Pine Script® v6 Documentation

5 FAQ

TradingView Pine Script Programming Language

Table of Contents

Redirect

Pine Script® v6 Documentation

FAQ

Pine Script® v6 Documentation

FAQ

Get real OHLC price on a Heikin Ashi chart

Suppose, we have a Heikin Ashi chart (or Renko, Kagi, PriceBreak etc) and we've added a Pine script on it:

//@version=6 indicator("Visible OHLC", overlay=true) c = close plot(c)

You may see that variable c is a Heikin Ashi *close* price which is not the same as real OHLC price. Because `close` built-in variable is always a value that corresponds to a visible bar (or candle) on the chart.

So, how do we get the real OHLC prices in Pine Script® code, if current chart type is non-standard? We should use `request.security` function in combination with `ticker.new` function. Here is an example:

//@version=6 indicator("Real OHLC", overlay = true) t = ticker.new(syminfo.prefix, syminfo.ticker) realC = request.security(t, timeframe.period, close) plot(realC)

In a similar way we may get other OHLC prices: *open*, *high* and *low*.

Get non-standard OHLC values on a standard chart

Backtesting on non-standard chart types (e.g. Heikin Ashi or Renko) is not recommended because the bars on these kinds of charts do not represent real price movement that you would encounter while trading.

If you want your strategy to enter and exit on real prices but still use Heikin Ashi-based signals, you can use the same method to get Heikin Ashi values on a regular candlestick chart:

//@version=6 strategy("BarUpDn Strategy", overlay = true, default_qty_type = strategy.percent_of_equity, default_qty_value = 10) maxIdLossPcntInput = input.float(1, "Max Intraday Loss(%)") strategy.risk.max_intraday_loss(maxIdLossPcntInput, strategy.percent_of_equity) needTrade() => close > open and open > close[1] ? 1 : close < open and open < close[1] ? -1 : 0 trade = request.security(ticker.heikinashi(syminfo.tickerid), timeframe.period, needTrade()) if trade == 1 strategy.entry("BarUp", strategy.long) if trade == -1 strategy.entry("BarDn", strategy.short)

Plot arrows on the chart

You may use plotshape with style `shape.arrowup` and `shape.arrowdown`:

//@version=6 indicator('Ex 1', overlay = true) condition = close >= open plotshape(condition, color = color.lime, style = shape.arrowup, text = "Buy") plotshape(not condition, color = color.red, style = shape.arrowdown, text = "Sell")

image

You may use the `plotchar` function with any unicode character:

//@version=6 indicator('buy/sell arrows', overlay = true) condition = close >= open plotchar(not condition, char='↓', color = color.lime, text = "Buy") plotchar(condition, char='↑', location = location.belowbar, color = color.red, text = "Sell")

image

Plot a dynamic horizontal line

There is the function `hline` in Pine Script, but it is limited to only plot a constant value. Here is a simple script with a workaround to plot a changing hline:

//@version=6 indicator("Horizontal line", overlay = true) plot(close[10], trackprice = true, offset = -9999) // `trackprice = true` plots horizontal line on close[10] // `offset = -9999` hides the plot plot(close, color = #FFFFFFFF) // forces display

Plot a vertical line on condition

//@version=6 indicator("Vertical line", overlay = true, scale = scale.none) // scale.none means do not resize the chart to fit this plot // if the bar being evaluated is the last baron the chart (the most recent bar), then cond is true cond = barstate.islast // when cond is true, plot a histogram with a line with height value of 100,000,000,000,000,000,000.00 // (10 to the power of 20) // when cond is false, plot no numeric value (nothing is plotted) // use the style of histogram, a vertical bar plot(cond ? 10e20 : na, style = plot.style_histogram)

Access the previous value

//@version=6 //... s = 0.0 s := nz(s[1]) // Accessing previous values if (condition) s := s + 1

Get a 5-days high

Lookback 5 days from the current bar, find the highest bar, plot a star character at that price level above the current bar

image

//@version=6 indicator("High of last 5 days", overlay = true) // Milliseconds in 5 days: millisecs * secs * mins * hours * days MS_IN_5DAYS = 1000 * 60 * 60 * 24 * 5 // The range check begins 5 days from the current time. leftBorder = timenow - time < MS_IN_5DAYS // The range ends on the last bar of the chart. rightBorder = barstate.islast // ———— Keep track of highest `high` during the range. // Intialize `maxHi` with `var` on bar zero only. // This way, its value is preserved, bar to bar. var float maxHi = na if leftBorder if not leftBorder[1] // Range's first bar. maxHi := high else if not rightBorder //

On other bars in the range, track highest `high`. maxHi := math.max(maxHi, high) // Plot level of the highest `high` on the last bar. plotchar(rightBorder ? maxHi : na, "Level", "—", location.absolute, size = size.normal) // When in range, color the background. bgcolor(leftBorder and not rightBorder ? color.new(color.aqua, 70) : na)

Count bars in a dataset

Get a count of all the bars in the loaded dataset. Might be useful for calculating flexible lookback periods based on number of bars.

//@version=6 indicator("Bar Count", overlay = true, scale = scale.none) plot(bar_index + 1, style = plot.style_histogram)

Enumerate bars in a day

//@version=6 indicator("My Script", overlay = true, scale = scale.none) isNewDay() => d = dayofweek na(d[1]) or d != d[1] plot(ta.barssince(isNewDay()), style = plot.style_cross)

Find the highest and lowest values for the entire dataset

//@version=6 indicator("", "", true) allTimetHi(source) => var atHi = source atHi := math.max(atHi, source) allTimetLo(source) => var atLo = source atLo := math.min(atLo, source) plot(allTimetHi(close), "ATH", color.green) plot(allTimetLo(close), "ATL", color.red)

Query the last non-na value

You can use the script below to avoid gaps in a series:

//@version=6 indicator("") series = close >= open ? close : na vw = fixnan(series) plot(series, style = plot.style_linebr, color = color.red) // series has na values plot(vw) // all na values are replaced with the last non-empty value

Previous

Next

Alerts

Pine Script® v6 Documentation

User Manual / FAQ / Alerts

Alerts

How do I make an alert available from my script?

In indicator scripts, there are two ways to define triggers for alerts:

- Using the alertcondition() function
- Using the alert() function

In strategy scripts, there are also two ways to define alert triggers:

- Using the alert() function
- Using order fill events

These methods make alert triggers available but do not *create* alerts directly. Users must create alerts using a script's alert triggers by selecting the appropriate trigger in the "Condition" dropdown of the "Create Alert" dialog box.

Programmers can define multiple alert triggers of one or more types in a script.

How are the types of alerts different?

Usability

Any script can include calls to the alertcondition() and alert() functions within their code. However, alertcondition() calls have no effect unless

the script is an *indicator*. Libraries can *export* functions containing alert() calls, but they cannot issue alert triggers directly.

Order fill alert triggers are available only from strategies.

Options for creating alerts

Each alertcondition() call in an indicator script defines one distinct trigger and one corresponding option in the “Condition” dropdown menu of the “Create Alert” dialog box. If the user wants multiple alerts, they must create each one *separately*.

By contrast, if a script includes one or more alert() function calls, only *one* option appears in the “Condition” dropdown menu, titled “Any alert() function call”. Selecting this option creates a *single alert* that activates based on the occurrences of *any* executed alert() call.

Similarly, for strategy scripts, the “Order fills and and alert() function calls” or “Order fills only” option in the “Condition” dropdown menu creates an alert that fires when *any* order fill event occurs.

How alerts activate

The alertcondition() function operates exclusively in an indicator’s global scope. Scripts cannot include calls to this function within any *local block*, such as the *indented* code within an if structure. The function triggers an alert when its specified `condition` is `true`. Users can set the allowed *frequency* of the alert trigger using the “Frequency” field in the “Create Alert” dialog box.

The alert() function has no `condition` parameter. Scripts trigger the alerts on any alert() call based on each call’s `freq` argument. Therefore, programmers typically include such calls within the local scopes of conditional structures to control when they execute.

Order fill alert triggers are available from strategies *automatically* without requiring extra code. However, programmers can customize the default alert messages. These alerts fire on order fill events, which occur when the broker emulator fills a strategy’s *orders*.

Messages

The `message` parameter of the alertcondition() function populates the "Message" field of the "Create Alert" dialog box with a default message, which script users can customize to suit their alert needs. It accepts a "const string" argument, meaning its value cannot change after compilation. However, the argument can include placeholders to make the message's information *dynamic*.

The `message` parameter of the alert() function accepts a "series string" argument, allowing programmers to create *dynamic messages* that can include "string" representations of a script's calculated values. Unlike alertcondition(), this function does not populate a "Message" field in the "Create Alert" dialog box, and it does not process placeholders. Programmers can allow users to customize alert() messages by creating inputs in the script's settings.

Order fill alerts have a default message that describes a strategy's order fill event. This default message contains strategy-specific placeholders, which the alert replaces with current strategy information each time it fires. Programmers can override the default message using the `//@strategy_alert_message` compiler annotation, which allows text and strategy placeholders, but *not* script variables. Script users can edit the default message from the "Message" field in the "Create Alert" dialog.

The `alert_message` parameter in a strategy's order placement commands allows programmers to define distinct messages for each order fill event. The parameter accepts "series string" values that can change on each event. To use the values from this parameter in a strategy's order fill alerts, include the `{{strategy.order.alert_message}}` placeholder in the `//@strategy_alert_message` annotation, or include it in the "Message" field when creating an alert.

Limitations

The alertcondition() function has some limitations:

- Each active alert condition counts toward the total number of alerts the user's plan allows.
- Every alertcondition() call contributes to the script's plot count.
- Only indicators can issue alert triggers with this function. Other script types do not raise a compilation error when they include calls to this function in their code, but each call has **no effect**.

By contrast, all calls to the alert() function count as **one** alert, regardless of the number of calls in the code. In addition, alert() calls *do not* contribute to a script's plot count.

Similarly, if a user creates a strategy alert based on order fill events, it counts as **one** alert, even though it can fire multiple times with distinct messages from different order executions.

Example `alertcondition()` alert

The script below demonstrates a simple alertcondition() call that triggers alerts when the current bar's close is above the value from the previous chart bar. It also uses a plotshape() call to indicate each bar where the `triggerCondition` occurred:

//@version=6 indicator("Simple alert demo", overlay = true) // Create condition to trigger alert. bool triggerCondition = close > close[1] // Use `triggerCondition` for the `condition` parameter. // Define a title for the alert in the menu and a message to send with the alert. alertcondition(condition = triggerCondition, title = "Example `alertcondition` Alert", message = "The example `alertcondition` alert was triggered.") // Plot a shape when `triggerCondition` is true to visually mark where alerts occur. plotshape(triggerCondition, "Trigger Condition", shape.xcross, color = color.fuchsia)

See this section of the Alerts page to learn more.

Example `alert()` alert

This example uses the `vstop()` function from our ta library to calculate a volatility stop value and trend information based on the Average True Range (ATR). The `stopValue` trails behind the chart's close to form a trend-following system.

The script triggers an alert with an alert() call each time the trend direction changes. The alert's message is a "series string" that shows the trend's new direction and the current stop value. An additional alert occurs whenever the `stopValue` moves in the current trend direction, with a message containing the updated value:

//@version=6 indicator("Vstop alert demo", overlay = true) import TradingView/ta/7 as TVta // Calculate ATR trailing stop and determine trend direction. [stopValue, trendUp] = TVta.vStop(close, 20, 2) // Round the stop value to mintick for accuracy in comparison operators. float stop = math.round_to_mintick(stopValue) // Check for trend changes. bool trendReversal = trendUp != trendUp[1] bool trendToDn = trendReversal and not trendUp bool trendToUp = trendReversal and trendUp // Create color variables for the plot display. color plotColor = trendUp ? color.green : color.red color lineColor = trendReversal ? color(na) : plotColor // Plot the stop value on the chart. Plot a circle on trend changes. plot(stop, "V-Stop", lineColor) plot(trendReversal ? stop : na, "Trend Change Circle", plotColor, 3, plot.style_circles) // Convert the stop value to string for use in the alert messages. string stopStr = str.tostring(stop) // If the trend changed to up, send a long alert with the initial stop value. if trendToUp alert("Long alert. Stop @ " + stopStr, alert.freq_once_per_bar_close) // If the trend changed to down, send a short alert with the initial stop value. if trendToDn alert("Short alert. Stop @ " + stopStr, alert.freq_once_per_bar_close) // If the stop value has progressed, send an alert to update the stop value. if (trendUp and stop > stop[1] or not trendUp and stop < stop[1]) and not trendReversal alert('Update stop to ' + stopStr, alert.freq_once_per_bar_close)

See this section of the Alerts page for more information.

Example strategy alert

This example strategy places a market order with strategy.entry() and a stop-loss and take-profit (bracket) order with strategy.exit() when a 5-bar moving average crosses over a 10-bar moving average. The stop-loss price is 1% below the current close and the take-profit price is 2% above the close. Order fill alerts occur when the broker emulator fills an entry or exit order. Both order placement commands include unique `alert_message` arguments that combine placeholders and “string” representations of the `limit` and `stop` values to output details like the trade action, position size, chart symbol, and order prices:

//@version=6 // This annotation auto-populates the alert dialogue with the `alert_message` string. // @strategy_alert_message {{strategy.order.alert_message}} strategy("Alert message demo", overlay = true) // Declare two moving averages to use for the entry condition. float fastMa = ta.sma(close, 5) float slowMa = ta.sma(close, 10) // Declare two persistent variables that will hold our stop-loss and take-profit values. var float limit = na var float stop = na // If `fastMa` has crossed over `slowMa` and we are not already in a position, // place an entry and exit order. // • Set the `limit` to 2% above the close and the stop to 1% below. // • Use a combination of script variables and placeholders in the alert strings. // • The exit alert shows the order direction, position size, ticker, and order price. // • The entry alert includes the same values plus the stop and limit price. if ta.crossover(fastMa, slowMa) and strategy.position_size == 0 limit := close * 1.02 stop := close * 0.99 string exitString = "{{strategy.order.action}} {{strategy.position_size}} {{ticker}} @ {{strategy.order.price}}" string entryString = exitString + " TP: " + str.tostring(limit, format.mintick) + " SL: " + str.tostring(stop, format.mintick) strategy.entry("Buy", strategy.long, alert_message = entryString) strategy.exit("Exit", "Buy", stop = stop, limit = limit, alert_message = exitString) // Plot the moving averages, stop, and limit values on the chart. plot(fastMa, "Fast Moving Average", color.aqua) plot(slowMa, "Slow Moving Average", color.orange) plot(strategy.position_size > 0 ? limit : na, "Limit", color.green, style = plot.style_linebr) plot(strategy.position_size > 0 ? stop : na, "Stop", color.red, style = plot.style_linebr)

For more information about order fill events, see this section of the Alerts page. To learn more about how strategy scripts work, see the Strategies page.

If I change my script, does my alert change?

No, not without creating a new alert.

When a user creates an alert using the “Create Alert” dialog box, that action saves a “snapshot” of the script, its inputs, and the current chart’s context on TradingView’s servers. This snapshot acts as an independent *copy* of the script instance and chart. Therefore, any changes to the script, its inputs, or the user’s chart **do not** affect that created alert. To update an alert after making changes, *delete* the existing alert and *create* a new one.

Why aren’t my alerts working?

Here are some common reasons why alerts might not work as expected, and how to solve them:

Make sure the alert is active and has not expired

Scripts that include alert triggers **do not** directly create alerts. Users must create alerts in the “Create Alert” dialog box, where they specify the “Condition” that triggers the alert and the “Expiration” time. Created alerts do not fire after they expire. See this Help Center article on Setting up alerts.

Check the alert logs

An alert can fire without a notification, depending on the alert’s settings. Check the logs in the alert manager to see whether an alert occurred. To set up notifications for an alert, use the options in the “Notifications” tab of the “Create/Edit Alert” dialog box.

Check for repainting

If an alert fires at a different time than expected, *repainting* might be the cause. Refer to the Repainting page for more information.

Limit the frequency of alerts

If more than 15 alerts occur within three minutes, the system automatically *halts* further alerts. This frequency limit helps prevent excessive notifications and potential server overload.

Debug script errors

If a script instance raises a *runtime error* at some point during its executions, alerts from that instance **cannot** fire because the error stops the script from continuing to execute its code. Some common issues that can halt alerts include:

- Attempting to store more than 100,000 elements within a collection
- Trying to access an item from a collection at an *out-of-bounds* index
- Referencing historical values of a time series outside its allocated memory buffer
- Using loops that take longer than 500 ms to complete their iterations

See this page for additional details about common error messages and troubleshooting tips.

Why is my alert firing at the wrong time?

Sometimes, alerts may fire when users do not expect according to what their script displays on the chart. Repainting is the typical cause of such issues.

A chart's realtime and historical bars often rely on *different* data feeds. Data providers may retroactively adjust the reported values on realtime bars, which the displayed data reflects *after* users refresh their charts or restart their scripts. Such adjustments can cause discrepancies

where a triggered alert's timing may not align with the script's output after reloading it.

Scripts may also behave differently on historical and realtime bars, which can lead to repainting. On historical bars, scripts execute once per bar close, whereas on realtime bars, where alerts fire, scripts execute once for *each new tick* from the data feed. Therefore, if a script behaves differently on those bars, users may see differences between its signals and triggered alerts after reloading the chart.

Below are some common repainting issues that can affect a script's alerts:

Alerts firing before bar close

Most scripts have fluid data values that update after new ticks during an unconfirmed realtime bar and finalize after the bar closes. Consequently, an alert that fires on an open bar may not reflect the *final state* of the condition after the bar's confirmation. Set the alert's frequency to "Once Per Bar Close" to avoid this issue.

Using `calc_on_every_tick` in strategies

When a strategy script includes `calc_on_every_tick = true` in its declaration statement or the user selects the "On every tick" option in the "Recalculate" section of the strategy's properties, it recalculates on *every* price update in the realtime data. This behavior can cause strategies to repaint because historical bars do not contain the same information as realtime bars. See this section of the Strategies page to learn more.

Incorrect usage of `request.security()` calls

Using request.security() calls to fetch data from alternative timeframes can cause discrepancies on historical bars that scripts **cannot** reproduce on realtime bars. Ensure you follow the best practices for *non-repainting* data requests to avoid such discrepancies, especially with higher-timeframe data. See the Avoiding repainting section of the Other timeframes and data page and the Higher-timeframe requests publication from PineCoders for more information.

Can I use variable messages with alertcondition()?

The `message` parameter of the alertcondition() function requires a "const string" argument, which **cannot change** after compilation. However, the "string" can include placeholders, which an alert substitutes with corresponding dynamic values from a script each time it fires.

The script below demonstrates two alertcondition() calls whose `message` arguments include placeholders for dynamic values. Each time alerts from these triggers occur, the message displays information about the current chart's exchange, symbol, price, and volume:

//@version=6 indicator("Placeholder demo", overlay = false) [macdLine, signalLine, histLine] = ta.macd(close, 12, 26, 9) plot(macdLine, "MACD", color.blue) plot(signalLine, "Signal", color.orange) plot(histLine, "Hist.", color.red, style = plot.style_histogram) bool crossUp = ta.crossover(macdLine, signalLine) bool crossDown = ta.crossunder(macdLine, signalLine) alertcondition(crossUp, "MACD Cross Up", "MACD cross up on {{exchange}}:{{ticker}}\nprice = {{close}}\nvolume = {{volume}}") alertcondition(crossDown, "MACD Cross Down", "MACD cross down on {{exchange}}:{{ticker}}\nprice = {{close}}\nvolume = {{volume}}")

How can I include values that change in my alerts?

The method for including dynamic values in alert messages varies with the type of alert trigger:

- The alertcondition() function accepts a "const string" `message` argument that can contain placeholders for dynamic values. See Can I use variable messages with `alertcondition()`? for more information.

- The alert() function accepts “series string” `message` arguments, which allows the convenient creation of dynamic messages that use a script’s calculated values. See this section for an example.
- Order fill alerts can use “series string” values and placeholders. Refer to the example here.

How can I get custom alerts on many symbols?

To manage alerts across multiple symbols using a custom script, one option is to set an individual alert on each symbol. There is no automated method to set the same alert across many symbols simultaneously in a single action. It’s also important to note that the TradingView screener uses built-in filters and does not support custom Pine Script® code.

Scripts can retrieve data from other *contexts* (symbols, timeframes, and modifiers such as non-standard chart calculations and extended sessions) using the functions in the `request.*()` namespace. With these functions, programmers can design scripts that retrieve data from up to 40 unique contexts. Search for “screener” in the Community scripts for in-depth examples.

Here is an example incorporating three symbols. The `checkForAlert()` function calls request.security() to fetch data from a specified context and evaluate the user-defined `checkForRsiConditions()` function using that data. Then, the function calls alert() using the result to create an alert trigger. The script calls this function three times, creating distinct alert triggers for each specified symbol:

//@version=6 indicator("Screener demo", overlay = true) // Declare inputs for the alert symbols and the timeframe to run the alerts on. The default is the current chart timeframe. string tfInput = input.timeframe("", "Timeframe") string symbol1Input = input.symbol("BINANCE:ETHUSDT", "Symbol 1") string symbol2Input = input.symbol("BINANCE:BATUSDT", "Symbol 2") string symbol3Input = input.symbol("BINANCE:SOLUSDT", "Symbol 3") // @function Generates alert messages for RSI crossing over or under 50, and crosses of price

and the 50 EMA. // @returns (string) Formatted alert messages with values for each crossover and crossunder event. checkForRsiConditions() => float rsi = ta.rsi(close, 14) float ema = ta.ema(close, 50) string alertMessage = "" if ta.crossover(rsi, 50) alertMessage += str.format("RSI ({0}) crossed over 50 for {1} on {2} timeframe.\n", rsi, syminfo.ticker, timeframe.period) if ta.crossunder(rsi, 50) alertMessage += str.format("RSI ({0}) crossed under 50 for {1} on {2} timeframe.\n", rsi, syminfo.ticker, timeframe.period) if ta.crossover(close, ema) alertMessage += str.format("Crossover of 50 EMA for {0} on {1} timeframe. Price is {2}", syminfo.ticker, timeframe.period, close) if ta.crossunder(close, ema) alertMessage += str.format("Crossunder of 50 EMA for {0} on {1} timeframe. Price is {2}", syminfo.ticker, timeframe.period, close) // @function Calls the `checkForRsiConditions()` function for the provided symbol and timeframe. // Triggers an alert if the function returns a message. // @param symbol (simple string) The symbol to check. // @param tf (simple string) The timeframe to check. // @param freq (const string) The frequency of the alert. Optional. Default is `alert.freq_once_per_bar`. // @returns (void) The function has no explicit return, but triggers an alert with the message if the // conditions defined within the `checkForRsiConditions()` function are met. checkForAlert(simple string symbol, simple string tf, const string freq = alert.freq_once_per_bar) => string msg = request.security(symbol, tf, checkForRsiConditions()) if msg != msg[1] and str.length(msg) > 0 alert(msg, freq) // Check for alerts on the input symbols and timeframe. checkForAlert(symbol1Input, tfInput) checkForAlert(symbol2Input, tfInput) checkForAlert(symbol3Input, tfInput) // Continue with additional symbols up to a maximum of 40...

Note that:

- A script can execute up to 40 *unique* `request.*()` calls. A `request.*()` call is not unique if a script already calls the same function with the same arguments. See this section of the Limitations page for more information.
- This script uses the alert() function because alertcondition() is not allowed within *local scopes*.

How can I trigger an alert for only the first instance of a condition?

Firing an alert only on its first occurrence can help avoid redundant notifications and isolate specific conditions or state changes, which is beneficial in several use cases. For instance, if a user relies on alerts to automate order placement, restricting redundant alerts to their first occurrence can help avoid accidentally placing excessive orders.

For alerts with alertcondition() triggers, setting them to fire once using the “Only Once” option in the “Create Alert” dialog box is not an optimal solution because it requires *manual* reactivation each time an alert occurs. Alerts from the alert() function do not have an “Only Once” frequency option. The programmer must use conditional logic to ensure the call executes at the appropriate time.

There are two primary ways to code repeating alerts that fire on only the first instance of a condition:

Using stricter criteria

Rather than relying on a continuous condition like `close > ma`, which may remain `true` for multiple consecutive bars, try using a more strict condition like `ta.crossover(close, ma)`. For simple cases, this is the easiest method.

Using state control

More complex scenarios might require controlling and tracking *states*, which entails setting flags or specific values to signify certain conditions.

The example script below manages separate bullish and bearish states, and it colors the background to represent each state. When a bullish or bearish state first occurs, an alert() call executes and the script plots a triangle on the chart. It also plots smaller triangles to show where other signals occur within a state, which do not trigger additional alerts:

// image

//@version=6 indicator("Single alert demo", overlay = true) // ——————— Calculations: Determine highest/lowest values over last `lengthInput` bars. int lengthInput = input.int(20, "Length") float highest = ta.highest(lengthInput) float lowest = ta.lowest(lengthInput) // ——————— Trigger conditions: Define bull and bear signals. Bull signal is triggered by a new high, and bear by a new low. bool bullSignal = high == highest bool bearSignal = low == lowest // ——————— State change flags: Set true on state transition bars only. bool changeToBull = false bool changeToBear = false // ——————— State tracking: `isBull` is set to true for bull state, false for bear. It's set only at the initial switch to the opposite condition. // This variable's state is retained from bar to bar because we use the `var` keyword to declare it. var bool isBull = false // ——————— State transitions: Allow a switch from bull to bear or bear to bull; ignore repeated signals in current state. // Set the state change flags to true only on the first bar where a new signal appears. if bullSignal and not isBull isBull := true changeToBull := true else if bearSignal and isBull isBull := false changeToBear := true // Plot highest and lowest values. plot(highest, "Highest", color.new(color.green, 80), 2) plot(lowest, "Lowest", color.new(color.red, 80), 2) // Background color: Green for bull, none for bear. bgcolor(isBull ? color.new(color.green, 90) : na) // State change markers: Display "ALERT" text on bars where a state change occurs and an alert would trigger. plotchar(changeToBull, "Change to Bull state", "▲", location.belowbar, color.new(color.lime, 30), size = size.small, text = "BULL\nALERT") plotchar(changeToBear, "Change to Bear state", "▼", location.abovebar, color.new(color.red, 30), size = size.small, text = "BEAR\nALERT") // Signal markers: Display for repeated signals within the current state. // These signals would trigger redundant alerts if not for the state tracking flag preventing them. plotchar(bullSignal and not changeToBull, "Bull signal", "▲", location.belowbar, color.green, size = size.tiny) plotchar(bearSignal and not changeToBear, "Bear signal", "▼", location.abovebar, color.maroon, size = size.tiny) // Alerts: Trigger on state changes only. if changeToBull alert("Change to bull state") if changeToBear alert("Change to bear state")

How can I run my alert on a timer or delay?

It is possible to program logic to delay alert triggers so that they occur *after* the initial condition. However, because Pine scripts execute on realtime bars only after new *price updates*, and an alert only fires when a script *executes*, it is difficult to predict the exact time of a delayed alert.

There are no price updates in a closed market, meaning an alert with a delay will not fire until the market opens again. Similarly, thinly traded securities may have very infrequent price updates in unpredictable intervals, which can cause a larger delay than intended.

The Pine script below implements a *time-delayed* alert, which is subject to the limitations above. When the current close is higher than a moving average, a delay counter starts. After the delay passes, the alert fires once, and another alert *cannot* fire until the timer resets. Users can specify whether the timer resets on each bar using the script's `resetInput`:

image

//@version=6 indicator("Delayed alert demo", overlay = true) import PineCoders/Time/4 as PCtime string TIME_TT = "The delay's duration and units. This specifies the continuous duration for which the condition must be true before triggering the alert." string RESET_TT = "When checked, the duration will reset every time a new realtime bar begins." enum TimeUnit seconds minutes hours int durationInput = input.int(20, "Condition must last", minval = 1, inline = "00") TimeUnit timeUnitInput = input.enum(TimeUnit.seconds, "", inline="00") bool resetInput = input.bool(false, "Reset timing on new bar", tooltip = RESET_TT) int maLengthInput = input.int(9, "MA length") // Calculate and plot a SMA with `maLengthInput` length. float ma = ta.sma(close, maLengthInput), plot(ma, "MA") // Check whether the close is greater than the SMA. bool cond = close > ma // Time the duration for which the condition has been true. int secSince = PCtime.secondsSince(cond, resetInput and barstate.isnew) // Check if the duration is greater than

the input timer. bool timeAlert = secSince > (PCtime.timeFrom("bar", durationInput, str.tostring(timeUnitInput)) - time) / 1000 // Format a time string for the timer label. string alertTime = str.format_time(secSince * 1000, "mm:ss") // Set the contents for the label depending on the stage of the alert timer. string alertString = switch timeAlert => "Timed Alert Triggered\n\n" + alertTime cond => "Condition Detected...\n\nTimer count\n" + alertTime => "Waiting for condition..." // Display alert timer using a label. Declare a basic label once and update location, color, and text on the last bar for efficiency. if barstate.islast var label condTime = label.new(na, na, yloc = yloc.abovebar, style = label.style_label_lower_left, textcolor = chart.fg_color) label.set_x(condTime, bar_index) label.set_text(condTime, alertString) label.set_color(condTime, color.new(timeAlert ? color.green : cond ? color.orange : color.red, 50)) // Create a flag to ensure alert is triggered only once each time the delay timer is exceeded. varip bool isFirstOccurrence = true // Fire alert if timer is triggered. if timeAlert and isFirstOccurrence alert(str.format("{0} {1} Delayed Alert Triggered", durationInput, str.tostring(timeUnitInput)), alert.freq_all) // Toggle the flag to `false` when alert triggers, and reset when the condition clears. isFirstOccurrence := not timeAlert

Note that:

- The `secondsSince()` function from the PineCoders' time library determines the duration, in seconds, for which a certain condition remains continuously `true`. The duration can be tracked within bars because it uses the varip keyword.
- The timing starts when the condition first becomes `true`. If the condition becomes `false` or an optional resetting condition occurs, the timer restarts. If "Reset timing on new bar" is enabled in the "Settings/Inputs" tab, the function restarts its timing at the start of a new bar.
- A colored label shows what state the script is in:
    1. **Red** - The condition has not occurred yet.
    2. **Orange** - The condition occurred and the delay timer is active.

