (e.g., "AAPL" for "NASDAQ:AAPL", "BTC.D" for "CRYPTOCAP:BTC.D", "ES1!" for "CME_MINI:ES1!"). // - `\\w+` means one or more word characters. // - `\\.` makes the `.` character literal. When not escaped, it matches any character. // - `{0,2}` means the regex matches the group's pattern (`\.\w+`) zero to two times. // - The `?` makes the `!` character optional. var string symbol = "\\w+(?:\\.\\w+){0,2}!?" //@variable A pattern that matches a symbol or comma-separated list of symbols with optional spaces. // (e.g., "AAPL, OANDA:EURUSD, BATS:SPY, BINANCE:BTCUSDT.P") // - `^` at the beginning matches the start of the text line. // - `*` after the spaces and group construction `(?:...)` mean the regex matches them zero or more times. // - `$` at the end matches the end of the text line. // The formatted result combines `exchange` and `symbol` to form this regex pattern: // `^ *(?:(?:\w+:)?\w+(?:\.\w+){0,2}!? *, *)*(?:\w+:)?\w+(?:\.\w+){0,2}!? *$` var string matchPattern = str.format("^ *(?:{0}{1} *, *)*{0}{1} *$", exchange, symbol) //@variable A copy of the `list` if the `matchPattern` produces a match. Otherwise, an empty string. string match = str.match(list, matchPattern) // If the `match` is empty, meaning the `list` does not have the required format, raise a runtime error. if match == "" runtime.error("Invalid list. The value must represent a comma-separated list of symbols with optional spaces.") // Log an `info` message showing the pattern and the match. log.info("\n\nThe pattern:\n\n{0}\n\nmatches:\n\n{1}", matchPattern, match) //@variable A copy of the `match` without space characters. string noSpaces = str.replace_all(match, " ", "") //@variable An array of substrings formed by splitting the `noSpaces` value by its commas. array<string> result = str.split(noSpaces, ",") //@variable An array of symbols from the processed `listInput`. var array<string> symbols = processList(listInput) if barstate.islast //@variable A two-column table with a row for each `symbols` item. var table display = table.new(position.middle_center, 2, symbols.size()) // Loop through the `symbols` array. for [i, symbol] in symbols //@variable The `close` value requested for the `symbol` on the chart's timeframe. float requestedValue = request.security(symbol, timeframe.period, close) // Initialize a cell for the `symbol` and a string representing the `requestedValue`. display.cell(0, i, symbol, text_color = chart.fg_color, text_size = 24) display.cell(1, i, str.tostring(requestedValue), text_color = chart.fg_color, text_size = 24)

Note that:

- When creating regex strings, it is often helpful to display them in a script's text outputs to ensure they are formatted as intended. In this script, we included a log.info() call to show the resulting regular expression and its match in the Pine Logs pane.
- Because the backslash (\) is an escape character in Pine strings, the value used as the regex argument in a str.match() call requires **two** consecutive backslashes for each single backslash in the regular expression.
- The `\\w` parts of the regex string specify the `\w` pattern, a *predefined character class* that matches *word characters* (letters, digits, or low lines).
- The * (asterisk) and `{0,2}` parts of the regular expression are *quantifiers*, similar to +. The asterisk requires the previous token to match *zero or more* times. The `{0,2}` quantifier requires the match to occur exactly *zero to two* times.
- The $ (dollar sign) character in this regular expression matches the *end* of the input, excluding any final line terminators (`\n`).
- Because the `.` (period) character has a special meaning in regex strings, we must prefix it with two backslashes (`\\`) in the string to match a literal period.

Regex syntax reference

Every programming language's regex engine has unique characteristics and syntax. Some regex syntax is universal across engines, while other patterns and modifiers are engine-specific.

The tables below provide a categorized overview of the syntax patterns supported by Pine's regex engine along with descriptions, remarks, and examples to explain how they work.

Escapes and character references

Click to show/hide

Token/ syntax	Description and remarks
\	

Token/ syntax	Description and remarks
	Changes the meaning of the next character. Because Pine strings natively use \ as an escape character, regex strings containing it must include an *additional* \ to use the token in the pattern. For example, "\\" represents a single \ (escape token) in the regex, and "\\\\" represents \\ (literal backslash). Some other characters always or conditionally represent regex syntax, including `.`, `^`, `$`, `*`, `+`, `?`, `(`, `)`, `[`, `]`, `{`, `}`, `|`, and `-`. To match a special character literally, include "\\" immediately before it in the string (e.g., "\\+" matches the literal + character). Note that some sequences of otherwise literal characters can also have syntactical meaning. See below for examples.
`\Q...\E`	Matches everything between `\Q` and `\E` *literally*, ignoring the syntactical meaning of special characters and sequences. For example, the regex string "\\Q[^abc]\\E" matches the literal sequence of `[`, `^`, `a`, `b`, `c`, and `]` characters instead of creating a *character class*.
a	Matches the literal character a (U+0061). By default, the regex engine is *case-sensitive*. If the string includes the `(?i)` modifier *before* the token, the match becomes *case-insensitive*. For example, the regex string "(?i)a" matches the a or A character.
`\t`	Matches the *tab space* character (U+0009).
`\n`	Matches the *newline* character (U+000A).
`\x61`	A *two-digit* Unicode reference that matches the hexadecimal point U+0061 (the a character). This shorthand syntax works only for codes with leading zeros and up to **two** nonzero end digits. It cannot reference other Unicode points. For example, the regex string "\\x2014" matches U+0020 (the *space* character) followed by U+0031 (the 1 character) and

Token/ syntax	Description and remarks
	U+0034 (the 4 character). It **does not** match U+2014 (the – character).
	A *four-digit* Unicode reference that matches the hexadecimal point U+2014 (–, Em Dash).
`\u2014`	This syntax works only for codes with leading zeros and up to **four** nonzero end digits. It cannot reference larger Unicode points. For example, the regex string `"\\u1F5E0"` matches U+1F5E (unassigned) followed by U+0030 (the `0` character), resulting in no match. It **does not** match U+1F5E0 (the Stock Chart character).
	The *full-range* Unicode reference syntax. The hexadecimal digits enclosed in the brackets can refer to *any* Unicode point.
`\x{...}`	Leading zeros in the digits *do not* affect the matched Unicode point. For example, the regex strings `"\\x{61}"`, `"\\x{061}"`, `"\\x{0061}"`, and `"\\x{000061}"` all match U+0061 (the a character).

Character class and logical constructions

Click to show/hide

Token/ syntax	Description and remarks
	A character class that matches only *one* of the characters listed (a, b, or c). It does **not** match the *entire* abc sequence.
`[abc]`	Each listed character, range, or nested class between two `[]` brackets represents a specific possible match. Note that several special characters have a *literal* meaning inside classes (e.g., the regex string `"[.+$]"` matches `.`, `+`, or `$` literally). However, regex strings should still escape the following characters to treat them literally because they maintain a special meaning in most cases: `\`, `[`, `]`, `^`, `-`.
`[a-z]`	

Token/ syntax	Description and remarks
	A class that matches a single character in the *range* from a (U+0061) to z (U+007A). It is equivalent to `[\x{61}-\x{7A}]`. Note that the left side of the `-` character must have a *smaller* Unicode value than the right. For example, the regex string `"[f-a]"` is *invalid* because `f` has the Unicode value U+0066, which is *larger* than the value of a. If the dash (`-`) is at the start or end of the enclosed text, the regex treats it as a *literal* character instead of a character range marker (e.g., `"[-abc]"` matches `-`, a, b, or c literally).
`[a-zA-Z]`	A class containing a *list* of character *ranges*. It matches any character from a (U+0061) to z (U+007A) or A (U+0041) to Z (U+005A) only. It is equivalent to `[\x{61}-\x{7A}\x{41}-\x{5A}]`. The syntax `[a-z[A-Z]]` also produces the same match.
`[^...]`	The syntax for a class that matches any character *except* for the ones specified. For example, the regex string `"[^abc\\n ]"` matches any character except for a, b, c, `\n` (newline), or (space). Note that only a caret (^) at the *start* of the enclosed text signifies *negation*. If the character comes after that point, the regex considers it a possible *literal* match (e.g., `"[ab^c]"` matches the a, b, ^, or c character literally).
`[...&&[...]]`	The syntax for a nested class structure that matches any character within the *intersection* of two character classes. Example 1: The regex string `"[abc&&[cde]]"` matches c exclusively because it is the only character common to both lists.

Token/ syntax	Description and remarks
	Example 2: The regex string `"[a-z&&[^def]]"` matches any character from lowercase a to `z` except for `d`, `e`, or `f`.
`expr1|expr2`	An OR operation that matches either the `expr1` or `expr2` substring. It does *not* include both in the match.

Predefined classes

Click to show/hide

Token/ syntax	Description and remarks
`.`	Matches any character on the line. By default, it *excludes* line terminators (e.g., `\n`). To include line terminators in the match, add the `(?s)` modifier *before* the token in the regex string (e.g., `"(?s)."`).
`\d`	Matches a decimal digit character. By default, it is equivalent to `[0-9]`. However, if the regex string includes the `"(?U)"` modifier before the token, it can match *other* Unicode characters with the “Digit” property. For example, the string `"(?U)\\d"` can match characters such as U+FF11 (Fullwidth Digit One). In contrast, the only “Digit One” character matched by the `[0-9]` class, even with the `(?U)` modifier, is U+0031 (the 1 character).
`\D`	Matches a *non-digit* character. By default, it is equivalent to `[^0-9]`, which does *not* negate other Unicode digits. To exclude other Unicode digits from the match, include the `(?U)` modifier before the token in the regex string (e.g., `"(?U)\\D"`).
`\w`	Matches a *word character* (letter, digit, or low line). By default, it is equivalent to `[a-zA-Z0-9_]`, which excludes other

Token/ syntax	Description and remarks
	Unicode characters. To include other Unicode letters, digits, or low lines in the match, add the `(?U)` modifier before the token in the regex string. For example, `"(?U)\\w"` can match characters such as U+FE4F (Wavy Low Line), whereas the only low line character the `[a-zA-Z0-9_]` class matches is U+005F (_).
`\W`	Matches a *non-word* character. By default, it is equivalent to `[^a-zA-Z0-9_]`, which does not negate other Unicode characters. To exclude other Unicode word characters from the match, include the `(?U)` modifier before the token (e.g., `"(?U)\\W"`).
`\h`	Matches a *horizontal whitespace* character, such as the tab space (`\t`), standard space, and other characters such as U+2003 (Em Space). The token matches other Unicode characters, even if the regex string includes the `(?-U)` modifier.
`\H`	Matches a character that is *not* a horizontal whitespace. It also excludes other Unicode spaces, even if the regex string includes the `(?-U)` modifier
`\s`	Matches a *whitespace* or other *control character*. In contrast to `\h`, this token covers a broader range of characters, including *vertical* spaces such as `\n`.
`\S`	Matches a *non-whitespace* character. In contrast to `\H`, this token excludes a broader character range of characters from the match.

Unicode property classes

Click to show/hide

Token/syntax	Description and remarks
`\p{...}`	The syntax to match a Unicode point that has a specific property, such as script type, block, general category, etc. See the following rows to learn the required syntax for different common

Token/syntax	Description and remarks
	Unicode property references. To match any character that does *not* have a specific Unicode property, use the *uppercase* P in the syntax (`\P{...}`)
`\p{IsScriptName}` or `\p{Script=ScriptName}`	Unicode script reference syntax. Matches any code point belonging to the `ScriptName` Unicode script. The specified name should not contain spaces. For example, the regex strings `"\\p{IsLatin}"` and `"\\p{Script=Latin}"` both match any Unicode point that is part of the Latin script.
`\p{InBlockName}` or `\p{Block=BlockName}`	Unicode block reference syntax. Matches any code point belonging to the `BlockName` Unicode block. The specified name should not contain spaces. For example, the regex string `"\\p{InBasicLatin}"` matches any Unicode point that is part of the Basic Latin block, and `"\\p{Block=Latin-1Supplement}"` matches any point belonging to the Latin-1 Supplement block.
`\p{category}` or `\p{gc=category}`	Unicode general category reference syntax. Matches any Unicode point with the assigned `category` *abbreviation*. For example, the regex string `"\\p{L}"` or `"\\p{gc=L}"` matches any Unicode point in the *Letter (L)* category, and `"\\p{N}"` matches any point in the *Number (N)* category. Note that, unlike some regex engines, Pine's regex engine does not support the *long form* of a category name, (e.g., `"Letter"` instead of `"L"`).
`\p{ClassName}`	The syntax for referencing the Unicode mapping of a POSIX character class, in Java notation.

Token/syntax	Description and remarks
	For example, the regex string `"\\p{XDigit}"` matches a *hexadecimal* digit. By default, it is equivalent to `"[A-Fa-f0-9]"`. Note that the default behavior for POSIX classes matches only *ASCII* characters. To allow other Unicode matches for a POSIX class, use the `(?U)` modifier. For instance, `"(?U)\\p{XDigit}"` can match non-ASCII characters that represent hexadecimal digits, such as U+1D7D9 (the 𝟙 character).

Group constructions

Click to show/hide

Token/ syntax	Description and remarks
`(...)`	A *capturing group* that matches the enclosed sequence and stores the matched substring for later reference. Each capturing group construction has an assigned *group number* starting from 1. The regex can reference a capturing group's match with the \# syntax, where # represents the group number. For example, the regex string `"(a|b)cde\\1"` matches a or b, followed by cde, and then another occurrence of the `(a|b)` group's initial match. If the group matched a, the \1 reference also matches a. If it matched b, the reference also matches b. If the regex does not need to use a group's match later, using a *non-capturing* group is the more efficient choice, e.g., `(?:...)`.
`(?<name>...)`	A *named* capturing group that matches the enclosed sequence and stores the matched substring with an assigned *identifier*. The regex can use the \k<name> syntax to reference the

Token/ syntax	Description and remarks
	group's match, where `name` is the assigned *identifier*. For example, the string `"(?<myGroup>a|b)cde\\k<myGroup>"` matches `a` or `b`, followed by `cde`, and then another instance of the substring (`a` or `b`) matched by the capturing group. As with a standard capturing group, a named capturing group contributes to the group count and has a *group number*, meaning the regex can also reference a named group with the `\#` syntax, for example, `"(?<myGroup>a|b)cde\\1"`.
`(?:...)`	A *non-capturing* group that matches the enclosed sequence *without* storing the matched substring. Unlike a capturing group, the regex string *cannot* reference a previous non-capturing group's match. For example, the regex string `"(?:a|b)\\1"` matches `a` or `b`, then references an *unassigned* group match, resulting in *no match*. In contrast to all other group constructions, standard non-capturing groups can contain *pattern modifiers* that apply exclusively to their scopes. For example, `"(?i)(?-i:a|b)c"` matches `a` or `b` followed by lowercase `c` or uppercase `C`. The `(?i)` part of the regex activates case-insensitive matching globally, but the `-i` token *deactivates* the behavior for the group's scope only. Note that non-capturing groups typically have a *lower* computational cost than capturing groups.
`(?>...)`	An *independent* non-capturing group (*atomic group*). Unlike a standard non-capturing group, an atomic group consumes as many characters as possible *without* allowing other parts of the pattern to use them. For example, the regex string `"(?s)(?>.+).+"` fails to produce a match because the atomic group `(?>.+)` consumes *every*

Token/ syntax	Description and remarks
	available character, leaving *nothing* for the following `.+` portion to match. In contrast, the regex string `"(?s)(?:.+).+"` matches the entire source string because the standard non-capturing group `(?:.+)` *releases* characters from its match as needed, allowing `.+` to match *at least one* character.

Quantifiers

Click to show/hide

Token/ syntax	Description and remarks
?	Appending ? to a character, group, or class specifies that the matched substring must contain the pattern *once* or *not at all*. For example, the regex string `"a?bc?"` matches abc, ab, bc, or b because a and c are *optional*. By default, regex quantifiers are *greedy*, meaning they match as many characters as possible, releasing some as necessary. Adding ? to another quantifier makes it *lazy*, meaning it matches the *fewest* characters possible, expanding its match only when required. For example, with a source string of `"a12b34b"`, the regex string `"a.*b"` matches the entire sequence, whereas `"a.*?b"` matches the *smallest* valid substring with the pattern, which is a12b.
*	Appending * to a character, group, or class specifies that the matched substring must contain the pattern *zero* or *more* times consecutively. For example, the regex string `"a*b"` matches zero or more consecutive a characters followed by a single b character.
+	

Token/ syntax	Description and remarks
	Appending + to a character, group, or class specifies that the matched substring must contain the pattern *one* or *more* times consecutively. For example, the regex string "\\w+abc" matches one or more consecutive word characters followed by abc. Adding + to another quantifier makes it *possessive*. Unlike a greedy quantifier (default), which *releases* characters from the match as necessary, a possessive quantifier consumes as many characters as possible *without* releasing them for use in other parts of the pattern. For instance, the regex string "\\w++abc" fails to produce a match because \w++ consumes *all* word characters in the pattern, including a, b, and c, leaving none for the abc portion to match.
{n}	Appending {n} to a character, group, or class specifies that the matched substring must contain the pattern exactly n times consecutively, where n >= 0. For example, the regex string "[abc]{2}" matches two consecutive characters from the [abc] class, meaning the possible substrings are aa, ab, ac, ba, bb, bc, ca, cb, or cc.
{n,}	Appending {n,} to a character, group, or class specifies that the matched substring must contain the pattern *at least* n times consecutively, where n >= 0. For example, the regex string "a{1,}b{2,}" matches one or more consecutive a characters followed by two or more consecutive b characters.
{n, m}	Appending {n, m} to a character, group, or class specifies that the matched substring must contain the pattern at least n times but no more than m times, where n >= 0, m >= 0, and m >= n. For example, the regex string "\\w{1,5}b{2,4}" matches one to

Token/ syntax	Description and remarks
	five consecutive word characters followed by two to four repeated `b` characters.

Boundary assertions

Click to show/hide

Token/ syntax	Description and remarks
`\A`	Matches the *starting point* of the source string without consuming characters. It enables the regex to isolate the initial pattern in a string without allowing matches in other locations. For example, the regex string `"\\A\\w+"` matches a sequence of one or more word characters only if the sequence is at the start of the source string.
`^`	When this character is outside a character class construction (i.e., `[^...]`), it matches the *starting point* of a *line* in the source string without consuming characters. By default, the character performs the same match as `\A`. However, if the regex string uses the `(?m)` modifier, it can also match a point immediately after a *newline* character (`\n`). For example, the regex string `"(?m)^[xyz]"` matches `x`, `y`, or `z` if the character is at the start of the source string or immediately after the `\n` character.
`\Z`	Matches the *ending point* of the source string, or the point immediately before the final character if it is `\n`, without consuming characters. It enables the regex to isolate the final pattern in a string without allowing matches in other locations. For example, the regex string `"\\w+\\Z"` matches a sequence of one or more word characters only if the sequence is at the end of the source string or immediately before the final line terminator.
`\z`	

Token/ syntax	Description and remarks
	Matches the *absolute* ending point of the source string without consuming characters. Unlike \Z (uppercase), this token does not match the point *before* any final line terminator. For example, the regex string "(?s)\\w+.*\\z" matches a sequence of one or more word characters, followed by zero or more extra characters, only if the sequence is at the absolute end of the source string.
$	Matches the *ending point* of a *line* in the source string without consuming characters. By default, it performs the same match as \Z (uppercase). However, if the regex string uses the (?m) modifier, it can match any point immediately before a newline (\n) character. For example, the regex string "(?m)[123]$" matches 1, 2, or 3 only if the character is at the end of the source string or immediately before the \n character.
\b	Matches a *word boundary*, which is the point immediately before or after a sequence of word characters (members of the \w class). For example, the regex string "\\babc" matches abc only if it is at the starting point of a word character sequence.
\B	Matches a *non-word* boundary, which is any point between characters that is *not* the start or end of a word character sequence. For example, the regex string "\\Babc" matches abc only if it is not at the start of a word character sequence.

Lookahead and lookbehind assertions

Click to show/hide

Token/ syntax	Description and remarks
(?=...)	

Token/ syntax	Description and remarks
`(?!...)`	A *positive lookahead* assertion that checks whether the specified sequence immediately *follows* the current match location, without consuming characters. For example, the regex string `"a(?=b)"` matches the a character only if b occurs immediately after that point, and it does not include b in the matched substring. A *negative lookahead* assertion that checks whether the specified sequence *does not* immediately follow the current match location, without consuming characters. For example, the regex string `"a(?!b)"` matches the a character only if the b character does not immediately follow it.
`(?<=...)`	A *positive lookbehind* assertion that checks whether the specified sequence immediately *precedes* the current match location, without consuming characters. For example, the regex string `"(?<=a)b"` matches the b character only if the a character occurs immediately before that point, and it does not include a in the matched substring.
`(?<!...)`	A *negative lookbehind* assertion that checks whether the specified sequence *does not* immediately precede the current match location, without consuming characters. For example, the regex string `"(?<!a)b"` matches the b character only if the a character does not immediately precede it.