3. **Green** - The timer has surpassed the set duration, simulating a delayed alert.

This script relies on variables declared with the varip keyword, which do not revert to their last committed states during realtime bar calculations. See this section of the User Manual to learn more about using this keyword. To learn about how *rollback* works, see the Execution model page.

How can I create JSON messages in my alerts?

Alerts can send messages containing JavaScript Object Notation (JSON) to webhooks. Pine Script does not include any built-in functions to produce JSON, but programmers can create JSON messages in Pine by constructing “string” representations.

When constructing JSON representations, ensure the keys and values intended as strings in the JSON-formatted text use *double quotes*, not single quotes.

The following example shows three ways to construct JSON strings in Pine Script:

1. **Static JSON Strings**

   Define separate alerts with predefined JSON-formatted strings. This method is the simplest.

2. **Placeholders**

   Use placeholders in the alert message, such as `{{close}}` and `{{volume}}`, to add *dynamic* values to the JSON. The alert instance replaces the placeholders with corresponding values when it fires. This method can create richer alerts, especially for strategies, which have extra placeholders for their calculated values. See this section above for an example.

3. **Dynamic strings**

Use the functions in the `str.*()` namespace and “string” concatenation to create dynamic JSON-formatted text. This method is the most customizable and advanced. Our script below shows a simple, straightforward example of this approach. When using dynamic string formatting to construct JSON strings, ensure the resulting JSON is *valid* for all the combined values.

```
//@version=6 indicator("JSON example", overlay = true) // Define EMA cross conditions to trigger alerts, and plot the ema on the chart. float ema = ta.ema(close, 21) bool crossUp = ta.crossover(close, ema) bool crossDown = ta.crossunder(close, ema) plot(ta.ema(close, 21)) // ——————— Method 1 - Separate alerts with static messages. string alertMessage1a = '{"method": 1, "action": "buy", "direction": "long", "text": "Price crossed above EMA"}' string alertMessage1b = '{"method": 1, "action": "sell", "direction": "short", "text": "Price crossed below EMA"}' alertcondition(crossUp, "Method 1 - Cross up", alertMessage1a) alertcondition(crossDown, "Method 1 - Cross down", alertMessage1b) // Rendered alert: // { // "method": 1, // "action": "buy", // "direction": "long", // "text": "Price crossed above EMA" // } // ——————— Method 2 - Using placeholders for dynamic values. string alertMessage2 = '{"method": 2, "price": {{close}}, "volume": {{volume}}, "ema": {{plot_0}}}' alertcondition(crossUp, "Method 2 - Cross Up", alertMessage2) // Rendered alert: // { // "method": 2, // "price": 2066.29, // "volume": 100.859, // "ema": 2066.286 // } // ——————— Method 3 - String concatenation using dynamic values. string alertMessage3 = '{"method": 3, "price": ' + str.tostring(close) + ', "volume": ' + str.tostring(volume) + ', "ema": ' + str.tostring(ema) + '}' if crossUp alert(alertMessage3, alert.freq_once_per_bar_close) // Rendered alert: // { // "method": 3, // "price": 2052.27, // "volume": 107.683, // "ema": 2052.168 // }
```

Before using the JSON-formatted string in alerts for real-world applications, such as sending messages to place orders, *test* and *validate* the JSON message to ensure it works as intended:

- Send alerts to an email address to see how the JSON message appears.
- Copy the alert message from the email into an online JSON validation tool.

- Use an API client application to check the server response to the request.

Refer to this Wikipedia page to learn more about JSON format. To learn more about how alerts send information using webhooks, see the Help Center article on webhooks.

How can I send alerts to Discord?

Sending alerts from a Pine script to a Discord chat room is possible using webhooks.

The message for Discord communication requires JSON format. The *minimum* requirement for a valid message is `{"content": "Your message here"}`.

The script example below uses placeholders to dynamically populate alert messages with script values, including the new high or low price, and the chart's symbol and timeframe:

//@version=6 indicator("Discord demo", overlay = true) // Calculate a Donchian channel using the TV ta library. import TradingView/ta/7 as TVta int lengthInput = input.int(10, "Channel length") [highest, lowest, middle] = TVta.donchian(lengthInput) // Create conditions checking for a new channel high or low. bool isNewHi = high > highest[1] bool isNewLo = low < lowest[1] // Plot the Donchian channel and fill between the midpoint and the upper and lower halves. hi = plot(highest, "Channel high", color.new(color.fuchsia, 70)) mid = plot(middle, "Channel mid.", color.new(color.gray, 70)) lo = plot(lowest, "Channel low", color.new(color.lime, 70)) fill(mid, hi, color.new(color.fuchsia, 95)) fill(mid, lo, color.new(color.lime, 95)) // Plot shapes to mark new highs and lows to visually identify where alert trigger conditions occur. plotshape(isNewHi, "isNewHi", shape.arrowup, location.abovebar, color.new(color.lime, 70)) plotshape(isNewLo, "isNewLo", shape.arrowdown, location.belowbar, color.new(color.fuchsia, 70)) // Create two alert conditions, one for new highs, and one for new lows. // Format the message for Discord in the following JSON format: {"content": "Your message here"} alertcondition(isNewHi, "New High (Discord Alert Demo)", '{"content":

"New high ({{high}}) on {{ticker}} on {{interval}} chart!"}') alertcondition(isNewLo, "New Low (Discord Alert Demo)", '{"content": "New low ({{low}}) on {{ticker}} on {{interval}} chart!"}') // The following test alert condition fires immediately. Set this alert frequency to "Only Once". alertcondition(true, "Test (Discord Alert Demo)", '{"content": "This is a test alert from TradingView to Discord."}')

To send these alert messages to Discord, follow these steps:

1. Create a Discord webhook

- Create a new webhook in a server using an account with webhook creation and management permissions. Refer to Discord's Intro to Webhooks article for instructions.
- Copy the Webhook URL. This URL represents the address where the alert sends a POST request.

2. Set up an alert on TradingView

- Add the above "Discord demo" script to a chart and open the "Create Alert" dialog box.
- Choose one of the script's alert conditions as the "Condition" in the dialog. If you select the "New High" or "New Low" alerts, choose the "Once Per Bar Close" option in the "Frequency" field to avoid triggering alerts for new highs or lows on an unconfirmed bar. When using the "Test" alert, choose "Only Once" as the "Frequency" option.
- In the "Notifications" tab of the "Create Alert" dialog, select "Webhook URL" and paste the URL of the Discord webhook.

3. Test the integration

- Check that alerts appear in the alert log on TradingView.
- Use the "Test" alert to check whether the webhook works as expected. After the alert fires, check your Discord to see if it received the message.
- If the alert message does not appear in Discord, check whether the pasted webhook URL is correct.

- If the message does not display correctly in Discord, check the JSON format. The minimum required format is `{"content": "Your message here"}`.

Consult Discord's Webhook Resource to learn about advanced JSON message configurations.

For more information about dynamic values in alert messages, refer to How can I include values that change in my alerts?.

To learn about using JSON format in script alerts, see How can I create JSON messages in my alerts?.

How can I send alerts to Telegram?

Sending TradingView alerts directly to Telegram is challenging due to protocol differences and formatting requirements. One solution is to use an intermediary service, which receives webhook alerts from TradingView, formats them as required by Telegram, and then forwards them to a Telegram bot.

1. Choose a platform like Zapier, Integromat, or Pipedream. Alternatively, programmers can consider developing a custom server script using Node.js or Python.
2. In TradingView, set up alerts to send webhook requests to the intermediary service's provided URL.
3. Configure the intermediary service to reformat TradingView's incoming requests for Telegram's API and send the formatted message to a Telegram bot using the `sendMessage` method.

See the Telegram Bot API documentation for detailed technical information.

Previous

Next

Data structures

Pine Script® v6 Documentation

Data structures

What data structures can I use in Pine Script®?

Pine data structures resemble those in other programming languages, with some important differences:

- **Tuple**: An arbitrary—and temporary—grouping of values of one or more types.
- **Array**: An ordered sequence of values of a single type.
- **Matrix**: A two-dimensional ordered sequence of values of a single type.
- **Object**: An arbitrary—and persistent—collection of values of one or more types.
- **Map**: An *unordered* sequence of key-value pairs, where the keys are of a single type and the values are of a single type.

The following sections describe each data structure in more detail.

Tuples

A tuple in Pine Script is a list of values that is returned by a function, method, or local block. Unlike in other languages, tuples in Pine serve no other function. Tuples do not have names and cannot be assigned to variables. Apart from the fact that the values are requested and

returned together, the values have no relation to each other, in contrast to the other data structures described here.

To define a tuple, enclose a comma-separated list of values in square brackets.

Using a tuple to request several values from the same symbol and timeframe using a request.security() call is more efficient than making several calls. For instance, consider a script that contains separate request.security() calls for the open, high, low, and close prices:

float o = request.security(syminfo.tickerid, "D", open) float h = request.security(syminfo.tickerid, "D", high) float l = request.security(syminfo.tickerid, "D", low) float c = request.security(syminfo.tickerid, "D", close)

Using a tuple can consolidate these calls into a single request.security() function call, reducing performance overhead:

[o, h, l, c] = request.security(syminfo.tickerid, "D", [open, high, low, close])

See the Tuples section in the User Manual for more information.

Arrays

Arrays store multiple values of the same type in a single variable. Each *element* in an array can be efficiently accessed by its *index*—an integer corresponding to its position within the array.

Arrays can contain an arbitrary number of elements. Scripts can loop through arrays, testing each element in turn for certain logical conditions. There are also many built-in functions to perform different operations on arrays. This flexibility makes arrays very versatile data structures.

Arrays can be created with either the array.new<type>() or array.from() function. In this simple example, we store the last five closing prices in an array and display it in a table:

//@version=6 indicator("Array example") // Declare an array with 5 `na` values on the first bar. var array<float> pricesArray = array.new<float>(5) // On each bar, add a new value to the end of the array and remove the first (oldest) element. array.push(pricesArray, close) array.shift(pricesArray) // Display the array and its contents in a table. var table displayTable = table.new(position.middle_right, 1, 1) if barstate.islast table.cell(displayTable, 0, 0, str.tostring(pricesArray), text_color = chart.fg_color)

See the Arrays section in the User Manual for more information.

Matrices

A matrix is a two-dimensional array, made of rows and columns, like a spreadsheet. Matrices, like arrays, store values of the same built-in or user-defined type.

Matrices have many built-in functions available to organize and manipulate their data. Matrices are useful for modeling complex systems, solving mathematical problems, and improving algorithm performance.

This script demonstrates a simple example of matrix addition. It creates a 3x3 matrix, calculates its transpose, then calculates the matrix.sum() of the two matrices. This example displays strings representing the original matrix, its transpose, and the resulting sum matrix in a table on the chart:

image

//@version=6 indicator("Matrix sum example") //@variable An empty matrix of type "float". m = matrix.new<float>() // Add rows to the matrix containing data. m.add_row(0, array.from(1, 2, 3)) m.add_row(1, array.from(0, 4, 2)) m.add_row(2, array.from(3, 1, 2)) var table displayTable = table.new(position.middle_right, 5, 2) if barstate.islast matrix<float> t = m.transpose() table.cell(displayTable, 0, 0, "A", text_color = chart.fg_color) table.cell(displayTable, 0, 1, str.tostring(m), text_color = chart.fg_color) table.cell(displayTable, 1, 1, "+", text_color = chart.fg_color) table.cell(displayTable, 2, 0, "Aᵀ", text_color =

chart.fg_color) table.cell(displayTable, 2, 1, str.tostring(t), text_color = chart.fg_color) table.cell(displayTable, 3, 1, "=", text_color = chart.fg_color) table.cell(displayTable, 4, 0, "A + Aᵀ", text_color = color.green) table.cell(displayTable, 4, 1, str.tostring(matrix.sum(m, t)), text_color = color.green)

See the Matrices section in the User Manual for more information.

Objects

Pine Script objects are containers that group together multiple fields into one logical unit.

Objects are *instances* of user-defined types (UDTs). UDTs are similar to *structs* in traditional programming languages. They define the rules for what an object can contain. Scripts first create a UDT by using the type keyword and then create one or more objects of that type by using the UDT's built-in `new()` method.

UDTs are *composite* types; they contain an arbitrary number of fields that can be of any type. A UDT's field type can even be another UDT, which means that objects can contain other objects.

Our example script creates a new `pivot` object each time a new pivot is found, and draws a label using each of the object's fields:

//@version=6 indicator("Object example", overlay = true) // Create the pivot type with 3 fields: the x coordinate, the y coordinate, and a formatted time string. type pivot int x float y string pivotTime // Check for new pivots. `ta.pivotHigh` returns the price of the pivot. float pivFound = ta.pivothigh(10, 10) // When a pivot is found, create a new pivot object and generate a label using the values from its fields. if not na(pivFound) pivot pivotObject = pivot.new(bar_index - 10, pivFound, str.format_time(time[10], "yyyy-MM-dd HH:mm")) label.new(pivotObject.x, pivotObject.y, pivotObject.pivotTime, textcolor = chart.fg_color)

See the User Manual page on Objects to learn more about working with UDTs.

Maps

Maps in Pine Script are similar to *dictionaries* in other programming languages, such as dictionaries in Python, objects in JavaScript, or HashMaps in Java. Maps store elements as key-value pairs, where each key is unique. Scripts can access a particular value by looking up its associated key.

Maps are useful when accessing data directly without searching through each element, as you need to do with arrays. For example, maps can be more performant and simpler than arrays for associating specific attributes with symbols, or dates with events.

The following example illustrates the practical application of maps for managing earnings dates and values as key-value pairs, with dates serving as the keys:

image

//@version=6 indicator("Earnings map", overlay = true) // Get the earnings value if present. We use `barmerge.gaps_on` to return `na` unless earnings occurred. float earnings = request.earnings(syminfo.tickerid, earnings.actual, barmerge.gaps_on) // Declare a map object for storing earnings dates and values. var map<string, float> earningsMap = map.new<string, float>() // If `request.security()` returned data, add an entry to the map with the date as the key and earnings as the value. if not na(earnings) map.put(earningsMap, str.format_time(time, "yyyy-MM-dd"), earnings) // On the last historical bar, loop through the map in the insertion order, writing the key-value pairs to the logs. if barstate.islastconfirmedhistory string logText = "\n" for [key, value] in earningsMap logText += str.format("{0}: {1}\n", key, value) log.info(logText)

Here, we use request.earnings() with the barmerge parameter set to barmerge.gaps_on to return the earnings value on bars where earnings data is available, and return na otherwise. We add non-na values to the map, associating the dates that earnings occurred with the earnings

numbers. Finally, on the last historical bar, the script loops through the map, logging each key-value pair to display the map's contents.

To learn more about working with maps, refer to the Maps section in the User Manual.

What's the difference between a series and an array?

In Pine Script, "series" variables are calculated on each bar. Historical values cannot change. Series values can change during the realtime bar, but when the bar closes, the value for that bar becomes fixed and immutable. These fixed values are automatically indexed for each bar. Scripts can access values from previous bars by using the [] history-referencing operator) to go back one or more bars.

Where "series" variables are strictly time-indexed, and the historical values are created automatically, arrays are created, filled, and manipulated arbitrarily by a script's logic. Programmers can change the size of arrays dynamically by using functions that insert or remove elements. Any element in an array can also be altered using the array.set() function.

The concept of time series is a fundamental aspect of Pine Script. Its series-based execution model processes scripts bar-by-bar. This built-in behavior mimics looping, allowing a series to track values, accumulate totals, or perform calculations across a sequence of data on each bar.

Simple calculations can thus be done efficiently using "series" variables. Using arrays for similar tasks requires manually creating a dataset, managing its size, and using loops to process the array's contents, which can be far less efficient.

Arrays, of course, can do many things that series variables cannot. Scripts can use arrays to store a fixed set of values, store complex data such as User-Defined Types, manipulate objects for visual display, and more. In general, use arrays to handle data that doesn't fit the time series model, or for complex calculations. Arrays can also mimic series by creating custom datasets, as in the getSeries library.

How do I create and use arrays in Pine Script?

Pine Script arrays are one-dimensional collections that can hold multiple values of a single type.

Declaring arrays

Declare an array by using one of the following functions: array.new<type>(), array.from(), or array.copy(). Arrays can be declared with the var keyword to have their values persist from bar to bar, or without it, so that the values initialize again on each bar. For more on the differences between declaring arrays with or without var, see this section of this FAQ.

Adding and removing elements

Pine Script provides several functions for dynamically adjusting the size and contents of arrays.

Adding elements

- array.unshift() inserts a new element at the beginning of an array (index 0) and increases the index values of any existing elements by one.
- array.insert() inserts a new element at the specified index and increases the index of existing elements at or after the insertion index by one. It accepts both positive and negative indices, which reference an element's position starting from the beginning of the array or from the end, respectively.
- array.push() adds a new element at the end of an array.

Removing elements

- array.remove() removes the element at the specified index and returns that element's value. It accepts both positive and negative indices, which reference an element's position starting from the beginning of the array or from the end, respectively.
- array.shift() removes the first element from an array and returns its value.

- array.pop() removes the last element of an array and returns its value.
- array.clear() removes all elements from an array. Note that clearing an array won't delete any objects that were referenced by its elements. If you want to delete objects contained by an array, loop through the array and delete the objects first, and then clear the array.

The flexibility afforded by these functions supports various data management strategies, such as queues or stacks, which are useful for custom datasets or sliding window calculations. Read more about implementing a stack or queue in this FAQ entry.

Calculations on arrays

Because arrays are not time series data structures, performing operations across an array's elements requires special functions designed for arrays. Programmers can write custom functions to perform calculations on arrays. Additionally, built-in functions enable computations like finding the maximum, minimum, or average values within an array. See the Calculation on arrays section of the User Manual for more information.

Script example

This script example demonstrates a practical application of arrays by tracking the opening prices of the last five sessions. The script declares a float array to hold the prices using the var keyword, allowing it to retain its values from bar to bar.

At the start of each session, we update the array by adding the new opening price and removing the oldest one. This process, resembling a queue, keeps the array's size constant while maintaining a moving window of the session opens for the last five days. Built-in array functions return the highest, lowest, and average opening price over the last five sessions. We plot these values to the chart.

image

```
//@version=6 indicator("Array demo", overlay = true) // Create an input to determine the number of session opens to track, with a default value of 5. int numOpensInput = input.int(5, "Number of opens to track") // Create an array to store open prices. Using `var` ensures the array retains its values from bar to bar. // Initially, the array is filled with placeholder values (`na`), which are later updated with actual open prices. var array<float> opensArray = array.new<float>(numOpensInput) // On the first bar of each session, update the array: add the current open price and remove the oldest entry. if session.isfirstbar_regular array.push(opensArray, open) array.shift(opensArray) // Plot the highest, lowest, and average open prices from the tracked sessions plot(array.max(opensArray), "Highest open in n sessions", color.lime) plot(array.min(opensArray), "Lowest open in n sessions", color.fuchsia) plot(array.avg(opensArray), "Avg. open of the last n sessions", color.gray) // Change the background color on the first bar of each session to visually indicate session starts. bgcolor(session.isfirstbar_regular ? color.new(color.gray, 80) : na)
```

For more information about arrays, see the Arrays page in the User Manual.

What’s the difference between an array declared with or without var?

Using the var keyword, a script can declare an array variable in a script that is initialized only once, during the first iteration on the first chart bar.

Persistent arrays

When an array is declared with var, it is initialized only once, at the first execution of the script. This allows the array to retain its contents and potentially grow in size across bars, making it ideal for cumulative data collection or tracking values over time.

Non-persistent arrays

Arrays declared without var are reinitialized on every new bar, effectively resetting their content. This behavior suits scenarios where

calculations are specific to the current bar, and historical data retention is unnecessary.

Example script

Here, we initialize two arrays. Array a is declared without using the var keyword, while array b is declared with var, allowing us to observe and compare their behavior. Throughout the runtime, we incrementally add an element to each array on each bar. We use a table to present and compare both the sizes of these arrays and the number of chart bars, effectively illustrating the impact of different declaration methods on array behavior:

//@version=6 indicator("Using `var` with arrays") //@variable An array that initializes on every bar. a = array.new<float>() array.push(a, close) //@variable An array that expands its size by 1 on each bar. var b = array.new<float>(0) array.push(b, close) // Populate a table on the chart's last bar to display the sizes of the arrays and compare it to the number of chart bars. if barstate.islast var table displayTable = table.new(position.middle_right, 2, 3) table.cell(displayTable, 0, 0, "Array A size:", text_color = chart.fg_color, text_halign = text.align_right) table.cell(displayTable, 1, 0, str.tostring(a.size()), text_color = chart.fg_color, text_halign = text.align_left) table.cell(displayTable, 0, 1, "Array B size:", text_color = chart.fg_color, text_halign = text.align_right) table.cell(displayTable, 1, 1, str.tostring(b.size()), text_color = chart.fg_color, text_halign = text.align_left) table.cell(displayTable, 0, 2, "Number of chart bars:", text_color = chart.fg_color, text_halign = text.align_right) table.cell(displayTable, 1, 2, str.tostring(bar_index + 1), text_color = chart.fg_color, text_halign = text.align_left)

Results

- **Array A (Non-Persistent):** This array is reset at the beginning of each new bar. As a result, despite adding elements on each bar, its size remains constant, reflecting only the most recent addition.
- **Array B (Persistent):** This array retains its elements and accumulates new entries across bars, mirroring the growing count

of chart bars. This persistent nature of the array shows its ability to track or aggregate data over the script's runtime.

For further details, consult the sections concerning variable declaration modes and their use in array declarations in the User Manual.

What are queues and stacks?

Scripts can use arrays to create queues and stacks.

Stacks

A stack uses the "last in, first out" (LIFO) principle, where the most recently added item is the first to be taken away. Think of this like a stack of plates, where you can only place a new plate on top or remove the top plate. To use an array as a stack, add elements to the end of the array using array.push() and remove elements from the end of the array using array.pop().

Queues

A queue uses the "first in, first out" (FIFO) principle, where the first item to be added is the first to be removed. This kind of queue in code is like a queue in real life, such as in a coffee shop, where no matter how many people join the end of the queue, the first person still gets served first. To use an array as a queue, add elements to the end of the array using array.push() and remove them from the beginning using array.shift().

Stacks are particularly useful for accessing the most recent data, such as for tracking price levels. Queues are used for sequential data processing tasks, like event handling. Two example scripts follow, to illustrate these different usages.

Example: Arrays as stacks

This script uses arrays as stacks to manage pivot points. It draws lines from the pivot points and extends the lines with each new bar until price intersects them. When the script detects a pivot point, it adds (pushes) a new line to the stack. With each new bar, the script extends

the end point of each line in the stack. It then checks whether price has intersected the high or low pivot lines at the top of the stack. If so, the script removes (pops) the intersected line from the stack, meaning that it will no longer be extended with new bars. Note that we do not need to iterate through the arrays to check all the lines, because price is always between only the high and low pivot lines at the end of each array.

image

//@version=6 indicator("Array as a stack", overlay = true) // @function Adds a new horizontal line to an array of lines at a specified pivot level. // @param id (array<line>) The array to which to add the new line. // @param pivot (float) The price level at which to draw the horizontal line. // @param lineColor (color) The color of the line. // @returns (void) The function has no explicit return. stackLine(array<line> id, float pivot, color lineColor) => if not na(pivot) array.push(id, line.new(bar_index - 10, pivot, bar_index, pivot, color = lineColor)) // @function Extends the endpoint (`x2`) of each line in an array to the current `bar_index`. // @param id (array<line>) The array containing the line objects to update. // @returns (void) The function has no explicit return. extendLines(array<line> id) => for eachLine in id eachLine.set_x2(bar_index) // @function Removes line objects from an array if they are above or below the current bar's high or low. // @param id (array<line>) The array from which to remove line objects. // @param isBull (bool) If true, remove bullish pivot lines below the high price; // if false, remove bearish pivot line above the low price. // @returns (void) The function has no explicit return. removeLines(array<line> id, bool isBull) => if array.size(id) > 0 float linePrice = line.get_price(array.last(id), bar_index) if isBull ? high > linePrice : low < linePrice array.pop(id) line(na) // Find the pivot high and pivot low prices. float pivotLo = ta.pivotlow(10, 10), float pivotHi = ta.pivothigh(10, 10) // Initialize two arrays on the first bar to stack our lines in. var array<line> pivotHiArray = array.new<line>() var array<line> pivotLoArray = array.new<line>() // If a pivot occurs, draw a line from the pivot to the current bar and add the line to the stack. stackLine(pivotHiArray, pivotHi, color.orange) stackLine(pivotLoArray, pivotLo, color.aqua) // Extend all lines in each array to the current bar

on each bar. extendLines(pivotHiArray) extendLines(pivotLoArray) // Check the final element of each array to see if price exceeded the pivot lines. // Pop the line off the stack if it was exceeded. removeLines(pivotHiArray, true) removeLines(pivotLoArray, false)

Example: Arrays as queues

This script uses arrays as queues to track pivot points for monitoring recent support and resistance levels. It dynamically updates lines extending from the four most recent pivot highs and lows to the current bar with each new bar. When the script detects a new pivot high or low, it adds a line that represents this pivot to the respective queue. To maintain the queue's size at a constant four items, the script removes the oldest line in the queue whenever it adds a new line.

image

//@version=6 indicator("Array as a queue", overlay = true) int PIVOT_LEGS = 10 // @function Queues a new `value` at the end of the `id` array and removes // the first element if the array size exceeds the specified `maxSize`. // @param id (<any array type>) The array in which to queue the element. // @param maxSize (int) The maximum allowed number of elements in the array. // If the array exceeds this size, the first element is removed. // @param value (<type of the array>) The new element to add to the array. // @returns (<type of the array>) The removed element. arrayQueue(id, int maxSize, value) => id.push(value) if id.size() > maxSize id.shift() // @function Adds a new horizontal line to an array at a certain pivot level and removes the oldest line. // @param id (array<line>) The array to which to add the new line. // @param pivot (float) The price level at which to draw the horizontal line. // @param numLines (int) The number of lines to keep in the queue. // @param lineColor (color) The color of the line to draw. // @returns (void) The function has no explicit return. queueLine(array<line> id, float pivot, int numLines, color lineColor) => if not na(pivot) arrayQueue(id, numLines, line.new(bar_index - PIVOT_LEGS, pivot, bar_index, pivot, color = lineColor)) // @function Extends the endpoint (`x2`) of each line in an array to the current `bar_index`. // @param id (array<line>) The array containing the line objects to update. // @returns (void) The function has no explicit return.

extendLines(array<line> id) => for eachLine in id eachLine.set_x2(bar_index) // Find the pivot high and pivot low price. float pivotLo = ta.pivotlow(PIVOT_LEGS, PIVOT_LEGS) float pivotHi = ta.pivothigh(PIVOT_LEGS, PIVOT_LEGS) // Initialize two arrays on the first bar to queue our lines in. var array<line> pivotHiArray = array.new<line>() var array<line> pivotLoArray = array.new<line>() // If a pivot occurs, draw a line from the pivot to the current bar, add it to the queue, and remove the oldest line. queueLine(pivotHiArray, pivotHi, 4, color.orange) queueLine(pivotLoArray, pivotLo, 4, color.aqua) // Extend all lines in each array to the current bar on each bar. extendLines(pivotHiArray) extendLines(pivotLoArray)

For more information on manipulating arrays, see the Arrays section in the User Manual.

How can I perform operations on all elements in an array?

In Pine Script, there are no built-in functions to apply operations across the entire array at once. Instead, scripts need to iterate through the array, performing the operation on each element one at a time.

The easiest way to retrieve each element in an array is by using a for… in structure. This type of loop retrieves each element in turn, without the need for specifying the number of iterations.

The simple form of the loop has the format `for element in array`, where `element` is a variable that is assigned the current array element being accessed.