Pattern modifiers

Click to show/hide

Token/ syntax	Description and remarks
`(?...)`	This syntax applies a global list of inline *pattern modifiers* (flags) to the regex string. Pattern modifiers change the matching behaviors of the regex engine. All parts of the regex string that come after

Token/ syntax	Description and remarks
	this syntax update their behaviors based on the specified modifiers, and those behaviors persist from that point until explicitly overridden. For example, `"(?mi)"` activates *multiline* and *case-insensitive* modes for the rest of the regex string. To deactivate modifiers, include the `-` character before the list of modifier tokens. For instance, `"(?-mi)"` deactivates multiline and case-insensitive modes for the rest of the regex string. Standard *non-capturing groups* can also utilize modifiers *locally*, allowing different behaviors exclusively within group constructions. For example, `"(?U:\\d)123"` activates *Unicode-aware* matching only for the specific group. The modifier does not apply globally, meaning the remaining 123 part of the regex string can only match *ASCII* characters. See the rows below for details about the most common, useful pattern modifiers for Pine regex strings.
`i`	The `i` character represents *case-insensitive* mode when used as a global modifier (`(?i)`) or group modifier (`(?i:...)`). For example, the regex string `"a(?i)b(?-i)c"` matches lowercase `a`, uppercase `B` or lowercase `b`, and then lowercase `c`. Note that case-insensitive mode only applies to ASCII characters unless Unicode-aware mode is active.
`m`	The `m` character represents multiline mode when used as a global modifier (`(?m)`) or group modifier (`(?m:...)`). By default, the `^` and `$` boundary assertions match the start and end of the source string, excluding final line terminators. With multiline mode enabled, they match the start and end boundaries

Token/ syntax	Description and remarks
	of any *separate line* in the string. For example, the regex string `"^abc"` matches `abc` only if the source string starts with that sequence, whereas `"(?m)^abc"` matches `abc` if it is at the start of the string or immediately follows a *newline character* (`\n`).
s	The lowercase `s` character represents *single-line mode* (*dotall mode*) when used as a global modifier (`(?s)`) or group modifier (`(?s:...)`). By default, the `.` character matches any character except for line terminators such as `\n`. With single-line mode enabled, the regex treats the source string as *one line*, allowing the character to match line terminators. For example, using the regex string `".+"` on the source string `"ab\nc"` matches `ab` only, whereas `"(?m).+"` matches the *entire* source string.
U	The uppercase `U` character represents *Unicode-aware* mode when used as a global modifier (`(?U)`) or group modifier (`(?U:...)`). By default, most of the regex engine's predefined character classes and mapped POSIX classes do not match *non-ASCII* characters. With Unicode-aware mode enabled, the regex allows these classes, and various ASCII character tokens, to match related Unicode characters. For example, the regex string `"\\d(?U)\\d+"` matches a single ASCII digit followed by one or more Unicode digit characters.
x	The lowercase `x` character represents *verbose mode* (*comments mode*) when used as a global modifier (`(?x)`) or group modifier (`(?x:...)`). In this mode, the regex string ignores *whitespace* characters and treats sequences starting with # as *comments*.

Token/ syntax	Description and remarks
	For example, the regex string `"(?x)[a-f ] 1 2\n3 # this is a comment!"` produces the same match as `"[a-f]123"`. It does **not** match the space or newline characters, or anything starting from the # character. Regex strings with this modifier can include multiple comments on *separate lines* (e.g., `"a #match 'a' \nb #followed by 'b'"` matches ab). To match whitespaces or the # character in this mode, *escape* them using backslashes or the `\Q...\E` syntax. For instance, `"(?x)\\#\\ \\# #comment"` and `"(?x)\\Q# #\\E #comment"` both literally match the sequence # #.

Previous

Next

Tables

Pine Script® v6 Documentation

Tables

Introduction

Tables are objects that can be used to position information in specific and fixed locations in a script's visual space. Contrary to all other plots or objects drawn in Pine Script®, tables are not anchored to specific bars; they *float* in a script's space, whether in overlay or pane mode, in studies or strategies, independently of the chart bars being viewed or the zoom factor used.

Tables contain cells arranged in columns and rows, much like a spreadsheet. They are created and populated in two distincts steps:

1. A table's structure and key attributes are defined using table.new(), which returns a table ID that acts like a pointer to the table, just like label, line, or array IDs do. The table.new() call will create the table object but does not display it.
2. Once created, and for it to display, the table must be populated using one table.cell() call for each cell. Table cells can contain text, or not. This second step is when the width and height of cells are defined.

Most attributes of a previously created table can be changed using `table.set_*()` setter functions. Attributes of previously populated cells can be modified using `table.cell_set_*()` functions.

A table is positioned in an indicator's space by anchoring it to one of nine references: the four corners or midpoints, including the center. Tables are positioned by expanding the table from its anchor, so a table anchored to the position.middle_right reference will be drawn by expanding up, down and left from that anchor.

Two modes are available to determine the width/height of table cells:

- A default automatic mode calculates the width/height of cells in a column/row using the widest/highest text in them.
- An explicit mode allows programmers to define the width/height of cells using a percentage of the indicator's available x/y space.

Displayed table contents always represent the last state of the table, as it was drawn on the script's last execution, on the dataset's last bar. Contrary to values displayed in the Data Window or in indicator values, variable contents displayed in tables will thus not change as a script user moves his cursor over specific chart bars. For this reason, it is strongly recommended to always restrict execution of all `table.*()` calls to either the first or last bars of the dataset. Accordingly:

- Use the var keyword to declare tables.
- Enclose all other calls inside an if barstate.islast block.

Multiple tables can be used in one script, as long as they are each anchored to a different position. Each table object is identified by its own ID. Limits on the quantity of cells in all tables are determined by the total number of cells used in one script.

Creating tables

When creating a table using table.new(), three parameters are mandatory: the table's position and its number of columns and rows. Five other parameters are optional: the table's background color, the color and width of the table's outer frame, and the color and width of the borders around all cells, excluding the outer frame. All table attributes except its number of columns and rows can be modified using setter functions: table.set_position(), table.set_bgcolor(), table.set_frame_color(), table.set_frame_width(), table.set_border_color() and table.set_border_width().

Tables can be deleted using table.delete(), and their content can be selectively removed using table.clear().

When populating cells using table.cell(), you must supply an argument for four mandatory parameters: the table id the cell belongs to, its column and

row index using indices that start at zero, and the text string the cell contains, which can be null. Other parameters are optional: the width and height of the cell, the text's attributes (color, horizontal and vertical alignment, size, formatting), and the cell's background color. All cell attributes can be modified using setter functions: table.cell_set_text(), table.cell_set_width(), table.cell_set_height(), table.cell_set_text_color(), table.cell_set_text_halign(), table.cell_set_text_valign(), table.cell_set_text_size(), table.cell_set_text_formatting(), and table.cell_set_bgcolor().

Keep in mind that each successive call to table.cell() redefines **all** the cell's properties, deleting any properties set by previous table.cell() calls on the same cell.

Placing a single value in a fixed position

Let's create our first table, which will place the value of ATR in the upper-right corner of the chart. We first create a one-cell table, then populate that cell:

//@version=6 indicator("ATR", "", true) // We use `var` to only initialize the table on the first bar. var table atrDisplay = table.new(position.top_right, 1, 1) // We call `ta.atr()` outside the `if` block so it executes on each bar. myAtr = ta.atr(14) if barstate.islast // We only populate the table on the last bar. table.cell(atrDisplay, 0, 0, str.tostring(myAtr))

image

Note that:

- We use the var keyword when creating the table with table.new().
- We populate the cell inside an if barstate.islast block using table.cell().
- When populating the cell, we do not specify the `width` or `height`. The width and height of our cell will thus adjust automatically to the text it contains.
- We call `ta.atr(14)` prior to entry in our if block so that it evaluates on each bar. Had we used `str.tostring(ta.atr(14))` inside the if block, the function would not have evaluated correctly because it would be called on the dataset's last bar without having calculated the necessary values from the previous bars.

Let’s improve the usability and aesthethics of our script:

//@version=6 indicator("ATR", "", true) atrPeriodInput = input.int(14, "ATR period", minval = 1, tooltip = "Using a period of 1 yields True Range.") var table atrDisplay = table.new(position.top_right, 1, 1, bgcolor = color.gray, frame_width = 2, frame_color = color.black) myAtr = ta.atr(atrPeriodInput) if barstate.islast table.cell(atrDisplay, 0, 0, str.tostring(myAtr, format.mintick), text_color = color.white)

image

Note that:

- We used table.new() to define a background color, a frame color and its width.
- When populating the cell with table.cell(), we set the text to display in white.
- We pass format.mintick as a second argument to the str.tostring() function to restrict the precision of ATR to the chart’s tick precision.
- We now use an input to allow the script user to specify the period of ATR. The input also includes a tooltip, which the user can see when he hovers over the “i” icon in the script’s “Settings/Inputs” tab.

Coloring the chart’s background

This example uses a one-cell table to color the chart’s background on the bull/bear state of RSI:

//@version=6 indicator("Chart background", "", true) bullColorInput = input.color(color.new(color.green, 95), "Bull", inline = "1") bearColorInput = input.color(color.new(color.red, 95), "Bear", inline = "1") // ——————— Function colors chart bg on RSI bull/bear state. colorChartBg(bullColor, bearColor) => var table bgTable = table.new(position.middle_center, 1, 1) float r = ta.rsi(close, 20) color bgColor = r > 50 ? bullColor : r < 50 ? bearColor : na if barstate.islast table.cell(bgTable, 0, 0, width = 100, height = 100, bgcolor = bgColor) colorChartBg(bullColorInput, bearColorInput)

Note that:

- We provide users with inputs allowing them to specify the bull/bear colors to use for the background, and send those input colors as arguments to our `colorChartBg()` function.
- We create a new table only once, using the var keyword to declare the table.
- We use table.cell() on the last bar only, to specify the cell's properties. We make the cell the width and height of the indicator's space, so it covers the whole chart.

Creating a display panel

Tables are ideal to create sophisticated display panels. Not only do they make it possible for display panels to always be visible in a constant position, they provide more flexible formatting because each cell's properties are controlled separately: background, text color, size and alignment, etc.

Here, we create a basic display panel showing a user-selected quantity of MAs values. We display their period in the first column, then their value with a green/red/gray background that varies with price's position with regards to each MA. When price is above/below the MA, the cell's background is colored with the bull/bear color. When the MA falls between the current bar's open and close, the cell's background is of the neutral color:

image

//@version=6 indicator("Price vs MA", "", true) var string GP1 = "Moving averages" int masQtyInput = input.int(20, "Quantity", minval = 1, maxval = 40, group = GP1, tooltip = "1-40") int masStartInput = input.int(20, "Periods begin at", minval = 2, maxval = 200, group = GP1, tooltip = "2-200") int masStepInput = input.int(20, "Periods increase by", minval = 1, maxval = 100, group = GP1, tooltip = "1-100") var string GP2 = "Display" string tableYposInput = input.string("top", "Panel position", inline = "11", options = ["top", "middle", "bottom"], group = GP2) string tableXposInput = input.string("right", "", inline = "11", options = ["left", "center", "right"], group = GP2) color bullColorInput = input.color(color.new(color.green, 30), "Bull", inline = "12", group = GP2) color bearColorInput = input.color(color.new(color.red, 30), "Bear", inline = "12", group = GP2) color

neutColorInput = input.color(color.new(color.gray, 30), "Neutral", inline = "12", group = GP2) var table panel = table.new(tableYposInput + "_" + tableXposInput, 2, masQtyInput + 1) if barstate.islast // Table header. table.cell(panel, 0, 0, "MA", bgcolor = neutColorInput) table.cell(panel, 1, 0, "Value", bgcolor = neutColorInput) int period = masStartInput for i = 1 to masQtyInput // ——————— Call MAs on each bar. float ma = ta.sma(close, period) // ——————— Only execute table code on last bar. if barstate.islast // Period in left column. table.cell(panel, 0, i, str.tostring(period), bgcolor = neutColorInput) // If MA is between the open and close, use neutral color. If close is lower/higher than MA, use bull/bear color. bgColor = close > ma ? open < ma ? neutColorInput : bullColorInput : open > ma ? neutColorInput : bearColorInput // MA value in right column. table.cell(panel, 1, i, str.tostring(ma, format.mintick), text_color = color.black, bgcolor = bgColor) period += masStepInput

Note that:

- Users can select the table's position from the inputs, as well as the bull/bear/neutral colors to be used for the background of the right column's cells.
- The table's quantity of rows is determined using the number of MAs the user chooses to display. We add one row for the column headers.
- Even though we populate the table cells on the last bar only, we need to execute the calls to ta.sma() on every bar so they produce the correct results. The compiler warning that appears when you compile the code can be safely ignored.
- We separate our inputs in two sections using `group`, and join the relevant ones on the same line using `inline`. We supply tooltips to document the limits of certain fields using `tooltip`.

Displaying a heatmap

Our next project is a heatmap, which will indicate the bull/bear relationship of the current price relative to its past values. To do so, we will use a table positioned at the bottom of the chart. We will display colors only, so our table will contain no text; we will simply color the background of its cells to produce our heatmap. The heatmap uses a user-selectable lookback period. It loops across that period to determine if price is above/below each bar in

that past, and displays a progressively lighter intensity of the bull/bear color as we go further in the past:

image

//@version=6 indicator("Price vs Past", "", true) var int MAX_LOOKBACK = 300 int lookBackInput = input.int(150, minval = 1, maxval = MAX_LOOKBACK, step = 10) color bullColorInput = input.color(#00FF00ff, "Bull", inline = "11") color bearColorInput = input.color(#FF0080ff, "Bear", inline = "11") // ——————— Function draws a heatmap showing the position of the current `_src` relative to its past `_lookBack` values. drawHeatmap(src, lookBack) => // float src : evaluated price series. // int lookBack: number of past bars evaluated. // Dependency: MAX_LOOKBACK // Force historical buffer to a sufficient size. max_bars_back(src, MAX_LOOKBACK) // Only run table code on last bar. if barstate.islast var heatmap = table.new(position.bottom_center, lookBack, 1) for i = 1 to lookBackInput float transp = 100. * i / lookBack if src > src[i] table.cell(heatmap, lookBack - i, 0, bgcolor = color.new(bullColorInput, transp)) else table.cell(heatmap, lookBack - i, 0, bgcolor = color.new(bearColorInput, transp)) drawHeatmap(high, lookBackInput)

Note that:

- We define a maximum lookback period as a `MAX_LOOKBACK` constant. This is an important value and we use it for two purposes: to specify the number of columns we will create in our one-row table, and to specify the lookback period required for the `_src` argument in our function, so that we force Pine Script to create a historical buffer size that will allow us to refer to the required quantity of past values of `_src` in our for loop.
- We offer users the possibility of configuring the bull/bear colors in the inputs and we use `inline` to place the color selections on the same line.
- Inside our function, we enclose our table-creation code in an if barstate.islast construct so that it only runs on the last bar of the chart.
- The initialization of the table is done inside the if statement. Because of that, and the fact that it uses the var keyword, initialization only occurs the first time the script executes on a last bar. Note that this behavior is different from the usual var declarations in the script's global scope, where initialization occurs on the first bar of the dataset, at bar_index zero.

- We do not specify an argument to the `text` parameter in our table.cell() calls, so an empty string is used.
- We calculate our transparency in such a way that the intensity of the colors decreases as we go further in history.
- We use dynamic color generation to create different transparencies of our base colors as needed.
- Contrary to other objects displayed in Pine scripts, this heatmap's cells are not linked to chart bars. The configured lookback period determines how many table cells the heatmap contains, and the heatmap will not change as the chart is panned horizontally, or scaled.
- The maximum number of cells that can be displayed in the scritp's visual space will depend on your viewing device's resolution and the portion of the display used by your chart. Higher resolution screens and wider windows will allow more table cells to be displayed.

Tips

- When creating tables in strategy scripts, keep in mind that unless the strategy uses `calc_on_every_tick = true`, table code enclosed in if barstate.islast blocks will not execute on each realtime update, so the table will not display as you expect.
- Keep in mind that successive calls to table.cell() overwrite the cell's properties specified by previous table.cell() calls. Use the setter functions to modify a cell's properties.
- Remember to control the execution of your table code wisely by restricting it to the necessary bars only. This saves server resources and your charts will display faster, so everybody wins.

Previous

Next

Text and shapes

Pine Script® v6 Documentation

Text and shapes

Introduction

Pine Script® features five different ways to display text or shapes on the chart:

- plotchar()
- plotshape()
- plotarrow()
- Labels created with label.new()
- Tables created with table.new() (see Tables)

Which one to use depends on your needs:

- Tables can display text in various relative positions on charts that will not move as users scroll of zoom the chart horizontally. Their content is not tethered to bars. In contrast, text displayed with plotchar(), plotshape() or label.new() is always tethered to a specific bar, so it will move with the bar's position on the chart. See the page on Tables for more information on them.
- Three function include are able to display pre-defined shapes: plotshape(), plotarrow() and Labels created with label.new().
- plotarrow() cannot display text, only up or down arrows.
- plotchar() and plotshape() can display non-dynamic text on any bar or all bars of the chart.
- plotchar() can only display one character while plotshape() can display strings, including line breaks.

- label.new() can display a maximum of 500 labels on the chart. Its text **can** contain dynamic text, or “series strings”. Line breaks are also supported in label text.
- While plotchar() and plotshape() can display text at a fixed offset in the past or the future, which cannot change during the script’s execution, each label.new() call can use a “series” offset that can be calculated on the fly.

These are a few things to keep in mind concerning Pine Script strings:

- Since the `text` parameter in both plotchar() and plotshape() require a “const string” argument, it cannot contain values such as prices that can only be known on the bar (“series string”).
- To include “series” values in text displayed using label.new(), they will first need to be converted to strings using str.tostring().
- The concatenation operator for strings in Pine is +. It is used to join string components into one string, e.g., `msg = "Chart symbol: " + syminfo.tickerid` (where syminfo.tickerid is a built-in variable that returns the chart’s exchange and symbol information in string format).
- Characters displayed by all these functions can be Unicode characters, which may include Unicode symbols. See this Exploring Unicode script to get an idea of what can be done with Unicode characters.
- Some functions have parameters that can specify the color, size, font family, and formatting of displayed text. For example, drawing objects like labels, tables, and boxes support text formatting such as bold, italics, and monospace.
- Pine scripts display strings using the system default font. The exact font may vary based on the user’s operating system.

This script displays text using the four methods available in Pine Script:

//@version=6 indicator("Four displays of text", overlay = true) plotchar(ta.rising(close, 5), "`plotchar()`", " ", location.belowbar, color.lime, size = size.small) plotshape(ta.falling(close, 5), "`plotchar()`", location = location.abovebar, color = na, text = "•`plotshape()•`\n ", textcolor = color.fuchsia, size = size.huge) if bar_index % 25 == 0 label.new(bar_index, na, "•LABEL•\nHigh = " + str.tostring(high, format.mintick) + "\n ", yloc = yloc.abovebar, style = label.style_none, textcolor = color.black, size = size.normal) printTable(txt) => var table t = table.new(position.middle_right,

1, 1), table.cell(t, 0, 0, txt, bgcolor = color.yellow) printTable("•TABLE•\n" + str.tostring(bar_index + 1) + " bars\nin the dataset")

image

Note that:

- The method used to display each text string is shown with the text, except for the lime up arrows displayed using plotchar(), as it can only display one character.
- Label and table calls can be inserted in conditional structures to control when their are executed, whereas plotchar() and plotshape() cannot. Their conditional plotting must be controlled using their first argument, which is a “series bool” whose `true` or `false` value determines when the text is displayed.
- Numeric values displayed in the table and labels is first converted to a string using str.tostring().
- We use the + operator to concatenate string components.
- plotshape() is designed to display a shape with accompanying text. Its `size` parameter controls the size of the shape, not of the text. We use na for its `color` argument so that the shape is not visible.
- Contrary to other texts, the table text will not move as you scroll or scale the chart.
- Some text strings contain the Unicode arrow (U+1F807).
- Some text strings contain the `\n` sequence that represents a new line.

plotchar()

This function is useful to display a single character on bars. It has the following syntax:

```
plotchar(series, title, char, location, color, offset, text, textcolor
```

See the Reference Manual entry for plotchar() for details on its parameters.