If the script's logic requires the position of the element in the array, use the two-argument form: `for [index, element] in array`. This form returns both the current element and its index in a tuple.

Example: retrieving array elements

This first example script uses an array as a queue to store lines representing the latest four pivot highs and lows. The for...in loop performs two tasks:

- It adjusts the x2 endpoint of each line to the current bar_index.
- It changes the colors of the lines to blue for support or orange for resistance, based on their position relative to the close price.

Note that neither of these operations requires knowing the index of the array element.

image

//@version=6 indicator("Example: `for...in` loop", overlay = true) // @function Queues a new `value` at the end of the `id` array and removes // the first element if the array size exceeds the specified `maxSize`. // @param id (<any array type>) The array in which to queue the element. // @param maxSize (int) The maximum allowed number of elements in the array. // If the array exceeds this size, the first element is removed. // @param value (<type of the array>) The new element to add to the array. // @returns (<type of the array>) The removed element. arrayQueue(id, int maxSize, value) => id.push(value) if id.size() > maxSize id.shift() // @function Adds a new horizontal line to an array at a certain pivot level and removes the oldest line. // @param id (array<line>) The array to which to add the new line. // @param pivot (float) The price level at which to draw the horizontal line. // @param numLines (int) The number of lines to keep in the queue. // @param lineColor (color) The color of the line to draw. // @returns (void) The function has no explicit return. queueLine(array<line> id, float pivot, int numLines, color lineColor) => if not na(pivot) arrayQueue(id, numLines, line.new(bar_index - 10, pivot, bar_index, pivot, color = lineColor)) // @function Extends the endpoint (`x2`) of each line in an array to the current `bar_index`. // @param id (array<line>) The array containing the line objects to update. // @returns (void) The function has no explicit return. extendLines(array<line> id) => for eachLine in id eachLine.set_x2(bar_index) // @function Adjusts the color of each line in an array. If the `close` is above the line, the line is // set to `bullColor` (support), else, `bearColor` (resistance). // @param id

(array<line>) The array containing the line objects. // @param bullColor (color) The color to apply to the line if `close` is equal to or higher than the line's price. // @param bearColor (color) The color to apply to the line if `close` is below the line's price. // @returns (void) The function has no explicit return. colorLines(array<line> id, color bullColor, color bearColor) => for eachLine in id if close >= eachLine.get_price(bar_index) eachLine.set_color(bullColor) else eachLine.set_color(bearColor) // Find the pivot high and pivot low prices. float pivotLo = ta.pivotlow(10, 10) float pivotHi = ta.pivothigh(10, 10) // Initialize two arrays on the first bar to queue our lines in. var array<line> pivotHiArray = array.new<line>(), var array<line> pivotLoArray = array.new<line>() // If a pivot occurs, draw a line from the pivot to the current bar, add it to the queue, and remove the oldest line. queueLine(pivotHiArray, pivotHi, 4, color.orange), queueLine(pivotLoArray, pivotLo, 4, color.aqua) // Extend all lines in each array to the current bar on each bar. extendLines(pivotHiArray), extendLines(pivotLoArray) // Set the color of lines as support or resistance by checking if the closing price is above or below the lines. colorLines(pivotHiArray, color.aqua, color.orange) colorLines(pivotLoArray, color.aqua, color.orange)

Example: retrieving array elements and indices

In our second script, we use the two-argument variant of the for...in loop to access elements and their indices in an array. This method facilitates operations that depend on element indices, such as managing parallel arrays or incorporating index values into calculations. The script pairs a boolean array with an array of positive and negative random integers. The boolean array flags whether each corresponding integer in the primary array is positive.

//@version=6 indicator("Example: `for...in` loop with index") // Create an array of random integers above and below 0. var valuesArray = array.from(4, -8, 11, 78, -16, 34, 7, 99, 0, 55) // Create an array to track the positive state of each integer. var isPos = array.new_bool(10, false) // Iterate over the valuesArray using a `for...in` loop and update each corresponding element in the bool array to true // if the value is above 0, or false if it is below 0. for [i, eachValue] in valuesArray if eachValue > 0 array.set(isPos, i, true) // Print both arrays in a label on

the last historical bar. if barstate.islastconfirmedhistory label.new(bar_index +1, high, str.tostring(valuesArray) + "\n" + str.tostring(isPos), style = label.style_label_left, textcolor = chart.fg_color)

What's the most efficient way to search an array?

The obvious way to search for an element in an array is to use a loop to check each element in turn. However, there are more efficient ways to search, which can be useful in different situations. Some of the following functions return only the index of a value. Programmers can then use array.get() if the script needs the actual value.

Checking if a value is present in an array

If all the script needs to do is to check whether a certain value is present in an array or not, use the array.includes() function. If the element is found, the function returns true; otherwise, it returns false. This method does not return the index of the element.

The following example script checks if the value 3 is present in the `values` array, and displays either "found" or "not found" in a label.

//@version=6 indicator("Example: Find whether an array element is present") array<int> values = array.from(1, 3, 5) int searchValue = input(3, "Value to Search For") bool valuePresent = array.includes(values, searchValue) if barstate.islast label.new(bar_index, low, valuePresent ? "Search value found" : "Search value not found", textcolor = color.white)

Finding the position of an element

If the script requires the *position* of an element, programmers can use the array.indexof() function. This function returns the index of the *first* occurrence of a value within an array. If the value is not found, the function returns `-1`. This method does not show whether there are multiple occurrences of the search value in the array. Depending on the

script logic, this method might not be suitable if the array contains values that are not unique.

The following script searches for the first occurrence of `101.2` in the `prices` array and displays “found” and the value’s index in a label, or “not found” otherwise.

```
//@version=6 indicator("Example: Find index of array element") array<float> prices = array.from(100.5, 101.2, 102.8, 100.5) float searchValue = input(101.2, "Value to Search For") int indexFound = array.indexof(prices, searchValue) if barstate.islast string lblString = switch indexFound < 0 => "Search value: not found" => "Search value: found\n Index: " + str.tostring(indexFound) label.new(bar_index, high, lblString, textcolor = color.white, textalign = text.align_left, text_font_family = font.family_monospace )
```

Binary search

If the script requires the position of the element in a sorted array, the function array.binary_search() returns the index of a value more efficiently than array.indexof(). The performance improvement is significant for large arrays. If the value is not found, the function returns `-1`.

This script uses a binary search to find the value `100.5` within an array of prices. The script displays the original array, the sorted array, the target value (100.5), and the result of the search. If the value is found, it displays “found”, along with the index of the value. If the value is not found, it displays “not found”.

```
//@version=6 indicator("Example: Binary search in sorted array") array<float> sortedPrices = array.from(100.5, 102.3, 98.7, 99.2) string originalArrayString = str.tostring(sortedPrices) float searchValue = input(100.5) // Ensure that the array is sorted (order is ascending by default); this step is crucial for binary search. array.sort(sortedPrices) string sortedArrayString = str.tostring(sortedPrices) int searchValueIndex = array.binary_search(sortedPrices, searchValue) bool valueFound = searchValueIndex >= 0 if barstate.islast string lblTxt = str.format("Original array: {0}\n Sorted Array: {1}\n Search value:
```

{2}\n Value found: {3}\n Position: {4}", originalArrayString, sortedArrayString, searchValue, valueFound, searchValueIndex) label.new(bar_index, high, lblTxt, textcolor = color.white, textalign = text.align_left, text_font_family = font.family_monospace)

If a script does not need the exact value, the functions array.binary_search_leftmost() and array.binary_search_rightmost() provide an effective way to locate the nearest index to a given value in sorted arrays. These functions return the index of the value, if it is present. If the value is not present, they return the index of the element that is closest to the search value on the left (smaller) or right (larger) side.

How can I debug arrays?

To debug arrays, scripts need to display the contents of the array at certain points in the script. Techniques that can display the contents of arrays include using plots, labels, tables, and Pine Logs.

For information about commonly encountered array-related errors, refer to the array Error Handling section in the User Manual.

Plotting

Using the plot() function to inspect the contents of an array can be helpful because this function can show numerical values on the script's status line, the price scale, and the Data Window. It is also easy to review historical values.

Limitations of this approach include:

- Arrays must be of type "float" or "int".
- The number of plots used for debugging counts towards the plot limit for a script.
- Plot calls must be in the global scope and scripts cannot call them conditionally. Therefore, if the size of the array varies across bars, using this technique can be impractical.

Here we populate an array with the open, high, low and close (OHLC) prices on each bar. The script retrieves all the elements of the array and plots them on the chart.

//@version=6 indicator("Plot array elements") array<float> ohlc = array.from(open, high, low, close) plot(ohlc.get(0), "Open", color.red) plot(ohlc.get(1), "High", color.yellow) plot(ohlc.get(2), "Low", color.blue) plot(ohlc.get(3), "Close", color.green)

Using labels

Using labels to display array values on certain bars is particularly useful for non-continuous data points or to view all elements of an array simultaneously. Scripts can create labels within any local scope, including functions and methods. Scripts can also position drawings at any available chart location, irrespective of the current bar_index. Unlike plots, labels can display the contents of a variety of array types, including boolean and string arrays.

Limitations of using labels include:

- Pine labels display only in the chart pane.
- Scripts can display only up to a maximum number of labels.

In the following example script, we monitor the close price at the last four moving average (MA) crosses in a queued array and use a label to display this array from a local scope whenever a cross occurs:

//@version=6 indicator("Array elements in a label", overlay = true) var array<float> crossPrices = array.new<float>(4) float fastMa = ta.ema(close, 9) float slowMa = ta.ema(close, 21) if ta.cross(fastMa, slowMa) crossPrices.push(close) crossPrices.shift() label.new(bar_index, high, str.tostring(crossPrices), textcolor = color.white) plot(fastMa, "Fast MA", color.aqua) plot(slowMa, "Slow MA", color.orange)

For more information, see the debugging with labels section in the User Manual.

Using label tooltips

If programmers want to be able to inspect the values in an array on every bar, displaying the contents of the array in a label is not convenient, because the labels overlap and become difficult to read. In this case, displaying the array contents in a label tooltip can be visually clearer. This method has the same advantages and limitations as using labels in the section above.

This example script plots a fast and a slow moving average (MA). It maintains one array of the most recent three values of the fast MA, and one array for the slow MA. The script prints empty labels on each bar. The tooltip shows the values of the MA arrays and whether or not the MAs crossed this bar. The labels are displayed in a semi-transparent color, and the tooltip is visible only when the cursor hovers over the label.

//@version=6 indicator("Array elements in a label tooltip", overlay = true) // Create two arrays to hold the MA values. var array<float> fastMaValues = array.new<float>(3) var array<float> slowMaValues = array.new<float>(3) // Calculate the MAs. float fastMa = ta.ema(close, 9) float slowMa = ta.ema(close, 21) // Load the current MA values into the arrays. fastMaValues.push(math.round(fastMa,2)), slowMaValues.push(math.round(slowMa,2)) // Remove the first element to keep the arrays at the same size. fastMaValues.shift(), slowMaValues.shift() // Define the string to print in the label tooltip. string labelString = str.format("Fast MA array: {0}\n Slow MA array: {1}\n Crossed this bar? {2}", str.tostring(fastMaValues), str.tostring(slowMaValues), ta.cross(fastMa, slowMa)) //Print the labels. label.new(bar_index, high, text="", color=color.new(chart.fg_color,90), textcolor = chart.fg_color, tooltip=labelString) plot(fastMa, "Fast MA", color.aqua) plot(slowMa, "Slow MA", color.orange)

Using tables

Using tables for debugging offers a more organized and scalable alternative to labels. Tables can display multiple “series” strings in a

clear format that remains unaffected by the chart's scale or the index of the bars.

Limitations of using tables for debugging include that, unlike labels, the state of a table can only be viewed from the most recent script execution, making it hard to view historical data. Additionally, tables are computationally more expensive than other debugging methods and can require more code.

In the following example script, we create and display two unrelated arrays, to show how flexible this approach can be. The first array captures the times of the last six bars where a Golden Cross occurred. The second array records the last eight bar indices where the Relative Strength Index (RSI) reached new all-time highs within the chart's history. We use the `whenSince()` function from the PineCoders' getSeries library to create and update the arrays. This function treats the arrays as queues, and limits their size.

//@version=6 indicator("Debugging arrays with tables", overlay = true) // Import the `getSeries` PineCoders library to build fixed-size arrays populated on specific conditions. // https://www.tradingview.com/v/Bn7QkdZR/ import PineCoders/getSeries/1 as PCgs // Calculate MAs and create cross condition. float ma50 = ta.sma(close, 50) float ma200 = ta.sma(close, 200) bool goldenCross = ta.cross(ma50, ma200) // Calculate the RSI and determine if it's hitting a new all-time high. float myRsi = ta.rsi(close, 20) bool newRsiAth = myRsi == ta.max(myRsi) // Create two arrays using the imported `whenSince()` function. array<float> goldenCrossesTimes = PCgs.whenSince(time_close, goldenCross, length = 6) array<float> barIndicesOfHiRSIs = PCgs.whenSince(bar_index, newRsiAth, length = 8) // Plot the MAs for cross reference. plot(ma50, "50 MA", color.aqua) plot(ma200, "200 MA", color.orange) // On the last historical bar, display the date and time of the last crosses. if barstate.islast // Declare our MA table to display the Golden Cross times. var table maTable = table.new(position.top_right, 2, 8, color.new(color.black, 100), color.gray, 1, color.gray, 1) // Create a title cell for the MA table and merge cells to form a banner two cells wide. table.cell(maTable , 0, 0, "Golden Cross Times", text_color = color.black, bgcolor = #FFD700) table.merge_cells(maTable , 0, 0, 1, 0) // Loop the array and write cells to the MA table containing the cross

time for each element of the array. Number each element in the left row. // Format the UNIX time value to a formatted time string using `str.format_time()`. for [i, timeValue] in goldenCrossesTimes table.cell(maTable, 0, i + 1, str.tostring(i + 1), text_color = #FFD700) table.cell(maTable, 1, i + 1, str.format_time(int(timeValue), "yyyy.MM.dd 'at' HH:mm:ss z"), text_color = chart.fg_color) // Create a second table to display the indices of the last eight RSI all-time highs. var table rsiTable = table.new(position.bottom_right, 1, 1, color.new(color.black, 100), color.gray, 1, color.gray, 1) table.cell(rsiTable, 0, 0, "Bar indices of RSI ATHs\n" + str.tostring(barIndicesOfHiRSIs), text_color = chart.fg_color)

Using Pine Logs

Pine Logs are messages that display in the Pine Logs pane, along with a timestamp when the logging function was called. Scripts can create log messages at specific points during the execution of a script. Pine Logs can display any script values or variables at any part of your code, including local scopes, functions, and loops.

By logging messages to the console whenever there is a modification to the array, programmers can track the logical flow of array operations in much more detail than by using other approaches.

The script below updates a previous example script from the section on queues and stacks to add logging. It uses arrays as stacks to track lines drawn from pivot points. When a pivot occurs, the script adds a new line to the stack and continues to extend the lines on each bar until an intersection with price occurs. If an intersection is found, the script removes (pops) the intersected line from the stack, meaning it will no longer be extended with new bars.

The console messages are time stamped and offer detailed information about when elements are added to and removed from the arrays, the current size of the arrays, and the specific prices at which elements were added.

image

```
//@version=6 indicator("Array as a stack", overlay = true) // @function Adds a new horizontal line to an array of lines at a specified pivot level. // @param id (array<line>) The array to which to add the new line. // @param pivot (float) The price level at which to draw the horizontal line. // @param lineColor (color) The color of the line. // @returns (void) The function has no explicit return. stackLine(array<line> id, float pivot, color lineColor) => if not na(pivot) array.push(id, line.new(bar_index - 10, pivot, bar_index, pivot, color = lineColor)) if barstate.isconfirmed log.info("\nNew line added at {0}\nArray size: {1}", pivot, id.size()) // @function Extends the endpoint (`x2`) of each line in an array to the current `bar_index`. // @param id (array<line>) The array containing the line objects to update. // @returns (void) The function has no explicit return. extendLines(array<line> id) => for eachLine in id eachLine.set_x2(bar_index) // @function Removes line objects from an array if they are above or below the current bar's high or low. // @param id (array<line>) The array from which to remove line objects. // @param isBull (bool) If true, remove bullish pivot lines below the high price; // if false, remove bearish pivot line above the low price. // @returns (void) The function has no explicit return. removeLines(array<line> id, bool isBull) => if array.size(id) > 0 float linePrice = line.get_price(array.last(id), bar_index) if isBull ? high > linePrice : low < linePrice array.pop(id) if barstate.isconfirmed log.warning( "\nLine removed from {0} array.\nPrice breached {1}\nArray size: {2}", isBull ? "Highs" : "Lows", linePrice, id.size()) // Find the pivot high and pivot low prices. float pivotLo = ta.pivotlow(10, 10), float pivotHi = ta.pivothigh(10, 10) // Initialize two arrays on the first bar to stack our lines in. var array<line> pivotHiArray = array.new<line>() var array<line> pivotLoArray = array.new<line>() // If a pivot occurs, draw a line from the pivot to the current bar and add the line to the stack. stackLine(pivotHiArray, pivotHi, color.orange), stackLine(pivotLoArray, pivotLo, color.aqua) // Extend all lines in each array to the current bar on each bar. extendLines(pivotHiArray), extendLines(pivotLoArray) // Check the final element of each array. If price exceeded the pivot lines, pop the line off the stack. removeLines(pivotHiArray, true), removeLines(pivotLoArray, false)
```

Can I use matrices or multidimensional arrays in Pine Script?

Pine Script does not directly support multidimensional arrays; however, it provides matrices and user-defined types (UDTs). Programmers can use these data structures to create and manipulate complex datasets.

Matrices

Pine Script matrices are like two-dimensional arrays. They organize data in a rectangular grid, facilitating operations like transformations, linear algebra, and other complex calculations. They are particularly useful for quantitative modeling, such as portfolio optimization, correlation matrix analysis, and more. Just as in arrays, all elements in a matrix must be of the same type, which can be a built-in or a user-defined type. Pine Script provides a range of functions for manipulating and performing calculations on matrices, including addition, subtraction, multiplication, and more.

Using UDTs for multidimensional structures

Programmers can achieve similar functionality to multidimensional arrays through defining user-defined types (UDTs). For example, a script can define a UDT that includes an array as one of its fields. UDTs themselves can be contained in arrays. In this way, scripts can effectively have arrays of arrays.

For more information, see the sections on Matrices, Maps, and Objects in the User Manual.

How can I debug objects?

To debug objects, create custom functions that break down an object into its constituent fields and convert these fields into strings. See the Debugging section of the User Manual for information about methods to display debug information. In particular, Pine Logs can display extensive and detailed debug information. See the FAQ section about

debugging arrays using Pine Logs for an explanation of using logs for debugging.

In our example script, we create a user-defined type (UDT) named `openLine`, which includes fields such as `price`, `openTime`, and a line object called `level`. On the first bar of each session, the script initializes a new `openLine` instance. This object tracks the session's opening price and time, and it draws a line at the open price, extending from the session's start to its close. An array stores each `openLine` object. A custom function `debugOpenLine()` breaks an `openLine` object into its individual fields, converts the fields to strings, and then logs a message that displays these strings in the console.

image

//@version=6 indicator("Debugging objects", overlay = true) // Define the user-defined type. type openLine float price int openTime line level // @function Queues a new `arrayElement` at the end of the `id` array and removes // the first element if the array size exceeds the specified `maxSize`. // @param id (<any array type>) The array in which the element is queued. // @param maxSize (int) The maximum allowed number of elements in the array. // If the array exceeds this size, the first element is removed. // @param arrayElement (<array type) The new element to add to the array. // @returns (<array type>) The removed element. arrayQueue(id, int maxSize, value) => id.push(value) if id.size() > maxSize id.shift() // @function Logs detailed information about an open line object for debugging purposes. // @param ol (openLine) The open line object to log. // @returns (void) Function has no explicit return. debugOpenLine(openLine ol) => if barstate.isconfirmed log.info("\nprice: {0}\nopenTime: {1}\nlevel line coords:\nx1: {2}\ny1: {3}\nx2: {4}\ny2: {5}", ol.price, ol.openTime, str.format_time(ol.level.get_x1()), ol.level.get_y1(), str.format_time(ol.level.get_x2()), ol.level.get_y2()) // Create an empty `openLine` array. var openLineArray = array.new<openLine>() // On session start, create a new `openLine` object and add it to the array. // Use the custom debug function to print the object's fields to the Pine Logs pane. if session.isfirstbar_regular openLine ol = openLine.new(open, time) ol.level := line.new(time, open,

time_close("D"), open, xloc.bar_time, color = color.aqua)
arrayQueue(openLineArray, 4, ol) debugOpenLine(ol)

Previous

Next

Functions

Pine Script® v6 Documentation

Functions

Can I use a variable length in functions?

Many built-in technical analysis (TA) functions have a `length` parameter, such as `ta.sma(source, length)`. A majority of these functions can process "series" lengths, i.e., lengths that can change from bar to bar. Some functions, however, only accept "simple" integer lengths, which must be known on bar zero and not change during the execution of the script.

Check the Reference Manual entry for a function to see what type of values a function can process.

Additional resources

For more advanced versions of functions that support "series" lengths, or for extra technical analysis tools explore the ta library on the TradingView profile. This library offers a range of extended TA-related capabilities and custom implementations.

User-defined functions

For built-in functions that do not accept "series" lengths and for which the functionality is not available in the ta library, consider creating a user-defined function.

How can I calculate values depending on variable lengths that reset on a condition?

To calculate certain values that are dependent on varying lengths, which also reset under specific conditions, the ta.barssince() function can be useful. This function counts the number of bars since the last occurrence of a specified condition, automatically resetting the count each time this condition is met. There are, however, some considerations to take into account when using this function for this purpose.

Firstly, before the condition is met for the first time in a chart's history, ta.barssince() returns na. This value is not usable as a length for functions and can cause errors, especially during execution on a chart's early bars. For a more robust version, use nz() to replace the na return of ta.barssince() with zero for early bars.

Secondly, when the condition is met, ta.barssince() returns zero for that bar, since zero bars have elapsed since the condition was last true.

Since lengths cannot be zero, it is necessary to add one to a returned value of zero, ensuring that the length is always at least one.

Here's an example of how to use these principles for a practical purpose. The following example script calculates the highest and lowest price points since the start of a new day. We use timeframe.change() to detect the start of a new day, which is our condition. The ta.barssince() function calculates the number of bars that elapsed since this condition was last met. The script passes this number, or "lookback", to the ta.lowest() and ta.highest() functions, which determine the highest and lowest points since the start of the new day:

image

//@version=6 indicator("Highest/lowest since new day", "", true) // Identify the start of a new day and calculate the number of bars since

then. bool newDay = timeframe.change("D") int lookback = nz(ta.barssince(newDay)) + 1 // Calculate the highest and lowest point since the new day began. float lowestSinceNewDay = ta.lowest(lookback) float highestSinceNewDay = ta.highest(lookback) // Plot the high/low level since the start of a new day. plot(lowestSinceNewDay, "High today", color.orange) plot(highestSinceNewDay, "Low today", color.aqua) // Change the background color to indicate the start of a new day. bgcolor(newDay ? color.new(color.gray, 80) : na) // Display the varying lookback period in Data Window. plot(lookback, "Lookback", display = display.data_window)

How can I round a number to x increments?

Rounding numbers to specific increments is useful for tasks like calculating levels for grid trading, dealing with fractional shares, or aligning trading parameters to specific pip values.

In this example, the `roundToIncrement()` function accepts a value and an increment as parameters. It divides the value by the increment, rounds the result, then multiplies it by the increment to give the rounded value. To demonstrate the function, the closing price is rounded to the nearest increment defined in the user menu:

//@version=6 indicator("Round to x increment demo", overlay = true) float incrementInput = input.float(0.75, "Increment", step = 0.25) // @function Rounds a value to the nearest multiple of a specified increment. // @param value The value to round. // @param increment The increment to round the value to. // @returns The rounded value. roundToIncrement(value, increment) => math.round(value / increment) * increment plot(series = roundToIncrement(close, incrementInput), color = chart.fg_color)

How can I control the precision of values my script displays?

The `precision` and `format` arguments in the indicator() or strategy() declaration statement control the number of decimals in the values that a script displays.

By default, scripts use the precision of the price scale. To display more decimal places, specify a `precision` argument that exceeds the value of the current price scale.

How can I control the precision of values used in my calculations?

The `math.round(number, precision)` variation of the math.round() function rounds values according to a specified precision. Alternatively, the math.round_to_mintick() function rounds values to the nearest tick precision of the chart's symbol.

How can I round to ticks?

To round values to the tick precision of a chart's symbol, use the function math.round_to_mintick(). To convert the resulting number to a string, use `str.tostring(myValue, format.mintick)` to first round the number to tick precision and then return its string representation, where `myValue` is the number to convert into a rounded string.

How can I abbreviate large values?

There are different ways to abbreviate large numerical values, such as volume. For instance, the number 1,222,333.0 can be simplified to 1.222M. Here are some methods to accomplish this:

Apply a global setting

Use the argument `format = format.volume` within either the indicator() or strategy() statements. Using this setting, displays all values in the script in their abbreviated forms.

Abbreviate specific values

To abbreviate only certain values for string display, use the `str.tostring(value, format.volume)` function.

Use a custom function

To specify a custom precision or abbreviate values up to trillions, use a custom function. In the following example script, the user-defined function `abbreviateValue()` divides the `value` by a power of ten based on its magnitude, and adds an abbreviation letter (K, M, B, or T) to represent the magnitude of the original value. The function also adds a subtle space between the value and the magnitude letter. The `print()` function displays the value on the chart for visualization.

//@version=6 indicator("Value abbreviation example") // @function Converts a numeric value into a readable string representation featuring the appropriate order // of magnitude abbreviation (K, M, B, T). // @param value (float) The value to format. // @param precision (string) The numerical precision of the result. ("" for none, ".00" for two digits, etc.) // @returns (string) The formatted value as a string with the appropriate abbreviation suffix. abbreviateValue(float value, string precision) => float digitsAmt = math.log10(math.abs(value)) string formatPrecision = "#" + precision string result = switch digitsAmt > 12 => str.tostring(value / 1e12, formatPrecision + " T") digitsAmt > 9 => str.tostring(value / 1e9, formatPrecision + " B") digitsAmt > 6 => str.tostring(value / 1e6, formatPrecision + " M") digitsAmt > 3 => str.tostring(value / 1e3, formatPrecision + " K") => str.tostring(value, "#" + formatPrecision) print(formattedString) => var table t = table.new(position.middle_right, 1, 1) table.cell(t, 0, 0, formattedString, bgcolor = color.yellow) print(abbreviateValue(volume, ".00"))

How can I calculate using pips?

Use the custom function `calcBaseUnit()` in the following example script to return the correct pip value for Forex symbols, or the base unit of change for non-forex symbols:

//@version=6 indicator("Pip calculation example") // @function Calculates the chart symbol's base unit of change in asset prices. // @returns (float) A ticks or pips value of base units of change. calcBaseUnit() => bool isForexSymbol = syminfo.type == "forex" bool isYenQuote = syminfo.currency == "JPY" bool isYenBase = syminfo.basecurrency == "JPY" float result = isForexSymbol ? isYenQuote ? 0.01 : isYenBase ? 0.00001 : 0.0001 : syminfo.mintick // Call the function and plot the result in a label var label baseUnitLabel = na if barstate.islast baseUnitLabel := label.new(x=bar_index + 1, y=open, text="Base Unit: " + str.tostring(calcBaseUnit(), "#.######"), style=label.style_label_left, color=color.new(color.blue, 0), textcolor=color.white) label.delete(baseUnitLabel[1])

How do I calculate averages?

The method of calculating averages depends on the type of values to average.

Distinct variables

To find the average of a small number of discrete variables, use the function math.avg(). Simply pass each of the variables as an argument to this function.

Bar prices

To find the average price of a single bar, use the built-in variables hl2, hlc3, and ohlc4.

Series values

To compute the average of the last *n* values in a series, use the function ta.sma().

Custom datasets

To average a custom set of values, organize them into an array and use array.avg(). For complex datasets, programmers can use the matrix.avg() function to average the contents of a matrix. For a deeper understanding of averaging custom datasets, refer to this conditional averages publication.

How can I calculate an average only when a certain condition is true?

The usual methods of calculating averages, which were discussed in the calculating averages section above, apply across *all* data points in a range. To calculate averages of only those values that occur under specific conditions, calculate *conditional averages* using custom functions.

The example script below imports a library called ConditionalAverages and uses two of its functions:

- The `avgWhen()` function calculates the average volume of session opening bars across the entire dataset.
- The `avgWhenLast()` function averages the opening volumes for the last five session opening bars.

The condition for these conditional averages is *session opening bars*, which we determine using the session.isfirstbar_regular variable.

image

//@version=6 indicator("Average session opening volume") import PineCoders/ConditionalAverages/1 as PCca // Color aqua for the session's opening bar, otherwise distinct colors for up/down volume columns. color volumeColor = switch session.isfirstbar_regular => color.aqua close > open => color.new(#D1D4DC, 65) => color.new(#787B86, 65) // Plot the volume columns. plot(volume, "volume", volumeColor, 4, plot.style_histogram) // Average volume over *all* session opening bars in the dataset. plot(PCca.avgWhen(src = volume, cond = session.isfirstbar_regular), "avg. When", #FF00FF) //

Average volume over the last five opening bars.
plot(PCca.avgWhenLast(src = volume, cond = session.isfirstbar_regular, cnt = 5), "avgWhenInLast()", #00FF00)

How can I generate a random number?

Use the math.random() function to generate random numbers. This example script creates a circle plot with random RGB color values and a random y value between 0 and 1:

//@version=6 indicator("Random demo", overlay = false) // Generate a random price value (the default range is 0 to 1). float y = math.random() // Generate a color with red, green, and blue values as separate random values between 0 and 255. color plotColor = color.rgb(math.random(0, 255), math.random(0, 255), math.random(0, 255)) plot(series = y, title = "Random number", color = plotColor, linewidth = 2, style = plot.style_circles)

How can I evaluate a filter I am planning to use?

To evaluate a filter, insert your filter code into the Filter Information Box - PineCoders FAQ script. This script conducts an impulse response analysis and shows the filter's characteristics in a label on the chart.

For further details and a guide on integrating your filter into the code, refer to the publication's description.

What does nz() do?

The nz() function replaces any na values with zero, or with a user-defined value if the `replacement` argument is specified. This function helps to prevent na values from interfering with calculations.

The following example script shows an exaggerated failure as a result of a single na value. The `barRangeRaw` variable is na only once, on the first bar, because it references a bar that does not exist, using the

history-referencing operator. The alternative variable `barRangeWithNz` uses nz() to prevent an na value from ever occurring.

The `dependentCalculation` variable takes one of these values and uses it to calculate a crude average of the bar range. If the input to this calculation is ever na, the series will be na forever after that.

Choose between the two values for bar range using the input setting, and the range either displays or not. In the latter case, the Data Window shows that the value of `dependentCalculation` is `0`, meaning na.

//@version=6 indicator("`na` values on first bar demo") bool useNzInput = input.bool(true, "Use `nz` to ensure value is never na") // This variable is na on the first bar. float barRangeRaw = close - close[1] // This variable is never na. float barRangeWithNz = close - nz(close[1], open) // Choose the value to use based on the input float barRange = useNzInput ? barRangeWithNz : barRangeRaw // Perform a calculation that depends on the barRange var float dependentCalculation = 0 dependentCalculation := ((dependentCalculation + barRange)/2) // Plot the results plot(dependentCalculation, title="Average Bar Range")

The nz() function is also useful to protect against any potential divide-by-zero errors. It guarantees a return value even when an equation unintentionally features a zero in the denominator. Consider the following code snippet that intentionally creates a divide-by-zero scenario by setting the denominator to zero. Without the nz() function, this expression would return na, instead of zero:

float dbzTest = nz(close / (close - close))

Previous

Next

Indicators

Pine Script® v6 Documentation

User Manual / FAQ / Indicators

Indicators

Can I create an indicator that plots like the built-in Volume or Volume Profile indicators?

The Volume and Visible Range Volume Profile indicators (along with some other built-in indicators) are written in Java. They display data on the main chart pane in a unique way:

- The bars are anchored to the bottom or right edge of the chart, not to an absolute x or y value.
- The length of the bars is a relative percentage of the available space and is not an absolute price or number of bars.
- The length of the bars adjusts automatically according to the data from the range of bars that are visible on the chart. The lengths of the bars are normalized so as never to appear too small or too large.
- The width of the bars adjusts automatically to fit the visible space.

It is difficult for Pine Script® indicators to plot values in the same way.

Limitations of `plot.style_columns`

If volume, or another series, plotted as columns, it is anchored to the bottom of the chart, and the width and length of the bars can adjust dynamically. However, the tops of the bars are defined by absolute price values. This means that it is not possible for the series to be plotted on the main chart without distorting the price scale. Also, plots

must be defined during processing of the bar they are plotted on, and cannot be plotted retroactively.

Limitations of drawings

Drawing objects such as lines and boxes are anchored to an absolute price scale, not to the edge of the chart. Drawing objects do not adjust their length automatically. Lines do not adjust their width automatically. Although boxes can be drawn exactly one bar wide, and so adjust their width automatically, they cannot be drawn so as to fit exactly in one bar; they always draw from the middle of one bar to the middle of another.

The following example script demonstrates some techniques for approximating the way that the built-in Volume indicator displays.

- We use the chart.right_visible_bar_time and chart.left_visible_bar_time built-in variables, through the PineCoders' VisibleChart library, to define the bars that are visible. Then we calculate the highest and lowest price, and the highest volume, for that period.
- We plot lines retroactively, after the visible window and all related values are known.
- We anchor the lines below the lowest visible price, so that it looks as if they are anchored to the bottom edge of the chart.
- We scale the length of all the volume bars so that the longest bar in the set is approximately 30% of the screen height, similar to the built-in Volume indicator.
- We adjust the width of the lines depending on how many bars are visible.

//@version=6 indicator("Dynamically scaled volume", overlay=true, max_lines_count=500) // Import the PineCoders' VisibleChart library import PineCoders/VisibleChart/4 as visibleChart const float RELATIVE_HEIGHT = 0.3 // 30% matches the built-in volume indicator. const string BOTTOM_TTIP = "Copy the bottom margin % from your chart settings to here, and then set the bottom margin to *zero* on the chart settings." int bottomInput = input.int(title = "Bottom Margin %", defval = 10, minval = 0, maxval = 100, tooltip = BOTTOM_TTIP) // Get

the highest volume, and highest and lowest price points, by calculating on each bar during the visible window. var float hiVol = na var float hiPrice = na var float loPrice = na if visibleChart.barIsVisible() hiVol := na(hiVol) ? volume : math.max(hiVol, volume) hiPrice := na(hiPrice) ? high : math.max(hiPrice, high) loPrice := na(loPrice) ? low : math.min(loPrice, low) int bars = visibleChart.bars() // Calculate the thickness for the lines based on how many bars are displayed. int lineWidth = math.ceil(1000/bars) // Draw the lines once, when the visible window ends. if time == chart.right_visible_bar_time // Calculate the bottom y coordinate for all lines once. float priceDifference = hiPrice - loPrice float scale = (priceDifference / hiVol) * RELATIVE_HEIGHT float bottomY = loPrice - (bottomInput / 100) * priceDifference // Loop through the visible window using the historical operator. for i = bars - 1 to 0 // Calculate the top y coordinate for each line. float topY = bottomY + (volume[i] * scale) // Draw the line. line.new(x1 = bar_index - i, y1 = bottomY, x2 = bar_index - i, y2 = topY, color = close[i] >= open[i] ? color.new(color.green, 50) : color.new(color.red, 50), width = lineWidth)

This script has some other limitations:

- The lines do not begin from the bottom of the chart if other indicators display plots or drawings below that level.
- In common with any script that uses the chart.right_visible_bar_time or chart.left_visible_bar_time built-in variables, the script must refresh each time the chart is moved or a new bar appears.
- There is a maximum limit of 500 lines per script.
- The width of the lines is calculated based on how many bars are visible. However, a Pine script has no way of knowing how much blank space there is to the right of the final bar. If the user scrolls to the right, the lines can appear too wide and overlap each other.

Can I use a Pine script with the TradingView screener?

The TradingView screener uses only its built-in filters, and cannot use a Pine script. Search for “screener” in the Community Collection to find

scripts that use the request.security() function to screen pre-set lists of symbols.

See also this FAQ entry for an example script that generates alerts on multiple symbols. Note that using several `request.security()` calls can cause scripts to compile and run more slowly than expected.

How can I use the output from one script as input to another?

Scripts with an input of type input.source() can take a plot from another script (up to a maximum of ten) as an input. Select the script and plot to take as input in the script's "Settings" tab. If the user removes the script from the chart and adds it again, they must select the correct inputs again.

The sources used as external inputs must originate from indicators; they cannot originate from strategies. However, plots originating from indicators *can* be used in strategies.

For further information, refer to this blog post and the Source input section in the User Manual.

Can my script draw on the main chart when it's running in a separate pane?

Scripts that have the `overlay` parameter in the indicator() or strategy() functions set to `false` appear in a separate pane to the main chart. Such scripts can affect the display of the main chart in only two ways:

- Changing bar colors, using the barcolor() function.
- Forcing plots to overlay, using `force_overlay = true` in the plotting function. The `force_overlay` parameter is available in most functions that draw on the chart.

Is it possible to export indicator data to a file?

The option “Export chart data…” in the dropdown menu at the top right corner of the chart exports a comma-separated values (CSV) file that includes time, OHLC data, and any plots generated by your script. This option can also export strategy data.

To include specific information in the CSV file, ensure that it is plotted by the script. If this extra information is far from the symbol’s price and the existing indicator plots, and plotting it on the chart could distort the scale of the script, or if you prefer not to display certain plots, consider using the `display` parameter in the plot() function.

Here is an example plot that displays the close only in the Data Window. The plot title “No chart display” becomes the column header for this value in the CSV file.

```
plot(close * 0.5, "No chart display", display = display.data_window)
```

Alternatively, the “Scale price chart only” in the chart settings maintains the script’s scale. To access these settings, right-click on the chart’s price scale.

To determine if a condition is true or false, use the plotshape() function, which records a 1 (for true) or 0 (for false) in the CSV file.

Previous

Next

Other data and timeframes

Other data and timeframes

What kinds of data can I get from a higher timeframe?

Generally speaking, the request.security() function can get the same kinds of data from another timeframe that is available on the chart timeframe. Scripts can retrieve built-in variables like open, high, low, close, volume, and bar_index.

The request.security() function can also evaluate built-in or user-defined functions in the requested *context* (timeframe and symbol). For example, the following example script retrieves the Average True Range (ATR) value from the daily (1D) timeframe by passing the ta.atr() function as the `expression` argument.

//@version=5 indicator("HTF ATR") float higherTfAtr = request.security(symbol = syminfo.tickerid, timeframe = "1D", expression = ta.atr(14)) plot(higherTfAtr)

Which `security.*` function should I use for lower timeframes?

The request.security() function is intended for accessing data at timeframes that are equal to or higher than the chart's current timeframe. It is *possible* to retrieve data from lower timeframes (LTF) using this function. However, the function returns the value from only *one* LTF bar within the chart's current bar (the last bar, by default).

If the script supplies the `expression` as a variable or simple calculation, directly or within a function, the data that request.security() returns from a lower timeframe is generally of limited use (see the first script in this section for an example). It is possible, however, to construct a function that performs meaningful calculations on the LTF bars and then returns the result on the last bar. The following example script counts the number of LTF bars in a chart bar and returns this value on the last LTF bar. For simplicity, the timeframes are hardcoded to `"1D"` and `"1W"` and the script should therefore be run from a chart on the weekly timeframe.

//@version=5 indicator("Counting intrabars using `request.security()`") // @function Calculates the quantity of 1D bars in a week of trading. // @returns (int) The number of intrabars within the current weekly bar up to the current moment. qtyIntrabars() => var int count = 0 count := timeframe.change("W") ? 1 : count + 1 int qtyIntrabars = request.security(syminfo.tickerid, "1D", qtyIntrabars()) plot(qtyIntrabars, "qtyIntrabars", style=plot.style_histogram)

When using the request.security() function on a lower timeframe, all calculations that reference individual LTF bars must be done *within the requested context*, and only the *result* of the calculation is returned. Using the request.security_lower_tf() function for intrabar analysis is usually easier and more powerful, because it returns an array of data from *all* available intrabars within a chart bar. Returning the data for each bar allows scripts to perform calculations on specific bars or all bars in the main script context.

In the following version of our example script, we use request.security_lower_tf() to perform the same calculations. With this approach, we do not need to explicitly define the current chart’s timeframe, nor do we need a custom function.

//@version=5 indicator("Counting intrabars using `request.security_lower_tf()`") // Count the number of elements in the array of close prices for each LTF bar in the current chart's bar. int qtyIntrabars = array.size(request.security_lower_tf(syminfo.tickerid, "1D", close)) plot(qtyIntrabars, "qtyIntrabars", style=plot.style_histogram)

See the sections in the User Manual page “Other timeframes and Data” about `request.security_lower_tf()` and using `request.security()` on lower timeframes to learn more about the differences between running these functions on a lower timeframe.

How to avoid repainting when using the `request.security()` function?

Repainting can be a problem when retrieving data from higher or lower timeframes using request.security().

Retrieving data from a different symbol on the chart’s timeframe does not risk repainting. Requesting data from the chart’s own symbol and timeframe does not result in repainting either, but it is usually unnecessary to use request.security() rather than simply referencing the chart’s own values (except when modifying the chart’s ticker using `ticker.*()` functions). When using the chart’s timeframe, there is no need to offset the data, change the default `lookahead` value, or use barmerge.lookahead_on in order to avoid repainting.

Higher timeframes

Values from a *higher timeframe* (HTF) often repaint because a historical bar on the chart might include data from a realtime bar on the HTF. Realtime values can change throughout the bar; for example, the close price reflects the *latest* price update in a realtime bar. When the HTF bar closes and its values become fixed, the relevant historical chart bars *change* to adjust to the fixed HTF values. This behavior is described in the Historical and realtime behavior section of the User Manual. Users expect historical bars not to change, which is one reason why repainting is such a concern.

To prevent repainting, use confirmed values that remain consistent across all bars. The most robust method is to offset all expressions by 1. For example, instead of `close`, which is equivalent to `close[0]`, use `close[1]`. The request.security() call must also use barmerge.lookahead_on. This method returns data that is up to one HTF bar “late”, and is thus not subject to change.

The following example script demonstrates the use of a single bar offset to the `expression` argument and barmerge.lookahead_on in request.security() to ensure that the data behaves the same on historical and realtime bars. The script calls runtime.error() to trigger a custom runtime error if the chart's timeframe exceeds or matches the daily timeframe, to prevent the return of inaccurate values.

//@version=5 indicator("HTF close" , overlay = true) float dailyClose = request.security(syminfo.tickerid, "1D", close[1], lookahead = barmerge.lookahead_on) plot(dailyClose) if timeframe.in_seconds() >= timeframe.in_seconds("1D") runtime.error("Chart timeframe must be less than 1D.")

See the Avoiding repainting section of the User Manual for more information.

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 a lower timeframe (LTF), with limitations. When accessing data from a LTF, the function evaluates the given expression in the LTF context and returns the result from a *single* LTF bar per chart bar. The specific LTF bar returned depends on the lookahead parameter:

- barmerge.lookahead_on returns the *first* intrabar of the period historically, but the *last* intrabar in realtime.
- barmerge.lookahead_off always returns the last intrabar for both historical and realtime data. To prevent repainting (in this case, inconsistent results between realtime and historical data) use barmerge.lookahead_off for lower timeframe data requests.

In most cases, using the request.security_lower_tf() function is more suitable for lower timeframes, as it returns an array containing data from *all* available intrabars within a chart bar. See the section on `request.security_lower_tf()` to learn more.

How can I convert the chart’s timeframe into a numeric format?

The timeframe.in_seconds() function converts a timeframe specified in timeframe.period format into an equivalent number of seconds. Having the timeframe in a numeric format means that scripts can calculate the number of time units within a specific timeframe, or perform operations that adjust the timeframe used in HTF calls in relation to the chart’s timeframe, as described in this FAQ entry.

In this script example, we use the timeframe.in_seconds() function to determine the chart’s timeframe, measured in seconds. Since no specific `timeframe` argument is specified, the function defaults to using `timeframe.period`, which returns the chart’s current timeframe. The script then converts the timeframe in seconds into various other units of time, including minutes, hours, and days, and displays the original string and converted numeric values in a table:

//@version=5 indicator("Timeframe to value") tfInSec = timeframe.in_seconds() tfInMin = tfInSec / 60 tfInHrs = tfInMin / 60 tfInDays = tfInHrs / 24 if barstate.islastconfirmedhistory var table displayTable = table.new(position.top_right, 2, 5, na, color.gray, 1, color.gray, 1) table.cell(displayTable, 0, 0, "Original TF string", text_color = chart.fg_color) table.cell(displayTable, 1, 0, "\"" + timeframe.period + "\"", text_color = chart.fg_color) table.cell(displayTable, 0, 1, "Timeframe in seconds", text_color = chart.fg_color) table.cell(displayTable, 1, 1, str.tostring(tfInSec), text_color = chart.fg_color) table.cell(displayTable, 0, 2, "Timeframe in minutes", text_color = chart.fg_color) table.cell(displayTable, 1, 2, str.tostring(tfInMin), text_color = chart.fg_color) table.cell(displayTable, 0, 3, "Timeframe in hours", text_color = chart.fg_color) table.cell(displayTable, 1, 3, str.tostring(tfInHrs), text_color = chart.fg_color) table.cell(displayTable, 0, 4, "Timeframe in days", text_color = chart.fg_color) table.cell(displayTable, 1, 4, str.tostring(tfInDays), text_color = chart.fg_color)

How can I convert a timeframe in “float” minutes into a string usable with request.security()?

The built-in function timeframe.from_seconds() function converts a number of seconds into a timeframe string that is compatible with request.security().

The example script below converts a user-defined number of minutes into a timeframe string using the timeframe.from_seconds() function. The script then requests the close price from that timeframe using request.security() and plots it. Additionally, we display the resulting timeframe string in a table on the chart’s top right corner:

//@version=5 indicator("Target TF in string from float minutes", "", true) float tfInMinInput = input.float(1440, "Minutes in target timeframe (<= 0.0167 [1 sec.])", minval = 0.0167) // Convert target TF in minutes from input into string. string targetTfString = timeframe.from_seconds(int(tfInMinInput * 60)) // Fetch target timeframe's close. float targetTfClose = request.security(syminfo.tickerid, targetTfString, close) // Plot target timeframe close. plot(targetTfClose, "Target TF close") // Display the target timeframe string in a table cell at the chart's top right. if barstate.islastconfirmedhistory var table displayTable = table.new(position.top_right, 1, 1, color.new(color.yellow, 70), color.gray, 1, color.gray, 1) table.cell(displayTable, 0, 0, str.format("Target TF (string): {0}", targetTfString), text_color = chart.fg_color)

How do I define a higher timeframe that is a multiple of the chart timeframe?

This example script uses the timeframe.in_seconds() and timeframe.from_seconds() functions to calculate a higher timeframe that is a fixed multiple of the chart’s current timeframe. Using the input for the multiplier, the user can define the ratio between the chart’s

timeframe and the higher timeframe. The script then calculates the Relative Strength Index (RSI) for both the chart's timeframe and the higher timeframe, plotting both in a separate pane for comparison. We display the calculated higher timeframe string in a table on the main chart pane by using `force_overlay`:

//@version=5 indicator("Multiple of current TF", overlay = false) // Provide an input to specify the multiple to apply to the chart's timeframe. float tfMult = input.float(4, minval = 1) // Get multiple of current timeframe. string targetTfString = timeframe.from_seconds(int(timeframe.in_seconds() * tfMult)) // Create RSI from the current timeframe. float myRsi = ta.rsi(close, 14) plot(myRsi, "Current TF RSI", color = color.silver) // Non-repainting HTF RSI. float myRsiHtf = request.security(syminfo.tickerid, targetTfString, myRsi[1], lookahead = barmerge.lookahead_on) plot(myRsiHtf, "Non-repainting HTF RSI", color = color.green) // Display the calculated timeframe at the top right of the main chart pane. if barstate.islastconfirmedhistory var table displayTable = table.new(position.top_right, 1, 1, color.new(color.yellow, 70), color.gray, 1, color.gray, 1, force_overlay = true) table.cell(displayTable, 0, 0, str.format("Target TF (string): {0}", targetTfString), text_color = chart.fg_color)

How can I plot a moving average only when the chart's timeframe is 1D or higher?

To plot a moving average on a chart only if it has a timeframe of daily ("1D") or higher, scripts can use the timeframe.in_seconds() function to convert the chart's current timeframe into seconds. Since a day consists of 86400 seconds, any timeframe equal to or exceeding this value corresponds to a daily or longer duration.

The example script below calculates and plots a Simple Moving Average (SMA) of the closing prices over the last 200 bars. The script uses a ternary operator to return the moving average on timeframes of 1D or greater, or na if the timeframe is shorter than one day. Because

plot() calls cannot be in a local scope, scripts cannot conditionally call this function. Passing an na value as the `series` argument is an effective way to not plot anything. Note that plotting an na value *does* count towards the script's plot limit.

//@version=5 indicator("Timeframe-dependent MA", overlay = true) bool tfIsDailyOrGreater = timeframe.in_seconds() >= 86400 float ma = ta.sma(close, 200) plot(tfIsDailyOrGreater ? ma : na, "MA", color.aqua)

What happens if I plot a moving average from the 1H timeframe on a different timeframe?

The request.security() function can access data from a different *context*, such as a different symbol or timeframe. There are different considerations when accessing data from a timeframe *higher* or *lower* than the chart timeframe.

First, let's consider an example of plotting data from a *lower* timeframe. The following script plots a 21-period Exponential Moving Average (EMA) derived from the 1-hour (1H) timeframe on any chart, irrespective of the timeframe of that chart:

//@version=5 indicator("1hr EMA", overlay = true) plot(request.security(syminfo.tickerid, "60", ta.ema(close, 21)), color = color.orange)

Assuming that we run this script on a chart with a daily timeframe, we encounter the following problems:

- For each daily bar, the chart can plot only 1 of the 24 MA values theoretically available. The plot misses out the intraday fluctuations and trends that a 1H moving average (MA) is typically used to identify.
- The script above displays only the EMA value calculated for the *final* 1-hour bar of each day. In realtime, the plot displays the most recently known value.

Unlike request.security(), the request.security_lower_tf() function is intended for use on lower timeframes. It returns an array containing data from all available intrabars within a chart bar. See this section of the User Manual to learn more.

We could rewrite the script to use request.security_lower_tf(), but plotting a moving average from a lower timeframe is still not very practical.

A more logical approach is to plot MAs from a *higher* timeframe. This strategy shows broader market trends within the context of shorter-term price movements. For example, plotting a daily MA on a 1H chart provides insights into how intraday prices are trending relative to the longer-term daily average.

In the following example script, we plot the 21 EMA calculated at the 1H timeframe, but only when the chart's timeframe is *equal to or lower than* 1H. We call the request.security() function in the recommended way to avoid repainting.

//@version=5 indicator("HTF EMA", overlay = true) // Input to specify the timeframe for `request.security() call. string tfinput = input.timeframe("60", "Timeframe for MA") // @function A wrapper for the `request.security()` function for non-repainting calls to HTFs. // @param timeframe Timeframe of the requested data. // To use the chart's timeframe, use an empty string or the `timeframe.period` variable. // @param expression An expression to calculate and returne from the request.security() call's context. // @returns The result of the calculated expression. htfSecurity(string timeframe, expression) => result = request.security(syminfo.tickerid, timeframe, expression[1], lookahead = barmerge.lookahead_on) // Calculate the moving average in the chart context. float ma = ta.ema(close, 21) // Calculate the moving average in the specified `tfInput` timeframe. float htfMA = htfSecurity(tfinput, ma) // Check whether the requested timeframe is greater or less than the chart's timeframe. bool tfIsGreater = timeframe.in_seconds() < timeframe.in_seconds(tfinput) bool tfIsLess = timeframe.in_seconds() > timeframe.in_seconds(tfinput) // Plot the HTF MA, the chart MA, or nothing, depending on the timeframe. float maPlot = tfIsGreater ? htfMA : tfIsLess ? na : ma plot(maPlot, "Requested MA",

color.orange) // Display a message in a table indicating that the requested timeframe is lower than the chart's timeframe, if applicable. if barstate.islastconfirmedhistory and tfIsLess var table displayTable = table.new(position.bottom_right, 1, 1, color.new(color.yellow, 70)) table.cell(displayTable, 0, 0, "Requested TF is lower than chart's TF\nNo MA displayed", text_color = color.red)

Why do intraday price and volume values differ from values retrieved with `request.security()` at daily timeframes and higher?

Intraday open, high, low, close, and volume (OHLCV) values can be different from those from request.security() at daily timeframes and higher for a number of reasons, including the following:

- **Different data feeds:** Certain trades (like block trades and OTC trades, especially in stocks) are recorded only at the end of the trading day, so their volume affects the End-of-Day (EOD) feed but not the intraday feed.
- **Price discrepancies:** There can be slight differences in prices between EOD and intraday data. For example, an EOD high might not match any intraday highs due to variations in data feeds.
- **Extended hours data:** EOD data feeds do not include information from trading outside regular hours, unlike some intraday feeds. For instance, the bars of an hourly chart might straddle the open of a session, mixing data from pre-market and regular trading.

For an extended list of factors with detailed explanations, refer to the Data feeds section in the User Manual.

Previous

Next

Programming

Pine Script® v6 Documentation

User Manual / FAQ / Programming

Programming

What does “scope” mean?

The *scope* of a variable is the part of a script that defines the variable and in which it can be referenced. There are two main types of scope: *global* and *local*.

Global Scope: The global scope is all of the script that is not inside a function, if statement, or other conditional structure. Code from anywhere in the script can access global variables. There is only one global scope.

Local Scope: Code that is inside a function or in any local block (one that is inset by four spaces) defines a local scope. Only code that is in the same local scope can access a local variable. There can be many local scopes.

The following example script gives an “Undeclared identifier” error when we try to access a local variable from the global scope.

//@version=5 indicator("Scope demo") // Global scope int globalValue = close > open ? 1 : -1 if barstate.isconfirmed // Local scope int localValue = close > open ? 1 : -1 plot(localValue, "Local variable", chart.fg_color, 2)

To fix this error, we can declare the variable in the global scope, thus making it accessible from any scope in the script, and then conditionally modify it within a local block:

//@version=5 indicator("Scope demo") // Global scope int globalValue = close > open ? 1 : -1 int localValue = na if barstate.isconfirmed // Local

scope localValue := close > open ? 1 : -1 plot(localValue, "Local variable", chart.fg_color, 2)

Similarly, the following script gives an “Undeclared identifier” error when we try to access a variable defined in one local scope from another local scope. In this case, local scope 1 *contains* local scope 2, but the same problem would be present if they were on the same level. When a scope contains another one, the inner scope can access variables declared in the outer one, but not vice versa.

//@version=5 indicator("Scope demo") bool isUpCandleWithLargerUpWick = false if barstate.isconfirmed // Local scope 1 bool upWickIsLarger = (high - math.max(open, close)) > (math.min(open,close) - low) if close > open // Local scope 2 bool isUpCandle = true isUpCandleWithLargerUpWick := upWickIsLarger and isUpCandle ? true : false plot(isUpCandleWithLargerUpWick, "Global variable depending on two local variables", chart.fg_color, 2)

For more information about scopes, see the Code section of the User Manual.

How can I convert a script to a newer version of Pine Script®?

See the Migration Guides section of the User Manual for instructions about upgrading the version of Pine that a script uses.

Can I access the source code of “invite-only” or “closed-source” scripts?

No; only *open* scripts have their source code visible. The source code of *protected* and *invite-only* scripts is hidden and can only be seen by the script author.

Refer to the Visibility types section of the Publishing scripts page to learn more about the differences between open-source, protected, and invite-only scripts. To learn about the difference between *public* and *private* scripts, see the Privacy types section on that page.

Is Pine Script an object-oriented language?

Although Pine Script is not strictly an object-oriented programming language, it incorporates some object-oriented features, notably user-defined types (UDTs). Scripts can create objects as instances of a UDT. These objects have one or more fields, which can store values of various data types.

Here is a simple example of how to use the type keyword to create an object:

//@version=5 indicator("Object demo") // Define a new type named `pivot`. type pivot int x float y bool isHigh // Create a new `pivot` with specific values. pivot newPivot = pivot.new(bar_index, close, true) // Plot the `y` component of `newPivot`. plot(newPivot.y)

In this example, we create an object `newPivot`, which is an instance of the user-defined type `pivot`. The script then plots the y field of `newPivot`.

How can I access the source code of built-in indicators?

There are two ways to access the source code of built-in indicators that are written in Pine:

Create a new indicator

In the Pine Script Editor, click the dropdown menu (the arrow in the upper-left corner of the editor pane) and choose the "*Create new*" > "*Built in...*" option. Select the built-in indicator that you want to work with.

Edit the code

With the indicator displayed on the chart, click on the curly braces {} next to the indicator name to open it in the Pine Editor. To edit the code, click the option to create a working copy.

Some built-in indicators, such as the Volume Profile or chart pattern indicators, are not written in Pine and so the code for these indicators is not accessible. These indicators are not included in the “*Built-in script*” menu, and curly braces are not displayed next to their names on the chart.

How can I examine the value of a string in my script?

Scripts can print strings to Pine Logs on any or every bar, along with messages about the logic of the script at that point. See the Pine Logs section of the User Manual for information about logging.

Scripts can also display string in labels or label tooltips. The following example script displays a string in a label on the last bar of the chart using a custom function.