As explained in the Without affecting the scale section of our page on Debugging, the function can be used to display and inspect values in the Data Window or in the indicator values displayed to the right of the script’s name on the chart:

//@version=6 indicator("", "", true) plotchar(bar_index, "Bar index", "", location.top)

image

Note that:

- The cursor is on the chart's last bar.
- The value of bar_index on **that** bar is displayed in indicator values (1) and in the Data Window (2).
- We use location.top because the default location.abovebar will put the price into play in the script's scale, which will often interfere with other plots.

plotchar() also works well to identify specific points on the chart or to validate that conditions are `true` when we expect them to be. This example displays an up arrow under bars where close, high and volume have all been rising for two bars:

//@version=6 indicator("", "", true) bool longSignal = ta.rising(close, 2) and ta.rising(high, 2) and (na(volume) or ta.rising(volume, 2)) plotchar(longSignal, "Long", "▲", location.belowbar, color = na(volume) ? color.gray : color.blue, size = size.tiny)

image

Note that:

- We use `(na(volume) or ta.rising(volume, 2))` so our script will work on symbols without volume data. If we did not make provisions for when there is no volume data, which is what `na(volume)` does by being `true` when there is no volume, the `longSignal` variable's value would never be `true` because `ta.rising(volume, 2)` yields `false` in those cases.
- We display the arrow in gray when there is no volume, to remind us that all three base conditions are not being met.
- Because plotchar() is now displaying a character on the chart, we use `size = size.tiny` to control its size.
- We have adapted the `location` argument to display the character under bars.

If you don’t mind plotting only circles, you could also use plot() to achieve a similar effect:

//@version=6 indicator("", "", true) longSignal = ta.rising(close, 2) and ta.rising(high, 2) and (na(volume) or ta.rising(volume, 2)) plot(longSignal ? low - ta.tr : na, "Long", color.blue, 2, plot.style_circles)

This method has the inconvenience that, since there is no relative positioning mechanism with plot() one must shift the circles down using something like ta.tr (the bar’s “True Range”):

image

plotshape()

This function is useful to display pre-defined shapes and/or text on bars. It has the following syntax:

```
plotshape(series, title, style, location, color, offset, text, textcol
```

See the Reference Manual entry for plotshape() for details on its parameters.

Let’s use the function to achieve more or less the same result as with our second example of the previous section:

//@version=6 indicator("", "", true) longSignal = ta.rising(close, 2) and ta.rising(high, 2) and (na(volume) or ta.rising(volume, 2)) plotshape(longSignal, "Long", shape.arrowup, location.belowbar)

Note that here, rather than using an arrow character, we are using the `shape.arrowup` argument for the `style` parameter.

image

It is possible to use different plotshape() calls to superimpose text on bars. You need to use the newline character sequence, \n. The newline needs to be the **last** one in the string for text going up, and the **first** one when you are plotting under the bar and text is going down:

//@version=6 indicator("Lift text", "", true) plotshape(true, "", shape.arrowup, location.abovebar, color.green, text = "A") plotshape(true, "", shape.arrowup,

location.abovebar, color.lime, text = "B\n") plotshape(true, "", shape.arrowdown, location.belowbar, color.red, text = "C") plotshape(true, "", shape.arrowdown, location.belowbar, color.maroon, text = "\nD")

image

The available shapes you can use with the `style` parameter are:

Argument	Shape	With Text	Argument
`shape.xcross`	Plotshape_xcross	Xcross_with_text	`shape.arrowup`
`shape.cross`	Plotshape cross	Cross with text	`shape.arrowdown`
`shape.circle`	Plotshape circle	Circle with text	`shape.square`
`shape.triangleup`	Plotshape triangleup	Triangleup with text	`shape.diamond`
`shape.triangledown`	Plotshape_triangledown	Triangledown_with_text	`shape.labelup`
`shape.flag`	Plotshape_flag	Flag_with_text	`shape.labeldown`

plotarrow()

The plotarrow function displays up or down arrows of variable length, based on the relative value of the series used in the function's first argument. It has the following syntax:

```
plotarrow(series, title, colorup, colordown, offset, minheight, maxhei
```

See the Reference Manual entry for plotarrow() for details on its parameters.

The `series` parameter in plotarrow() is not a "series bool" as in plotchar() and plotshape(); it is a "series int/float" and there's more to it than a simple `true` or `false` value determining when the arrows are plotted. This is the logic governing how the argument supplied to `series` affects the behavior of plotarrow():

- `series > 0`: An up arrow is displayed, the length of which will be proportional to the relative value of the series on that bar in relation to other series values.
- `series < 0`: A down arrow is displayed, proportionally-sized using the same rules.
- `series == 0 or na(series)`: No arrow is displayed.

The maximum and minimum possible sizes for the arrows (in pixels) can be controlled using the `minheight` and `maxheight` parameters.

Here is a simple script illustrating how plotarrow() works:

//@version=6 indicator("", "", true) body = close - open plotarrow(body, colorup = color.teal, colordown = color.orange)

image

Note how the heigth of arrows is proportional to the relative size of the bar bodies.

You can use any series to plot the arrows. Here we use the value of the “Chaikin Oscillator” to control the location and size of the arrows:

//@version=6 indicator("Chaikin Oscillator Arrows", overlay = true) fastLengthInput = input.int(3, minval = 1) slowLengthInput = input.int(10, minval = 1) osc = ta.ema(ta.accdist, fastLengthInput) - ta.ema(ta.accdist, slowLengthInput) plotarrow(osc)

image

Note that we display the actual “Chaikin Oscillator” in a pane below the chart, so you can see what values are used to determine the position and size of the arrows.

Labels

Labels are only available in v4 and higher versions of Pine Script. They work very differently than plotchar() and plotshape().

Labels are objects, like lines and boxes, or tables. Like them, they are referred to using an ID, which acts like a pointer. Label IDs are of “label” type. As with other objects, labels IDs are “time series” and all the functions used to manage them accept “series” arguments, which makes them very flexible.

Labels are advantageous because:

- They allow “series” values to be converted to text and placed on charts. This means they are ideal to display values that cannot be known before

time, such as price values, support and resistance levels, of any other values that your script calculates.
- Their positioning options are more flexible that those of the `plot*()` functions.
- They offer more display modes.
- Contrary to `plot*()` functions, label-handling functions can be inserted in conditional or loop structures, making it easier to control their behavior.
- You can add tooltips to labels.

One drawback to using labels versus plotchar() and plotshape() is that you can only draw a limited quantity of them on the chart. The default is ~50, but you can use the `max_labels_count` parameter in your indicator() or strategy() declaration statement to specify up to 500. Labels, like lines and boxes, are managed using a garbage collection mechanism which deletes the oldest ones on the chart, such that only the most recently drawn labels are visible.

Your toolbox of built-ins to manage labels are all in the `label` namespace. They include:

- label.new() to create labels.
- `label.set_*()` functions to modify the properties of an existing label.
- `label.get_*()` functions to read the properties of an existing label.
- label.delete() to delete labels
- The label.all array which always contains the IDs of all the visible labels on the chart. The array's size will depend on the maximum label count for your script and how many of those you have drawn. `array.size(label.all)` will return the array's size.

Creating and modifying labels

The label.new() function creates a new label. It has the following signature:

```
label.new(x, y, text, xloc, yloc, color, style, textcolor, size, texta
```

The *setter* functions allowing you to change a label's properties are:

- label.set_x()
- label.set_y()

- label.set_xy()
- label.set_text()
- label.set_xloc()
- label.set_yloc()
- label.set_color()
- label.set_style()
- label.set_textcolor()
- label.set_size()
- label.set_textalign()
- label.set_tooltip()

They all have a similar signature. The one for label.set_color() is:

```
label.set_color(id, color) → void
```

where:

- `id` is the ID of the label whose property is to be modified.
- The next parameter is the property of the label to modify. It depends on the setter function used. label.set_xy() changes two properties, so it has two such parameters.

This is how you can create labels in their simplest form:

//@version=6 indicator("", "", true) label.new(bar_index, high)

image

Note that:

- The label is created with the parameters `x = bar_index` (the index of the current bar, bar_index) and `y = high` (the bar's high value).
- We do not supply an argument for the function's `text` parameter. Its default value being an empty string, no text is displayed.
- No logic controls our label.new() call, so labels are created on every bar.
- Only the last 54 labels are displayed because our indicator() call does not use the `max_labels_count` parameter to specify a value other than the ~50 default.
- Labels persist on bars until your script deletes them using label.delete(), or garbage collection removes them.

In the next example we display a label on the bar with the highest high value in the last 50 bars:

//@version=6 indicator("", "", true) // Find the highest `high` in last 50 bars and its offset. Change it's sign so it is positive. LOOKBACK = 50 hi = ta.highest(LOOKBACK) highestBarOffset = - ta.highestbars(LOOKBACK) // Create label on bar zero only. var lbl = label.new(na, na, "", color = color.orange, style = label.style_label_lower_left) // When a new high is found, move the label there and update its text and tooltip. if ta.change(hi) != 0 // Build label and tooltip strings. labelText = "High: " + str.tostring(hi, format.mintick) tooltipText = "Offest in bars: " + str.tostring(highestBarOffset) + "\nLow: " + str.tostring(low[highestBarOffset], format.mintick) // Update the label's position, text and tooltip. label.set_xy(lbl, bar_index[highestBarOffset], hi) label.set_text(lbl, labelText) label.set_tooltip(lbl, tooltipText)

image

Note that:

- We create the label on the first bar only by using the var keyword to declare the `lbl` variable that contains the label's ID. The `x`, `y` and `text` arguments in that label.new() call are irrelevant, as the label will be updated on further bars. We do, however, take care to use the `color` and `style` we want for the labels, so they don't need updating later.
- On every bar, we detect if a new high was found by testing for changes in the value of `hi`
- When a change in the high value occurs, we update our label with new information. To do this, we use three `label.set*()` calls to change the label's relevant information. We refer to our label using the `lbl` variable, which contains our label's ID. The script is thus maintaining the same label throughout all bars, but moving it and updating its information when a new high is detected.

Here we create a label on each bar, but we set its properties conditionally, depending on the bar's polarity:

//@version=6 indicator("", "", true) lbl = label.new(bar_index, na) if close >= open label.set_text(lbl, "green") label.set_color(lbl, color.green) label.set_yloc(lbl, yloc.belowbar) label.set_style(lbl, label.style_label_up) else

label.set_text(lbl, "red") label.set_color(lbl, color.red) label.set_yloc(lbl, yloc.abovebar) label.set_style(lbl, label.style_label_down)

image

Positioning labels

Labels are positioned on the chart according to *x* (bars) and *y* (price) coordinates. Five parameters affect this behavior: x, y, `xloc`, `yloc` and `style`:

x

Is either a bar index or a time value. When a bar index is used, the value can be offset in the past or in the future (up to a maximum of 500 bars in the future and 10,000 bars in the past). Past or future offsets can also be calculated when using time values. The x value of an existing label can be modified using label.set_x() or label.set_xy().

`xloc`

Is either xloc.bar_index (the default) or xloc.bar_time. It determines which type of argument must be used with x. With xloc.bar_index, x must be an absolute bar index. With xloc.bar_time, x must be a UNIX time in milliseconds corresponding to the time value of a bar's open. The `xloc` value of an existing label can be modified using label.set_xloc().

y

Is the price level where the label is positioned. It is only taken into account with the default `yloc` value of `yloc.price`. If `yloc` is yloc.abovebar or yloc.belowbar then the y argument is ignored. The y value of an existing label can be modified using label.set_y() or label.set_xy().

`yloc`

Can be yloc.price (the default), yloc.abovebar or yloc.belowbar. The argument used for y is only taken into account with yloc.price. The `yloc` value of an existing label can be modified using label.set_yloc().

`style`

The argument used has an impact on the visual appearance of the label and on its position relative to the reference point determined by either the y value or the top/bottom of the bar when yloc.abovebar or yloc.belowbar are used. The `style` of an existing label can be modified using label.set_style().

These are the available `style` arguments:

Argument	Label	Label with text	Argum
`label.style_xcross`	label_style_xcross	label_style_xcross_t	label.
`label.style_cross`	label style cross	label style cross t	label.
`label.style_flag`	label style flag	label style flag t	label.
`label.style_circle`	label style circle	label style circle t	label.
`label.style_square`	label_style_square	label_style_square_t	label.
`label.style_diamond`	label_style_diamond	label_style_diamond_t	label.
`label.style_triangleup`	label style triangleup	label style triangleup t	label.
`label.style_triangledown`	label style triangledown	label style triangledown t	label.
`label.style_arrowup`	label_style_arrowup	label_style_arrowup_t	label.
`label.style_arrowdown`	label_style_arrowdown	label_style_arrowdown_t	label.

When using xloc.bar_time, the x value must be a UNIX timestamp in milliseconds. See the page on Time for more information. The start time of the current bar can be obtained from the time built-in variable. The bar time of previous bars is `time[1]`, `time[2]` and so on. Time can also be set to an absolute value with the timestamp function. You may add or subtract periods of time to achieve relative time offset.

Let’s position a label one day ago from the date on the last bar:

//@version=6 indicator("") daysAgoInput = input.int(1, tooltip = "Use negative values to offset in the future") if barstate.islast MS_IN_ONE_DAY = 24 * 60 * 60 * 1000 oneDayAgo = time - (daysAgoInput * MS_IN_ONE_DAY) label.new(oneDayAgo, high, xloc = xloc.bar_time, style = label.style_label_right)

Note that because of varying time gaps and missing bars when markets are closed, the positioning of the label may not always be exact. Time offsets of the sort tend to be more reliable on 24x7 markets.

You can also offset using a bar index for the x value, e.g.:

label.new(bar_index + 10, high) label.new(bar_index - 10, high[10]) label.new(bar_index[10], high[10])

Reading label properties

The following *getter* functions are available for labels:

- label.get_x()
- label.get_y()
- label.get_text()

They all have a similar signature. The one for label.get_text() is:

```
label.get_text(id) → series string
```

where `id` is the label whose text is to be retrieved.

Cloning labels

The label.copy() function is used to clone labels. Its syntax is:

```
label.copy(id) → void
```

Deleting labels

The label.delete() function is used to delete labels. Its syntax is:

```
label.delete(id) → void
```

To keep only a user-defined quantity of labels on the chart, one could use code like this:

//@version=6 MAX_LABELS = 500 indicator("", max_labels_count = MAX_LABELS) qtyLabelsInput = input.int(5, "Labels to keep", minval = 0, maxval = MAX_LABELS) myRSI = ta.rsi(close, 20) if myRSI > ta.highest(myRSI, 20)[1] label.new(bar_index, myRSI, str.tostring(myRSI, "#.00"), style = label.style_none) if array.size(label.all) > qtyLabelsInput label.delete(array.get(label.all, 0)) plot(myRSI)

image

Note that:

- We define a `MAX_LABELS` constant to hold the maximum quantity of labels a script can accommodate. We use that value to set the `max_labels_count` parameter's value in our indicator() call, and also as the `maxval` value in our input.int() call to cap the user value.
- We create a new label when our RSI breaches its highest value of the last 20 bars. Note the offset of `[1]` we use in `if myRSI > ta.highest(myRSI, 20)[1]`. This is necessary. Without it, the value returned by ta.highest() would always include the current value of `myRSI`, so `myRSI` would never be higher than the function's return value.
- After that, we delete the oldest label in the label.all array that is automatically maintained by the Pine Script runtime and contains the ID of all the visible labels drawn by our script. We use the array.get() function to retrieve the array element at index zero (the oldest visible label ID). We then use label.delete() to delete the label linked with that ID.

Note that if one wants to position a label on the last bar only, it is unnecessary and inefficent to create and delete the label as the script executes on all bars, so that only the last label remains:

// INEFFICENT! //@version=6 indicator("", "", true) lbl = label.new(bar_index, high, str.tostring(high, format.mintick)) label.delete(lbl[1])

This is the efficient way to realize the same task:

//@version=6 indicator("", "", true) if barstate.islast // Create the label once, the first time the block executes on the last bar. var lbl = label.new(na, na) // On all iterations of the script on the last bar, update the label's information. label.set_xy(lbl, bar_index, high) label.set_text(lbl, str.tostring(high, format.mintick))

Realtime behavior

Labels are subject to both *commit* and *rollback* actions, which affect the behavior of a script when it executes in the realtime bar. See the Execution model page to learn more.

This script demonstrates the effect of rollback when running in the realtime bar:

//@version=6 indicator("", "", true) label.new(bar_index, high)

On realtime bars, label.new() creates a new label on every script update, but because of the rollback process, the label created on the previous update on the same bar is deleted. Only the last label created before the realtime bar's close will be committed, and thus persist.

Text formatting

Drawing objects like labels, tables, and boxes have text-related properties that allow users to customize how an object's text appears on the chart. Some common properties include the text color, size, font family, and typographic emphasis.

Programmers can set an object's text properties when initializing it using the label.new(), box.new(), or table.cell() parameters. Alternatively, they can use the corresponding setter functions, e.g., label.set_text_font_family(), table.cell_set_text_color(), box.set_text_halign(), etc.

All three drawing objects have a `text_formatting` parameter, which sets the typographic emphasis to display **bold**, *italicized*, or unformatted text. It accepts the constants text.format_bold, text.format_italic, or text.format_none (no special formatting; default value). It also accepts `text.format_bold + text.format_italic` to display text that is both ***bold and italicized***.

The `size` parameter in label.new() and the `text_size` parameter in box.new() and table.cell() specify the size of the text displayed in the drawn objects. The parameters accept both "string" `size.*` constants and "int" typographic sizes. A "string" `size.*` constant represents one of six fixed sizing options. An "int" size value can be any positive integer, allowing scripts to replicate the `size.*` values or use other customized sizing.

This table lists the `size.*` constants and their equivalent "int" sizes for tables, boxes, and labels:

“string” constant	”int” text_size in tables and boxes	”int” size in labels
size.auto	0	0
size.tiny	8	~7
size.small	10	~10
size.normal	14	12
size.large	20	18
size.huge	36	24

The example below creates a label and table on the last available bar. The label displays a string representation of the current close value. The single-cell table displays a string representing the price and percentage difference between the current close and open values. The label’s text size is defined by a string input that returns the value of a built-in size.* constant, and the table’s text size is defined by an integer input. Additionally, the script creates a box that visualizes the range from the highest to lowest price over the last 20 bars. The box displays custom text, with a constant text_size of 19, to show the distance from the close value to the current highest or lowest price. The two Boolean inputs specify whether all three drawings apply bold and italic text formats to their displayed text:

image

//@version=6 indicator("Text formatting demo", overlay = true) //@variable The size of the `closeLabel` text, set using "string" `size.*` constants. string closeLabelSize = input.string(size.large, "Label text size", [size.auto, size.tiny, size.small, size.normal, size.large, size.huge], group = "Text size") // @variable The size of the `barMoveTable` text, set using "int" sizes. int tableTextSize = input.int(25, "Table text size", minval = 0, group = "Text size") // Toggles for the text formatting of all the drawing objects (`label`, `table` cell, and `box` texts). bool formatBold = input.bool(false, "Bold emphasis", group = "Text formatting (all objects)") bool formatItalic = input.bool(true, "Italic emphasis", group = "Text formatting (all objects)") // Track the highest and lowest prices in 20 bars. Used to draw a `box` of the high-low range. float recentHighest = ta.highest(20) float recentLowest = ta.lowest(20) if barstate.islast //@variable Label displaying `close` price on last bar. Text size is set using "string" constants. label closeLabel =

label.new(bar_index, close, "Close price: " + str.tostring(close, "$0.00"), color = #EB9514D8, style = label.style_label_left, size = closeLabelSize) // Create a `table` cell to display the bar move (difference between `open` and `close` price). float barMove = close - open //@variable Single-cell table displaying the `barMove`. Cell text size is set using "int" values. var table barMoveTable = table.new(position.bottom_right, 1, 1, bgcolor = barMove > 0 ? #31E23FCC : #EE4040CC) barMoveTable.cell(0, 0, "Bar move = " + str.tostring(barMove, "$0.00") + "\n Percent = " + str.tostring(barMove / open, "0.00%"), text_halign = text.align_right, text_size = tableTextSize) // Draw a box to show where current price falls in the range of `recentHighest` to `recentLowest`. //@variable Box drawing the range from `recentHighest` to `recentLowest` in last 20 bars. Text size is set at 19. box rangeBox = box.new(bar_index - 20, recentHighest, bar_index + 1, recentLowest, text_size = 19, bgcolor = #A4B0F826, text_valign = text.align_top, text_color = #4A07E7D8) // Set box text to display how far current price is from the high or low of the range, depending on which is closer. rangeBox.set_text("Current price is " + (close >= (recentHighest + recentLowest) / 2 ? str.tostring(recentHighest - close, "$0.00") + " from box high" : str.tostring(close - recentLowest, "$0.00") + " from box low")) // Set the text formatting of the `closeLabel`, `barMoveTable` cell, and `rangeBox` objects. // `formatBold` and `formatItalic` can both be `true` to combine formats, or both `false` for no special formatting. switch formatBold and formatItalic => closeLabel.set_text_formatting(text.format_bold + text.format_italic) barMoveTable.cell_set_text_formatting(0, 0, text.format_bold + text.format_italic) rangeBox.set_text_formatting(text.format_bold + text.format_italic) formatBold => closeLabel.set_text_formatting(text.format_bold) barMoveTable.cell_set_text_formatting(0, 0, text.format_bold) rangeBox.set_text_formatting(text.format_bold) formatItalic => closeLabel.set_text_formatting(text.format_italic) barMoveTable.cell_set_text_formatting(0, 0, text.format_italic) rangeBox.set_text_formatting(text.format_italic) => closeLabel.set_text_formatting(text.format_none) barMoveTable.cell_set_text_formatting(0, 0, text.format_none) rangeBox.set_text_formatting(text.format_none)

Previous

Next

Time

Pine Script® v6 Documentation

Time

Introduction

In Pine Script®, the following key aspects apply when working with date and time values:

- **UNIX timestamp**: The native format for time values in Pine, representing the absolute number of *milliseconds* elapsed since midnight UTC on 1970-01-01. Several built-ins return UNIX timestamps directly, which users can format into readable dates and times. See the UNIX timestamps section below for more information.
- **Exchange time zone**: The time zone of the instrument's exchange. All calendar-based variables hold values expressed in the exchange time zone, and all built-in function overloads that have a `timezone` parameter use this time zone by default.
- **Chart time zone**: The time zone the chart and Pine Logs message prefixes use to express time values. Users can set the chart time zone using the "Timezone" input in the "Symbol" tab of the chart's settings. This setting only changes the *display* of dates and times on the chart and the times that prefix logged messages. It does **not** affect the behavior of Pine scripts because they cannot access a chart's time zone information.
- **`timezone` parameter**: A "string" parameter of time-related functions that specifies the time zone used in their calculations. For calendar-based functions, such as dayofweek(), the `timezone` parameter determines the time zone of the returned value. For functions that return UNIX timestamps, such as time(), the specified `timezone` defines the time zone of other applicable parameters, e.g., `session`. See the Time zone strings section to learn more.

UNIX timestamps

UNIX time is a standardized date and time representation that measures the number of *non-leap seconds* elapsed since January 1, 1970 at 00:00:00 UTC (the *UNIX Epoch*), typically expressed in seconds or smaller time units. A UNIX time value in Pine Script is an “int” *timestamp* representing the number of *milliseconds* from the UNIX Epoch to a specific point in time.

Because a UNIX timestamp represents the number of consistent time units elapsed from a fixed historical point (epoch), its value is **time zone-agnostic**. A UNIX timestamp in Pine always corresponds to the same distinct point in time, accurate to the millisecond, regardless of a user’s location.

For example, the UNIX timestamp `1723472500000` always represents the time 1,723,472,500,000 milliseconds (1,723,472,500 seconds) after the UNIX Epoch. This timestamp’s meaning does **not** change relative to any time zone.

To *format* an “int” UNIX timestamp into a readable date/time “string” expressed in a specific time zone, use the str.format_time() function. The function does not *modify* UNIX timestamps. It simply *represents* timestamps in a desired human-readable format.

For instance, the function can represent the UNIX timestamp `1723472500000` as a “string” in several ways, depending on its `format` and `timezone` arguments, without changing the *absolute* point in time that it refers to. The simple script below calculates three valid representations of this timestamp and displays them in the Pine Logs pane:

image

//@version=6 indicator("UNIX timestamps demo") //@variable A UNIX time value representing the specific point 1,723,472,500,000 ms after the UNIX Epoch. int unixTimestamp = 1723472500000 // These are a few different ways to express the `unixTimestamp` in a relative, human-readable format. // Despite their format and time zone differences, all the calculated strings represent the SAME distinct point: string isoExchange = str.format_time(unixTimestamp) string utcDateTime = str.format_time(unixTimestamp, "MM/dd/yyyy HH:mm:ss.S", "UTC+0") string utc4TimeDate = str.format_time(unixTimestamp, "hh:mm:ss a, MMMM dd,

yyyy z", "UTC+4") // Log the `unixTimestamp` and the custom "string" representations on the first bar. if barstate.isfirst log.info("\nUNIX time (ms): {0, number, #}\n ISO 8601 representation (Exchange time zone): {1}\n Custom date and time representation (UTC+0 time zone): {2}\n Custom time and date representation (UTC+4 time zone): {3}", unixTimestamp, isoExchange, utcDateTime, utc4TimeDate)

Note that:

- The value enclosed within square brackets in the logged message is an *automatic* prefix representing the historical time of the log.info() call in ISO 8601 format, expressed in the chart time zone.

See the Formatting dates and times section to learn more about representing UNIX timestamps with formatted strings.

Time zones

A time zone is a geographic region with an assigned *local time*. The specific time within a time zone is consistent throughout the region. Time zone boundaries typically relate to a location's longitude. However, in practice, they tend to align with administrative boundaries rather than strictly following longitudinal lines.

The local time within a time zone depends on its defined *offset* from Coordinated Universal Time (UTC), which can range from UTC-12:00 (12 hours *behind* UTC) to UTC+14:00 (14 hours *ahead* of UTC). Some regions maintain a consistent offset from UTC, and others have an offset that changes over time due to daylight saving time (DST) and other factors.

Two primary time zones apply to data feeds and TradingView charts: the *exchange time zone* and the *chart time zone*.

The exchange time zone represents the time zone of the current symbol's *exchange*, which Pine scripts can access with the syminfo.timezone variable. Calendar-based variables, such as month, dayofweek, and hour, always hold values expressed in the exchange time zone, and all time function overloads that have a `timezone` parameter use this time zone by default.

The chart time zone is a *visual preference* that defines how the chart and the time prefixes of Pine Logs represent time values. To set the chart time zone, use the “Timezone” input in the “Symbol” tab of the chart’s settings or click on the current time shown below the chart. The specified time zone does **not** affect time calculations in Pine scripts because they cannot access this chart information. Although scripts cannot access a chart’s time zone, programmers can provide inputs that users can adjust to match the time zone.

For example, the script below uses str.format_time() to represent the last historical bar’s opening and closing UNIX timestamps (time and time_close values) as date-time strings expressed in the function’s default time zone, the exchange time zone, UTC-0, and a user-specified time zone. It uses a table to display all four representations in the bottom-right corner of the chart for comparison:

image

//@version=6 indicator("Time zone comparison demo", overlay = true) // @variable The time zone of the time values in the last table row. // The "string" can contain either UTC offset notation or an IANA time zone identifier. string timezoneInput = input.string("UTC+4:00", "Time zone") //@variable A `table` showing strings representing bar times in three preset time zones and a custom time zone. var table displayTable = table.new(position.bottom_right, columns = 3, rows = 5, border_color = chart.fg_color, border_width = 2) //@function Initializes three `displayTable` cells on the `row` that show the `title`, `text1`, and `text2` strings. tableRow(int row, string title, string text1, string text2, color titleColor = #9b27b066, color infoColor = na) => displayTable.cell(0, row, title, bgcolor = titleColor, text_color = chart.fg_color) displayTable.cell(1, row, text1, bgcolor = infoColor, text_color = chart.fg_color) displayTable.cell(2, row, text2, bgcolor = infoColor, text_color = chart.fg_color) if barstate.islastconfirmedhistory // Draw an empty label to signify the bar that the displayed time strings represent. label.new(bar_index, high, color = #9b27b066, size = size.huge) //@variable The formatting string for all `str.format_time()` calls. Sets the format of the date-time strings. var string formatString = "yyyy-MM-dd HH:mm:ss" // Initialize a header row at the top of the `displayTable`. tableRow(0, "", "OPEN time", "CLOSE time", na, #9b27b066) // Initialize a row showing the bar's times in the default time

zone (no specified `timezone` arguments). tableRow(1, "Default", str.format_time(time, formatString), str.format_time(time_close, formatString)) // Initialize a row showing the bar's times in the exchange time zone (`syminfo.timezone`). tableRow(2, "Exchange: " + syminfo.timezone, str.format_time(time, formatString, syminfo.timezone), str.format_time(time_close, formatString, syminfo.timezone)) // Initialize a row showing the bar's times in the UTC-0 time zone (using "UTC" as the `timezone` arguments). tableRow(3, "UTC-0", str.format_time(time, formatString, "UTC"), str.format_time(time_close, formatString, "UTC")) // Initialize a row showing the bar's times in the custom time zone (`timezoneInput`). tableRow(4, "Custom: " + timezoneInput, str.format_time(time, formatString, timezoneInput), str.format_time(time_close, formatString, timezoneInput))

Note that:

- The label on the chart signifies which bar's times the displayed strings represent.
- The "Default" and "Exchange" rows in the table show identical results because syminfo.timezone is the str.format_time() function's default `timezone` argument.
- The exchange time zone on our example chart appears as `"America/New_York"`, the IANA identifier for the NASDAQ exchange's time zone. It represents UTC-4 *or* UTC-5, depending on the time of year. See the next section to learn more about time zone strings.

Time zone strings

All built-in functions with a `timezone` parameter accept a "string" argument specifying the time zone they use in their calculations. These functions can accept time zone strings in either of the following formats:

- **UTC** (or *GMT*) offset notation, e.g., `"UTC-5"`, `"UTC+05:30"`, `"GMT+0100"`
- **IANA database** notation, e.g., `"America/New_York"`, `"Asia/Calcutta"`, `"Europe/Paris"`

The IANA time zone database reference page lists possible time zone identifiers and their respective UTC offsets. The listed identifiers are valid as `timezone` arguments.

Note that various time zone strings expressed in UTC or IANA notation can represent the *same* offset from Coordinated Universal Time. For instance, these strings all represent a local time three hours ahead of UTC:

- `"UTC+3"`
- `"GMT+03:00"`
- `"Asia/Kuwait"`
- `"Europe/Moscow"`
- `"Africa/Nairobi"`

For the str.format_time() function and the functions that calculate calendar-based values from a UNIX timestamp, including month(), dayofweek(), and hour(), the “string” passed to the `timezone` parameter changes the returned value’s calculation to express the result in the specified time zone. See the Formatting dates and times and Calendar-based functions sections for more information.

The example below shows how time zone strings affect the returned values of calendar-based functions. This script uses three hour() function calls to calculate “int” values representing the opening hour of each bar in the exchange time zone, UTC-0, and a user-specified UTC offset. It plots all three calculated hours in a separate pane for comparison:

image

//@version=6 indicator("Time zone strings in calendar functions demo") // @variable An "int" representing the user-specified hourly offset from UTC. int utcOffsetInput = input.int(defval = 4, title ="Timezone offset UTC (+/-)", minval = -12, maxval = 14) //@variable A valid time zone string based on the `utcOffsetInput`, in UTC offset notation (e.g., "UTC-4"). string customOffset = "UTC" + (utcOffsetInput > 0 ? "+" : "") + str.tostring(utcOffsetInput) // @variable The bar's opening hour in the exchange time zone (default). Equivalent to the `hour` variable. int exchangeHour = hour(time) //@variable The bar's opening hour in the "UTC-0" time zone. int utcHour = hour(time, "UTC-0") //@variable The bar's opening hour in the `customOffset` time zone. int customOffsetHour = hour(time, customOffset) // Plot the `exchangeHour`, `utcHour`, and `customOffsetHour` for comparison. plot(exchangeHour, "Exchange hour", #E100FF5B, 8) plot(utcHour, "UTC-0 hour", color.blue, 3) plot(customOffsetHour, "Custom offset hour", color.orange, 3)

Note that:

- The `exchangeHour` value is four *or* five hours behind the `utcHour` because the NASDAQ exchange is in the “America/New_York” time zone. This time zone has a UTC offset that *changes* during the year due to daylight saving time (DST). The script’s default `customOffsetHour` is consistently four hours ahead of the `utcHour` because its time zone is UTC+4.
- The call to the hour() function without a specified `timezone` argument returns the same value that the hour *variable* holds because both represent the bar’s opening hour in the exchange time zone (syminfo.timezone).

For functions that return UNIX timestamps directly, such as time() and timestamp(), the `timezone` parameter defines the time zone of the function’s calendar-based *parameters*, including `session`, `year`, `month`, `day`, `hour`, `minute`, and `second`. The parameter does *not* determine the time zone of the returned value, as UNIX timestamps are *time zone-agnostic*. See the Testing for sessions and `timestamp()` sections to learn more.

The following script calls the timestamp() function to calculate the UNIX timestamp of a specific date and time, and it draws a label at the timestamp’s corresponding bar location. The user-selected `timezone` argument (`timezoneInput`) determines the time zone of the call’s calendar-based arguments. Consequently, the calculated timestamp varies with the `timezoneInput` value because identical local times in various time zones correspond to *different* amounts of time elapsed since the UNIX Epoch:

image

//@version=6 indicator("Time zone strings in UNIX timestamp functions demo", overlay = true) //@variable The `timezone` argument of the `timestamp()` call, which sets the time zone of all date and time parameters. string timezoneInput = input.string("Etc/UTC", "Time zone") //@variable The UNIX timestamp corresponding to a specific calendar date and time. // The specified `year`, `month`, `day`, `hour`, `minute`, and `second` represent calendar values in the // `timezoneInput` time zone. // Different `timezone` arguments produce different UNIX timestamps because an identical date in another // time zone does NOT represent the same absolute point in time. int

unixTimestamp = timestamp(timezone = timezoneInput, year = 2024, month = 10, day = 31, hour = 0, minute = 0, second = 0) //@variable The `close` value when the bar's opening time crosses the `unixTimestamp`. float labelPrice = ta.valuewhen(ta.cross(time, unixTimestamp), close, 0) // On the last historical bar, draw a label showing the `unixTimestamp` value at the corresponding bar location. if barstate.islastconfirmedhistory label.new(unixTimestamp, nz(labelPrice, close), "UNIX timestamp: " + str.tostring(unixTimestamp), xloc.bar_time, yloc.price, chart.fg_color, label.style_label_down, chart.bg_color, size.large)

Note that:

- `"Etc/UTC"` is the *IANA identifier* for the UTC+0 time zone.
- The label.new() call uses xloc.bar_time as its `xloc` argument, which is required to anchor the drawing to an absolute time value. Without this argument, the function treats the `unixTimestamp` as a relative bar index, leading to an incorrect location.
- The label's y value is the close of the bar where the time value crosses the `unixTimestamp` value. If the timestamp represents a future time, the label displays the last historical bar's price.

Although time zone strings can use either UTC or IANA notation, we recommend using *IANA notation* for `timezone` arguments in most cases, especially if a script's time calculations must align with the observed time offset in a specific country or subdivision. When a time function call uses an IANA time zone identifier for its `timezone` argument, its calculations adjust automatically for historical and future changes to the specified region's observed time, such as daylight saving time (DST) and updates to time zone boundaries, instead of using a fixed offset from UTC.

The following script demonstrates how UTC and IANA time zone strings can affect time calculations differently. It uses two calls to the hour() function to calculate the hour from the current bar's opening timestamp using `"UTC-4"` and `"America/New_York"` as `timezone` arguments. The script plots the results of both calls for comparison and colors the main pane's background when the returned values do not match. Although these two hour() calls may seem similar because UTC-4 is an observed UTC offset in New York, they *do not* always return the same results, as shown below:

image

//@version=6 indicator("UTC vs IANA time zone strings demo") //@variable The hour of the current `time` in the "UTC-4" time zone. // This variable's value represents the hour in New York only during DST. It is one hour ahead otherwise. int hourUTC = hour(time, "UTC-4") //@variable The hour of the current `time` in the "America/New_York" time zone. // This form adjusts to UTC offset changes automatically, so the value always represents the hour in New York. int hourIANA = hour(time, "America/New_York") //@variable Is translucent blue when `hourUTC` does not equal `hourIANA`, `na` otherwise. color bgColor = hourUTC != hourIANA ? color.rgb(33, 149, 243, 80) : na // Plot the values of `hourUTC` and `hourIANA` for comparison. plot(hourUTC, "UTC-4", color.blue, linewidth = 6) plot(hourIANA, "America/New_York", color.orange, linewidth = 3) // Highlight the main chart pane with the `bgColor`. bgcolor(bgColor, title = "Unequal result highlight", force_overlay = true)

The plots in the chart above diverge periodically because New York observes daylight saving time, meaning its UTC offset *changes* at specific points in a year. During DST, New York's local time follows UTC-4. Otherwise, it follows UTC-5. Because the script's first hour() call uses `"UTC-4"` as its `timezone` argument, it returns the correct hour in New York *only* during DST. In contrast, the call that uses the `"America/New_York"` time zone string adjusts its UTC offset automatically to return the correct hour in New York at *any* time of the year.

Time variables

Pine Script has several built-in variables that provide scripts access to different forms of time information:

- The time and time_close variables hold UNIX timestamps representing the current bar's opening and closing times, respectively.
- The time_tradingday variable holds a UNIX timestamp representing the starting time of the last UTC calendar day in a session.
- The timenow variable holds a UNIX timestamp representing the current time when the script executes.

- The year, month, weekofyear, dayofmonth, dayofweek, hour, minute, and second variables reference calendar values based on the current bar's opening time, expressed in the exchange time zone.
- The last_bar_time variable holds a UNIX timestamp representing the last available bar's opening time.
- The chart.left_visible_bar_time and chart.right_visible_bar_time variables hold UNIX timestamps representing the opening times of the leftmost and rightmost visible chart bars.
- The syminfo.timezone variable holds a "string" value representing the time zone of the current symbol's exchange in IANA database notation. All time-related function overloads with a `timezone` parameter use this variable as the default argument.

`time` and `time_close` variables

The time variable holds the UNIX timestamp of the current bar's *opening time*, and the time_close variable holds the UNIX timestamp of the bar's *closing time*.

These timestamps are unique, time zone-agnostic "int" values, which programmers can use to anchor drawing objects to specific bar times, calculate and inspect bar time differences, construct readable date/time strings with the str.format_time() function, and more.

The script below displays bar opening and closing times in different ways. On each bar, it formats the time and time_close timestamps into strings containing the hour, minute, and second in the exchange time zone, and it draws labels displaying the formatted strings at the open and close prices. Additionally, the script displays strings containing the unformatted UNIX timestamps of the last chart bar within a table in the bottom-right corner:

image

//@version=6 indicator("`time` and `time_close` demo", overlay = true, max_labels_count = 500) //@variable A "string" representing the hour, minute, and second of the bar's opening time in the exchange time zone. string openTimeString = str.format_time(time, "HH:mm:ss") //@variable A "string" representing the hour, minute, and second of the bar's closing time in the exchange time zone. string closeTimeString =

str.format_time(time_close, "HH:mm:ss") //@variable Is `label.style_label_down` when the `open` is higher than `close`, `label.style_label_up` otherwise. string openLabelStyle = open > close ? label.style_label_down : label.style_label_up //@variable Is `label.style_label_down` when the `close` is higher than `open`, `label.style_label_up` otherwise. string closeLabelStyle = close > open ? label.style_label_down : label.style_label_up // Draw labels anchored to the bar's `time` to display the `openTimeString` and `closeTimeString`. label.new(time, open, openTimeString, xloc.bar_time, yloc.price, color.orange, openLabelStyle, color.white) label.new(time, close, closeTimeString, xloc.bar_time, yloc.price, color.blue, closeLabelStyle, color.white) if barstate.islast //@variable A `table` displaying the last bar's *unformatted* UNIX timestamps. var table t = table.new(position.bottom_right, 2, 2, bgcolor = #ffe70d) // Populate the `t` table with "string" representations of the the "int" `time` and `time_close` values. t.cell(0, 0, "`time`") t.cell(1, 0, str.tostring(time)) t.cell(0, 1, "`time_close`") t.cell(1, 1, str.tostring(time_close))

Note that:

- This script's label.new() calls include xloc.bar_time as the `xloc` argument and time as the x argument to anchor the drawings to bar opening times.
- The formatted strings express time in the exchange time zone because we did not specify `timezone` arguments in the str.format_time() calls. NYSE, our chart symbol's exchange, is in the "America/New_York" time zone (UTC-4/-5).
- Although our example chart uses an *hourly* timeframe, the table and the labels at the end of the chart show that the last bar closes only *30 minutes* (1,800,000 milliseconds) after opening. This behavior occurs because the chart aligns bars with session opening and closing times. A session's final bar closes when the session ends, and a new bar opens when a new session starts. A typical session on our 60-minute chart with regular trading hours (RTH) spans from 09:30 to 16:00 (6.5 hours). The chart divides this interval into as many 60-minute bars as possible, starting from the session's opening time, which leaves only 30 minutes for the final bar to cover.

It’s crucial to note that unlike the time variable, which has consistent behavior across chart types, time_close behaves differently on *time-based* and *non-time-based* charts.

Time-based charts have bars that typically open and close at regular, *predictable* times within a session. Thanks to this predictability, time_close can accurately represent the *expected* closing time of an open bar on a time-based chart, as shown on the last bar in the example above.