//@version=5 indicator("print()", "", true) print(string txt) => // Create a persistent label var label myLabel = label.new(bar_index, na, txt, xloc.bar_index, yloc.price, color(na), label.style_label_left, chart.fg_color, size.large, text.align_left) // Update the label's x and y position, and the text it displays. label.set_xy(myLabel, bar_index, open) label.set_text(myLabel, txt) if barstate.islast print("Timeframe = " + timeframe.period)

How can I visualize my script’s conditions?

If a script contains complex logical conditions, it can be difficult to debug the output. Visualizing each condition separately can help to debug any problems. See the Compound and nested conditions section of the User Manual for an example.

How can I make the console appear in the editor?

To display the console in the editor, either press the keyboard shortcut Ctrl + ` (grave accent), or right-click within the editor and choose the “Toggle Console” option.

How can I plot numeric values so that they don’t affect the indicator’s scale?

Plotting numerical values on the main chart pane can distort the price scale if the values differ too much from the price.

One way around this is not to plot the values on the chart, but use the Data Window to inspect them. Add `display = display.data_window` to the plot() call, and the values are visible in the Data Window for any single historical or realtime bar that the cursor hovers over.

Another option is to set the script to display in a separate pane by using `overlay = false` in the indicator() declaration. The user needs to delete and re-add the script to the chart if this parameter is changed. Plot the numeric values to track in the separate pane, and draw the rest of the script visuals on the main chart pane by using the `force_overlay` parameter.

Additionally, right-clicking on the scale on the chart brings out the dropdown menu. The “Scale Price Chart Only” option there makes it so the Auto mode of the chart scale only takes the chart itself into account, without adjusting for plots or other graphics of all indicators that overlay that chart.

Previous

Next

Strategies

Pine Script® v6 Documentation

User Manual / FAQ / Strategies

Strategies

Using Pine Script® strategy scripts, users can test *simulated* trades on historical and realtime data, to backtest and forward test trading systems. Strategies are similar to indicators, but with added capabilities such as placing, modifying, and canceling simulated orders and analyzing their results. Scripts that use the strategy() function as their declaration statement gain access to the `strategy.*` namespace, which contains functions and variables for simulating orders and retrieving strategy information.

When a user applies a strategy that uses order placement commands to the chart, the strategy uses the broker emulator to calculate simulated trades, and displays the results in the Strategy Tester tab.

Strategies support various types of orders including market, limit, stop, and stop-limit orders, allowing programmers to simulate different trading scenarios. Strategy order commands can send alerts when order fill events occur. An *order fill event* is triggered by the broker emulator when it executes a simulated order in realtime.

For a thorough exploration of strategy features, capabilities, and usage, refer to the Strategies section in the User Manual.

Strategy basics

How can I turn my indicator into a strategy?

To convert an indicator to a strategy, begin by replacing the indicator() declaration with the strategy() declaration. This designates the script as a strategy.

Add order placement commands for simulating orders. Use logical conditions from the initial indicator to trigger the commands in the converted strategy.

The following example includes two scripts: an initial indicator script and a strategy script converted from the indicator. We use a simple RSI oscillator as a momentum indicator to gauge the direction of a market's momentum, with values above 50 indicating an upward (bullish) trend and values below 50 signaling a downward (bearish) trend:

image

The initial indicator colors the plot line and the bars on the chart in a lime color when the RSI is greater than 50 and fuchsia when less than 50. We use plotshape() to plot triangles at the top and bottom of the oscillator on bars where the RSI crosses over or under the 50 level.

//@version=6 indicator("Example RSI indicator") float rsi = ta.rsi(close, 14) plot(rsi, "RSI", rsi >= 50 ? color.lime : color.fuchsia) hline(50, "Middle line", linestyle = hline.style_solid) plotshape(ta.crossover(rsi, 50), "Cross up", shape.arrowup, location.bottom, color.lime) plotshape(ta.crossunder(rsi, 50), "Cross Down", shape.arrowdown, location.top, color.fuchsia) barcolor(rsi >= 50 ? color.lime : color.fuchsia)

In the converted strategy version, we maintain the same RSI crossover and crossunder conditions used in the indicator script. These conditions, which previously only drew the plotshape() triangles, now also trigger entry orders for long and short positions using the strategy.entry() function. A long entry is called when the RSI crosses over 50, and a short entry is initiated when it crosses under 50. A long entry cancels a short trade, and vice-versa.

//@version=6 strategy("Example RSI strategy") float rsi = ta.rsi(close, 14) plot(rsi, "RSI", rsi >= 50 ? color.lime : color.fuchsia) hline(50, "Middle line", linestyle = hline.style_solid) plotshape(ta.crossover(rsi, 50), "Cross up", shape.triangleup, location.bottom, color.lime) plotshape(ta.crossunder(rsi, 50), "Cross Down", shape.triangledown, location.top, color.fuchsia) barcolor(rsi >= 50 ? color.lime : color.fuchsia) if ta.crossover(rsi, 50) strategy.entry("Long",

strategy.long, comment = "Long") if ta.crossunder(rsi, 50) strategy.entry("Short", strategy.short, comment = "Short")

How do I set a basic stop-loss order?

Stop losses are a risk management method that traders use to limit potential losses. The strategy.exit() function sets an order to exit a trade once it hits a specified price, thus preventing the loss from exceeding a predetermined amount.

To implement a basic stop loss in Pine Script, use the strategy.exit() function with either the `stop` or the `loss` parameter. The `stop` parameter specifies the *price* for the stop loss order, while the `loss` parameter sets the stop loss a certain number of *ticks* away from the entry order's price. Similarly, to set a take-profit level, use either the `limit` parameter, specifying the exact price for taking profit, or the `profit` parameter, defining the profit size in ticks from the entry price.

If a strategy.exit() call includes both the `stop` *and* `loss` parameters, or both the `limit` *and* `profit` parameters, the function uses the price level that is expected to trigger an exit first.

The following example script uses the tick-based `loss` parameter for long positions and the price-based `stop` parameter for short positions, and plots these stop levels on the chart. The script enters positions on the crossover or crossunder of two simple moving averages.

image

//@version=6 strategy("Stop using `loss` and `stop`", overlay = true) int lossTicksInput = input.int(60, "Stop loss in ticks (for longs)") float atrMultInput = input.float(1.0, "ATR multiplier (for shorts)", minval = 0) // Calculate the ATR value, adjusted by the multiplier, for setting dynamic stop loss levels on short positions. float atr = ta.atr(14) * atrMultInput // A persistent short stop loss level, updated based on short entry signals. var float shortStopLevel = na // Define conditions for entering long and short positions based on the crossover and crossunder of two SMAs. float ma1 = ta.sma(close, 14) float ma2 = ta.sma(close, 28) bool longCondition = ta.crossover(ma1, ma2) bool

shortCondition = ta.crossunder(ma1, ma2) // On detecting a long condition, place a long entry. if longCondition strategy.entry("Long", strategy.long) // For a short condition, place a short entry and set the stop loss level by adding the ATR value to the closing price. if shortCondition strategy.entry("Short", strategy.short) shortStopLevel := close + atr // Apply a fixed-size stop loss for long positions using the specified input tick size in the `loss` parameter. strategy.exit(id = "Long Exit", from_entry = "Long", loss = lossTicksInput) // For short positions, set the stop loss at the calculated price level using the `stop` parameter. strategy.exit(id = "Short Exit", from_entry = "Short", stop = shortStopLevel) // Calculate the long stop loss price by subtracting the loss size from the average entry price. // Set the price to `na` if the strategy is not in a long position. float longStopPlot = strategy.position_size > 0 ? strategy.position_avg_price - lossTicksInput * syminfo.mintick : na // The short stop price is already calculated. Set to `na` if the strategy is not in a short position. float shortStopPlot = strategy.position_size < 0 ? shortStopLevel : na // Plot the moving averages and stop loss levels. plot(ma1, "MA 1", color.new(color.lime, 50)) plot(ma2, "MA 2", color.new(color.fuchsia, 50)) plot(longStopPlot, "Long Stop", color.red, style = plot.style_steplinebr) plot(shortStopPlot, "Short Stop", color.red, style = plot.style_steplinebr) // Color the background when long or short conditions are met. bgcolor(longCondition ? color.new(color.aqua, 80) : shortCondition ? color.new(color.orange, 90) : na)

For more information, see the entry in the User Manual on strategy.exit().

How do I set an advanced stop-loss order?

Scripts can use different types of exits that are more advanced than simply closing the position at a predetermined level.

Bracket orders

A bracket order is a pair of orders that close the position if price moves far enough in either direction. Scripts can combine a stop-loss and take-profit order within a single strategy.exit() function call. See the FAQ entry about bracket orders for more details.

Trailing stop losses

A trailing stop loss is a stop loss that moves with price, but in the profitable direction only. To create a trailing stop, either adjust the stop price with each new bar, or use the built-in trailing stop parameters in the strategy.exit() function. Refer to the FAQ on implementing a trailing stop loss for information and examples.

Scaled exits

Scaled exits use multiple exit orders at varied price levels. When using tiered exit strategies, which progressively scale out of a position, ensure that the total quantity of all exit orders does not surpass the size of the initial entry position. Consult the FAQ on multiple exits for more information.

Moving a stop loss to breakeven

Adjusting a stop loss to the breakeven point once a specific condition is met can help in risk management. Details can be found in the FAQ on moving stop losses to breakeven.

Adjusting position size based on stop loss

Modify the position size relative to the stop loss to maintain a constant risk percentage of total equity. For more insights, see the FAQ on position sizing.

How can I save the entry price in a strategy?

Scripts can access the entry price for a *specific trade*, or the average entry price for a *position*.

Average entry price

The strategy.position_avg_price variable automatically updates to the average entry price of the current position. If the position consist of only one trade, the average price of the position is equal to the entry price of that single trade. If a strategy closes a market position that consists of multiple trades, trades are closed in the order they were opened, by default. Since the average price of the open position

changes according to which positions are still open, be aware of the order in which trades are closed, and if necessary, configure it using the `close_entries_rule` parameter of the strategy() declaration function.

Specific entry price

The strategy.opentrades.entry_price() function returns the entry price for a given trade ID. To find the entry price for the most recent open trade, and remembering that the trade indexes start at zero, use `float entryPrice = strategy.opentrades.entry_price(strategy.opentrades - 1)`.

How do I filter trades by a date or time range?

Using a date and time range filter in a strategy allows trades to be simulated only during a certain time period. Such filters can be useful to backtest specific historical periods, or to focus on particular times of the trading day.

Additionally, if the strategy sends signals for live trading, consider excluding all trades earlier than the trading start date and time, to ensure that the broker emulator starts in a neutral state.

The following example script restricts trading if a bar falls within a defined `startTime` and endTime, or outside of an optional intraday session window. The script colors the background red for bars that fall outside the time windows. On the screenshot, we've limited the trading range from June 1st 2024 to June 10th 2024, and additionally forbidden trading from 0000-0300 UTC:

image

//@version=6 strategy("Date/time filtering demo", "", true) // Timezone setting for date and time calculations. Adjust to the chart timezone. string TZ = "GMT+0" // Define the date window, an intraday time session to exclude, and the filtering to apply. bool useDateFilterInput = input.bool(true, "Allow trades only between the following dates (" + TZ + ")") int startTimeInput = input.time(timestamp("01 Jan 2000 00:00 " + TZ), " Start date", confirm = true) int endTimeInput =

input.time(timestamp("01 Jan 2099 00:00 " + TZ), " End date", confirm = true) bool useTimeFilterInput = input.bool(false, "Restrict trades during the following times (" + TZ + ")") string sessionStringInput = input.session("0000-0300", "") // @function Determines whether the current bar falls within a specified date and time range. // @param startTime (int) A timestamp marking the start of the time window. // @param endTime (int) A timestamp marking the end of the time window. // @param useDateFilter (bool) Whether to filter between `startTime` and `endTime`. Optional. // @param useTimeFilter (bool) Whether to restrict trades in the time session. Optional. // @param timeSession (string) Session time range in 'HHMM-HHMM' format, used if `useTimeFilter` is true. // @param timeZone (string) Timezone for the session time, used if `useTimeFilter` is true. // @returns (bool) `true` if the current bar is within the specified date and time range. timeWithinAllowedRange(int startTime, int endTime, bool useDateFilter = true, bool useTimeFilter = false, string timeSession = "0000-0000", string timeZone = "GMT-0") => bool isOutsideTime = na(time(timeframe.period, timeSession, timeZone)) bool timeIsAllowed = useTimeFilter and isOutsideTime or not useTimeFilter bool dateIsAllowed = time >= startTime and time <= endTime or not useDateFilter bool result = timeIsAllowed and dateIsAllowed // Determine if each bar falls within the date window or outside the ignored time session. bool isWithinTime = timeWithinAllowedRange(startTimeInput, endTimeInput, useDateFilterInput, useTimeFilterInput, sessionStringInput, TZ) // Calculate RSI for simple trading signals. float rsi = ta.rsi(close, 14) // Generate trading signals based on RSI conditions, provided they occur within the permissible date/time range. bool enterLong = ta.crossover(rsi, 50) and isWithinTime bool enterShort = ta.crossunder(rsi, 50) and isWithinTime // Simulate trades only if they meet the filtering criteria. if enterLong strategy.entry("Long", strategy.long) if enterShort strategy.entry("Short", strategy.short) // Color the background red for bars falling outside the specified date/time range. bgcolor(isWithinTime ? na : color.new(color.red, 80), title = "Exempt times")

Note that:

- We use the time() function to calculate whether bars are outside the user-defined session times. For additional details on integrating session data in Pine Script, refer to the Sessions section in the User Manual.
- We set the `confirm` argument to `true` for the inputs that define the time range. When the script is first added to the chart, it prompts the user to confirm the values by clicking on the chart.

Order execution and management

Why are my orders executed on the bar following my triggers?

Each historical bar in a chart is composed of a single set of open, high, low and close (OHLC) data. Pine scripts execute on this data once per historical bar, at the **close** of the bar.

So that results are consistent between historical and realtime bars, strategies also execute at the close of realtime bars. The next possible moment for an order to be filled is the beginning of the next bar.

Users can alter a strategy's calculation behavior by configuring strategies to process orders at the close of the signal bar instead, by selecting the "Fill orders/On bar close" setting in the "Settings/ Properties" tab. Programmers can do the same by setting the process_orders_on_close parameter to `true` in the strategy() declaration statement:

//@version=6 strategy("My Strategy", process_orders_on_close = true, ...)

An alternative method is to specify the `immediately` parameter as `true` in a strategy.close() or strategy.close_all function call. This setting causes the broker emulator to close a position on the same tick that the strategy creates the close order — meaning, when bar closes instead of the beginning of the next one. The process_orders_on_close

parameter affects all closing orders in the strategy, whereas the `immediately` parameter affects only the close order in which it is used.

However, processing orders on close might not give accurate results. For instance, if an alert occurs at the close of the session's last bar, the actual order can be executed only on the next trading day, since the bar is already closed. In contrast, the emulator would simulate the order being filled at the previous day's close. This discrepancy can lead to repainting, where the behavior of the strategy's simulation on historical bars differs from that seen in live trading.

How can I use multiple take-profit levels to close a position?

Setting up a strategy with multiple take profit levels enables traders to scale out of trades in segments to secure profits incrementally.

There are two main methods for scaling out at varying levels:

- Multiple strategy.exit calls. This method is most suitable when each take-profit level has a corresponding stop loss.
- An OCA reduce group. This method is ideal for a different number of take-profit levels and stop losses.

Multiple `strategy.exit()` functions

Each strategy.exit() call can set a bracket order for a specific take-profit and stop-loss level. However, if a strategy uses multiple strategy.exit() functions with the **same** stop level, each function call triggers a *separate* order (and therefore multiple order alerts). If order alerts are configured to trigger real trades, ensure that the trade system handles multiple alerts at the same stop level appropriately.

The following example script uses two separate strategy.exit() functions, each with its own stop-loss and take-profit levels. The quantity for the first bracket order is set to 50% of the total position size. This combination of orders creates a scaled exit with distinct stop levels.

image

//@version=6 strategy("Multiple exit demo", overlay = true) int exitPercentInput = input.int(1, "Exit %", minval = 1, maxval = 99) float exitPercent = exitPercentInput / 100 //@variable Is `true` on every 100th bar. bool buyCondition = bar_index % 100 == 0 var float stopLoss1 = na, var float takeProfit1 = na // Exit levels for `Exit1` var float stopLoss2 = na, var float takeProfit2 = na // Exit levels for `Exit2` // Place orders when `buyCondition` is true and we are not in a position. if buyCondition and strategy.position_size == 0.0 stopLoss1 := close * (1 - exitPercent), takeProfit1 := close * (1 + exitPercent) // Update the levels based on the current price stopLoss2 := close * (1 - (2 * exitPercent)), takeProfit2 := close * (1 + (2 * exitPercent)) strategy.entry("Buy", strategy.long, qty = 2) strategy.exit("Exit1", "Buy", stop = stopLoss1, limit = takeProfit1, qty_percent = 50) strategy.exit("Exit2", "Buy", stop = stopLoss2, limit = takeProfit2) // Set `stopLoss1` and `takeProfit1` to `na` when price touches either. if low <= stopLoss1 or high >= takeProfit1 stopLoss1 := na takeProfit1 := na // Set `stopLoss2` and `takeProfit2` to `na` when price touches either. if low <= stopLoss2 or high >= takeProfit2 stopLoss2 := na takeProfit2 := na plot(stopLoss1, "SL1", color.red, style = plot.style_circles) plot(stopLoss2, "SL2", color.red, style = plot.style_circles) plot(takeProfit1, "TP1", color.green, style = plot.style_circles) plot(takeProfit2, "TP2", color.green, style = plot.style_circles)

Note that:

- We use persistent global variables for the take-profit and stop-loss levels so that we can plot them. Otherwise, declaring the variables in the first if block would be simpler.

Using `strategy.oca.reduce`

Creating exit orders as a group, using the strategy.oca.reduce type, ensures that when one exit order from the group is filled, the quantity of the remaining orders is reduced accordingly. This method is ideal in scripts that have an unequal number of take-profit levels to stops.

The following example script uses two take-profit levels but only one stop level. All three sell orders have the same `oca_name`, which means they form a group. They have `oca_type = strategy.oca.reduce` set, so that filling one of the limit orders reduces the quantity of the remaining orders. The total quantity of the exit orders matches the entry order quantity, preventing the strategy from trading an excessive number of units and causing a reversal.

image

//@version=6 strategy("Multiple TP, one stop demo", overlay = true) int exitPercentInput = input.int(1, "Exit %", minval = 1, maxval = 99) float exitPercent = exitPercentInput / 100 var float stop = na var float limit1 = na var float limit2 = na bool buyCondition = bar_index % 100 == 0 // Is `true` on every 100th bar. // Place orders when `buyCondition` is true and we are not in a position. if buyCondition and strategy.position_size == 0 stop := close * (1 - exitPercent) limit1 := close * (1 + exitPercent) limit2 := close * (1 + (2 * exitPercent)) strategy.entry("Long", strategy.long, 6) // All three sell orders use the "Bracket" OCA group; filling one order reduces the quantity of the remaining orders. 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) // Set `limit1` to `na` when price exceeds it. if high >= limit1 limit1 := na // Set `stop`, `limit11`, and `limit2` to `na` when price surpasses either the last take-profit, or the stop. if low <= stop or high >= limit2 stop := na, limit1 := na, limit2 := na plot(stop, "Stop", color.red, style = plot.style_linebr) plot(limit1, "Limit 1", color.green, style = plot.style_linebr) plot(limit2, "Limit 2", color.green, style = plot.style_linebr)

How can I execute a trade partway through a bar?

On historical bars, Pine scripts can access only a single set of open, high, low and close (OHLC) data per bar. Consequently, strategies are

calculated once, at the close of each bar. This limitation means it’s not possible to evaluate logical conditions that occur mid-bar, such as a price cross, on historical data.

Using `calc_on_every_tick`

Strategies running on realtime bars can execute orders partway through a bar by enabling the calc_on_every_tick parameter. This setting allows the strategy to process each tick (incoming price update) and execute trades on the tick after a logical condition occurs.

Using predefined prices

Stop or limit orders at predefined prices *can* execute orders partway through a bar, even when the strategy does not enable the `calc_on_every_tick` parameter. This method is effective on both realtime *and* historical data. Even though orders are processed on the close of historical bars, the broker emulator simulates an order fill at the predefined price level, if the broker determines that price has hit that level during the bar. For information about the assumptions that the broker emulator makes about price movements, see the Broker emulator section of the User Manual.

The following example script uses stop and limit orders to exit a trade partway through a bar. The script calls the strategy.exit() function with the `stop` and `limit` parameters, determining the specific price levels at which the trade will exit.

image

//@version=6 strategy("Predefined price exit demo", overlay = true) int exitPercentInput = input.int(1, "Exit %", minval = 1, maxval = 99) float exitPercent = exitPercentInput / 100 //@variable Is `true` on every 100th bar. bool buyCondition = bar_index % 100 == 0 var float stopLoss = na var float takeProfit = na // Place orders when `buyCondition` is true and we are not in a position. if buyCondition and strategy.position_size == 0.0 stopLoss := close * (1 - exitPercent) takeProfit := close * (1 + exitPercent) strategy.entry("buy",

strategy.long) strategy.exit("exit", "buy", stop = stopLoss, limit = takeProfit) // Set `stopLoss` and `takeProfit` to `na` when price touches either, i.e., when the strategy simulates an exit. if low <= stopLoss or high >= takeProfit stopLoss := na takeProfit := na plot(stopLoss, "SL", color.red, style = plot.style_linebr) plot(takeProfit, "TP", color.green, style = plot.style_linebr)

How can I exit a trade in the same bar as it opens?

Sometimes, strategy testers want to be able to exit a trade in the same bar as the entry. By default, if an exit condition occurs during the same bar that a trade is opened, the broker emulator closes the trade at the open of the *next* bar. To learn why this happens, refer to this FAQ entry.

To override this default behavior, either specify exit prices, or exit with a market order at the bar close.

Specifying exit prices

If the entry command also sets stop-loss or take-profit orders to trigger an exit when certain price levels are reached, then the trade can exit during the same bar that it opens.

In the following example script, the trade exits within the same bar *if* the price hits either of the defined profit or loss levels. Setting small profit and loss values increases the likelihood of triggering an exit within the entry bar, although the trade could hit those levels for the first time in a subsequent bar instead.

//@version=6 strategy("Exit on entry bar with specific price", overlay = true) int exitTickSizeInput = input.int(10, "Exit if price moves this many ticks", minval = 1) //@variable Is `true` on every 100th bar. bool buyCondition = bar_index % 10 == 0 // Place orders when `buyCondition` is true and we are not in a position. if buyCondition and strategy.position_size == 0.0 strategy.entry("buy", strategy.long) strategy.exit("exit", "buy", profit = exitTickSizeInput, loss = exitTickSizeInput)

Using a market order at bar close

Another method to exit a trade in the same bar that it opens is to use a market order at the bar's close, by setting the `immediately` argument to `true` in the strategy.close() function.

In the following example script, if the buy order is opened, the strategy closes the position at the end of the entry bar. Scripts can call the strategy.close() function conditionally within a local block if necessary. For simplicity, in this example we apply the command to every entry.

//@version=6 strategy("Exit on entry bar with market order", overlay = true) //@variable Is `true` on every 100th bar. bool buyCondition = bar_index % 10 == 0 // Place orders when `buyCondition` is true and we are not in a position. if buyCondition and strategy.position_size == 0.0 strategy.entry("buy", strategy.long) strategy.close("buy", immediately = true)

Advanced order types and conditions

How can I set stop-loss and take-profit levels as a percentage from my entry point?

To set exit orders as a percentage from the entry price, the script needs the average entry price calculated by the broker emulator (which is affected by conditions including multiple entries and slippage). However, the built-in variable strategy.position_avg_price returns na until the close of the entry bar. This means that take-profit and stop-loss orders based on the entry price can only be placed during the *next* bar.

If programmers want strategies to be able to close trades on the same bar that they are opened, there are two workarounds, each of which have their own benefits and limitations: altering the emulator behavior and using a different, fixed value.

Using `calc_on_order_fills`

Setting the calc_on_order_fills argument of the strategy() declaration function to `true` recalculates the strategy immediately after simulating an order fill. This setting provides access to data such as the current average price of a position on an unconfirmed bar.

The following example script sets take-profit and stop-loss orders during the entry bar, based on the entry price strategy.position_avg_price. The script uses the `calc_on_order_fills` setting to enable this behavior.

//@version=6 strategy("Exit demo using `calc_on_order_fills`", overlay = true, calc_on_order_fills = true) float stopSizeInput = input.float(1.0, "SL %", minval = 0.0) / 100.0 float profitSizeInput = input.float(1.0, "TP %", minval = 0.0) / 100.0 //@variable Is `true` on every 100th bar. bool buyCondition = bar_index % 100 == 0 //@variable Stop-loss price for exit commands. var float stopLoss = na //@variable Take-profit price for exit commands. var float takeProfit = na // Place orders when `buyCondition` is true and we are not in a position. if buyCondition and strategy.position_size == 0.0 strategy.entry("buy", strategy.long) // If we are in a position, set the exit orders. if strategy.position_size != 0.0 stopLoss := strategy.position_avg_price * (1.0 - stopSizeInput) takeProfit := strategy.position_avg_price * (1.0 + profitSizeInput) strategy.exit("exit", "buy", stop = stopLoss, limit = takeProfit) // Set `stopLoss` and `takeProfit` to `na` when price touches either, i.e., when the strategy simulates an exit. if low <= stopLoss or high >= takeProfit stopLoss := na takeProfit := na plot(stopLoss, "SL", color.red, style = plot.style_linebr) plot(takeProfit, "TP", color.green, style = plot.style_linebr)

Note that:

- If we change `calc_on_order_fills` to `false` in this script, the exit orders are placed on the bar *after* the entry bar, and can fill at very different levels depending on the movement of price.

Using predefined prices

The following example script calculates the stop and limit orders based on the *closing price* of the signal bar. The disadvantage of this approach is that the close price might not match the average opening price exactly. The advantage is that this method doesn't introduce potential *lookahead bias* like using `calc_on_order_fills`.

//@version=6 strategy("Exit demo using predefined prices", overlay = true) float stopSizeInput = input.float(1.0, "SL %", minval = 0.0) / 100.0 float profitSizeInput = input.float(1.0, "TP %", minval = 0.0) / 100.0 // @variable Is `true` on every 100th bar. bool buyCondition = bar_index % 100 == 0 //@variable Stop-loss price for exit commands. var float stopLoss = na //@variable Take-profit price for exit commands. var float takeProfit = na // Place orders when `buyCondition` is true and we are not in a position. if buyCondition and strategy.position_size == 0.0 stopLoss := close * (1.0 - stopSizeInput) takeProfit := close * (1.0 + profitSizeInput) strategy.entry("buy", strategy.long) strategy.exit("exit", "buy", stop = stopLoss, limit = takeProfit) // Set `stopLoss` and `takeProfit` to `na` when price touches either, i.e., when the strategy simulates an exit. if low <= stopLoss or high >= takeProfit stopLoss := na takeProfit := na plot(stopLoss, "SL", color.red, style = plot.style_linebr) plot(takeProfit, "TP", color.green, style = plot.style_linebr)

How do I move my stop-loss order to breakeven?

Moving a stop-loss order to breakeven can be a useful technique to manage risk.

The following example script sets a persistent `stopLoss` variable when the strategy enters a position. The script then updates the stop price to the entry price when the market price gets halfway to the take-profit level. The script calls the strategy.exit() function on every bar to ensure that the broker emulator receives any updates made to the `stopLoss` value. Lastly, it plots the average price according to the strategy.position_avg_price variable for reference.

image

```
//@version=6 strategy("Move stop to breakeven", overlay = true) float stopSizeInput = input.float(5.0, "SL %", minval = 0.0) / 100.0 float profitSizeInput = input.float(5.0, "TP %", minval = 0.0) / 100.0 float breakEvenInput = input.float(50, "BE %", minval = 0.0, maxval = 100) / 100.0 //@variable Is `true` on every 100th bar. bool buyCondition = bar_index % 100 == 0 //@variable Stop-loss price for exit commands. var float stopLoss = na //@variable Take-profit price for exit commands. var float takeProfit = na //@variable Price that, if breached, sets the stop to breakeven. var float breakEvenThreshold = na // Place orders when `buyCondition` is true and we are not in a position. if buyCondition and strategy.position_size == 0.0 stopLoss := close * (1.0 - stopSizeInput) takeProfit := close * (1.0 + profitSizeInput) // Set the breakeven threshold. breakEvenThreshold := close * (1.0 + profitSizeInput * breakEvenInput) strategy.entry("buy", strategy.long) // If the breakeven threshold is exceeded while in a position, set the stop to the entry price. if high >= breakEvenThreshold and strategy.position_size != 0 stopLoss := strategy.position_avg_price // @variable Is `true` on the bar on which a trade exits. bool isExitBar = strategy.closedtrades.exit_bar_index(strategy.closedtrades - 1) == bar_index //@variable Condition to determine when plots are displayed. bool showPlots = strategy.position_size != 0 or buyCondition or isExitBar // Plot the entry price, stop loss, take-profit, and the breakeven threshold. plot(strategy.position_avg_price, "BE", chart.fg_color, style = plot.style_linebr) plot(showPlots ? stopLoss : na, "SL", color.red, style = plot.style_linebr) plot(showPlots ? takeProfit : na, "TP", color.green, style = plot.style_linebr) plot(showPlots ? breakEvenThreshold : na, "TG", color.blue, style = plot.style_circles) // Place a bracket order using the `stopLoss` and `takeProfit` values. // We call it on every bar so that the stop level is updated when the breakeven threshold is exceeded. strategy.exit("exit", "buy", stop = stopLoss, limit = takeProfit)
```

How do I place a trailing stop loss?