In contrast, the bars on tick charts and *price-based* charts (all non-standard charts excluding Heikin Ashi) cover *irregular* time intervals. Tick charts construct bars based on successive ticks in the data feed, and price-based charts construct bars based on significant price movements. The time it takes for new ticks or price changes to occur is *unpredictable*. As such, the time_close value is na on the *open realtime bars* of these charts.

The following script uses the time and time_close variables with str.tostring() and str.format_time() to create strings containing bar opening and closing UNIX timestamps and formatted date-time representations, which it displays in labels at each bar’s high and low prices.

When applied to a Renko chart, which forms new bars based on *price movements*, the labels show correct results on all historical bars. However, the last bar has a time_close value of na because the future closing time is unpredictable. Consequently, the bar’s closing time label shows a timestamp of `"NaN"` and an *incorrect* date and time:

image

//@version=6 indicator("`time_close` on non-time-based chart demo", overlay = true) //@variable A formatted "string" containing the date and time that `time_close` represents in the exchange time zone. string formattedCloseTime = str.format_time(time_close, format = "'Date and time:' yyyy-MM-dd HH:mm:ss") //@variable A formatted "string" containing the date and time that `time` represents in the exchange time zone. string formattedOpenTime = str.format_time(time, format = "'Date and time:' yyyy-MM-dd HH:mm:ss") //@variable A "string" containing the `time_close` UNIX timestamp and the `formattedCloseTime`. string closeTimeText = str.format("Close timestamp: {0,number,#}\n{1}", time_close, formattedCloseTime) //@variable A "string" containing the `time` UNIX

timestamp and the `formattedOpenTime`. string openTimeText = str.format("Open timestamp: {0,number,#}\n{1}", time, formattedOpenTime) // Define label colors for historical and realtime bars. color closeLabelColor = barstate.islast ? color.purple : color.aqua color openLabelColor = barstate.islast ? color.green : color.orange // Draw a label at the `high` to display the `closeTimeText` and a label at the `low` to display the `openTimeText`, // both anchored to the bar's `time`. label.new(time, high, closeTimeText, xloc.bar_time, color = closeLabelColor, textcolor = color.white, size = size.large, textalign = text.align_left) label.new(time, low, openTimeText, xloc.bar_time, color = openLabelColor, style = label.style_label_up, size = size.large, textcolor = color.white, textalign = text.align_left) // Highlight the background yellow on the latest bar. bgcolor(barstate.islast ? #f3de22cb : na, title = "Latest bar highlight")

Note that:

- The script draws up to 50 labels because we did not specify a `max_labels_count` argument in the indicator() declaration statement.
- The str.format_time() function replaces na values with 0 in its calculations, which is why it returns an incorrect date-time “string” on the last bar. A timestamp of 0 corresponds to the *UNIX Epoch* (00:00:00 UTC on January 1, 1970). However, the str.format_time() call does not specify a `timezone` argument, so it expresses the epoch’s date and time in the *exchange time zone*, which was five hours behind UTC at that point in time.
- The time_close() function, which returns the closing timestamp of a bar on a specified timeframe within a given session, also returns na on the open realtime bars of tick-based and price-based charts.

Scripts can retrieve a realtime bar’s closing time on tick charts and price-based charts once the bar is *confirmed*. The closing timestamp of an *elapsed realtime bar* is committed to the realtime data feed as soon as the bar closes, so its time_close value is no longer na.

To demonstrate this, we can modify the script above to use `time_close[1]` to output the previous bar’s closing time on each bar. The image below shows two highlighted realtime bars. When we executed the script on the chart, the first bar was initially an unconfirmed realtime bar. Its label shows the previous *historical* bar’s closing time. After some time, this realtime bar

closed, and the second highlighted bar opened. The label on the new realtime bar shows the *elapsed realtime* bar's closing time:

image

//@version=6 indicator("`time_close[1]` on non-time-based chart demo", overlay = true) //@variable A formatted "string" containing the date and time that `time_close[1]` represents in the exchange time zone. string formattedCloseTime = str.format_time(time_close[1], format = "'Date and time:' yyyy-MM-dd HH:mm:ss") //@variable A "string" containing the `time_close[1]` UNIX timestamp and the `formattedCloseTime`. string closeTimeText = str.format("Close timestamp of previous bar: {0,number,#} \n{1}", time_close[1], formattedCloseTime) // Define label colors for historical and realtime bars. color closeLabelColor = barstate.islast ? color.purple : color.aqua // Draw a label at the `high` to display the `closeTimeText` anchored to the bar's `time`. label.new(time, high, closeTimeText, xloc.bar_time, color = closeLabelColor, textcolor = color.white, size = size.large, textalign = text.align_left) // Highlight the background yellow on the realtime bar. bgcolor(barstate.islast ? #f3de22cb : na, title = "Realtime bar highlight")

Note that:

- A confirmed realtime bar is **not** the same as a historical bar. Pine's execution model uses separate data feeds for realtime and historical data. The closing time of a confirmed realtime bar is committed to the *realtime* feed, until the script re-executes on the chart. Only then will this bar's closing time load *historically* along with all the other closed bars.
- The barstate.islast value is `true` for all realtime bars in the dataset. Therefore, the elapsed realtime bar and the latest realtime bar both display a purple label and highlighted background. See the Bar states page to learn more about the different `barstate.*` variables in Pine Script.
- The time_close() function can similarly retrieve the previous bar's closing time on price-based charts using a `bar_back = 1` argument.

time_tradingday

The time_tradingday variable holds a UNIX timestamp representing the starting time (00:00 UTC) of the last trading day in the current bar's final session. It is helpful primarily for date and time calculations on *time-based* charts for symbols with overnight sessions that start and end on *different* calendar days.

On "1D" and lower timeframes, the time_tradingday timestamp corresponds to the beginning of the day when the current session *ends*, even for bars that open and close on the previous day. For example, the "Monday" session for "EURUSD" starts on Sunday at 17:00 and ends on Monday at 17:00 in the exchange time zone. The time_tradingday values of *all* intraday bars within the session represent Monday at 00:00 UTC.

On timeframes higher than "1D", which can cover *multiple* sessions, time_tradingday holds the timestamp representing the beginning of the last calendar day of the bar's *final* trading session. For example, on a "EURUSD, 1W" chart, the timestamp represents the start of the last trading day in the week, which is typically Friday at 00:00 UTC.

The script below demonstrates how the time_tradingday and time variables differ on Forex symbols. On each bar, it draws labels to display strings containing the variables' UNIX timestamps and formatted dates and times. It also uses the dayofmonth() function to calculate the UTC calendar day from both timestamps, highlighting the background when the calculated days do not match.

When applied to the "FXCM:EURUSD" chart with the "3h" ("180") timeframe, the script highlights the background of the *first bar* in each session, as each session opens on the *previous* calendar day. The dayofmonth() call that uses time calculates the opening day on the session's first bar, whereas the call that uses time_tradingday calculates the day when the session *ends*:

image

//@version=6 indicator("`time_tradingday` demo", overlay = true) // @variable A concatenated "string" containing the `time_tradingday` timestamp and a formatted representation in UTC. string tradingDayText = "`time_tradingday`: " + str.tostring(time_tradingday) + "\n" + "Date and

time: " + str.format_time(time_tradingday, "dd MMM yyyy, HH:mm (z)", "UTC+0") //@variable A concatenated "string" containing the `time` timestamp and a formatted representation in UTC. string barOpenText = "`time`: " + str.tostring(time) + "\n" + "Date and time: " + str.format_time(time, "dd MMM yyyy, HH:mm (z)", "UTC+0") //@variable Is `true` on every even bar, `false` otherwise. This condition determines the appearance of the labels. bool isEven = bar_index % 2 == 0 // The `yloc` and `style` properties of the labels. They alternate on every other bar for visibility. labelYloc = isEven ? yloc.abovebar : yloc.belowbar labelStyle = isEven ? label.style_label_down : label.style_label_up // Draw alternating labels anchored to the bar's `time` to display the `tradingDayText` and `barOpenText`. if isEven label.new(time, 0, tradingDayText + "\n\n\n", xloc.bar_time, labelYloc, color.teal, labelStyle, color.white) label.new(time, 0, barOpenText, xloc.bar_time, labelYloc, color.maroon, labelStyle, color.white) else label.new(time, 0, "\n\n\n" + barOpenText, xloc.bar_time, labelYloc, color.maroon, labelStyle, color.white) label.new(time, 0, tradingDayText, xloc.bar_time, labelYloc, color.teal, labelStyle, color.white) //@variable The day of the month, in UTC, that the `time_tradingday` timestamp corresponds to. int tradingDayOfMonth = dayofmonth(time_tradingday, "UTC+0") //@variable The day of the month, in UTC, that the `time` timestamp corresponds to. int openingDayOfMonth = dayofmonth(time, "UTC+0") // Highlight the background when the `tradingDayOfMonth` does not equal the `openingDayOfMonth`. bgcolor(tradingDayOfMonth != openingDayOfMonth ? color.rgb(174, 89, 243, 85) : na, title = "Different day highlight")

Note that:

- The str.format_time() and dayofmonth() calls use `"UTC+0"` as the `timezone` argument, meaning the results represent calendar time values with no offset from UTC. In the screenshot, the first bar opens at 21:00 UTC, 17:00 in the exchange time zone (“America/New_York”).
- The formatted strings show `"GMT"` as the acronym of the time zone, which is equivalent to `"UTC+0"` in this context.
- The time_tradingday value is the same for *all* three-hour bars within each session, even for the initial bar that opens on the previous UTC calendar day. The assigned timestamp changes only when a new session starts.

timenow

The timenow variable holds a UNIX timestamp representing the script's *current time*. Unlike the values of other variables that hold UNIX timestamps, the values in the timenow series correspond to times when the script *executes*, not the times of specific bars or trading days.

A Pine script executes only *once* per historical bar, and all historical executions occur when the script first *loads* on the chart. As such, the timenow value is relatively consistent on historical bars, with only occasional millisecond changes across the series. In contrast, on realtime bars, a script executes once for *each new update* in the data feed, which can happen several times per bar. With each new execution, the timenow value updates on the latest bar to represent the current time.

This variable is most useful on realtime bars, where programmers can apply it to track the times of the latest script executions, count the time elapsed within open bars, control drawings based on bar updates, and more.

The script below inspects the value of timenow on the latest chart bars and uses it to analyze realtime bar updates. When the script first reaches the last chart bar, it declares three variables with the varip keyword to hold the latest timenow value, the total time elapsed between the bar's updates, and the total number of updates. It uses these values to calculate the average number of milliseconds between updates, which it displays in a label along with the current execution's timestamp, a formatted time and date in the exchange time zone, and the current number of bar updates:

image

//@version=6 indicator("`timenow` demo", overlay = true, max_labels_count = 500) if barstate.islast //@variable Holds the UNIX timestamp of the latest script execution for timing updates in the data feed. varip int lastUpdateTimestamp = timenow //@variable The total number of milliseconds elapsed across the bar's data updates. varip int totalUpdateTime = 0 //@variable The number of updates that have occurred on the current bar. varip int numUpdates = 0 // Add the time elapsed from the `lastUpdateTimestamp` to `totalUpdateTime`, increase the `numUpdates` counter, and // update the `lastUpdateTimestamp` when the `timenow` value

changes. if timenow != lastUpdateTimestamp totalUpdateTime += timenow - lastUpdateTimestamp numUpdates += 1 lastUpdateTimestamp := timenow // @variable The average number of milliseconds elapsed between the bar's updates. float avgUpdateTime = nz(totalUpdateTime / numUpdates) // @variable Contains the `timenow` value, a custom representation, and the `numUpdates` and `avgUpdateTime` values. string displayText = "`timenow`: " + str.tostring(timenow) + "\nTime and date (exchange): " + str.format_time(timenow, "HH:mm:ss MM/dd/yy") + "\nNumber of updates: " + str.tostring(numUpdates) + "\nAvg. time between updates: " + str.tostring(avgUpdateTime, "#.###") + " ms" //@variable The color of the label. Is blue when the bar is open, and gray after it closes. color labelColor = barstate.isconfirmed ? color.gray : color.blue //@variable The label's y-coordinate. Alternates between `high` and `low` on every other bar. float labelPrice = bar_index % 2 == 0 ? high : low //@variable The label's style. Alternates between "lower-right" and "upper-right" styles. labelStyle = bar_index % 2 == 0 ? label.style_label_lower_right : label.style_label_upper_right // Draw a `labelColor` label anchored to the bar's `time` to show the `displayText`. label.new(time, labelPrice, displayText, xloc.bar_time, color = labelColor, style = labelStyle, textcolor = color.white, size = size.large) // Reset the `totalUpdateTime` and `numUpdates` counters when the bar is confirmed (closes). if barstate.isconfirmed totalUpdateTime := 0 numUpdates := 0

Note that:

- When a bar is open, the drawn label is blue to signify that additional updates can occur. After the bar closes, the final label’s color is gray.
- Although we’ve set the chart time zone to match the exchange time zone, the formatted time in the open bar’s label and the time shown below the chart *do not* always align. The script records a new timestamp only when a *new execution* occurs, whereas the time below the chart updates *continuously*.
- The varip keyword specifies that a variable does not revert to the last committed value in its series when new updates occur. This behavior allows the script to use variables to track changes in timenow on an open bar.
- Updates to timenow on open realtime bars do not affect the recorded timestamps on confirmed bars as the script executes. However, the

historical series changes (*repaints*) after reloading the chart because timenow references the script's *current time*, not the times of specific bars.

Calendar-based variables

The year, month, weekofyear, dayofmonth, dayofweek, hour, minute, and second variables hold *calendar-based* "int" values calculated from the current bar's *opening time*, expressed in the exchange time zone. These variables reference the same values that calendar-based functions return when they use the default `timezone` argument and time as the `time` argument. For instance, the year variable holds the same value that a `year(time)` call returns.

Programmers can use these calendar-based variables for several purposes, such as:

- Identifying a bar's opening date and time.
- Passing the variables to the timestamp() function to calculate UNIX timestamps.
- Testing when date/time values or ranges occur in a data feed.

One of the most common use cases for these variables is checking for date or time ranges to control when a script displays visuals or executes calculations. This simple example inspects the year variable to determine when to plot a visible value. If the year is 2022 or higher, the script plots the bar's close. Otherwise, it plots na:

image

//@version=6 indicator("`year` demo", overlay = true) // Plot the `close` price on bars that open in the `year` 2022 onward. Otherwise, plot `na` to display nothing. plot(year >= 2022 ? close : na, "`close` price (year 2022 and later)", linewidth = 3)

When using these variables in conditions that isolate specific dates or times rather than ranges, it's crucial to consider that certain conditions might not detect some occurrences of the values due to a chart's timeframe, the opening times of chart bars, or the symbol's active session.

For instance, suppose we want to detect when the first calendar day of each month occurs on the chart. Intuitively, one might consider simply checking when the dayofmonth value equals 1. However, this condition only identifies when a bar *opens* on a month's first day. The bars on some charts can open and close in *different* months. Additionally, a chart bar might not contain the first day of a month if the market is *closed* on that day. Therefore, we must create extra conditions that work in these scenarios to identify the first day in *any* month on the chart.

The script below uses the dayofmonth and month variables, and the month() function, to create three conditions that detect the first day of the month in different ways. The first condition detects if the bar opens on the first day, the second checks if the bar opens in one month and closes in another, and the third checks if the chart skips the date entirely. The script draws labels showing bar opening dates and highlights the background with different colors to visualize when each condition occurs:

image

//@version=6 indicator("Detecting the first day of the month demo", overlay = true, max_labels_count = 500) //@variable Is `true` only if the current bar opens on the first day of the month in exchange time, `false` otherwise. bool opensOnFirst = dayofmonth == 1 //@variable Is `true` if the bar opens in one month and closes in another, meaning its time span includes the first day. bool containsFirst = month != month(time_close) //@variable Is `true` only if the bar opens in a new month and the current or previous bar does not cover the first day. bool skipsFirst = month != month[1] and not (opensOnFirst or containsFirst[1]) //@variable The name of the current bar's opening weekday. string weekdayName = switch dayofweek dayofweek.sunday => "Sunday" dayofweek.monday => "Monday" dayofweek.tuesday => "Tuesday" dayofweek.wednesday => "Wednesday" dayofweek.thursday => "Thursday" dayofweek.friday => "Friday" dayofweek.saturday => "Saturday" //@variable A custom "string" representing the bar's opening date, including the weekday name. string openDateText = weekdayName + str.format_time(time, ", MMM d, yyyy") // Draw a green label when the bar opens on the first day of the month. if opensOnFirst string labelText = "Bar opened on\n" + openDateText label.new(time, open, labelText, xloc.bar_time, color = color.green, textcolor = color.white, size = size.large) // Draw a blue label when the bar opens and

closes in different months. if containsFirst string labelText = "Bar includes the first day,\nbut opened on\n" + openDateText label.new(time, open, labelText, xloc.bar_time, color = color.blue, textcolor = color.white, size = size.large) // Draw a red label when the chart skips the first day of the month. if skipsFirst string labelText = "Chart doesn't include the first day.\nBar opened on\n" + openDateText label.new(time, open, labelText, xloc.bar_time, color = color.red, textcolor = color.white, size = size.large) // Highlight the background when the conditions occur. bgcolor(opensOnFirst ? color.new(color.green, 70) : na, title = "`opensOnFirst` condition highlight") bgcolor(containsFirst ? color.new(color.blue, 70) : na, title = "`containsFirst` condition highlight") bgcolor(skipsFirst ? color.new(color.red, 70) : na, title = "`skipsFirst` condition highlight")

Note that:

- The script calls the month() function with time_close as the `time` argument to calculate each bar's *closing* month for the `containsFirst` condition.
- The `dayofweek.*` namespace contains variables that hold each possible dayofweek value, e.g., dayofweek.sunday has a constant value of 1 and dayofweek.saturday has a constant value of 7. The script compares dayofweek to these variables in a switch structure to determine the weekday name shown inside each label.
- To detect the *first opening time* in a monthly timeframe, not strictly the first day in a calendar month, use `ta.change(time("1M")) > 0` or `timeframe.change("1M")` instead of conditions based on these variables. See the Testing for changes in higher timeframes section to learn more.

`last_bar_time`

The last_bar_time variable holds a UNIX timestamp representing the *last* available bar's opening time. It is similar to last_bar_index, which references the latest bar index. On historical bars, last_bar_time consistently references the time value of the last bar available when the script first *loads* on the chart. The only time the variable's value updates across script executions is when a new realtime bar opens.

The following script uses the last_bar_time variable to get the opening timestamp of the last chart bar during its execution on the *first bar*. It displays the UNIX timestamp and a formatted date and time using a single-cell table created only on that bar. When the script executes on the last available bar, it creates a label to show the bar's time value and its formatted representation for visual comparison.

As the chart below shows, both drawings display *identical* times, verifying that last_bar_time correctly references the last bar's time value on previous historical bars:

image

//@version=6 indicator("`last_bar_time` demo", overlay = true) if barstate.isfirst //@variable A single-cell `table`, created only on the *first* bar, showing the *last* available bar's opening time. table displayTable = table.new(position.bottom_right, 1, 1, color.aqua) //@variable A "string" containing the `last_bar_time` UNIX timestamp and a custom date and time representation. string lastBarTimeText = "`last_bar_time`: " + str.tostring(last_bar_time) + "\nDate and time (exchange): " + str.format_time(last_bar_time, "dd/MM/yy HH:mm:ss") // Initialize the `displayTable` cell with the `lastBarTimeText`. displayTable.cell(0, 0, lastBarTimeText, text_color = color.white, text_size = size.large, text_halign = text.align_left) //@variable Is `true` only on the first occurrence of `barstate.islast`, `false` otherwise. // This condition occurs on the bar whose `time` the `last_bar_time` variable refers to on historical bars. bool isInitialLastBar = barstate.islast and not barstate.islast[1] if isInitialLastBar // @variable A "string" containing the last available bar's `time` value and a custom date and time representation. // Matches the `lastBarTimeText` from the first bar because `last_bar_time` equals this bar's `time`. string openTimeText = "`time`: " + str.tostring(time) + "\nDate and time (exchange): " + str.format_time(time, "dd/MM/yy HH:mm:ss") // Draw a label anchored to the bar's `time` to display the `openTimeText`. label.new(time, high, openTimeText, xloc.bar_time, color = color.purple, textcolor = color.white, size = size.large, textalign = text.align_left) // Highlight the background when `isInitialLastBar` is `true` for visual reference. bgcolor(barstate.islast ? color.rgb(155, 39, 176, 80) : na, title = "Initial last bar highlight")

Note that:

- The script creates the label only on the *first* bar with the barstate.islast state because that bar's time value is what last_bar_time equals on all historical bars. On subsequent bars, the last_bar_time value *updates* to represent the latest realtime bar's opening time.
- Updates to last_bar_time on realtime bars do not affect the values on historical bars as the script executes. However, the variable's series *repaints* when the script restarts because last_bar_time always references the latest available bar's opening time.
- This script expresses dates using the `"dd/MM/yy"` format, meaning the two-digit day appears before the two-digit month, and the month appears before the two-digit representation of the year. See this section below for more information.

Visible bar times

The chart.left_visible_bar_time and chart.right_visible_bar_time variables reference the opening UNIX timestamps of the chart's leftmost (first) and rightmost (last) *visible bars* on every script execution. When a script uses these variables, it responds dynamically to visible chart changes, such as users scrolling across bars or zooming in/out. Each time the visible window changes, the script *re-executes* automatically to update the variables' values on all available bars.

The example below demonstrates how the chart.left_visible_bar_time and chart.right_visible_bar_time variables work across script executions. The script draws labels anchored to the visible bars' times to display the UNIX timestamps. In addition, it draws two single-cell tables showing corresponding dates and times in the standard ISO 8601 format. The script creates these drawings only when it executes on the first bar. As the script continues to execute on subsequent bars, it identifies each bar whose time value equals either visible bars' timestamp and colors it on the chart:

image

//@version=6 indicator("Visible bar times demo", overlay = true) // Create strings on the first chart bar that contain the first and last visible bars' UNIX timestamps. var string leftTimestampText = str.format("UNIX timestamp:

{0,number,#}", chart.left_visible_bar_time) var string rightTimestampText = str.format("UNIX timestamp: {0,number,#}", chart.right_visible_bar_time) // Create strings on the first bar that contain the exchange date and time of the visible bars in ISO 8601 format. var string leftDateTimeText = "Date and time: " + str.format_time(chart.left_visible_bar_time) var string rightDateTimeText = "Date and time: " + str.format_time(chart.right_visible_bar_time) // @variable A `label` object anchored to the first visible bar's time. Shows the `leftTimestampText`. var label leftTimeLabel = label.new(chart.left_visible_bar_time, 0, leftTimestampText, xloc.bar_time, yloc.abovebar, color.purple, label.style_label_lower_left, color.white, size.large) //@variable A `label` object anchored to the last visible bar's time. Shows the `rightTimestampText`. var label rightTimeLabel = label.new(chart.right_visible_bar_time, 0, rightTimestampText, xloc.bar_time, yloc.abovebar, color.teal, label.style_label_lower_right, color.white, size.large) //@variable A single-cell `table` object showing the `leftDateTimeText`. var table leftTimeTable = table.new(position.middle_left, 1, 1, color.purple) //@variable A single-cell `table` object showing the `rightDateTimeText`. var table rightTimeTable = table.new(position.middle_right, 1, 1, color.teal) // On the first bar, initialize the `leftTimeTable` and `rightTimeTable` with the corresponding date-time text. if barstate.isfirst leftTimeTable.cell(0, 0, leftDateTimeText, text_color = color.white, text_size = size.large) rightTimeTable.cell(0, 0, rightDateTimeText, text_color = color.white, text_size = size.large) // @variable Is purple at the left visible bar's opening time, teal at the right bar's opening time, `na` otherwise. color barColor = switch time chart.left_visible_bar_time => color.purple chart.right_visible_bar_time => color.teal // Color the leftmost and rightmost visible bars using the `barColor`. barcolor(barColor, title = "Leftmost and rightmost visible bar color")

Note that:

- The chart.left_visible_bar_time and chart.right_visible_bar_time values are consistent across all executions, which allows the script to identify the visible bars' timestamps on the *first* bar and check when the time value equals them. The script restarts on any chart window changes, updating the variables' series to reference the new timestamps on every bar.

- The str.format_time() function uses ISO 8601 format by default when the call does not include a `format` argument because it is the *international standard* for expressing dates and times. See the Formatting dates and times section to learn more about time string formats.

`syminfo.timezone`

The syminfo.timezone variable holds a time zone string representing the current symbol's *exchange* time zone. The "string" value expresses the time zone as an *IANA identifier* (e.g., `"America/New_York"`). The overloads of time functions that include a `timezone` parameter use syminfo.timezone as the default argument.

Because this variable is the default `timezone` argument for all applicable time function overloads, it is unnecessary to use as an explicit argument, except for stylistic purposes. However, programmers can use the variable in other ways, such as:

- Displaying the "string" in Pine Logs or drawings to inspect the exchange time zone's IANA identifier.
- Comparing the value to other time zone strings to create time zone-based conditional logic.
- Requesting the exchange time zones of other symbols with `request.*()` function calls.

The following script uses the timenow variable to retrieve the UNIX timestamp of its latest execution. It formats the timestamp into date-time strings expressed in the main symbol's exchange time zone and a requested symbol's exchange time zone, which it displays along with the IANA identifiers in a table on the last chart bar:

image

//@version=6 indicator("`syminfo.timezone` demo", overlay = true) // @variable The symbol to request exchange time zone information for. string symbolInput = input.symbol("NSE:BANKNIFTY", "Requested symbol") // @variable A `table` object displaying the exchange time zone and the time of the script's execution. var table t = table.new(position.bottom_right, 2, 3, color.yellow, border_color = color.black, border_width = 1) //@variable The

IANA identifier of the exchange time zone requested for the `symbolInput` symbol. var string requestedTimezone = na if barstate.islastconfirmedhistory // Retrieve the time zone of the user-specified symbol's exchange. requestedTimezone := request.security(symbolInput, "", syminfo.timezone, calc_bars_count = 2) // Initialize the `t` table's header cells. t.cell(0, 0, "Exchange prefix and time zone string", text_size = size.large) t.cell(1, 0, "Last execution date and time", text_size = size.large) t.cell(0, 1, syminfo.prefix(syminfo.tickerid) + " (" + syminfo.timezone + ")", text_size = size.large) t.cell(0, 2, syminfo.prefix(symbolInput) + " (" + requestedTimezone + ")", text_size = size.large) if barstate.islast //@variable The formatting string for all `str.format_time()` calls. var string formatString = "HH:mm:ss 'on' MMM dd, YYYY" // Initialize table cells to display the formatted text. t.cell(1, 1, str.format_time(timenow, formatString), text_size = size.large) t.cell(1, 2, str.format_time(timenow, formatString, requestedTimezone), text_size = size.large)

Note that:

- Pine scripts execute on realtime bars only when new updates occur in the data feed, and timenow updates only on script executions. As such, when no realtime updates are available, the timenow timestamp does not change. See this section above for more information.
- The default `symbolInput` value is `"NSE:BANKNIFTY"`. NSE is in the “Asia/Kolkata” time zone, which is 9.5 hours ahead of the main symbol’s exchange time zone (“America/New_York”) at the time of the screenshot. Although the *local* time representations differ, both refer to the same *absolute* time that the timenow timestamp represents.
- Pine v6 scripts use dynamic `request.*()` calls by default, which allows the script to call request.security() dynamically inside the if structure’s *local scope*. See the Dynamic requests section of the Other timeframes and data page to learn more.

Time functions

Pine Script features several built-in functions that scripts can use to retrieve, calculate, and express time values:

- The time() and time_close() functions allow scripts to retrieve UNIX timestamps for the opening and closing times of bars within a session on a specified timeframe, without requiring `request.*()` function calls.
- The year(), month(), weekofyear(), dayofmonth(), dayofweek(), hour(), minute(), and second() functions calculate calendar-based values, expressed in a specified time zone, from a UNIX timestamp.
- The timestamp() function calculates a UNIX timestamp from a specified calendar date and time.
- The str.format_time() function formats a UNIX timestamp into a human-readable date/time "string", expressed in a specified time zone. The Formatting dates and times section below provides detailed information about formatting timestamps with this function.
- The input.time() function returns a UNIX timestamp corresponding to the user-specified date and time, and the input.session() function returns a valid session string corresponding to the user-specified start and end times. See the Time input and Session input sections of the Inputs page to learn more about these functions.

`time()` and `time_close()` functions

The time() and time_close() functions return UNIX timestamps representing the opening and closing times of bars on a specified timeframe. Both functions can *filter* their returned values based on a given session in a specific time zone. They each have the following signatures:

```
functionName(timeframe, bars_back) → series int
functionName(timeframe, session, bars_back) → series int
functionName(timeframe, session, timezone, bars_back) → series int
```

Where:

- `functionName` is the function's identifier.

- The `timeframe` parameter accepts a timeframe string. The function uses the script's main timeframe if the argument is timeframe.period or an empty "string".
- The `session` parameter accepts a session string defining the session's start and end times (e.g., `"0930-1600"`) and the days for which it applies (e.g., `":23456"` means Monday - Friday). If the value does not specify the days, the session applies to *all* weekdays automatically. The function returns UNIX timestamps only for the bars *within* the session. It returns na if a bar's time is *outside* the session. If the `session` argument is an empty "string" or not specified, The function uses the symbol's session information.
- The `timezone` parameter accepts a valid time zone string that defines the time zone of the specified `session`. It does **not** change the meaning of returned UNIX timestamps, as they are *time zone-agnostic*. If the `timezone` argument is not specified, the function uses the exchange time zone (syminfo.timezone).
- The `bars_back` parameter accepts an "int" value specifying which bar's time the returned timestamp represents. If the value is positive, the function returns the timestamp from that number of bars back relative to the current bar. If the value is negative and greater than or equal to -500, the function returns the *expected* timestamp of a future bar. The default value is 0.

Similar to the time and time_close variables, these functions behave differently on *time-based* and *non-time-based* charts.

Time-based charts have bars that open and close at *predictable* times, whereas the bars on tick charts and all non-standard charts, excluding Heikin Ashi, open and close at irregular, *unpredictable* times. Consequently, time_close() cannot calculate the expected closing time of an open realtime bar on non-time-based charts, so it returns na on that bar. Similarly, the time() function with a negative `bars_back` value cannot accurately calculate the expected opening time of a future realtime bar on these charts. See the *second example* in this section above. That example script exhibits the same behavior on a price-based chart if it uses a `time_close("")` call instead of the time_close variable.

On non-time-based charts, a new bar's closing time is available in the realtime data feed once the bar closes. Therefore, time_close() can retrieve

valid closing timestamps for *confirmed realtime bars* without needing to restart the script to load them historically. See the *third example* in this section above. Replacing the `time_close[1]` variable in the example script with a `time_close("", 1)` call achieves the same result to retrieve the closing time of the elapsed realtime bar.

Typical use cases for the time() and time_close() functions include:

- Testing for bars that open or close in specific sessions defined by the `session` and `timezone` parameters.
- Testing for changes or measuring time differences on specified higher timeframes.

Testing for sessions

The time() and time_close() functions' `session` and `timezone` parameters define the sessions for which they can return *non-na* values. If a call to either function references a bar that opens/closes within the defined session in a given time zone, it returns a UNIX timestamp for that bar. Otherwise, it returns na. Programmers can pass the returned values to the na() function to identify which bars open or close within specified intervals, which is helpful for session-based calculations and logic.

This simple script identifies when a bar on the chart's timeframe opens at or after 11:00 and before 13:00 in the exchange time zone on any trading day. It calls time() with timeframe.period as the `timeframe` argument and the `"1100-1300"` session string as the `session` argument, and then verifies whether the returned value is na with the na() function. When the value is **not** na, the script highlights the chart's background to indicate that the bar opened in the session:

image

//@version=6 indicator("Testing for session bars demo", overlay = true) // @variable Checks if the bar opens between 11:00 and 13:00. Is `true` if the `time()` function does not return `na`. bool inSession = not na(time(timeframe.period, "1100-1300")) // Highlight the background of the bars that are in the session "11:00-13:00". bgcolor(inSession ? color.rgb(155, 39, 176, 80) : na)

Note that:

- The `session` argument in the time() call represents an interval in the *exchange* time zone because syminfo.timezone is the default `timezone` argument.
- The session string expresses the start and end times in the `"HHmm-HHmm"` format, where `"HH"` is the two-digit *hour* and `"mm"` is the two-digit *minute*. Session strings can also specify the *weekdays* a session applies to. However, the time() call's `session` argument (`"1100-1300"`) does not include this information, which is why it considers the session valid for *every* day. See the Sessions page to learn more.

When using session strings in time() and time_close() calls, it's crucial to understand that such strings define start and end times in a *specific* time zone. The local hour and minute in one region may not correspond to the same point in UNIX time as that same hour and minute in another region. Therefore, calls to these functions with different `timezone` arguments can return non-na timestamps at *different* times, as the specified time zone string changes the meaning of the local times represented in the `session` argument.

This example demonstrates how the `timezone` parameter affects the `session` parameter in a time() function call. The script calculates an `opensInSession` condition that uses a time() call with arguments based on inputs. The session input for the `session` argument includes four preset options: `"0000-0400"`, `"0930-1400"`, `"1300-1700"`, and `"1700-2100"`. The string input that defines the `timezone` argument includes four IANA time zone options representing different offsets from UTC: `"America/Vancouver"` (UTC-7/-8), `"America/New_York"` (UTC-4/-5), `"Asia/Dubai"` (UTC+4), and `"Austrailia/Sydney"` (UTC+10/+11).

For any chosen `sessionInput` value, changing the `timezoneInput` value changes the specified session's time zone. The script highlights *different bars* with each time zone choice because, unlike UNIX timestamps, the *absolute* times that local hour and minute values correspond to *varies* across time zones:

image

```
//@version=6 indicator("Testing session time zones demo", overlay = true) // @variable The timeframe of the analyzed bars. string timeframeInput = input.timeframe("60", "Timeframe") //@variable The session to check. Features four interval options. string sessionInput = input.session("1300-1700", "Session", ["0000-0400", "0930-1400", "1300-1700", "1700-2100"]) //@variable An IANA identifier representing the time zone of the `sessionInput`. Fetures four preset options. string timezoneInput = input.string("America/New_York", "Time zone", ["America/Vancouver", "America/New_York", "Asia/Dubai", "Australia/Sydney"] ) // @variable Is `true` if a `timeframeInput` bar opens within the `sessionInput` session in the `timezoneInput` time zone. // The condition detects the session on different bars, depending on the chosesn time zone, because identical // local times in different time zones refer to different absolute points in UNIX time. bool opensInSession = not na(time(timeframeInput, sessionInput, timezoneInput)) // Highlight the background when `opensInSession` is `true`. bgcolor(opensInSession ? color.rgb(33, 149, 243, 80) : na, title = "Open in session highlight")
```

Note that:

- This script uses *IANA notation* for all time zone strings because it is the recommended format. Using an IANA identifier allows the time() call to automatically adjust the session's UTC offset based on a region's local time policies, such as daylight saving time.

Testing for changes in higher timeframes

The `timeframe` parameter of the time() and time_close() functions specifies the timeframe of the bars in the calculation, allowing scripts to retrieve opening/closing UNIX timestamps from *higher timeframes* than the current chart's timeframe without requiring `request.*()` function calls.

Programmers can use the opening/closing timestamps from higher-timeframe (HTF) bars to detect timeframe changes. One common approach is to call time() or time_close() with a consistent `timeframe` argument across all executions on a time-based chart and measure the one-bar change in the returned value with the ta.change() function. The result is a *nonzero* value only when an HTF bar opens. One can also check whether the data has a time gap at that point by comparing the time() value to the previous bar's

time_close() value. A gap is present when the opening timestamp on the current bar is greater than the closing timestamp on the previous bar.

The script below uses the call `time("1M")` to get the opening UNIX timestamp of the current bar on the “1M” timeframe, then assigns the result to the `currMonthlyOpenTime` variable. It detects when bars on that timeframe open by checking when the one-bar change in the `currMonthlyOpenTime` value is above zero. On each occurrence of the condition, the script detects whether the HTF bar opened after a gap by checking if the new opening timestamp is greater than the “1M” closing timestamp from the previous chart bar (`time_close("1M")[1]`).

The script draws labels containing formatted “1M” opening times to indicate the chart bars that mark the start of monthly bars. If a monthly bar opens without a gap from the previous closing time, the script draws a blue label. If a monthly bar starts after a gap, it draws a red label. Additionally, if the “1M” opening time does not match the opening time of the chart bar, the script displays that bar’s formatted time in the label for comparison:

image

//@version=6 indicator("Detecting changes in higher timeframes demo", overlay = true) //@variable The opening UNIX timestamp of the current bar on the "1M" timeframe. int currMonthlyOpenTime = time("1M") //@variable The closing timestamp on the "1M" timeframe as of the previous chart bar. int prevMonthlyCloseTime = time_close("1M")[1] //@variable Is `true` when the opening time on the "1M" timeframe changes, indicating a new monthly bar. bool isNewTf = ta.change(currMonthlyOpenTime) > 0 if isNewTf // Initialize variables for the `text` and `color` properties of the label drawing. string lblText = "New '1M' opening time:\n" + str.format_time(currMonthlyOpenTime) color lblColor = color.blue //@variable Is `true` when the `currMonthlyOpenTime` exceeds the `prevMonthlyCloseTime`, indicating a time gap. bool hasGap = currMonthlyOpenTime > prevMonthlyCloseTime // Modify the `lblText` and `lblColor` based on the `hasGap` value. if hasGap lblText := "Gap from previous '1M' close.\n\n" + lblText lblColor := color.red // Include the formatted `time` value if the `currMonthlyOpenTime` is before the first available chart bar's `time`. if time > currMonthlyOpenTime lblText += "\nFirst chart bar has a later time:\n" + str.format_time(time) // Draw a

`lblColor` label anchored to the `time` to display the `lblText`.
label.new(time, high, lblText, xloc.bar_time, color = lblColor, style = label.style_label_lower_right, textcolor = color.white, size = size.large)

Note that:

- Using ta.change() on a time() or time_close() call's result is not the *only* way to detect changes in a higher timeframe. The timeframe.change() function is an equivalent, more convenient option for scripts that do not need to *use* the UNIX timestamps from HTF bars in other calculations, as it returns a "bool" value directly without extra code.
- The detected monthly opening times do not always correspond to the first calendar day of the month. Instead, they correspond to the first time assigned to a "1M" bar, which can be *after* the first calendar day. For symbols with overnight sessions, such as "USDJPY" in our example chart, a "1M" bar can also open *before* the first calendar day.
- Sometimes, the opening time assigned to an HTF bar might *not* equal the opening time of any chart bar, which is why other conditions such as `time == time("1M")` cannot detect new monthly bars consistently. For example, on our "USDJPY" chart, the "1M" opening time `2023-12-31T17:00:00-0500` does not match an opening time on the "1D" timeframe. The first available "1D" bar after that point opened at `2024-01-01T17:00:00-0500`.

Calendar-based functions

The year(), month(), weekofyear(), dayofmonth(), dayofweek(), hour(), minute(), and second() functions calculate *calendar-based* "int" values from a UNIX timestamp. Unlike the calendar-based variables, which always hold exchange calendar values based on the current bar's opening timestamp, these functions can return calendar values for any valid timestamp and express them in a chosen time zone.

Each of these calendar-based functions has the following two signatures:

```
functionName(time) → series int
functionName(time, timezone) → series int
```

Where:

- `functionName` is the function's identifier.
- The `time` parameter accepts an "int" UNIX timestamp for which the function calculates a corresponding calendar value.
- The `timezone` parameter accepts a time zone string specifying the returned value's time zone. If the `timezone` argument is not specified, the function uses the exchange time zone (syminfo.timezone).

In contrast to the functions that return UNIX timestamps, a calendar-based function returns different "int" results for various time zones, as calendar values represent parts of a *local time* in a *specific region*.

For instance, the simple script below uses two calls to dayofmonth() to calculate each bar's opening day in the exchange time zone and the "Australia/Sydney" time zone. It plots the results of the two calls in a separate pane for comparison:

image

//@version=6 indicator("`dayofmonth()` demo", overlay = false) //@variable An "int" representing the current bar's opening calendar day in the exchange time zone. // Equivalent to the `dayofmonth` variable. int openingDay = dayofmonth(time) //@variable An "int" representing the current bar's opening calendar day in the "Australia/Sydney" time zone. int openingDaySydney = dayofmonth(time, "Australia/Sydney") // Plot the calendar day values. plot(openingDay, "Day of Month (Exchange)", linewidth = 6, color = color.blue) plot(openingDaySydney, "Day of Month (Sydney)", linewidth = 3, color = color.orange)

Note that:

- The first dayofmonth() call calculates the bar's opening day in the exchange time zone because it does not include a `timezone` argument. This call returns the same value that the dayofmonth variable references.
- Our example symbol's exchange time zone is "America/New_York", which follows UTC-5 during standard time and UTC-4 during daylight saving time (DST). The "Australia/Sydney" time zone follows UTC+10 during standard time and UTC+11 during DST. However, Sydney

observes DST at *different* times of the year than New York. As such, its time zone is 14, 15, or 16 hours ahead of the exchange time zone, depending on the time of year. The plots on our “1D” chart diverge when the difference is at least 15 hours because the bars open at 09:30 in exchange time, and 15 hours ahead is 00:30 on the *next* calendar day.