A trailing stop loss limits a trader’s losses while allowing a position to remain open as long as the price moves favorably.

Strategies can create trailing stops either by using the built-in functionality of the strategy.exit() function or by creating custom trailing stop-loss logic.

Trailing stops set in the strategy.exit() use live price updates in realtime but assumed price movements for historical bars. These assumptions can cause repainting. This type of trailing stop is therefore potentially more responsive but less accurate.

Custom trailing stop values are typically updated at the close of each bar, and so do not capture realtime intrabar price movements with the same responsiveness. This delay helps to avoid repainting strategy results.

Using built-in trailing stop functionality

To set a trailing stop in the strategy.exit() function, specify both *when* the trail should activate and *how far* behind price it should trail.

Activation level

When price crosses this level, the trailing stop activates. The activation level can be set as a number of ticks past the entry price via the `trail_points` parameter, or as a price value via the `trail_price` parameter.

Trail offset

After it activates, the stop loss trails behind the bar's high or low price by this distance, defined in ticks using the `trail_offset` parameter.

In the following long-only example script, the strategy.exit() function uses the `trail_points` and `trail_offset` parameters to set a trailing stop. The stop-loss trails the high, minus the offset points, after it activates. The script creates and plots a separate `trailingStop` variable to visualize the trailing stop price that the function calculates internally, although this is not necessary for the trailing stop to function. We also set a separate stop-loss order to close trades that go too low before they trigger the trailing stop.

image

//@version=6 strategy("Trailing stop order demo", overlay = true) string TT_SO = "The trailing stop offset in ticks. Also used as the initial stop loss distance from the entry price." //@variable The activation level is this number of ticks above the entry price. int activationOffsetInput = input.int(1000, "Activation Level Offset (in ticks)") //@variable The trailing stop trails this many ticks below the high price. int stopOffsetInput = input.int(2000, "Stop Offset (in ticks)", tooltip = TT_SO) //@variable The price at which the trailing stop activates. float trailPriceActivationLevel = activationOffsetInput * syminfo.mintick + strategy.position_avg_price //@variable The price at which the trailing stop itself is located. var float trailingStop = na // Calculate a fast and slow Simple Moving Average. float ma1 = ta.sma(close, 14) float ma2 = ta.sma(close, 28) //@variable Is `true` when `ma1` crosses over `ma2` and we are not in a position. bool longCondition = ta.crossover(ma1, ma2) and strategy.position_size == 0 //@variable Is `true` on the bar that a trade exits. bool isExitBar = strategy.closedtrades.exit_bar_index(strategy.closedtrades - 1) == bar_index float exitPrice = strategy.closedtrades.exit_price(strategy.closedtrades - 1) // Generate a long market order when `longCondition` is `true`. // Set a static abd trailing stop loss. if longCondition strategy.entry("Long", strategy.long) strategy.exit("Stop", from_entry = "Long", trail_points = activationOffsetInput, trail_offset = stopOffsetInput, loss = stopOffsetInput) // If the high exceeds the activation level, set the `trailingStop` to whichever is higher: // the current high minus the price equivalent of `stopOffsetInput` or the previous `trailingStop` value. if high > trailPriceActivationLevel or isExitBar and exitPrice > trailingStop trailingStop := math.max(high - stopOffsetInput * syminfo.mintick, nz(trailingStop)) //@variable The price of the active stop price, using the trailing stop when activated, or a static stop loss otherwise. float stopLevel = na(trailingStop) ? strategy.position_avg_price - stopOffsetInput * syminfo.mintick : trailingStop // Visualize the movement of the trailing stop and the activation level. plot(stopLevel, "Stop Level", chart.fg_color, 2, plot.style_linebr) plot(trailPriceActivationLevel, "Activation level", color.aqua, 1, plot.style_linebr) // Display the two simple moving averages on the

chart. plot(ma1, "MA 1", color.new(color.lime, 60)) plot(ma2, "MA 2", color.new(color.fuchsia, 60)) // Mark the point where the trailing stop is activated with a shape and text. plotshape(high > trailPriceActivationLevel and na(trailingStop)[1], "Trail Activated", shape.triangledown, size = size.small, color = color.aqua, text = "Trailing stop\nactivated", textcolor = color.aqua) // Set the trailing stop to `na` when not in a position. if strategy.position_size == 0 trailingStop := na

Coding a custom trailing stop

A custom trailing stop can use different activation conditions, and can trail in a different way, to the trailing stop built into the strategy.exit() function. To work correctly, a custom trailing stop must calculate the stop price on each bar that the stop is active, and call the strategy.exit() function on each bar to set the `stop` price.

The following example script triggers long and short trades based on crosses of two moving averages. A custom function calculates the trailing stop using the highest or lowest price from the last five bars, adjusted by an Average True Range (ATR) buffer. This method of distancing the stop by a measure of average price movement attempts to reduce premature stop triggers in volatile conditions.

//@version=6 strategy("ATR trailing stop demo", overlay = true) // Set the lookback period in bars to identify the highest or lowest point for trailing stop calculations. int SWING_LOOKBACK = 5 // @function Calculates a dynamic trailing stop by adjusting the highest // (bearish) or lowest (bullish) swing points over a set `length` // of bars using the ATR, for a stop distance proportional to average bar size. // @param calcStop (series bool) A condition that activates the trailing stop, e.g., being in a trade. // @param length (simple int) The number of bars to look back to determine the highest or lowest point for // the trailing stop calculation. // @param isLong (simple bool) Indicator of the trailing stop's orientation: true for long trades // (stop below price) and false for short trades (stop above price). // @param atrMultiplier (simple float) The multiplier applied to the ATR, adjusting the stop's distance from the // identified extreme price point. Optional. Default is 1.0, or 100% of

the ATR value. // @returns (float) The trailing stop price, or `na` if `calcStop` is false. atrTrailingStop(series bool calcStop, simple int length, simple bool isLong, simple float atrMultiplier = 1.0) => var float trailPrice = na int m = isLong ? 1 : -1 float atr = ta.atr(14) * atrMultiplier float swingPoint = switch isLong => ta.lowest(length) - atr => ta.highest(length) + atr trailPrice := switch calcStop and not calcStop[1] => swingPoint calcStop[1] and not calcStop => na => math.max(trailPrice * m, swingPoint * m) * m // Calculate a fast and slow simple moving average. float ma1 = ta.sma(close, 14) float ma2 = ta.sma(close, 28) // Conditions for long/short entries on MA crossover/crossunder, if we are not in a position. bool longCondition = ta.crossover(ma1, ma2) and strategy.position_size == 0 bool shortCondition = ta.crossunder(ma1, ma2) and strategy.position_size == 0 // Determine when to calculate trailing stops for long/short positions, based on entries and position. bool isExitBar = strategy.closedtrades.exit_bar_index(strategy.closedtrades - 1) == bar_index bool isLong = longCondition or strategy.position_size > 0 or isExitBar bool isBear = shortCondition or strategy.position_size < 0 or isExitBar // Use `atrTrailingStop()` to calculate trailing stops for both long and short positions. float longStop = atrTrailingStop(isLong, SWING_LOOKBACK, true) float shortStop = atrTrailingStop(isBear, SWING_LOOKBACK, false) // Place long entry order when `longCondition` occurs. if longCondition strategy.entry("long", strategy.long) // Place short entry order when `shortCondition` occurs. if shortCondition strategy.entry("short", strategy.short) // Create exit orders for long/short trades with ATR trailing stop, called on each bar to update to the latest price. strategy.exit("long exit", "long", stop = longStop) strategy.exit("short exit", "short", stop = shortStop) // Display the two simple moving averages and stop levels on the chart. plot(ma1, "MA 1", color.new(color.lime, 60)) plot(ma2, "MA 2", color.new(color.fuchsia, 60)) plot(isExitBar ? longStop[1] : longStop, "Long Stop", color.red, 2, plot.style_linebr) plot(isExitBar ? shortStop[1] : shortStop, "Short Stop", color.red, 2, plot.style_linebr)

Note that:

- Because strategies run once per bar, the trailing stop price in this example script updates at the close of each bar. During realtime

bars the *previous* bar's stop value is used. This approach, while slightly delayed compared to using the built-in trailing stop described in the FAQ entry about how to place a trailing stop loss using built-in trailing stop functionality, ensures that the trailing stop price is not subject to assumptions about intrabar price movements, and thus avoids repainting.

How can I set a time-based condition to close out a position?

To close positions after a certain amount of time has passed, track the entry time for each trade and close the position using strategy.close() after the timeout.

Because strategies calculate at the close of each bar on historical data, time-based conditions can only be evaluated at the close, so **trade times are assessed in multiples of the chart bar's duration**. Further, if the timeout value is not divisible by the duration of a chart bar, each trade will last at least one additional chart bar. For instance, setting a timeout of 100 seconds on a 1-minute chart effectively means a minimum of two bars before a position can be closed.

In realtime, the same logic applies unless the strategy uses the calc_on_every_tick parameter, in which case the trade closes as soon as the first tick exceeds the timeout value. Remember that altering emulator behavior typically introduces repainting.

The following example script calculates the duration of each open trade by comparing the current time against the trade entry time. If a trade's duration exceeds the specified timeout, the script closes the trade and marks the event with a comment on the chart including the trade's duration in seconds.

image

//@version=6 strategy("Close position by timeout", overlay = true) // @function Automatically closes all positions that have been open for longer than a specified period. // @param timeoutInSeconds (int) The maximum allowed duration for an open trade, measured in seconds. //

@returns (void) The function has no explicit return. closePositionsAfter(int timeoutInSeconds) => if strategy.opentrades > 0 for i = 0 to strategy.opentrades - 1 int timeNow = barstate.isrealtime ? timenow : time_close int tradeDurationInSeconds = (timeNow - strategy.opentrades.entry_time(i)) / 1000 if tradeDurationInSeconds >= timeoutInSeconds string entryName = strategy.opentrades.entry_id(i) string tradeComment = str.format("Close \"{0}\" by timeout {1}s", entryName, tradeDurationInSeconds) strategy.close(entryName, comment = tradeComment, immediately = true) // Create long and short conditions based on the crossover/under of 2 moving averages. bool longCondition = ta.crossover(ta.sma(close, 14), ta.sma(close, 28)) bool shortCondition = ta.crossunder(ta.sma(close, 14), ta.sma(close, 28)) // Place long entry order upon `longCondition`. if longCondition strategy.entry("long", strategy.long) // Place short entry order upon `shortCondition`. if shortCondition strategy.entry("short", strategy.short) // Close positions after a configurable number of seconds. closePositionsAfter(input(1200, "Timeout (seconds)"))

Note that:

- The script uses either the time of the bar's close using the time_close variable, or the current time from the timenow variable (if the strategy uses the calc_on_every_tick parameter).
- The script uses the built-in functions strategy.opentrades.entry_time() and strategy.opentrades.entry_id() to measure trade duration and identify individual trades.
- The strategy.close() function uses the `immediately` argument to simulate trades at the end of the bar that exceeds the timer, rather than waiting for the opening of the next bar. Consequently, when a 120-second timeout is applied and the script runs on a 1-minute chart, it gives the appearance that trades last exactly two bars.

How can I configure a bracket order with a specific risk-to-reward (R:R) ratio?

To create a bracket order, define a stop-loss and a take-profit order using a single strategy.exit() call. To apply a specific risk-to-reward ratio, calculate the distance between the entry point and the stop-loss level. This stop distance represents the “risk”. Then place the take-profit order a certain multiple of the stop distance away. The distance to the take-profit order represents the “reward”, and the ratio between them is the risk:reward (R:R) ratio.

The following example script simulates long and short trades using inputs to define the stop distance in ticks and the R:R ratio. The `loss` parameter of the strategy.exit() function is simply the stop distance. The `profit` parameter is the stop distance multiplied by the R:R ratio. The script fills the areas between the entry and stop-loss points, and between the entry and take-profit points, to illustrate the risk and reward.

image

//@version=6 strategy("R:R demo", overlay = true) // Declare the stop size in ticks and the risk-to-reward ratio as inputs. int lossSizeInput = input.int(300, "Loss size (in ticks)", minval = 0) float riskRewardInput = input.float(2.0, "Risk/Reward multiple", minval = 0) // Create long and short entry conditions on MA crossover/crossunder, as long as we are not in a position. float ma1 = ta.sma(close, 14), float ma2 = ta.sma(close, 28) bool buyCondition = ta.crossover(ma1, ma2) and strategy.position_size == 0 bool sellCondition = ta.crossunder(ma1, ma2) and strategy.position_size == 0 // Place orders when `buyCondition` or `sellCondition` is true. if buyCondition strategy.entry("buy", strategy.long) if sellCondition strategy.entry("sell", strategy.short) // Define exit point for the entries based on a predefined loss size. // Calculate the profit target by multiplying the loss size with the user-defined risk-to-reward ratio. strategy.exit("exit", loss = lossSizeInput, profit = lossSizeInput * riskRewardInput) // Calculate the price equivalent of the profit and loss level. float tradeBias = math.sign(strategy.position_size) float stopLossPrice =

strategy.position_avg_price - (tradeBias * lossSizeInput * syminfo.mintick) float takeProfitPrice = strategy.position_avg_price + (tradeBias * lossSizeInput * syminfo.mintick * riskRewardInput) // Plot the entry price, the stop price, and the price of the take-profit order. plotEntry = plot(strategy.position_avg_price, "Entry price", color.new(color.gray, 70), style = plot.style_linebr) plotStop = plot(stopLossPrice, "Stop-loss price", color.red, style = plot.style_linebr) plotTP = plot(takeProfitPrice, "Take-profit price", color.green, style = plot.style_linebr) // Highlight the R:R ratio by shading the area between the entry and the stop and the entry and the take-profit. fill(plotStop, plotEntry, color.new(color.red, 80)) fill(plotTP, plotEntry, color.new(color.green, 80))

How can I risk a fixed percentage of my equity per trade?

Adjusting the position size to risk a fixed percentage of equity normalizes risk exposure, regardless of equity fluctuations, and helps avoid disproportionate risks across a strategy's trading history.

Calculate the position size so that as the stop distance increases, the position size decreases, and vice-versa, to maintain a constant risk percentage:

1. Calculate monetary *risk per contract* by multiplying the stop distance in ticks by the monetary value of each tick (syminfo.mintick) and by the number of units each contract represents (syminfo.pointvalue).

2. Determine *risk amount* by multiplying the current equity (strategy.equity) by the percentage of equity that you want to risk.

3. Calculate position size by dividing the *risk amount* by the *risk per contract*.

The following example script uses moving average crosses to generate long and short orders. The stop distance, risk:reward ratio, and percentage of equity to risk are all configurable via inputs. The script plots the current equity, the current value of a new position, and the

percentage change in equity to the Data Window. Note that the actual exposure level can be less than intended if the available capital does not divide neatly by the unit value, particularly with small equity amounts, high unit prices, or assets such as stocks where trading partial shares is not possible.

Additionally, we display lines on the chart for the current total equity (in green) and the value of a position needed for the specified risk exposure at the current price (in blue). If the position value exceeds the total equity, the strategy requires leverage to achieve the required exposure, and the script colors the background red and displays the minimum leverage ratio needed in the data window.

image

//@version=6 strategy("Fixed risk", overlay = false, initial_capital = 100000) // Specify the desired stop distance (in ticks), the trade R:R ratio, and the percentage of equity to risk. int lossSizeInput = input.int(300, "Loss size (in ticks)", minval = 0) float riskRewardInput = input.float(2.0, "Risk/Reward multiple", minval = 0) float pctRiskInput = input.float(1.0, "% of equity to risk") / 100 // Create conditions for long/short entries on MA crossover/crossunder, if we are not in a position. float ma1 = ta.sma(close, 14), float ma2 = ta.sma(close, 28) bool buyCondition = ta.crossover(ma1, ma2) and strategy.position_size == 0 bool sellCondition = ta.crossunder(ma1, ma2) and strategy.position_size == 0 // Store the equity value at each trade entry, in order to calculate the percent change in equity. var float equityAtEntry = 0.0 // Calculate the risk per contract of the instrument. float riskPerContract = lossSizeInput * syminfo.mintick * syminfo.pointvalue // Calculate the amount of equity to risk. float equityToRisk = strategy.equity * pctRiskInput // Determine the position size necessary to risk the specified percentage of the equity. float positionSize = equityToRisk / riskPerContract // Place orders when `buyCondition` or `sellCondition` is true. if buyCondition strategy.entry("buy", strategy.long, positionSize) equityAtEntry := strategy.equity // Set the `equityAtEntry` variable to the current equity on each entry. if sellCondition strategy.entry("sell", strategy.short, positionSize) equityAtEntry := strategy.equity // Stop-loss level is from the user input. Profit target is the multiple of the loss size with the risk-

to-reward ratio. strategy.exit("exit", loss = lossSizeInput, profit = lossSizeInput * riskRewardInput) // Calculate the percent equity change between the current equity and the equity at entry. // On the exit bar of each trade, this value can be used to verify the percentage of equity risked. float equityChgPercent = 100 * (strategy.equity - equityAtEntry) / equityAtEntry color equityChgColor = equityChgPercent < 0 ? color.red : color.green, // Display current equity and current value of a new position on the chart, and % change in equity to the Data Window. plot(strategy.equity, "Current Total Equity", color.green, 2, display = display.all - display.pane) plot(positionSize * close, "Value of New Position at Current Price", color.aqua, 2, display = display.all - display.pane) plot(equityChgPercent, "% Change in Equity per Trade", equityChgColor, display = display.data_window, format = format.percent) // Color the background red if the calculated risk value exceeds the available equity (leverage required). bgcolor(strategy.equity < positionSize * close ? color.new(color.red, 80) : na) // Plot the minimum leverage multiple required to open the position, applicable only if leverage is necessary. plot(strategy.equity < positionSize * close ? positionSize * close / strategy.equity : na, "Leverage multiple required", display = display.data_window)

Note that:

- The stop distance in our example script is set to a constant value for demonstration purposes. In practice, the stop distance normally varies for each trade.

Strategy optimization and testing

Why did my trade results change dramatically overnight?

Strategy results can vary over time depending on where the historical data starts. The starting point of the data set aligns with the start of the nearest day, week, month or year, depending on the chart timeframe. Additionally, different TradingView plans provide access to varying amounts of historical bars. Refer to the User Manual entry on starting points for a discussion of these factors.

For strategies, this means the historical results seen today might change as the dataset's starting point moves. This can lead to a natural repainting of strategy results over time. To reduce the effect of these changes on backtesting, follow these tips:

Export strategy results

Regularly exporting strategy results to file maintains a record of performance over time despite changes in historical data. Use the "Export Data" option in the top of the Strategy Tester to export data.

Use Deep Backtesting

Users with Premium and higher plans have access to the Deep Backtesting feature, which provides results from the entire available dataset of a symbol. Deep Backtesting results are displayed in the Strategy Tester but are not visible on the chart.

Use Bar Replay

Use the Bar Replay feature on the first chart bar to extend the dataset backward, allowing a strategy to run on an additional full dataset prior to the current range. This process can be repeated a few times to analyze multiple datasets.

Why is backtesting on Heikin Ashi and other non-standard charts not recommended?

Non-standard charts like Heikin Ashi, Renko, Line Break, Kagi, Point & Figure, and Range Charts offer unique perspectives on price action. However, these chart types are not suited for strategy backtesting or automated trading systems execution, because the prices and time intervals do not match market prices and times.

Renko, Line Break, Kagi, Point & Figure, and Range Charts simplify price action, losing some price detail. Heikin Ashi charts calculate synthetic prices for each bar's open, high, low, and close (OHLC) values based on averages.

Further, all non-standard chart types with the exception of Heikin Ashi charts form new price units based on price movement only and omit the element of time.

Both the distortion of price data and the omission of time in non-standard charts lead to unrealistic and potentially misleading backtesting results.

Programmers can specify the `fill_orders_on_standard_ohlc` parameter of the strategy declaration, which causes the strategy to calculate on standard chart data even if the current view is of Heikin Ashi candles. The user can do the same thing by by enabling the “Fill orders on standard OHLC” option in the strategy’s properties. This option has no effect on other non-standard chart types, because they use non-standard time as well as price.

For a more detailed analysis of how non-standard chart types affect strategy results, refer to this script from the PineCoders account.

How can I backtest deeper into history?

Different TradingView plans give access to different amounts of historical information. To conduct more comprehensive backtesting in Pine Script, exploring further into an asset’s historical data, use Bar Replay or Deep Backtesting.

Bar Replay

Starting the Bar Replay from the first chart bar in history effectively rolls back the dataset to an earlier point in time. Each iteration of the bar replay extends the dataset further back, offering analysis of multiple historical datasets. However, there is a limit to the number of times this process can be repeated. This method has the added benefit of visualizing the strategy’s performance directly on the chart, which can be insightful for understanding trade entries, exits, and behavior during specific historical market conditions.

Deep Backtesting

For TradingView users with Premium and higher plans, the Deep Backtesting feature calculates the strategy on *all* historical data available for the selected symbol. The results are displayed in the Strategy Tester but are not visible on the chart. The results from Deep Backtesting might be different from results from the Strategy Tester in regular mode, as explained in this Help Center article.

How can I backtest multiple symbols?

Each Pine Script strategy runs on one symbol at a time. To evaluate a strategy across various markets or instruments:

- Apply the strategy to the chart and then switch the chart to the desired symbol.
- Use TradingView's watchlist feature to organize and quickly access different symbols.
- Export the results from the Strategy Tester and use external tools such as spreadsheet software to compare the performance of a strategy on different symbols.

What does Bar Magnifier do?

The Bar Magnifier feature, available for TradingView Premium account holders, significantly enhances the accuracy of order fills in strategy backtests. This tool uses data from lower timeframes to obtain more detailed price movement within a bar, which can result in more precise order fills. When selected, Bar Magnifier mode replaces the assumptions that the broker emulator must make about price movement using only a single set of OHLC values for each historical bar.

The Bar Magnifier chooses the lower timeframe based on the chart timeframe:

Chart Timeframe	Intrabar Timeframe
1S	1S
30S	5S
1	10S

Chart Timeframe	Intrabar Timeframe
5	30S
10	1
15	2
30	5
60	10
240	30
1D	60
3D	240
1W	1D

To fully appreciate the effectiveness of Bar Magnifier, refer to the script demonstrations in the section about Bar Magnifier in the User Manual.

Advanced features and integration

Can my strategy script place orders with TradingView brokers?

Pine Script strategies and indicators cannot directly place orders on exchanges. Traders can use external tools or platforms that can interpret alert signals from Pine scripts using webhooks and execute trades accordingly.

How can I add a time delay between orders?

Adding a time delay between orders can help to prevent too many trades in a short time. Strategies can also prevent trading for a time after a series of losses. Here’s how to set up a time delay between orders:

- Define the delay duration, whether in time units (minutes, hours, days) or a number of bars. For time-based delays, convert the chosen time unit into milliseconds, because Pine time variables use milliseconds.

- Check the time or bar_index of the last trade using strategy.closedtrades.exit_time() or strategy.closedtrades.exit_bar_index().
- If the difference between the current bar `time` or `bar_index` and that of the last trade's exit exceeds the delay duration, set a boolean flag to allow new orders. Make sure to include the flag in the strategy entry conditions.

The following example script provides two methods for delaying orders: a time-based delay or a specified number of bars. The strategy creates a long entry order when either the `time` of a bar or its `bar_index` exceeds the set delay from the last active trade bar. No other conditions are used for entry in this demonstration, but users can add their own logic to these conditions.

To keep the chart clean, the script calls the strategy.close() function to close active trades after they have been open for 10 bars. The script uses background shading, labels and arrows to illustrate the trade entries and exits.

image

//@version=6 strategy("Time-delayed orders", overlay=true, max_labels_count = 500, max_lines_count = 500) import PineCoders/Time/4 as PCtime // Constants string TU1 = "seconds", string TU2 = "minutes", string TU3 = "hours", string TU4 = "days" string TU5 = "weeks", string TU6 = "months", string DT1 = "bars", string DT2 = "time" // Tooltips for inputs string D_TT = "Delay orders for a specific number of bars or a specific duration of time since the last trade." string N_TT = "Specify the number of bars or time units for the delay." string U_TT = "Unit of time; relevant only if the delay type is 'time'." // User inputs for delay type, number of units/bars, and time units. string delayTypeInput = input.string(DT2, "Delay type", tooltip = D_TT, options = [DT1, DT2]) int nInput = input.int(15, "Number of bars or time units", tooltip = N_TT) string unitsInput = input.string(TU2, "Time units", tooltip = U_TT, options = [TU1, TU2, TU3, TU4, TU5]) // Convert the time unit string input to a value in milliseconds for use in the time delay calculation. int mult = switch unitsInput TU1 => 1000 TU2 => 60000 TU3 => 3600000 TU4 => 86400000 TU5 => 604800000 =>

2628003000 bool useTimeDelay = delayTypeInput == DT2 // Use time delay or not. int timeOfExit = strategy.closedtrades.exit_time(strategy.closedtrades - 1) // Time of last trade exit. int barOfExit = strategy.closedtrades.exit_bar_index(strategy.closedtrades - 1) // Bar index of last trade exit. int timeSinceExit = time - timeOfExit // Calculate the time since the last trade. int barsSinceExit = bar_index - barOfExit // Calculate the number of bars since the last trade. bool timeAllowed = (timeSinceExit >= nInput * mult or na(timeOfExit)) and useTimeDelay bool barAllowed = (bar_index - barOfExit >= nInput or na(barOfExit)) and not useTimeDelay // Allow entry of a trade if the delay has passed and we're not in a position. bool entryCondition = (timeAllowed or barAllowed) and strategy.position_size == 0 bool tradeExited = barOfExit == bar_index // Did the trade exit on the current bar? if entryCondition // Enter the trade if conditions allow. strategy.entry("Long", strategy.long) // Set label text: format time or show bar count since last trade. string labelTxt = useTimeDelay ? PCtime.formattedNoOfPeriods(timeSinceExit, unitsInput) : str.format("{0} bars", barsSinceExit) label.new(bar_index, low, labelTxt, color = color.new(color.lime, 80), textcolor = color.lime, style = label.style_label_up) line.new(timeOfExit, low, time, low, xloc.bar_time, color = color.new(color.lime, 50), style = line.style_arrow_left, width = 2) if bar_index % 10 == 0 // Close any open position on every tenth bar. strategy.close("Long") bgcolor(entryCondition ? color.new(color.lime, 85) : tradeExited ? color.new(color.fuchsia, 85) : na)

Consider the following limitations when adding time-based delays.

Historical bars

Strategies calculate at the close of each bar, so they can only evaluate time-based conditions at that moment. This constraint entails that on historical bars, **delay times are assessed in increments equal to the chart bar’s duration**.

Session times

Strategies cannot evaluate delays when the market is closed, because there are no price updates to trigger script execution. This means that if a delay extends beyond the end of a trading session, the delay condition cannot be identified until the script runs again on the next session, resulting in a longer-than-anticipated time between orders.

Delay duration on different timeframes

If the delay value is not divisible by the duration of a chart bar, each delay lasts at least one additional chart bar. For instance, setting a delay of 100 seconds on a 1-minute chart effectively means a minimum of two bars before the delay is exceeded.

How can I calculate custom statistics in a strategy?

To track metrics other than the default metrics that the Strategy Tester tracks, strategies can calculate custom statistics. These calculations might need to detect order executions, track closed trades, monitor entries into trades, and assess whether a trade is active. Changes in built-in variables such as strategy.opentrades and strategy.closedtrades can track the execution of orders.