It’s important to understand that although the `time` argument in a calendar-based function call represents a single, absolute point in time, each function returns only *part* of the date and time information available from the timestamp. Consequently, a calendar-based function’s returned value does **not** directly correspond to a *unique* time point, and conditions based on individual calendar values can apply to *multiple* bars.

For example, this script uses the timestamp() function to calculate a UNIX timestamp from a date “string”, and it calculates the calendar day from that timestamp, in the exchange time zone, with the dayofmonth() function. The script compares each bar’s opening day to the calculated day and highlights the background when the two are equal:

image

//@version=6 indicator("`dayofmonth()` demo", overlay = false) //@variable The UNIX timestamp corresponding to August 29, 2024 at 00:00:00 UTC. const int fixedTimestamp = timestamp("29 Aug 2024") //@variable The day of the month calculated from the `fixedTimestamp`, expressed in the exchange time zone. // If the exchange time zone has a negative UTC offset, this variable's value is 28 instead of 29. int dayFromTimestamp = dayofmonth(fixedTimestamp) //@variable An "int" representing the current bar's opening calendar day in the exchange time zone. // Equivalent to the `dayofmonth` variable. int openingDay = dayofmonth(time) // Plot the `openingDay`. plot(openingDay, "Opening day of month", linewidth = 3) // Highlight the background when the `openingDay` equals the `dayFromTimestamp`. bgcolor(openingDay == dayFromTimestamp ? color.orange : na, title = "Day detected highlight")

Note that:

- The timestamp() call treats its argument as a *UTC* calendar date because its `dateString` argument does not specify time zone

information. However, the dayofmonth() call calculates the day in the *exchange time zone*. Our example symbol's exchange time zone is "America/New_York" (UTC-4/-5). Therefore, the returned value on this chart is 28 instead of 29.

- The script highlights *any* bar on our chart that opens on the 28th day of *any* month instead of only a specific bar because the dayofmonth() function's returned value does **not** represent a specific point in time on its own.
- This script highlights the bars that *open* on the day of the month calculated from the timestamp. However, some months on our chart have no trading activity on that day. For example, the script does not highlight when the July 28, 2024 occurs on our chart because NASDAQ is closed on Sundays.

Similar to calendar-based variables, these functions are also helpful when testing for dates/times and detecting calendar changes on the chart. The example below uses the year(), month(), weekofyear(), and dayofweek() functions on the time_close timestamp to create conditions that test if the current bar is the first bar that closes in a new year, quarter, month, week, and day. The script plots shapes, draws labels, and uses background colors to visualize the conditions on the chart:

image

//@version=6 indicator("Calendar changes demo", overlay = true, max_labels_count = 500) // Calculate the year, month, week of year, and day of week corresponding to the `time_close` UNIX timestamp. // All values are expressed in the exchange time zone. int closeYear = year(time_close) int closeMonth = month(time_close) int closeWeekOfYear = weekofyear(time_close) int closeDayOfWeek = dayofweek(time_close) // @variable Is `true` when the change in `closeYear` exceeds 0, marking the first bar that closes in a new year. bool closeInNewYear = ta.change(closeYear) > 0 //@variable Is `true` when the difference in `closeMonth` is not 0, marking the first bar that closes in a new month. bool closeInNewMonth = closeMonth - closeMonth[1] != 0 //@variable Is `true` when `closeMonth - 1` becomes divisible by 3, marking the first bar that closes in a new quarter. bool closeInNewQuarter = (closeMonth[1] - 1) % 3 != 0 and (closeMonth - 1) % 3 == 0 //@variable Is `true` when the change in `closeWeekOfYear` is not 0, marking the first bar that closes in a new week.

bool closeInNewWeek = ta.change(closeWeekOfYear) != 0 //@variable Is `true` when the `closeDayOfWeek` changes, marking the first bar that closes in the new day. bool closeInNewDay = closeDayOfWeek != closeDayOfWeek[1] //@variable Switches between `true` and `false` after every `closeInNewDay` occurrence for background color calculation. var bool alternateDay = true if closeInNewDay alternateDay := not alternateDay // Draw a label above the bar to display the `closeWeekOfYear` when `closeInNewWeek` is `true`. if closeInNewWeek label.new(time, 0, "W" + str.tostring(closeWeekOfYear), xloc.bar_time, yloc.abovebar, color.purple, textcolor = color.white, size = size.normal) // Plot label shapes at the bottom and top of the chart for the `closeInNewYear` and `closeInNewMonth` conditions. plotshape(closeInNewYear, "Close in new year", shape.labelup, location.bottom, color.teal, text = "New year", textcolor = color.white, size = size.huge) plotshape(closeInNewMonth, "Close in new month", shape.labeldown, location.top, text = "New month", textcolor = color.white, size = size.large) // Plot a triangle below the chart bar when `closeInNewQuarter` occurs. plotshape(closeInNewQuarter, "Close in new quarter", shape.triangleup, location.belowbar, color.maroon, text = "New quarter", textcolor = color.maroon, size = size.large) // Highlight the background in alternating colors based on occurrences of `closeInNewDay`. bgcolor(alternateDay ? color.new(color.aqua, 80) : color.new(color.fuchsia, 80), title = "Closing day change")

Note that:

- This script's conditions check for the first bar that closes after each calendar unit changes its value. The bar where each condition is `true` varies with the data available on the chart. For example, the `closeInNewMonth` condition can be `true` *after* the first calendar day of the month if a chart bar did not close on that day.
- To detect when new bars start on a specific *timeframe* rather than strictly calendar changes, check when the ta.change() of a time() or time_close() call's returned value is nonzero, or use the timeframe.change() function. See this section above for more information.

timestamp()

The timestamp() function calculates a UNIX timestamp from a specified calendar date and time. It has the following three signatures:

```
timestamp(year, month, day, hour, minute, second) → simple/series int
timestamp(timezone, year, month, day, hour, minute, second) → simple/s
timestamp(dateString) → const int
```

The first two signatures listed include `year`, `month`, `day`, `hour`, `minute`, and `second` parameters that accept "int" values defining the calendar date and time. A timestamp() call with either signature must include `year`, `month`, and `day` arguments. The other parameters are optional, each with a default value of 0. Both signatures can return either *"simple"* or *"series"* values, depending on the qualified types of the specified arguments.

The primary difference between the first two signatures is the `timezone` parameter, which accepts a time zone string that determines the time zone of the *date and time* specified by the other parameters. If a timestamp() call with "int" calendar arguments does not include a `timezone` argument, it uses the exchange time zone (syminfo.timezone) by default.

The third signature listed has only *one* parameter, `dateString`, which accepts a "string" representing a valid calendar date (e.g., `"20 Aug 2024"`). The value can also include the time of day and time zone (e.g., `"20 Aug 2024 00:00:00 UTC+0"`). If the `dateString` argument does not specify the time of day, the timestamp() call considers the time 00:00 (midnight).

Unlike the other two signatures, the default time zone for the third signature is **GMT+0**. It does **not** use the exchange time zone by default because it interprets time zone information from the `dateString` directly. Additionally, the third signature is the only one that returns a *"const int"* value. As shown in the Time input section of the Inputs page, programmers can use this overload's returned value as the `defval` argument in an input.time() function call.

When using the timestamp() function, it's crucial to understand how time zone information affects its calculations. The *absolute* point in time represented by a specific calendar date *depends* on its time zone, as an identical date and time in various time zones can refer to **different** amounts

of time elapsed since the *UNIX Epoch*. Therefore, changing the time zone of the calendar date and time in a timestamp() call *can change* its returned UNIX timestamp.

The following script compares the results of four different timestamp() calls that evaluate the date 2021-01-01 in different time zones. The first timestamp() call does not specify time zone information in its `dateString` argument, so it treats the value as a *UTC* calendar date. The fourth call also evaluates the calendar date in UTC because it includes `"UTC0"` as the `timezone` argument. The second timestamp() call uses the first signature listed above, meaning it uses the exchange time zone, and the third call uses the second signature with `"America/New_York"` as the `timezone` argument.

The script draws a table with rows displaying each timestamp() call, its assigned variable, the calculated UNIX timestamp, and a formatted representation of the time. As we see on the “NASDAQ:MSFT” chart below, the first and fourth table rows show *different* timestamps than the second and third, leading to different formatted strings in the last column:

image

//@version=6 indicator("`timestamp()` demo", overlay = false) //@variable A `table` that displays the different `timestamp()` calls, their returned timestamps, and formatted results. var table displayTable = table.new(position.middle_center, 4, 5, color.white, border_color = color.black, border_width = 2) //@function Initializes a `displayTable` cell showing the `displayText` with an optional `specialFormat`. printCell(int colID, int rowID, string displayText, string specialFormat = "") => displayTable.cell(colID, rowID, displayText, text_size = size.large) switch specialFormat "header" => displayTable.cell_set_bgcolor(colID, rowID, color.rgb(76, 175, 79, 70)) "code" => displayTable.cell_set_text_font_family(colID, rowID, font.family_monospace) displayTable.cell_set_text_size(colID, rowID, size.normal) displayTable.cell_set_text_halign(colID, rowID, text.align_left) if barstate.islastconfirmedhistory //@variable The UNIX timestamp corresponding to January 1, 2021 in the UTC+0 time zone. int dateTimestamp1 = timestamp("2021-01-01") //@variable The UNIX timestamp corresponding to January 1, 2021 in the exchange time zone. int dateTimestamp2 = timestamp(2021, 1, 1, 0, 0) //@variable The UNIX

timestamp corresponding to January 1, 2021 in the "America/New_York" time zone. int dateTimestamp3 = timestamp("America/New_York", 2021, 1, 1, 0, 0) //@variable The UNIX timestamp corresponding to January 1, 2021 in the "UTC0" (UTC+0) time zone. int dateTimestamp4 = timestamp("UTC0", 2021, 1, 1, 0, 0) // Initialize the top header cells in the `displayTable`. printCell(0, 0, "Variable", "header") printCell(1, 0, "Function call", "header") printCell(2, 0, "Timestamp returned", "header") printCell(3, 0, "Formatted date/time", "header") // Initialize a table row for `dateTimestamp1` results. printCell(0, 1, "`dateTimestamp1`", "header") printCell(1, 1, "`timestamp(\"2021-01-01\")`", "code") printCell(2, 1, str.tostring(dateTimestamp1)) printCell(3, 1, str.format_time(dateTimestamp1, "yyyy.MM.dd HH:mm (Z)")) // Initialize a table row for `dateTimestamp2` results. printCell(0, 2, "`dateTimestamp2`", "header") printCell(1, 2, "`timestamp(2021, 1, 1, 0, 0)`", "code") printCell(2, 2, str.tostring(dateTimestamp2)) printCell(3, 2, str.format_time(dateTimestamp2, "yyyy.MM.dd HH:mm (Z)")) // Initialize a table row for `dateTimestamp3` results. printCell(0, 3, "`dateTimestamp3`", "header") printCell(1, 3, "`timestamp(\"America/New_York\", 2021, 1, 1, 0, 0)`", "code") printCell(2, 3, str.tostring(dateTimestamp3)) printCell(3, 3, str.format_time(dateTimestamp3, "yyyy.MM.dd HH:mm (Z)")) // Initialize a table row for `dateTimestamp4` results. printCell(0, 4, "`dateTimestamp4`", "header") printCell(1, 4, "`timestamp(\"UTC0\", 2021, 1, 1, 0, 0)`", "code") printCell(2, 4, str.tostring(dateTimestamp4)) printCell(3, 4, str.format_time(dateTimestamp4, "yyyy.MM.dd HH:mm (Z)"))

Note that:

- The formatted date-time strings express results in the exchange time zone because the str.format_time() function uses syminfo.timezone as the default `timezone` argument. The formatted values on our example chart show the offset string `"-0500"` because NASDAQ's time zone ("America/New_York") follows UTC-5 during *standard time*.
- The formatted strings on the first and fourth rows show the date and time five hours *before* January 1, 2021, because the timestamp() calls evaluated the date in *UTC* and the str.format_time() calls used a time zone five hours *behind* UTC.
- On our chart, the second and third rows have matching timestamps because both corresponding timestamp() calls evaluated the date in the "America/New_York" time zone. The two rows would show different

results if we applied the script to a symbol with a different exchange time zone.

Formatting dates and times

Programmers can format UNIX timestamps into human-readable dates and times, expressed in specific time zones, using the str.format_time() function. The function has the following signature:

```
str.format_time(time, format, timezone) → series string
```

Where:

- The `time` parameter specifies the "int" UNIX timestamp to express as a readable time.
- The `format` parameter accepts a "string" consisting of *formatting tokens* that determine the returned information. If the function call does not include a `format` argument, it uses the ISO 8601 standard format: `"yyyy-MM-dd'T'HH:mm:ssZ"`. See the table below for a list of valid tokens and the information they represent.
- The `timezone` parameter determines the time zone of the formatted result. It accepts a time zone string in UTC or IANA notation. If the call does not specify a `timezone`, it uses the exchange time zone (syminfo.timezone).

The general-purpose str.format() function can also format UNIX timestamps into readable dates and times. However, the function **cannot** express time information in *different* time zones. It always expresses dates and times in **UTC+0**. In turn, using this function to format timestamps often results in *erroneous* practices, such as mathematically modifying a timestamp to try and represent the time in another time zone. However, a UNIX timestamp is a unique, **time zone-agnostic** representation of a specific point in time. As such, modifying a UNIX timestamp changes the *absolute time* it represents rather than expressing the same time in a different time zone.

The str.format_time() function does not have this limitation, as it can calculate dates and times in *any* time zone correctly without changing the meaning of a UNIX timestamp. In addition, unlike str.format(), it is optimized specifically for processing time values. Therefore, we recommend that

programmers use str.format_time() instead of str.format() to format UNIX timestamps into readable dates and times.

A str.format_time() call's `format` argument determines the time information its returned value contains. The function treats characters and sequences in the argument as *formatting tokens*, which act as *placeholders* for values in the returned date/time "string". The following table outlines the most commonly used formatting tokens and explains what each represents:

Token	Represents	Remarks and examples
`"y"`	Year	Use `"yy"` to include the last two digits of the year (e.g., `"00"`), or `"yyyy"` to include the complete year number (e.g., `"2000"`).
`"M"`	Month	Uppercase `"M"` for the month, not to be confused with *lowercase* `"m"` for the minute. Use `"MM"` to include the two-digit month number with a leading zero for single-digit values (e.g., `"01"`), `"MMM"` to include the three-letter abbreviation of month (e.g., `"Jan"`), or `"MMMM"` for the full month name (e.g., `"January"`).
`"d"`	Day of the **month**	Lowercase `"d"`. Includes the numeric day of the month (`"1"` to `"31"`). Use `"dd"` for the two-digit day number with a leading zero for single-digit values. It is *not* a placeholder for the day number of the *week* (1-7). Use dayofweek() to calculate that value.
`"D"`	Day of the **year**	Uppercase `"D"`. Includes the numeric day of the year (`"1"` to `"366"`). Use `"DD"` or `"DDD"` for the two-digit or three-digit day number with leading zeros.
`"E"`	Day of the **week**	Includes the abbreviation of the weekday *name* (e.g., `"Mon"`).

Token	Represents	Remarks and examples
		Use `"EEEE"` for the weekday's full name (e.g., `"Monday"`)
`"w"`	Week of the **year**	Lowercase `"w"`. Includes the week number of the year (`"1"` to `"53"`). Use `"ww"` for the two-digit week number with a leading zero for single-digit values.
`"W"`	Week of the **month**	Uppercase `"W"`. Includes the week number of the month (`"1"` to `"5"`).
`"a"`	AM/PM postfix	Lowercase `"a"`. Includes `"AM"` if the time of day is before noon, `"PM"` otherwise.
`"h"`	Hour in the **12-hour** format	Lowercase `"h"`. The included hour number from this token ranges from `"0"` to `"11"`. Use `"hh"` for the two-digit hour with a leading zero for single-digit values.
`"H"`	Hour in the **24-hour** format	Uppercase `"H"`. The included hour number from this token ranges from `"0"` to `"23"`. Use `"HH"` for the two-digit hour with a leading zero for single-digit values.
`"m"`	Minute	Lowercase `"m"` for the minute, not to be confused with *uppercase* `"M"` for the month. Use `"mm"` for the two-digit minute with a leading zero for single-digit values.
`"s"`	Second	Lowercase `"s"` for the second, not to be confused with *uppercase* `"S"` for fractions of a second. Use `"ss"` for the two-digit second with a leading zero for single-digit values.
`"S"`	Fractions of a second	Uppercase `"S"`. Includes the number of milliseconds in the fractional second (`"0"` to `"999"`).

Token	Represents	Remarks and examples
		Use `"SS"` or `"SSS"` for the two-digit or three-digit millisecond number with leading zeros.
`"Z"`	Time zone (**UTC offset**)	Uppercase `"Z"`. Includes the hour and minute UTC offset value in `"HHmm"` format, preceded by its sign (e.g., `"-0400"`).
`"z"`	Time zone (**abbreviation or name**)	Lowercase `"z"`. A single `"z"` includes the abbreviation of the time zone (e.g., `"EDT"`). Use `"zzzz"` for the time zone's name (e.g., `"Eastern Daylight Time"`). It is not a placeholder for the *IANA identifier*. Use syminfo.timezone to retrieve the exchange time zone's IANA representation.
`":"`, `"/"`, `"-"`, `"."`, `","`, `"("`, `")"`, `" "`	Separators	These characters are separators for formatting tokens. They appear as they are in the formatted text. (e.g., `"01/01/24"`, `"12:30:00"`, `"Jan 1, 2024"`). Some other characters can also act as separators. However, the ones listed are the most common.
`"'"`	Escape character	Characters enclosed within *two single quotes* appear as they are in the result, even if they otherwise act as formatting tokens. For example, `" 'Day' "` appears as-is in the resulting "string" instead of listing the day of the year, AM/PM postfix, and year.

The following example demonstrates how various formatting tokens affect the str.format_time() function's result. The script calls the function with different `format` arguments to create date/time strings from time, timenow, and time_close timestamps. It displays each `format` value and the corresponding formatted result in a table on the last bar:

image

```
//@version=6 indicator("Formatting dates and times demo", overlay = false) //@variable A `table` that displays different date/time `format` strings and their results. var table displayTable = table.new( position.middle_center, 2, 15, bgcolor = color.white, frame_color = color.black, frame_width = 1, border_width = 1 ) //@function Initializes a `displayTable` row showing a `formatString` and its formatted result for a specified // `timeValue` and `timezoneValue`. displayText(rowID, formatString, timeValue = time, timezoneValue = syminfo.timezone) => //@variable Is light blue if the `rowID` is even, white otherwise. Used to set alternating table row colors. color rowColor = rowID % 2 == 0 ? color.rgb(33, 149, 243, 75) : color.white // Display the `formatString` in the row's first cell. displayTable.cell( 0, rowID, formatString, text_color = color.black, text_halign = text.align_left, bgcolor = rowColor ) // Show the result of formatting the `timeValue` based on the `formatString` and `timezoneValue` in the second cell. displayTable.cell( 1, rowID, str.format_time(timeValue, formatString, timezoneValue), text_color = color.black, text_halign = text.align_right, bgcolor = rowColor ) if barstate.islast // Initialize the table's header cells. displayTable.cell(0, 0, "FORMAT STRINGS") displayTable.cell(1, 0, "FORMATTED DATE/TIME OUTPUT") // Initialize a row to show the default date-time "string" format and its result for `time`. displayTable.cell( 0, 1, "(Default `str.format_time()` format)", text_color = color.black, text_halign = text.align_left, bgcolor = color.yellow) displayTable.cell( 1, 1, str.format_time(time), text_color = color.black, text_halign = text.align_right, bgcolor = color.yellow) // Initialize rows to show different formatting strings and their results for `time`, `time_close`, and `timenow`. displayText(2, "dd/MM/yy") displayText(3, "MMMM dd, yyyy") displayText(4, "hh:mm:ss.SS a", timenow) displayText(5, "HH:mm 'UTC'Z") displayText(6, "H:mm a (zzzz)") displayText(7, "my day / 'my day' ('escaped')") displayText(8, "'Month' M, 'Week' w, 'Day' DDD") displayText(9, "'Bar expected closing time': ha", time_close) displayText(10, "'Current date/time': MMM-d-y HH:mm:ss z", timenow) displayText(11, "'New Time zone': zzzz", timezoneValue = "Australia/Sydney") displayText(12, "'Time zone change': MMM-d-y HH:mm:ss z", timenow, "Australia/Sydney")
```

Expressing time differences

Every UNIX timestamp represents a specific point in time as the absolute *time difference* from a fixed historical point (epoch). The specific epoch all

UNIX timestamps reference is *midnight UTC on January 1, 1970*. Programmers can format UNIX timestamps into readable date-time strings with the str.format_time() function because it uses the time difference from the UNIX Epoch in its date and time calculations.