The following example script uses a moving average crossover strategy to generate orders. It calculates custom metrics, including the price risk at entry, average position size, and the average percentage of bars involved in trades across the dataset, and plots the custom metrics and some built-in variables to the Data Window. Users can view the history of values plotted in the Data Window by moving the cursor over any bar. In contrast, the Strategy Tester summarizes data over the entire testing period.

image

//@version=6 strategy("Custom strategy metrics", "", true, initial_capital = 10000, commission_type = strategy.commission.percent, commission_value = 0.075, max_bars_back = 1000, default_qty_type = strategy.percent_of_equity, default_qty_value = 100) // Calculate entry conditions. float c = math.round_to_mintick(close) // Round OHLC to chart prices. float maF

= math.round_to_mintick(ta.sma(hlc3, 10)), float maS = math.round_to_mintick(ta.sma(hlc3, 60)) bool enterLong = ta.crossover(maF, maS), bool enterShort = ta.crossunder(maF, maS) // Entry conditions. float stopLong = ta.lowest(20)[1], float stopShort = ta.highest(20)[1] // Stop-loss order levels. // Enter a new position or reverse, unless stop could not be calculated yet. if enterLong and not na(stopLong) strategy.entry("Long", strategy.long, comment = "►Long") if enterShort and not na(stopShort) strategy.entry("Short", strategy.short, comment = "►Short") // Modify existing exit orders using the current stop value. strategy.exit("◄Long", "Long", stop = stopLong), strategy.exit("◄Short", "Short", stop = stopShort) // Generate custom statistics. float riskOnEntry = math.abs(c - (enterLong ? stopLong : enterShort ? stopShort : na)) // Trade risk at entry. int changeInClosedTrades = ta.change(strategy.closedtrades) int changeInOpenTrades = ta.change(strategy.opentrades) bool tradeWasClosed = changeInClosedTrades != 0 bool tradeWasEntered = changeInOpenTrades > 0 or (strategy.opentrades == strategy.opentrades[1] and tradeWasClosed) or changeInClosedTrades > 1 bool tradeIsActive = strategy.opentrades != 0 // Check if a trade is currently active. float barsInTradePct = 100 * ta.cum(tradeIsActive ? 1 : 0) / bar_index // Percentage of bars on which a trade was open. float tradesEntered = ta.cum(tradeWasEntered ? 1 : 0) float positionSize = math.abs(strategy.position_size) float avgPositionSize = ta.cum(nz(positionSize))[1] / tradesEntered // Calculate average position size. float positionValue = positionSize * close // Position monetary value float priceRiskPct = riskOnEntry / close // Risk percentage of trade relative to entry price. float tradeRiskPct = positionSize * riskOnEntry // Monetary risk of the trade. float stop = strategy.position_size > 0 ? stopLong : strategy.position_size < 0 ? stopShort : na // Plot the MAs, stop price, and markers for entries and exits to the chart. plot(maF,"MA Fast"), plot(maS, "MA Slow", color.silver), plot(stop, "Stop", color.fuchsia, 1, plot.style_circles) plotchar(tradeWasClosed, "tradeWasClosed", "—", location.bottom, color.fuchsia, size = size.tiny) plotchar(tradeWasEntered, "tradeWasEntered", "+", location.top, color.lime, size = size.tiny) // Highlight the background while long and short positions are active. bgcolor(strategy.position_size > 0 ? color.new(color.teal, 80) :

```
strategy.position_size < 0 ? color.new(color.maroon, 80) : na) // Plot statistics to the Data Window. plot(na, "═══════ Built-ins", display = display.data_window) plot(strategy.opentrades, "strategy.opentrades", display = display.data_window) plot(strategy.closedtrades, "strategy.closedtrades", display = display.data_window) plot(strategy.position_size, "strategy.position_size", display = display.data_window) plot(strategy.equity, "Equity", display = display.data_window) plot(na, "═══════ Custom Metrics", display = display.data_window) plot(riskOnEntry, "Risk On Entry", display = display.data_window) plot(positionSize, "Position Size", display = display.data_window) plot(tradesEntered, "tradesEntered", display = display.data_window) plot(barsInTradePct, "barsInTradePct", display = display.data_window) plot(avgPositionSize, "avgPositionSize", display = display.data_window) plot(positionValue, "Position Value", display = display.data_window) plot(priceRiskPct, "Price Risk %", display = display.data_window) plot(tradeRiskPct, "Trade Risk Value", display = display.data_window)
```

Note that:

- The strategy incorporates trading costs. Failing to account for these costs can lead to an unrealistic perception of strategy performance and diminish the credibility of test results.
- We round the open, high, low and close (OHLC) built-in variables to the symbol's precision. This rounding ensures that any statistics the script calculates align within the Strategy Tester and with strategy order-related built-in variables.
- The script creates global variables for the changes in built-in variables for open and closed trades so that the ta.change function is called on every bar for consistency.

How do I incorporate leverage into my strategy?

Trading with *leverage* means borrowing capital from a broker to control larger position sizes than the amount of capital risked. This amplifies both potential profits and losses, making it a powerful but risky tool. The amount of the trader's capital that they risk is called the *margin*.

For example, setting a 20% margin ratio means that the trader's balance funds only 20% of the position's value, allowing positions up to five times the account balance. A margin ratio of 20% is therefore the same as 5:1 leverage. With an available balance of $10,000 and a 20% margin setting, a strategy can open positions up to $50,000 in value.

Pine Script strategies can simulate trading with leverage by specifying margin requirements for long and short positions. Users can adjust the "Margin for long positions" and Margin for short positions" in the strategy's "Properties" tab. Programmers can set the default margin in the script using the `margin_long` and `margin_short` parameters in the strategy() declaration function.

For more information on using leverage in strategies, see the Help Center article How do I simulate trading with leverage?

Can you hedge in a Pine Script strategy?

When traders offset the risk of one position by opening another position at the same time, this is called *hedging*.

The main ways to hedge an open position are:

- By opening a second position in a related asset that is expected to move in the opposite direction to the first asset.
- By opening a short position to offset a long position or vice-versa.
- By using derivatives such as options.

Strategies cannot use these methods, because Pine strategies can only have positions open in one direction at a time, either long or short. Pine strategies run on only the chart asset and cannot open positions in different assets.

Can I connect my strategies to my paper trading account?

Pine Script does not support placing orders using the brokers integrated via the Trading Panel, or using TradingView's built-in paper trading account. The Strategy Tester closely mimics a paper trading account by

simulating orders and tracking theoretical positions and capital in a risk-free environment.

Strategies can customize order fill alerts to include detailed results and performance metrics in the alert strings, providing a record of the strategy's theoretical fills and overall performance in realtime.

Troubleshooting and specific issues

Why are no trades executed after I add the strategy to the chart?

If a strategy that is running on the chart does not place any orders, the Strategy Tester's "Overview" tab displays the message, "This strategy did not generate any orders throughout the testing range." By contrast, while no strategy is loaded and visible on the chart, the Strategy Tester displays a different message: "To test a strategy, apply it to the chart."

If a valid script that uses the strategy() declaration statement is running but is not placing any orders, consider the following potential problems and their solutions:

Lack of order placement commands

The strategy must use either the strategy.order() or strategy.entry() order placement commands to place orders. Add log.info messages and review the Pine Logs to check whether the conditions to run the commands are met.

Insufficient capital

Verify that the strategy has enough initial capital to cover the position sizes it attempts to open. Remember, the cost of entering a futures contract position is the chart price multiplied by the syminfo.pointvalue, which can be significantly greater than the chart price. For a quick fix, increase the initial capital to a very high value in the *Properties* tab.

Runtime errors

Check for runtime errors indicated by a red exclamation mark on the chart pane next to the script's title. Resolve any issues by correcting the script as necessary.

For more detailed guidance and troubleshooting tips, refer to the dedicated article on this topic in the Help Center.

Why does my strategy not place any orders on recent bars?

If a strategy places one or more orders early in the testing range but then stops placing orders, check the following issues.

Total account loss

Check whether the simulated account balance experienced a total loss of equity earlier in the available history. As a result, the account might lack sufficient capital to continue trading the symbol and fail to show trades only in the chart's recent history.

No exit condition

Some programmers define entry conditions that rely on having no positions currently open. Make sure to explicitly close trades by specifying corresponding exit conditions for all trades. Without explicit instructions to close an open position using strategy.close() or strategy.exit() commands, the strategy might display only a single entry order early in the chart's history and in the *List of Trades* tab. If trades are not closed, they do not generate results in the *Overview*.

Why is my strategy repainting?

Pine scripts *repaint* if they behave differently on historical and realtime bars. If strategies repaint, their backtesting results are not reliable because they do not accurately represent the strategy's behavior in realtime.

Some strategy properties cause repainting:

- The calc_on_every_tick setting causes a strategy to recalculate with every price update, which may cause orders and alerts to trigger during the formation of a bar in realtime. By contrast, on historical bars, calculations are performed at the close of the bar.
- The calc_on_order_fills setting causes a strategy to recalculate immediately after simulating an order fill. For example, this feature is particularly useful for strategies that rely on entry prices to set exit prices on the entry bar, rather than waiting for the bar to close, such as the first example script in the FAQ entry How can I set stop-loss and take-profit levels as a percentage from my entry point using `calc_on_order_fills`? However, using this setting can introduce *lookahead bias* into the strategy, leading to potentially unrealistic outcomes. For instance, if a strategy's entry conditions are met within a bar that also triggers an exit, the strategy would execute an entry order within the same bar on the next tick. On historical bars, such entries could occur at any of the bar's open, high, low, or close (OHLC) prices, resulting in entry prices that are unrealistically favorable.
- Since strategies and their alerts execute at the close of a historical bar, the next possible moment for an entry order to be filled is the beginning of the next bar. However, the process_orders_on_close setting causes the strategy to use the close price of the bar where the condition is met for its order prices instead. See the FAQ entry Why are my orders executed on the bar following my triggers? for more information.

To avoid repainting, set the calc_on_every_tick, calc_on_order_fills, and process_orders_on_close parameters to `false` in the strategy() declaration statement.

Additionally, using unfixed data from a higher timeframe can cause repainting. If the data from the higher timeframe changes during the higher timeframe bar, this can change the script's oputput for historical bars. Ensure that strategies use only fixed values from a higher timeframe, as described in Avoiding repainting.

Although these are the most common causes of repainting in strategies, they are not the only causes. For additional information, refer to the section on repainting in the User Manual.

How do I turn off alerts for stop loss and take profit orders?

In automated trading strategies, it is common practice to set stop-loss and take-profit orders at the same time as an entry order, using the alert from the entry order as a trigger. In this case, sending alerts for the stop-loss and take-profit order fills can be unnecessary or even problematic. To disable alerts for a specific order placement command, set the `disable_alert` parameter to `true`. The broker emulator still simulates the fills for these orders, but sends no alerts for them.

Here is an example of an order fill command with this parameter set:

strategy.exit("Exit", stop = stopLevel, limit = limitLevel, disable_alert = true)

Previous

Next

Strings and formatting

Strings and formatting

How can I place text on the chart?

Scripts can display text using the following methods:

- The plotchar() or plotshape() functions for static text, which doesn't change.
- Labels and boxes for dynamic text, which can vary bar to bar.
- Tables for more complex text (static or dynamic) that stays in the same region of the chart.

Plotting text

The plotchar() and plotshape() functions can display fixed text on bars:

- A single plotchar() function call can print a string using the `text` parameter, but only one character using the `char` parameter. To plot only the text and not the character, set `char` to `""`.
- A plotshape() function call can print a string using the `text` parameter. To plot only the text and not the shape, set the `color` (for the shape) to na and the `textcolor` to something visible.

Plots appears on the bar where the script calls the function, by default, but scripts can offset a plot by a dynamic number of bars to the left or right. On the Y axis, the plots appear above/below the bar, at the top/bottom of the chart, or at an arbitrary price level. Scripts can call a plotchar() or plotshape() function on any number of bars and it counts as a single plot towards the plot limit.

When using these functions, the text cannot change during the execution of the script. The `text` parameter accepts an argument of type “const string”, which means it cannot cannot change from bar to bar and cannot be supplied by an input.

This script, for example, does not compile, because the argument to the `text` parameter is a “series string”:

//@version=6 indicator("Plotting text demo: incorrect", overlay = true) float rsi = ta.rsi(close, 14) bool rsiUp = ta.crossover(rsi, 50) bool rsiDn = ta.crossunder(rsi, 50) string txt = rsiUp ? "RSI\nUp" : rsiDn ? "RSI\nDown" : "" plotchar(series = rsiUp or rsiDn, title = "Up/Down", char = "R", text = txt, location = location.top, size = size.tiny)

To print different text depending on a logical condition, use two function calls and control them using the `series` parameter. Note that even if the `series` for one or both of the function calls is never true during a script’s execution, and so no shape, character or text is ever plotted, *both* functions still count towards the plot limit.

The following script corrects the earlier example, and shows the use of both plotchar() and plotshape() to display text:

//@version=6 indicator("Plotting text demo", overlay = true) float rsi = ta.rsi(close, 14) bool rsiUp = ta.crossover(rsi, 50) bool rsiDn = ta.crossunder(rsi, 50) plotchar(series = rsiUp, title = "Up", char = "▲", location = location.belowbar, color = color.lime, text = "RSI\nUp", size = size.tiny) plotshape(series = rsiDn, title = "Down", style = shape.triangledown, location = location.abovebar, color = color.fuchsia, text = "RSI\nDown", size = size.tiny, textcolor = color.fuchsia)

Labels

Labels are particularly useful for displaying text that can change from one bar to another. The `text` parameter of the label.new function takes a “series string”, so it can change whenever necessary.

Labels do not count towards the plot limit, but there is a separate limit of how many labels can display on the chart. By default, up to approximately 50 of the *most recent* labels appear on the chart.

Programmers can adjust this limit up to 500 by setting the `max_labels_count` parameter in the indicator() or strategy() functions.

The parameters to the label.new function for text, color, etc., take "series" arguments. This makes labels much more flexible than plots.

The following example script displays the same information as the previous script, but using labels. The background to the labels is transparent (set to na) in this example, to more closely match the style of the previous scripts.

//@version=6 indicator("Drawing labels demo", "", true) float rsi = ta.rsi(close, 14) bool rsiUp = ta.crossover(rsi, 50) bool rsiDn = ta.crossunder(rsi, 50) if rsiUp or rsiDn string labelText = rsiUp ? "▲\nRSI Up" : "RSI Down\n▼" color textColor = rsiUp ? color.lime : color.fuchsia string labelPos = rsiUp ? yloc.belowbar : yloc.abovebar label.new(bar_index, na, labelText, yloc = labelPos, color = color(na), textcolor = textColor)

As well as showing historical information, labels can also be used to show only the latest information on the current bar. The following example script displays the value of RSI in a different color depending on whether it is above or below 50, for the most recent bar only. This is not possible using plotchar() or plotshape(), because the text is fixed, and too many plots would be required to plot every value separately.

//@version=6 indicator("Single label demo", "", true) float rsi = ta.rsi(close, 14) bool rsiAbove50 = rsi >= 50 bool rsiBelow50 = rsi < 50 var label rsiLabel = label.new(na, na, style = label.style_label_left, yloc = yloc.price, color = color.new(color.gray,70)) if barstate.islast color textColor = rsiAbove50 ? color.lime : rsiBelow50 ? color.fuchsia : color(na) rsiLabel.set_x(bar_index + 1) rsiLabel.set_y(open) rsiLabel.set_text(str.format("RSI: {0, number, #.##}", rsi)) rsiLabel.set_textcolor(textColor)

Note that:

- We create the label once, on the first bar, with all its unchanging properties such as style and background color already set.
- We do nothing with the label for all historical bars.

- We update the changing properties of the label such as text and position on the most recent bar and on every realtime bar. This method is more performant than updating the label on all bars or creating and deleting it each bar.

Boxes

Boxes can also display text on the chart, by providing the text to the `text` parameter of the box.new() function. Boxes work with text in a similar way to labels, but with some additional features.

Labels exist specifically to display text — and so the label adjusts to the size of the text. Labels always resize so that all of the text is visible inside of the label.

The main use of boxes is to display the drawing itself. A box attaches to specific points on the chart, and its text might or might not fit into it. To ensure that the text displays in the best possible way, boxes provide some additional features that can not be used in labels: text wrapping and text alignment.

Text contained in the box can automatically wrap if it reaches the border of the box, if the `text_wrap` parameter is set to `text.wrap_auto`. Additionally, scripts can align the text inside the box along the vertical and horizontal axes. Using the `text_halign` and `text_valign` parameters of `box.new()`, text can display at one of the nine possible positions inside of the box.

In the example below, we draw a box that spans the last 50 historical bars on the chart, and a label. We add long text to both. With `text_wrap = text.wrap_auto`, the text inside the box automatically wraps to fit the box itself, while the text inside of the label stays unchanged:

image

//@version=6 indicator("Box and label text", overlay = true) if barstate.islastconfirmedhistory bt = "This long text is inside of a box, which means it is automatically wrapped and scaled to be visible in the constraints of the box." lt = "This long text is inside of a label, which

means that it is displayed as is, and the label is simply drawn around it. It doesn't change when the chart is scaled." box.new(bar_index[50], close * 1.1, bar_index, close, text = bt, text_wrap = text.wrap_auto, text_size = 36) label.new(bar_index[25], close * 1.1, lt, size = 36)

Tables

Tables are useful to display information in a fixed position on the chart. Whereas plots and labels can easily show historical information because they are, or can be, linked to specific bars, table contents do not change as users move the cursor over past chart bars. This makes tables best suited for showing *current* information.

The following example script displays the value of RSI in a different color depending on whether it is above or below 50, for the most recent bar only.

//@version=6 indicator("RSI table", "", true) var table rsiDisplay = table.new(position.top_right, 1, 1, bgcolor = color.gray, frame_width = 2, frame_color = color.black) float rsi = ta.rsi(close, 14) bool rsiAbove50 = rsi >= 50 bool rsiBelow50 = rsi < 50 color textColor = rsiAbove50 ? color.lime : rsiBelow50 ? color.fuchsia : color(na) if barstate.isfirst table.cell(rsiDisplay, 0, 0, "") else if barstate.islast table.cell_set_text(rsiDisplay, 0, 0, str.format("RSI: {0, number, #.##}", rsi)) table.cell_set_text_color(rsiDisplay, 0, 0, textColor)

Note that:

- We create the table and its single cell only once, and update the text and color on the most recent bar and on every realtime bar, for performance.
- This script displays the same information as the preceding example did using a label.
- Although this is a simple example, for more complex information, tables are easier to organise and read than labels.

How can I position text on either side of a single bar?

Scripts can position a label to the *right* of a bar by using `style = label.style_label_left`. This style *points* the label to the **right** and *places* it to the **left**. Likewise, a label with `style = label.style_label_right` displays to the right of the bar, pointing left.

To manage the alignment of the text within the label, use the `textalign` parameter.

The following example script draws three labels on the chart's last bar, with different `style` and `textalign` values. User inputs control whether individual labels appear, and the central label is off by default for readability. If the input to hide the background is enabled, the color is set to na so that it does not appear. Note that the proper way to do this is to cast it to a color by using `color(na)`.

image

//@version=6 indicator("Text position demo", "", true) hideBackgroundInput = input.bool(false, "Hide Background") color backgroundColor = hideBackgroundInput ? color(na) : color.new(color.gray,70) // @function Prints a label with the specified text at a specific position and alignment. // @param txt (string) The text to be displayed in the label. // @param pos (string) The label style. // @param align (string) The horizontal alignment of text within the label. // @returns (void) Function has no explicit return. print(string txt, string pos, string align) => var label lbl = label.new(na, na, na, xloc.bar_index, yloc.price, backgroundColor, pos, chart.fg_color, size.huge, align, text_font_family = font.family_monospace) label.set_xy(lbl, bar_index, high) label.set_text(lbl, txt) if input.bool(true, "Show Left Label") print("label_left\ntext.align_left", label.style_label_left, text.align_left) if input.bool(true, "Show Right Label") print("label_right\ntext.align_right", label.style_label_right, text.align_right) if input.bool(false, "Show Center Label")

print("label_center\nalign_center", label.style_label_center, text.align_center)

How can I stack plotshape() text?

To make multiple text plots visible on the same bar, the text on one plot must be raised or lowered so that it does not overlap with another plot.

To add a blank line in a plotchar() or plotshape() call, add the newline character \n. Text above the bar or at the top of the chart can only be *raised*, by adding a newline *after* the text. Newlines added before the text are ignored. Likewise, text below the bar or at the bottom of the chart can only be *lowered*, by adding a newline *before* the text.

The following example script shows how to correctly stack text by inserting a blank line over or under other text:

//@version=6 indicator("Stack text demo", "", true) plotshape(true, "", shape.arrowup, location.abovebar, color.green, textcolor = color.green, text = "A") plotshape(true, "", shape.arrowup, location.abovebar, color.lime, textcolor = color.lime, text = "B\n") plotshape(true, "", shape.arrowdown, location.belowbar, color.red, textcolor = color.red, text = "C") plotshape(true, "", shape.arrowdown, location.belowbar, color.maroon, textcolor = color.maroon, text = "\nD")

image

How can I print a value at the top right of the chart?

Refer to the Placing a single value in a fixed position section of the Tables page. The example in that section uses a single-cell table to display a string representation of a value in the top-right corner of the chart.

How can I split a string into characters?

The str.split() function splits a string into parts and stores the parts in an array. To split a string into individual characters, use an empty string `""` as the `separator` argument. Here is a code example:

//@version=6 indicator("Split a string into characters") string sourceStringInput = input.string("123456789", "String to Split") var array<string> charactersArray = str.split(sourceStringInput, "") if barstate.islast string txt = sourceStringInput + "\n" + str.tostring(charactersArray) var label = label.new(na, na, txt, xloc.bar_index, yloc.price, color(na), label.style_label_left, chart.fg_color, size.large, text.align_left) label.set_xy(label, bar_index, open)

Previous

Next

Techniques

Pine Script® v6 Documentation

User Manual / FAQ / Techniques

Techniques

How can I prevent the “Bar index value of the x argument is too far from the current bar index. Try using `time` instead” and “Objects positioned using xloc.bar_index cannot be drawn further than X bars into the future” errors?

Both these errors occur when creating objects too distant from the current bar. An x point on a line, label, or box can not be more than 9999 bars in the past or more than 500 bars in the future relative to the bar on which the script draws it.

Scripts *can* draw objects beyond these limits, however, using xloc.bar_time instead of the `xloc` parameter, and time as an alternative to bar_index for the x arguments.

Note that, by default, all drawings use `xloc.bar_index`, which means that the values passed to their x-coordinates are treated as if they are bar indices. If drawings use a `time`-based value without specifying `xloc = xloc.bar_time`, the timestamp — which is usually an `int` value of trillions of milliseconds — is treated as an index of a bar in the future, and inevitably exceeds the 500 future bars limit. To use `time`-based values for drawings, always specify `xloc.bar_time`.

How can I update the right side of all lines or boxes?

Scripts can update the x2 value of all lines or boxes by storing them in an array and using a for...in loop to iterate over each object. Update the x2 value using the line.set_x2() or box.set_right() functions.

In the example below, we create a custom array and go over it to extend lines with each new bar:

//@version=6 indicator("Update x2 demo", "", true) int activeLevelsInput = input.int(10, "Number of levels") int pivotLegsInput = input.int(5, "Pivot length") // Save pivot prices. float pHi = ta.pivothigh(pivotLegsInput, pivotLegsInput) // Initialize an array for lines on the first bar, sized to match the number of levels to track. var array<line> pivotLines = array.new<line>(activeLevelsInput) // Check for a pivot. Add a new line to the array. Remove and delete the oldest line. if not na(pHi) line newPivotLine = line.new(bar_index[pivotLegsInput], pHi, bar_index, pHi) pivotLines.push(newPivotLine) pivotLines.shift().delete() // Update all line x2 values. if barstate.islast for eachLine in pivotLines eachLine.set_x2(bar_index)

As an alternative to adding new drawings to a custom array, scripts can use the appropriate built-in variable that collects all instances of a drawing type. These arrays use the `<drawingNamespace>.all` naming scheme: for example, scritps can access all drawn labels by referring to label.all, all polylines with polyline.all, etc. Scripts can iterate over these arrays in the same way as with custom arrays.

This example implements gets the same result using the line.all built-in array instead:

//@version=6 indicator("Update x2 demo", "", true) int activeLevelsInput = input.int(10, "Number of levels") int pivotLegsInput = input.int(5, "Pivot length") // Save pivot prices. float pHi = ta.pivothigh(pivotLegsInput, pivotLegsInput) // Check for a pivot. Delete the oldest line if the array is over the "Number of levels" limit. if not na(pHi) line newPivotLine = line.new(bar_index[pivotLegsInput], pHi,

bar_index, pHi) if line.all.size() > activeLevelsInput line.all.first().delete() // Update all line x2 values. if barstate.islast for eachLine in line.all eachLine.set_x2(bar_index)

How to avoid repainting when *not* using the `request.security()` function?

Scripts can give deceptive output if they repaint by behaving differently on historical and elapsed realtime bars. This type of repainting is most commonly caused by requesting data from another context using the request.security() function.

Scripts can also change their output during a realtime bar, as the volume, close, high, and low values change. This form of repainting is not normally deceptive or detrimental.

To avoid this kind of repainting and ensure that outputs do not change during a bar, consider the following options:

- Use confirmed values, or the values from the previous bar.
- Set alerts to fire on bar close. Read more about repainting alerts in the FAQ entry Why is my alert firing at the wrong time?
- Use the open in calculations instead of the close.

For further exploration of these methods, see the PineCoders publication “How to avoid repainting when NOT using security()“.

How can I trigger a condition n bars after it last occurred?

Using the ta.barssince() function, scripts can implement a condition when a certain number of bars have elapsed since the last occurrence of that condition.

The following example script uses the `cond` condition to plot a blue star when the close value is greater than the open value for two consecutive bars. Then, the `trigger` variable is true only if the `cond` condition is already true *and* the number of bars elapsed since `cond`

was last true is greater than `lengthInput`. The script plots a red “O” on the chart, overlaying the blue star, each time these conditions are met. The Data Window displays the count since cond was last true.

image

//@version=6 indicator("`barssince` demo", overlay = true) int lengthInput = input.int(3, "Length") bool cond = close > open and close[1] > open[1] int count = ta.barssince(cond[1]) + 1 bool trigger = cond and count > lengthInput plot(cond ? 0 : count, "Count", display = display.data_window) plotchar(cond) plotchar(trigger, "", "O", color = color.red)

How can my script identify what chart type is active?

Various boolean built-in variables within the `chart.*` namespace enable a script to detect the type of chart it is running on.

The following example script defines a function, `chartTypeToString()`, which uses the `chart.*` built-ins to identify the chart type and convert this information into a string. It then displays the detected chart type in a table on the chart.

//@version=6 indicator("Chart type", "", true) chartTypeToString() => string result = switch chart.is_standard => "Standard" chart.is_heikinashi => "Heikin-Ashi" chart.is_kagi => "Kagi" chart.is_linebreak => "Line Break" chart.is_pnf => "Point and Figure" chart.is_range => "Range" chart.is_renko => "Renko" if barstate.islastconfirmedhistory var table display = table.new(position.bottom_right, 1, 1, bgcolor = chart.fg_color) table.cell(display, 0, 0, str.format("Chart type: {0}", chartTypeToString()), text_color = chart.bg_color)

How can I plot the highest and lowest visible candle values?

To plot the highest high and lowest low within the range of visible bars, a script can use the chart.left_visible_bar_time and chart.right_visible_bar_time built-ins. These variables allow the script to identify the times of the earliest and latest visible bars on the chart and calculate the maximum or minimum values within that range.

The VisibleChart library by PineCoders offers such functionality with its `high()` and `low()` functions, which dynamically calculate the highest and lowest values of the currently visible bars.

The following example script uses functions from this library to create two horizontal lines on the chart, signifying the highest and lowest price points within the range of visible bars. The script draws labels for these lines, displaying both the price and the corresponding timestamp for each high and low point. As the chart is manipulated through scrolling or zooming, these lines and labels dynamically update to reflect the highest and lowest values of the newly visible bars:

image

//@version=6 indicator("Chart's visible high/low", "", true) import PineCoders/VisibleChart/4 as PCvc // Calculate the chart's visible high and low prices and their corresponding times. int x1 = PCvc.highBarTime() int x2 = PCvc.lowBarTime() float chartHi = PCvc.high() float chartLo = PCvc.low() // Draw lines and labels on the last bar. if barstate.islast line.new(x1, chartHi, x2, chartHi, xloc.bar_time, extend.both, color.lime) line.new(x1, chartLo, x2, chartLo, xloc.bar_time, extend.both, color.fuchsia) string hiTxt = str.format("{0}\n{1}", str.tostring(chartHi, format.mintick), str.format_time(x1, format = "dd/MM/yy @ HH:mm")) string loTxt = str.format("{0}\n{1}", str.tostring(chartLo, format.mintick), str.format_time(x2, format = "dd/MM/yy @ HH:mm")) label.new(x1, chartHi, hiTxt, xloc.bar_time, yloc.price, color.new(color.lime, 80), label.style_label_down, color.lime) label.new(x2, chartLo, loTxt,

xloc.bar_time, yloc.price, color.new(color.fuchsia, 80),
label.style_label_up, color.fuchsia)

Note that:

- Values derived from visible chart variables can change throughout the script's runtime. To accurately reflect the entire visible range, the script defers drawing the lines until the last bar (using barstate.islast).
- Because the visible chart values are defined in the global scope, *outside* the local block defined by barstate.islast, the functions process the entire dataset before determining the final high and low values.

For more information, refer to the VisibleChart library's documentation.

How to remember the last time a condition occurred?