In contrast, the difference between two nonzero UNIX timestamps represents the number of milliseconds elapsed from one absolute point to another. The difference does not directly refer to a specific point in UNIX time if neither timestamp in the operation has a value of 0 (corresponding to the UNIX Epoch).

Programmers may want to express the millisecond difference between two UNIX timestamps in *other time units*, such as seconds, days, etc. Some might assume they can use the difference as the `time` argument in a str.format_time() call to achieve this result. However, the function always treats its `time` argument as the time elapsed from the *UNIX Epoch* to derive a *calendar date/time* representation in a specific time zone. It **does not** express time differences directly. Therefore, attempting to format timestamp *differences* rather than timestamps with str.format_time() leads to unintended results.

For example, the following script calculates the millisecond difference between the current execution time (timenow) and the “1M” bar’s closing time (`time_close("1M")`) for a monthly countdown timer display. It attempts to express the time difference in another format using str.format_time(). It displays the function call’s result in a table, along with the original millisecond difference (`timeLeft`) and formatted date-time representations of the timestamps.

As we see below, the table shows correct results for the formatted timestamps and the `timeLeft` value. However, the formatted time difference appears as `"1970-01-12T16:47:10-0500"`. Although the `timeLeft` value is supposed to represent a difference between timestamps rather than a specific point in time, the str.format_time() function still treats the value as a **UNIX timestamp**. Consequently, it creates a “string” expressing the value as a *date and time* in the UTC-5 time zone:

image

```
//@version=6 indicator("Incorrectly formatting time difference demo", overlay = true) //@variable A table that displays monthly close countdown information. var table displayTable = table.new(position.top_right, 1, 4, color.rgb(0, 188, 212, 60)) if barstate.islast //@variable A UNIX timestamp representing the current time as of the script's latest execution. int currentTime = timenow //@variable A UNIX timestamp representing the expected closing time of the current "1M" bar. int monthCloseTime = time_close("1M") //@variable The number of milliseconds between the `currentTime` and the `monthCloseTime`. // This value is NOT intended as a UNIX timestamp. int timeLeft = monthCloseTime - currentTime //@variable A "string" representing the `timeLeft` as a date and time in the exchange time zone, in ISO 8601 format. // This format is INCORRECT for the `timeLeft` value because it's supposed to represent the time between // two nonzero UNIX timestamps, NOT a specific point in time. string incorrectTimeFormat = str.format_time(timeLeft) // Initialize `displayTable` cells to initialize the `currentTime` and `monthCloseTime`. displayTable.cell( 0, 0, "Current time: " + str.format_time(currentTime, "HH:mm:ss.S dd/MM/yy (z)"), text_size = size.large, text_halign = text.align_right ) displayTable.cell( 0, 1, "`1M` Bar closing time: " + str.format_time(monthCloseTime, "HH:mm:ss.SS dd/MM/yy (z)"), text_size = size.large, text_halign = text.align_right ) // Initialize a cell to display the `timeLeft` millisecond difference. displayTable.cell( 0, 2, "`timeLeft` value: " + str.tostring(timeLeft), text_size = size.large, bgcolor = color.yellow ) // Initialize a cell to display the `incorrectTimeFormat` representation. displayTable.cell( 0, 3, "Time left (incorrect format): " + incorrectTimeFormat, text_size = size.large, bgcolor = color.maroon, text_color = color.white )
```

To express the difference between timestamps in other time units correctly, programmers must write code that *calculates* the number of units elapsed instead of erroneously formatting the difference as a specific date or time.

The calculations required to express time differences depend on the chosen time units. The sections below explain how to express millisecond differences in weekly and smaller units, and monthly and larger units.

Weekly and smaller units

Weeks and smaller time units (days, hours, minutes, seconds, and milliseconds) cover *consistent* blocks of time. These units have the following relationship:

- One week equals seven days.
- One day equals 24 hours.
- One hour equals 60 minutes.
- One minute equals 60 seconds.
- One second equals 1000 milliseconds.

Using this relationship, programmers can define the span of these units by the number of *milliseconds* they contain. For example, since every hour has 60 minutes, every minute has 60 seconds, and every second has 1000 milliseconds, the number of milliseconds per hour is `60 * 60 * 1000`, which equals `3600000`.

Programmers can use *modular arithmetic* based on the milliseconds in each unit to calculate the total number of weeks, days, and smaller spans covered by the difference between two UNIX timestamps. The process is as follows, starting from the *largest* time unit in the calculation:

1. Calculate the number of milliseconds in the time unit.
2. Divide the remaining millisecond difference by the calculated value and round down to the nearest whole number. The result represents the number of *complete* time units within the interval.
3. Use the *remainder* from the division as the new remaining millisecond difference.
4. Repeat steps 1-3 for each time unit in the calculation, in *descending* order based on size.

The following script implements this process in a custom `formatTimeSpan()` function. The function accepts two UNIX timestamps defining a start and end point, and its “bool” parameters control whether it calculates the number of weeks or smaller units covered by the time range. The function calculates the millisecond distance between the two timestamps. It then calculates the numbers of complete units covered by that distance and formats the results into a “string”.

The script calls `formatTimeSpan()` to express the difference between two separate time input values in selected time units. It then displays the resulting “string” in a table alongside formatted representations of the start and end times:

image

//@version=6 indicator("Calculating time span demo") // Assign the number of milliseconds in weekly and smaller units to "const" variables for convenience. const int ONE_WEEK = 604800000 const int ONE_DAY = 86400000 const int ONE_HOUR = 3600000 const int ONE_MINUTE = 60000 const int ONE_SECOND = 1000 //@variable A UNIX timestamp calculated from the user-input start date and time. int startTimeInput = input.time(timestamp("1 May 2022 00:00 -0400"), "Start date and time", group = "Time between") //@variable A UNIX timestamp calculated from the user-input end date and time. int endTimeInput = input.time(timestamp("7 Sep 2024 20:37 -0400"), "End date and time", group = "Time between") // Create "bool" inputs to toggle weeks, days, hours, minutes, seconds, and milliseconds in the calculation. bool weeksInput = input.bool(true, "Weeks", group = "Time units", inline = "A") bool daysInput = input.bool(true, "Days", group = "Time units", inline = "A") bool hoursInput = input.bool(true, "Hours", group = "Time units", inline = "B") bool minutesInput = input.bool(true, "Minutes", group = "Time units", inline = "B") bool secondsInput = input.bool(true, "Seconds", group = "Time units", inline = "B") bool millisecondsInput = input.bool(true, "Milliseconds", group = "Time units", inline = "B") //@function Calculates the difference between two UNIX timestamps as the number of complete time units in // descending order of size, formatting the results into a "string". The "int" parameters accept timestamps, // and the "bool" parameters determine which units the function uses in its calculations. formatTimeSpan(int startTimestamp, int endTimestamp, bool calculateWeeks, bool calculateDays, bool calculateHours, bool calculateMinutes, bool calculateSeconds, bool calculateMilliseconds) => //@variable The milliseconds between the `startTimestamp` and `endTimestamp`. int timeDifference = math.abs(endTimestamp - startTimestamp) //@variable A "string" representation of the interval in mixed time units. string formattedString = na // Calculate complete units within the interval for each toggled unit, reducing the `timeDifference` by those sizes. if calculateWeeks int

totalWeeks = math.floor(timeDifference / ONE_WEEK) timeDifference %= ONE_WEEK formattedString += str.tostring(totalWeeks) + (totalWeeks == 1 ? " week " : " weeks ") if calculateDays int totalDays = math.floor(timeDifference / ONE_DAY) timeDifference %= ONE_DAY formattedString += str.tostring(totalDays) + (totalDays == 1 ? " day " : " days ") if calculateHours int totalHours = math.floor(timeDifference / ONE_HOUR) timeDifference %= ONE_HOUR formattedString += str.tostring(totalHours) + (totalHours == 1 ? " hour " : " hours ") if calculateMinutes int totalMinutes = math.floor(timeDifference / ONE_MINUTE) timeDifference %= ONE_MINUTE formattedString += str.tostring(totalMinutes) + (totalMinutes == 1 ? " minute " : " minutes ") if calculateSeconds int totalSeconds = math.floor(timeDifference / ONE_SECOND) timeDifference %= ONE_SECOND formattedString += str.tostring(totalSeconds) + (totalSeconds == 1 ? " second " : " seconds ") if calculateMilliseconds // `timeDifference` is in milliseconds already, so add it to the `formattedString` directly. formattedString += str.tostring(timeDifference) + (timeDifference == 1 ? " millisecond" : " milliseconds") // Return the `formattedString`. formattedString if barstate.islastconfirmedhistory //@variable A table that that displays formatted start and end times and their custom-formatted time difference. var table displayTable = table.new(position.middle_center, 1, 2, color.aqua) // @variable A "string" containing formatted `startTimeInput` and `endTimeInput` values. string timeText = "Start date and time: " + str.format_time(startTimeInput, "dd/MM/yy HH:mm:ss (z)") + "\n End date and time: " + str.format_time(endTimeInput, "dd/MM/yy HH:mm:ss (z)") // @variable A "string" representing the span between `startTimeInput` and `endTimeInput` in mixed time units. string userTimeSpan = formatTimeSpan(startTimeInput, endTimeInput, weeksInput, daysInput, hoursInput, minutesInput, secondsInput, millisecondsInput) // Display the `timeText` in the table. displayTable.cell(0, 0, timeText, text_color = color.white, text_size = size.large, text_halign = text.align_left) // Display the `userTimeSpan` in the table. displayTable.cell(0, 1, "Time span: " + userTimeSpan, text_color = color.white, text_size = size.large, text_halign = text.align_left, bgcolor = color.navy)

Note that:

- The user-defined function uses math.floor() to round each divided result down to the nearest “int” value to get the number of *complete* units in the interval. After division, it uses the modulo assignment operator (%=) to get the *remainder* and assign that value to the `timeDifference` variable. This process repeats for each selected unit.

The image above shows the calculated time difference in mixed time units. By toggling the “bool” inputs, users can also isolate specific units in the calculation. For example, this image shows the result after enabling only the “Milliseconds” input:

image

Monthly and larger units

Unlike weeks and smaller units, months and larger units *vary* in length based on calendar rules. For example, a month can contain 28, 29, 30, or 31 days, and a year can contain 365 or 366 days.

Some programmers prefer to use the modular arithmetic outlined in the previous section, with *approximate lengths* for these irregular units, to calculate large-unit durations between UNIX timestamps. With this process, programmers usually define the units in either of the following ways:

- Using *common* lengths, e.g., a common year equals 365 days, and a common month equals 30 days.
- Using the *average* lengths, e.g., an average year equals 365.25 days, and an average month equals 30.4375 days.

Calculations involving approximate units produce *rough estimates* of the elapsed time. Such estimates are often practical when expressing relatively short durations. However, their precision diminishes with the size of the difference, drifting further away from the actual time elapsed.

Therefore, expressing time differences in monthly and larger units with precision requires a different calculation than the process outlined above. For a more precise estimate of months, years, and larger units elapsed, the calculations should use the *actual* span of each individual unit rather than

approximations, meaning it must account for *leap years* and *variations* in month sizes.

The advanced example below contains a custom `formatTimeDifference()` function that calculates the years and months, in addition to days and smaller units, elapsed between two UNIX timestamps.

The function uses the process outlined in the previous section to calculate the daily and smaller units within the interval. For the monthly and yearly units, which have *irregular* lengths, the function uses a while loop to iterate across calendar months. On each iteration, it increments monthly and yearly counters and subtracts the number of days in the added month from the day counter. After the loop ends, the function adjusts the year, month, and day counters to account for partial months elapsed between the timestamps. Finally, it uses the counters in a str.format() call to create a formatted "string" containing the calculated values.

The script calls this `formatTimeDifference()` function to calculate the years, months, days, hours, minutes, seconds, and milliseconds elapsed between two separate time input values and displays the result in a label:

image

//@version=6 indicator("Calculating time span for larger units demo") // @variable The starting date and time of the time span, input by the user. int startTimeInput = input.time(timestamp("3 Apr 2022 20:00 -0400"), "Start date", group = "Time between") //@variable The ending date and time of the time span, input by the user. int endTimeInput = input.time(timestamp("3 Sep 2024 15:45 -0400"), "End date", group = "Time between") //@function Returns the number of days in the `monthNumber` month of the `yearNumber` year. daysPerMonth(int yearNumber, int monthNumber) => // @variable Is `true` if the `yearNumber` represents a leap year. bool leapYear = (yearNumber % 4 == 0 and yearNumber % 100 != 0) or (yearNumber % 400 == 0) //@variable The number of days calculated for the month. int result = switch monthNumber == 2 => leapYear ? 29 : 28 => 31 - (monthNumber - 1) % 7 % 2 //@function Calculates the relative time difference between two timestamps, covering monthly and larger units. formatTimeDifference(int timestamp1, int timestamp2) => // The starting time and ending time. int startTime = math.min(timestamp1, timestamp2),

int endTime = math.max(timestamp1, timestamp2) // The year, month, and day of the `startTime` and `endTime`. int startYear = year(startTime), int startMonth = month(startTime), int startDay = dayofmonth(startTime) int endYear = year(endTime), int endMonth = month(endTime), int endDay = dayofmonth(endTime) // Calculate the total number of days, hours, minutes, seconds, and milliseconds in the interval. int milliseconds = endTime - startTime int days = math.floor(milliseconds / 86400000), milliseconds %= 86400000 int hours = math.floor(milliseconds / 3600000), milliseconds %= 3600000 int minutes = math.floor(milliseconds / 60000), milliseconds %= 60000 int seconds = math.floor(milliseconds / 1000), milliseconds %= 1000 // Calculate the number of days in the `startMonth` and `endMonth`. int daysInStartMonth = daysPerMonth(startYear, startMonth), int daysInEndMonth = daysPerMonth(endYear, endMonth) //@variable The number of days remaining in the `startMonth`. int remainingInMonth = daysInStartMonth - startDay + 1 // Subtract `remainingInMonth` from the `days`, and offset the `startDay` and `startMonth`. days -= remainingInMonth, startDay := 1, startMonth += 1 // Set `startMonth` to 1, and increase the `startYear` if the `startMonth` exceeds 12. if startMonth > 12 startMonth := 1, startYear += 1 // Initialize variables to count the total number of months and years in the interval. int months = 0, int years = 0 // Loop to increment `months` and `years` values based on the `days`. while days > 0 //@variable The number of days in the current `startMonth`. int daysInMonth = daysPerMonth(startYear, startMonth) // Break the loop if the number of remaining days is less than the `daysInMonth`. if days < daysInMonth break // Reduce the `days` by the `daysInMonth` and increment the `months`. days -= daysInMonth, months += 1 // Increase the `years` and reset the `months` to 0 when `months` is 12. if months == 12 months := 0, years += 1 // Increase the `startMonth` and adjust the `startMonth` and `startYear` if its value exceeds 12. startMonth += 1 if startMonth > 12 startMonth := 1, startYear += 1 // Re-add the `remainingInMonth` value to the number of `days`. Adjust the `days`, `months`, and `years` if the // new value exceeds the `daysInStartMonth` or `daysInEndMonth`, depending on the `startDay`. days += remainingInMonth if days >= (startDay < daysInStartMonth / 2 ? daysInStartMonth : daysInEndMonth) months += 1 if months == 12 months := 0, years += 1 days -= remainingInMonth // Format the calculated values into a "string" and return the result. str.format(" Years: {0}\n Months: {1}\n Days: {2}\n Hours: {3}\n Minutes: {4}\n

Seconds: {5} \n Milliseconds: {6}", years, months, days, hours, minutes, seconds, milliseconds) if barstate.islastconfirmedhistory //@variable A "string" representing the time between the `startTimeInput` and `endTimeInput` in mixed units. string userTimeSpan = formatTimeDifference(startTimeInput, endTimeInput) //@variable Text shown in the label. string labelText = "Start time: " + str.format_time(startTimeInput, "dd/MM/yy HH:mm:ss (z)") + "\n End time: " + str.format_time(endTimeInput, "dd/MM/yy HH:mm:ss (z)") + "\n---------\nTime difference:\n" + userTimeSpan label.new(bar_index, high, labelText, color = #FF946E, size = size.large, textalign = text.align_left, style = label.style_label_center)

Note that:

- The script determines the number of days in each month with the user-defined `daysPerMonth()` function. The function identifies whether a month has 28, 29, 30, or 31 days based on its month number and the year it belongs to. Its calculation accounts for leap years. A leap year occurs when the year is divisible by 4 or 400 but not by 100.
- Before the while loop, the function subtracts the number of days in a partial starting month from the initial day count, aligning the counters with the beginning of a new month. It re-adds the subtracted days after the loop to adjust the counters for partial months. It adjusts the month and year counters based on the days in the `startMonth` if the `startDay` is less than halfway through that month. Otherwise, it adjusts the values based on the days in the `endMonth`.

Previous

Next

Timeframes

Pine Script® v6 Documentation

Timeframes

Introduction

The *timeframe* of a chart is sometimes also referred to as its *interval* or *resolution*. It is the unit of time represented by one bar on the chart. All standard chart types use a timeframe: “Bars”, “Candles”, “Hollow Candles”, “Line”, “Area” and “Baseline”. One non-standard chart type also uses timeframes: “Heikin Ashi”.

Programmers interested in accessing data from multiple timeframes will need to become familiar with how timeframes are expressed in Pine Script®, and how to use them.

Timeframe strings come into play in different contexts:

- They must be used in request.security() when requesting data from another symbol and/or timeframe. See the page on Other timeframes and data to explore the use of request.security().
- They can be used as an argument to time() and time_close() functions, to return the time of a higher timeframe bar. This, in turn, can be used to detect changes in higher timeframes from the chart’s timeframe without using request.security(). See the Testing for changes in higher timeframes section to see how to do this.
- The input.timeframe() function provides a way to allow script users to define a timeframe through a script’s “Inputs” tab (see the Timeframe input section for more information).
- The indicator() declaration statement has an optional `timeframe` parameter that can be used to provide multi-timeframe capabilities to simple scripts without using request.security().

- Many built-in variables provide information on the timeframe used by the chart the script is running on. See the Chart timeframe section for more information on them, including timeframe.period which returns a string in Pine Script's timeframe specification format.

Timeframe string specifications

Timeframe strings follow these rules:

- They are composed of the multiplier and the timeframe unit, e.g., "1S", "30" (30 minutes), "1D" (one day), "3M" (three months).
- The unit is represented by a single letter, with no letter used for minutes: "T" for ticks, "S" for seconds, "D" for days, "W" for weeks, and "M" for months.
- When no multiplier is used, 1 is assumed: "S" is equivalent to "1S", "D" to "1D", etc. If only "1" is used, it is interpreted as 1 minute, since no unit letter identifier is used for minutes.
- There is no "hour" unit; "1H" is **not** valid. The correct format for one hour is "60" (remember no unit letter is specified for minutes).
- The valid multipliers vary for each timeframe unit:
    - For ticks, only the discrete 1, 10, 100, and 1000 multipliers are valid.
    - For seconds, only the discrete 1, 5, 10, 15, 30, and 45 multipliers are valid.
    - For minutes, 1 to 1440.
    - For days, 1 to 365.
    - For weeks, 1 to 52.
    - For months, 1 to 12.

Comparing timeframes

It can be useful to compare different timeframe strings to determine, for example, if the timeframe used on the chart is lower than the higher timeframes used in the script.

Converting timeframe strings to a representation in fractional minutes provides a way to compare them using a universal unit. This script uses the timeframe.in_seconds() function to convert a timeframe into float seconds and then converts the result into minutes:

//@version=6 indicator("Timeframe in minutes example", "", true) string tfInput = input.timeframe(defval = "", title = "Input TF") float chartTFInMinutes = timeframe.in_seconds() / 60 float inputTFInMinutes = timeframe.in_seconds(tfInput) / 60 var table t = table.new(position.top_right, 1, 1) string txt = "Chart TF: " + str.tostring(chartTFInMinutes, "#.##### minutes") + "\nInput TF: " + str.tostring(inputTFInMinutes, "#.##### minutes") if barstate.isfirst table.cell(t, 0, 0, txt, bgcolor = color.yellow) else if barstate.islast table.cell_set_text(t, 0, 0, txt) if chartTFInMinutes > inputTFInMinutes runtime.error("The chart's timeframe must not be higher than the input's timeframe.")

Note that:

- We use the built-in timeframe.in_seconds() function to convert the chart timeframe and the timeframe selected by the user into seconds, then divide by 60 to convert into minutes.
- We use two calls to the timeframe.in_seconds() function in the initialization of the `chartTFInMinutes` and `inputTFInMinutes` variables. In the first instance, we do not supply an argument for its `timeframe` parameter, so the function returns the chart's timeframe in seconds. In the second call, we supply the timeframe selected in the timeframe input.
- Next, we validate the timeframes to ensure that the input timeframe is equal to or higher than the chart's timeframe. If it is not, the script generates a custom runtime error.
- We finally print the two timeframe values converted to minutes.

Previous