Scripts can store the number of bars between the current bar and a bar on which a condition occurred in various ways:

- Using ta.barssince(). This built-in function is the simplest way to track the distance from the condition.
- Manually replicating the functionality of ta.barssince() by initializing the distance to zero when the condition occurs, then incrementing it by one on each bar, resetting it if the condition occurs again.
- Saving the bar_index when the condition occurs, and calculating the difference from the current bar_index.

Programmers can then use the number of bars with the history-referencing operator [] to retrieve the value of a variable, such as the close, on that bar.

Alternatively, if the script needs *only* the value itself and not the number of bars, simply save the value each time the condition occurs. This method is more efficient because it avoids referencing the series multiple times throughout its history. This method also reduces the risk

of runtime errors in scripts if the size of the historical reference is too large.

Here’s a script that demonstrates these methods:

//@version=6 indicator("Track distance from condition", "", true) // Plot the high/low from the bar where a condition occurred the last time. // Conditions bool upBar = close > open bool dnBar = close < open bool up3Bars = dnBar and upBar[1] and upBar[2] and upBar[3] bool dn3Bars = upBar and dnBar[1] and dnBar[2] and dnBar[3] display = display.data_window // Method 1: Using "ta.barssince()". plot(high[ta.barssince(up3Bars)], color = color.new(color.blue, 80), linewidth = 16) plot(low[ta.barssince(dn3Bars)], color = color.new(color.red, 80), linewidth = 16) plot(ta.barssince(up3Bars), "1. ta.barssince(up3Bars)", display = display) plot(ta.barssince(dn3Bars), "1. ta.barssince(dn3Bars)", display = display) // Method 2: Manually replicating the functionality of the "ta.barssince()" function. var int barsFromUp = na var int barsFromDn = na barsFromUp := up3Bars ? 0 : barsFromUp + 1 barsFromDn := dn3Bars ? 0 : barsFromDn + 1 plot(high[barsFromUp], color = color.blue, linewidth = 3) plot(low[barsFromDn], color = color.red, linewidth = 3) plot(barsFromUp, "3. barsFromUp", display = display) plot(barsFromDn, "3. barsFromDn", display = display) // Method 3: Storing the `bar_index` value when a condition is met. var int barWhenUp = na var int barWhenDn = na if up3Bars barWhenUp := bar_index if dn3Bars barWhenDn := bar_index plot(high[bar_index - barWhenUp], color = color.new(color.blue, 70), linewidth = 8) plot(low[bar_index - barWhenDn], color = color.new(color.red, 70), linewidth = 8) plot(bar_index - barWhenUp, "2. bar_index - barWhenUp", display = display) plot(bar_index - barWhenDn, "2. bar_index - barWhenDn", display = display) // Method 4: Storing the value when a condition is met. var float highWhenUp = na var float lowWhenDn = na if up3Bars highWhenUp := high if dn3Bars lowWhenDn := low plot(highWhenUp, color = color.new(color.white, 70), linewidth = 1) plot(lowWhenDn, color = color.new(color.white, 70), linewidth = 1)

How can I plot the previous and current day’s open?

There are several methods for plotting prices from a higher timeframe (we assume that these scripts are to be run on intraday timeframes).

Using `timeframe.change()`

The timeframe.change() function identifies when a bar in a specified timeframe opens. When a new daily bar opens, the following example script first copies the existing daily opening value to the variable for the previous day, and then updates the opening price for the current day.

//@version=6 indicator("Previous and current day open using `timeframe.change()`", "", true) bool newDay = timeframe.change("1D") var float yesterdayOpen = na var float todayOpen = na if newDay yesterdayOpen := todayOpen // We reassign this value first todayOpen := open // and then store today's open plot(yesterdayOpen, "Yesterday's Open", newDay ? na : color.red, 2, plot.style_line) plot(todayOpen, "Today's Open", newDay ? na : color.green, 2, plot.style_line) bgcolor(newDay ? color.new(color.gray, 80) : na)

Note that:

- This method uses the chart’s timeframe transitions to establish open prices and does not make adjustments for session times.
- For some markets and instrument types, the intraday data and the daily data is expected to differ. For example, the US exchanges like NASDAQ and NYSE include more trades in daily bars than in intraday ones, which results in different OHLC values between intraday and daily data, and in daily volume being far greater than intraday one. As a result, the first open of a trading session on an intraday chart can differ from the open of its respective 1D candle.

Using request.security()

To match the values on the chart with the values on higher timeframe charts, it’s necessary to access the higher timeframe data feeds. Scripts can achieve this by using the request.security() function.

The following example script requests two data feeds from a higher timeframe. To reduce the risk of repainting, we use only confirmed values for historical bars. The script plots confirmed values retroactively on each preceding day when a new day begins. For the real-time bar of the higher timeframe, which represents the current day, we draw a separate set of lines. The realtime lines can change during the day. While this type of repainting is not apparent here when using the opening price, which does not change after the bar opens, it is more obvious for scripts that use the closing price, which takes the current price until the bar closes.

//@version=6 indicator("Previous and current day open using `request.security()`", "", true, max_lines_count = 500) string periodInput = input.timeframe("1D", "Higher timeframe") [htfOpen1, htfOpen2, htfTime, htfTimeClose] = request.security(syminfo.tickerid, periodInput, [open[1], open[2], time[1], time_close[1]], lookahead = barmerge.lookahead_on) [htfRtOpen, htfRtOpen1] = request.security(syminfo.tickerid, periodInput, [open, open[1]]) var line rtOpen = line.new(na, na, na, na, xloc.bar_time, color = color.lime) var line rtOpen1 = line.new(na, na, na, na, xloc.bar_time, color = color.gray) var int rtStart = time var int rtEnd = time_close(periodInput) if ta.change(htfTime) != 0 line.new(htfTime, htfOpen1, htfTimeClose, htfOpen1, xloc.bar_time, color = color.lime) line.new(htfTime, htfOpen2, htfTimeClose, htfOpen2, xloc.bar_time, color = color.gray) rtStart := time rtEnd := time_close(periodInput) line.set_xy1(rtOpen1, rtStart, htfRtOpen1), line.set_xy2(rtOpen1, rtEnd, htfRtOpen1) line.set_xy1(rtOpen, rtStart, htfRtOpen), line.set_xy2(rtOpen, rtEnd, htfRtOpen) bgcolor(timeframe.change(periodInput) ? color.new(color.gray, 80) : na)

Using `timeframe`

Instead of writing custom logic to retrieve or calculate prices for a particular timeframe, programmers can run the entire script in that timeframe.

If scripts include the `timeframe` parameter in the indicator declaration, the user can choose the timeframe in which the script runs. The script can set a default timeframe.

By default, the following script plots the current and previous day's opening prices, similar to the previous examples. It is much simpler, but behaves quite differently. For historical bars, the script returns values when the day closes, effectively one day "late". For realtime and elapsed realtime bars, the script returns live values, if the option "Wait for timeframe closes" is not selected in the script settings.

//@version=6 indicator("Previous and current day open using `timeframe`", "", true, timeframe = "1D", timeframe_gaps = true) plot(open[1], "Yesterday's Open", color.red, 2, plot.style_line) plot(open, "Today's Open", color.green, 2, plot.style_line)

Note that:

- Only simple scripts that do not use drawings can use the `timeframe` parameter.
- Scripts that use the `timeframe` parameter can plot values quite differently depending on which settings are chosen. For an explanation, see this Help Center article.

How can I count the occurrences of a condition in the last x bars?

One obvious method is to use a for loop to retrospectively review each of the last x bars and check for the condition. However, this method is inefficient, because it examines all bars in range *again* on every bar, even though it already examined all but the last bar.

In general, using unnecessary, large, or nested for loops can result in slower processing and longer chart loading times.

The simplest and most efficient method is to use the built-in math.sum() function, and pass it a conditional series to count. This function maintains a running total of the count as each bar is processed, and can take a simple or series length.

The following example script uses both of these calculation methods. It also uses a series length that adjusts for the first part of the chart, where the number of bars available is less than the length. This way, the functions do not return na values.

image

//@version=6 indicator("Number of occurrences demo", overlay = false) int lengthInput = input.int(100, "Length", minval = 1) // Condition to count. bool isUpBar = close > open // Count using a loop (inefficient). countWithLoop(bool condition, int length) => int count = 0 for i = 0 to length - 1 if condition[i] count += 1 count // Count using Pine's built-in function. Can be "simple" or "series" length. countWithSum(bool condition, int length) => float result = math.sum(condition ? 1 : 0, length) float v1 = countWithSum(isUpBar, math.min(lengthInput, bar_index + 1)) int v2 = countWithLoop(isUpBar, math.min(lengthInput, bar_index + 1)) plot(v1, "Efficient count", color.red, 4) plot(v2, "Inefficient count", color.black, 1)

How can I implement an on/off switch?

An on/off switch is a persistent state that can be turned on once, and persists across bars until it is turned off. Scripts can use the var keyword to initialize a variable only once, and maintain its most recent value across subsequent bars unless it is reassigned. Such persistent states can be boolean values, or integers, or any other type.

The following example script show how to implement this. Each instance of the on and off triggers displays with an arrow and the word “On” or “Off”. A green background highlights the bars where the switch is in the “On” state.

image

//@version=6 indicator("On/Off condition example", overlay = true) bool upBar = close > open // On/off conditions. bool triggerOn = upBar and upBar[1] and upBar[2] bool triggerOff = not upBar and not upBar[1] // Switch state is saved across bars. var bool onOffSwitch = false // Turn the switch on or off, otherwise persist its state. onOffSwitch := triggerOn ? true : triggerOff ? false : onOffSwitch bgcolor(onOffSwitch ? color.new(color.green, 90) : na) plotchar(triggerOn, "triggerOn", "▲", location.belowbar, color.lime, size = size.tiny, text = "On") plotchar(triggerOff, "triggerOff", "▼", location.abovebar, color.red, size = size.tiny, text = "Off")

How can I alternate conditions?

Scripts can alternate from one state to another strictly, even when the triggers to change state do not occur in strict order. This can be useful to mark only the first trigger and not any subsequent triggers, or to prevent multiple alerts.

The following example script plots all pivots, defined by Williams fractals. These pivots can occur in any order. The script stores the type of the most recent pivot, and confirms the next pivot *only* if it is of the opposite type, such that confirmed pivots appear strictly high-low-high or low-high-low, etc. Confirmed pivots are plotted in a larger size and different color. The chart background color is colored according to the type of the most recent confirmed pivot.

image

//@version=6 indicator("Alternating states", "", true) lookback = input.int(2, title="Lookback & Lookahead") // Define an enum of allowed pivot types. enum PivotType high low undefined const color red80 = color.new(color.red, 80) const color green80 = color.new(color.green, 80) const color yellow80 = color.new(color.yellow, 80) // Define a variable of type PivotType to track the pivot direction. var PivotType lastPivot = PivotType.undefined // Define pivots. float pivotLowPrice = ta.pivotlow(lookback, lookback) float pivotHighPrice =

ta.pivothigh(lookback, lookback) bool isPivotLow = not na(pivotLowPrice) bool isPivotHigh = not na(pivotHighPrice) // Plot triangles for pivot points. plotshape(isPivotLow ? pivotLowPrice : na, "Low", shape.triangleup, location.belowbar, color.yellow, offset = -lookback, size = size.tiny) plotshape(isPivotHigh ? pivotHighPrice : na, "High", shape.triangledown, location.abovebar, color.yellow, offset = -lookback, size = size.tiny) // Confirm highs and lows strictly in order. `PivotType.undefined` handles the case where no pivot has yet occurrred. bool confirmedLow = isPivotLow and (lastPivot == PivotType.high or lastPivot == PivotType.undefined) bool confirmedHigh = isPivotHigh and (lastPivot == PivotType.low or lastPivot == PivotType.undefined) // Plot larger triangles for confirmed pivots. plotshape(confirmedLow ? pivotLowPrice : na, "Low Confirmed", shape.triangleup, location.belowbar, color.green, offset = -lookback, size = size.normal) plotshape(confirmedHigh ? pivotHighPrice : na, "High Confirmed", shape.triangledown, location.abovebar, color.red, offset = -lookback, size = size.normal) // Update last pivot direction. lastPivot := confirmedLow ? PivotType.low : confirmedHigh ? PivotType.high : lastPivot // Color the background of the chart based on the direction of the most recent confirmed pivot. bgcolor(lastPivot == PivotType.low ? green80 : lastPivot == PivotType.high ? red80 : lastPivot == PivotType.undefined ? yellow80 : na)

Note that:

- The script uses an enum variable with three possible values to store the type of the last pivot and to decide whether to confirm subsequent pivots.
- A single boolean value cannot reliably do this, because boolean values can only be `true` or `false` and not `na`. Using a boolean value can cause unexpected behavior, for example, at the beginning of the chart history where no trigger condition has occurred.
- A pair of boolean variables can replicate this behavior, with careful handling. See the FAQ entry “How can I accumulate a value for two exclusive states?” for an example of using two boolean values in this way.

- A string variable can also do the same thing. The advantage of an enum over a string is that all possible allowed values are known, thus avoiding the case where a condition tests for a value that is misspelled, outdated or otherwise not relevant. Such a test silently fails in every possible case, and the corresponding logic never runs. Such tests can therefore cause bugs that are difficult to find.

Can I merge two or more indicators into one?

It is possible to combine indicators, paying attention to the following points:

- Ensure that the scales that the indicators use are compatible, or re-scale them to be compatible. For example, combining a moving average indicator, designed to overlay the bar chart, with a volume bar indicator that's meant for a separate indicator pane is unlikely to display as expected.
- Check that variable names do not overlap.
- Convert each script to the most recent version of Pine Script®, or at least the same version, before combining them.
- Ensure that there is only one version declaration and script declaration in the resulting script.

How can I rescale an indicator from one scale to another?

Rescaling an indicator from one scale to another means trying to ensure that the values display within a similar range to other values, from the same indicator or from the chart.

For example, consider a script that displays volume typically measuring in the millions of units, and also RSI, which ranges from zero to one hundred. If the script displays these values in the same pane, the volume is visible but the RSI will be so small as to be unreadable.

Where values are dissimilar like this, they must be *rescaled* or *normalized*:

- If the minimum and maximum possible values are known, or *bounded*, the values can be *rescaled*, that is, adjusted to a new range bounded by different maximum and minimum values. Each value differs in absolute terms, but retains the same *relative* proportion to other rescaled values.
- If the values are *unbounded*, meaning that either the maximum or minimum values, or both, are not known, they must instead be *normalized*. Normalizing means scaling the values relative to historical maximum and minimum values. Because the maximum and minimum historical values can change over time as the script runs on more historical and realtime bars, the new scale is *dynamic* and therefore the new values are *not* exactly proportional to each other.

The example script below uses a `rescale()` function to rescale RSI values, and a `normalize()` function to normalize Commodity Channel Index (CCI) and volume values. Although normalizing is an imperfect solution, it is more complete than using ta.lowest() and ta.highest(), because it uses the minimum and maximum values for the complete set of elapsed bars instead of a subset of fixed length.

//@version=6 indicator("Rescaling and normalizing values", "", overlay = false) // @function Rescales a signal with a known scale (bounded) to a new scale. // @param src (series float) The series to rescale. // @param oldMin (simple float) The minimum value of the original signal's scale. // @param oldMax (simple float) The maximum value of the original signal's scale. // @param newMin (simple float) The minimum value of the new scale. // @param newMax (simple float) The maximum value of the new scale. // @returns (float) The rescaled value of the signal. rescale(series float src, simple float oldMin, simple float oldMax, simple float newMin, simple float newMax) => float result = newMin + (newMax - newMin) * (src - oldMin) / math.max(oldMax - oldMin, 10e-10) // @function Rescales a signal with an unknown scale (unbounded) using its historical low and high values. // @param src (series float) The series to rescale. // @param min (simple float) The minimum value of the rescaled series. // @param max (simple float)

The maximum value of the rescaled series. // @returns (float) The rescaled value of the signal. normalize(series float src, simple float min, simple float max) => var float historicMin = 10e10 var float historicMax = -10e10 historicMin := math.min(nz(src, historicMin), historicMin) historicMax := math.max(nz(src, historicMax), historicMax) float result = min + (max - min) * (src - historicMin) / math.max(historicMax - historicMin, 10e-10) // ——————— Plot normalized CCI cci = ta.cci(close, 20) plot(normalize(cci, 100, 300), "Normalized CCI", #2962FF) // Arbitrary and inexact equivalent of 100 and -100 levels rescaled to the 100/300 scale. band00 = hline(150, "Lower Band", color.new(#C0C0C0, 90), hline.style_solid) band01 = hline(250, "Upper Band", color.new(#C0C0C0, 90), hline.style_solid) fill(band01, band00, color.new(#21328F, 80), "Background") // ——————— Plot normalized volume in the same region as the rescaled RSI color volColor = close > open ? #26a69a : #ef5350 plot(normalize(volume, -100, 100), "Normalized volume", volColor, style = plot.style_columns, histbase = -100) hline(100, "", color.new(color.gray, 50), hline.style_dashed) hline(-100, "", color.new(color.gray, 50), hline.style_solid) // ——————— Plot rescaled RSI plot(rescale(ta.rsi(close, 14), 0, 100, -100, 100), "Rescaled RSI", #8E1599) hline(0, "RSI 50 level", color.new(color.gray, 70), hline.style_solid) // Precise equivalent of 70 and 30 levels rescaled to the -100/100 scale. band10 = hline(-40, "Lower Band", color.new(#9915FF, 80), hline.style_solid) band11 = hline(40, "Upper Band", color.new(#9915FF, 80), hline.style_solid) fill(band11, band10, color.new(#9915FF, 90), "Background") // ——————— Plot original values in Data Window plot(na, "═══════════════", display = display.data_window) plot(cci, "Original CCI", display = display.data_window) plot(volume, "Original volume", display = display.data_window) plot(ta.rsi(close, 14), "Original RSI", display = display.data_window)

image

How can I calculate my script’s run time?

Programmers can measure the time that a script takes to run and see detailed information about which parts of the code take longest in the Pine Profiler. See the section of the User Manual on Profiling and optimization for more information.

How can I save a value when an event occurs?

To save a value when an event occurs, use a *persistent variable*. Scripts declare persistent variables by using the var keyword. Such variables are initialized only once, at bar_index zero, instead of on each bar, and maintain the same value after that unless changed.

In the following example script, the var keyword allows the `priceAtCross` variable to maintain its value between bars until a crossover event occurs, when the script updates the variable with the current close price. The := reassignment operator ensures that the global variable `priceAtCross` is modified. Using the = assignment operator instead would create a new local variable that is inaccessible outside the if block. The new local variable would have the same name as the global variable, which is called *shadowing*. The compiler warns about shadow variables.

//@version=6 indicator("Save a value when an event occurs", "", true) float hiHi = ta.highest(high, 5)[1] var float priceAtCross = na if ta.crossover(close, hiHi) // When a crossover occurs, assign the current close price to `priceAtCross`. priceAtCross := close plot(hiHi) plot(priceAtCross, "Price At Cross", color.orange, 3, plot.style_circles)

How can I count touches of a specific level?

The most efficient way to count touches of a specific level is by tracking the series on each bar. A robust approach requires maintaining separate tallies for up and down bar touches and taking into account any gaps across the level. Using loops instead would be inefficient and impractical in this case.

The following example script records a value of 1 in a series whenever a touch occurs, and uses the math.sum() function to count these instances within the last `touchesLengthInput` bars. This script displays the median and touches on the chart using the `force_overlay` parameter of the `plot*()` functions, and displays the count in a separate pane.

image

//@version=6 indicator("Median Touches", "", overlay = false) int medianLengthInput = input.int(100, "Median calculation: Number of previous closes") int touchesLengthInput = input.int(50, "Number of previous bars to check for price touches") float median = ta.percentile_nearest_rank(close, medianLengthInput, 50) // Don"t count neutral touches when price doesn't move. bool barUp = close > open bool barDn = close < open // Bar touches median. bool medianTouch = high > median and low < median bool gapOverMedian = high[1] < median and low > median bool gapUnderMedian = low[1] > median and high < median // Record touches. int medianTouchUp = medianTouch and barUp or gapOverMedian ? 1 : 0 int medianTouchDn = medianTouch and barDn or gapUnderMedian ? 1 : 0 // Count touches over the last n bars. float touchesUp = math.sum(medianTouchUp, touchesLengthInput) float touchesDn = math.sum(medianTouchDn, touchesLengthInput) // ———————————— Plots // Markers plotchar(medianTouchUp, "medianTouchUp", "▲", location.belowbar, color.lime, force_overlay = true) plotchar(medianTouchDn, "medianTouchDn", "▼", location.abovebar, color.red, force_overlay = true) // Median plot(median, "Median", color.orange, force_overlay = true) // Base areas. plot(touchesUp, "Touches Up", color.green, style =

plot.style_columns) plot(-touchesDn, "Touches Dn", color.maroon, style = plot.style_columns) // Exceeding area. float minTouches = math.min(touchesUp, touchesDn) bool minTouchesIsUp = touchesUp < touchesDn basePlus = plot(minTouches, "Base Plus", display = display.none) hiPlus = plot(not minTouchesIsUp ? touchesUp : na, "High Plus", display = display.none) baseMinus = plot(-minTouches, "Base Plus", display = display.none) loMinus = plot(minTouchesIsUp ? -touchesDn : na, "Low Minus", display = display.none) fill(basePlus, hiPlus, color.lime) fill(baseMinus, loMinus, color.red)

How can I know if something is happening for the first time since the beginning of the day?

One way is to use the ta.barssince() function to check if the number of bars since the last occurrence of a condition, plus one, is greater than the number of bars since the beginning of the new day.

Another method is to use a *persistent state* to decide whether an *event* can happen. When the timeframe changes to a new day, the state is reset to allow the event. If the condition occurs while the state allows it, an event triggers. When the event triggers, the state is set so so as not to allow the event.

The following example script shows both methods.

image

//@version=6 indicator("First time today example", "", true) bool isUpCandle = close > open // ——————— Method 1. int barsSincePreviousUpCandle = ta.barssince(isUpCandle[1]) int barsSinceStartOfDay = ta.barssince(timeframe.change("1D")) - 1 bool previousUpCandleWasNotToday = barsSincePreviousUpCandle > barsSinceStartOfDay bool isFirstToday1 = isUpCandle and previousUpCandleWasNotToday plotchar(isFirstToday1, "isFirstToday1", "•", location.top, color = color.silver, size = size.normal) plot(barsSinceStartOfDay, "barsSinceStartOfDay", display=display.data_window) // ——————— Method 2. var bool

```
hadUpCandleToday = false // This is a persistent state.
bool isFirstToday2 = false // This is a one-off event.
if timeframe.change("1D") // When the day begins..
    hadUpCandleToday := false // we have not yet had an up candle today, so reset the state.
if isUpCandle and not hadUpCandleToday // If this is the first up candle today..
    hadUpCandleToday := true // set the persistent state
    isFirstToday2 := true // and update the event.
plotchar(isFirstToday2, "isFirstToday2", "•", location.top, color = color.yellow, size = size.small)
```

How can I optimize Pine Script code?

Optimizing Pine Script code can make scripts run faster and use less memory. For large or complex scripts, optimization can avoid scripts reaching the computational limits.

The Pine Profiler analyzes all significant code in a script and displays how long each line or block takes to run. Before optimizing code, run the Pine Profiler to identify which parts of the code to optimize first. The Pine Profiler section of the User Guide contains an extensive discussion of how to optimize code. In addition, consider the following tips:

- Use strategy scripts only to model trades. Otherwise, use indicator scripts, which are faster.
- Become familiar with the Pine execution model and time series to structure code effectively.
- Declare variables with the var keyword when initialization involves time-consuming operations like complex functions, arrays, objects, or string manipulations.
- Keep operations on strings to a necessary minimum, because they can be more resource-intensive than operations on other types.
- Using built-in functions is usually faster than writing custom functions that do the same thing. Sometimes, alternative logic can be more efficient than using standard functions. For example, use a persistent variable when an event occurs, to avoid using ta.valuewhen(), as described in the FAQ entry How can I save a value when an event occurs?. Or save the bar_index when a

condition occurs to avoid using ta.barssince(), as described in the FAQ entry How to remember the last time a condition occurred?.

How can I access a stock's financial information?

In Pine, the request.financial() function can directly request financial data.

On the chart, users can open financial indicators in the "Financials" section of the "Indicators, Metrics & Strategies" window.

How can I find the maximum value in a set of events?

Finding the maximum value of a variable that has a meaningful value *on every bar*, such as the high or low in price, is simple, using the ta.highest() function.

However, if the values do not occur on every bar, we must instead store each value when it occurs and then find the maximum. The most flexible way to do this is by using an array.

The following example script stores pivot highs in a fixed-length array. The array is managed as a queue: the script adds new pivots to the end, and removes the oldest element from the array. To identify the highest value among the stored pivots, we use the array.max() function and plot this maximum value on the chart. Additionally, we place markers on the chart to indicate when the pivots are detected, and the bars where the pivots occurred. By definition, these points are not the same, because a pivot is only confirmed after a certain number of bars have elapsed.

//@version=6 indicator("Max pivot demo", "", true) // Create inputs to specify the pivot legs and the number of last pivots to keep to compare. int pivotLengthInput = input.int(5, "Pivot length", minval = 1) int numPivotsInput = input.int(3, "Number of pivots to check") // Initialize an array with a size based on the number of recent pivots to

evaluate. var array<float> pivotsArray = array.new<float>(numPivotsInput) // Find the pivot value and set up a condition to verify if a value has been found. float ph = ta.pivothigh(pivotLengthInput, pivotLengthInput) bool newPH = not na(ph) // When a new pivot is found, add it to the array and discard the oldest value. if newPH pivotsArray.push(ph) pivotsArray.shift() // Display the max value from the array on the chart, along with markers indicating the positions and detection times of the pivot highs. plot(pivotsArray.max()) plotchar(newPH, "newPH", "•", location.abovebar, offset = - pivotLengthInput) plotchar(newPH, "newPH", "▲", location.top)

How can I display plot values in the chart’s scale?

To display the names and values of plots from an indicator in the chart’s scale, right-click on the chart to open the chart “Settings” menu. In the “Scales and lines” tab, select “Name” and “Value” from the “Indicators and financials” drop-down menu.

image

How can I reset a sum on a condition?

To sum a series of values, initialize a persistent variable by using the var keyword to the track the sum. Then use a logical test to reset the values when a condition occurs.

In the following example script, we initialize a persistent variable called `cumulativeVolume` to track the sum of the volume. Then we reset it to zero on a Moving Average Convergence/Divergence (MACD) cross up or down.

We plot the cumulative volume on the chart, as well as arrows to show the MACD crosses.

image

//@version=6 indicator("Reset sum on condition example", overlay = false) const color TEAL = color.new(color.teal, 50) const color RED = color.new(color.red, 50) [macdLine, signalLine, _] = ta.macd(close, 12, 26, 9) bool crossUp = ta.crossover(macdLine, signalLine) bool crossDn = ta.crossunder(macdLine, signalLine) bool doReset = crossUp or crossDn var float cumulativeVolume = na cumulativeVolume += volume // On every bar, we sum the volume. cumulativeVolume := doReset ? 0. : cumulativeVolume // But when we get a cross, we reset it to zero. plot(cumulativeVolume, "Cumulative volume", close >= open ? TEAL : RED, 1, plot.style_columns) plotshape(crossUp, "crossDn", shape.arrowup, location.top, color.lime) plotshape(crossDn, "crossUp", shape.arrowdown, location.top, color.fuchsia)

Note that:

- In the ta.macd() function call, we only require two of the three values returned in the tuple. To avoid unnecessary variable declarations, we assign the third tuple value to an underscore. Here, the underscore acts like a dummy variable.

How can I accumulate a value for two exclusive states?

Consider a simple indicator defined by two exclusive states: *buy* and *sell*. The indicator cannot be in both *buy* and *sell* states simultaneously. In the *buy* state, the script accumulates the volume of shares being traded. In the *sell* state, the accumulation of volume begins again from zero.

There are different ways to code this kind of logic. See the FAQ entry "How can I alternate conditions" for an example of using an enum to manage two exclusive states. The following example script uses two boolean variables to do the same thing.

Additionally, this script demonstrates the concept of *events* and *states*. An event is a condition that occurs on one or more arbitrary bars. A state is a condition that persists over time. Typically, programmers use

events to turn states on and off. In turn, states can allow or prevent other processing.

The script plots arrows for events, which are based on rising or falling values of the close price. These events determine which of the two exclusive states is active; the script colors the background according to the current state. The script accumulates bullish and bearish volume only in the corresponding bullish or bearish state, displaying it in a Weis Wave fashion.

image

//@version=6 indicator("Cumulative volume", "") bool upEvent = ta.rising(close, 2) bool dnEvent = ta.falling(close, 2) var bool upState = false, var bool dnState = false // When the right event occurs, turn the state on; when a counter-event occurs, turn it off; otherwise, persist it. upState := upEvent ? true : dnEvent ? false : upState dnState := upEvent ? false : dnEvent ? true : dnState var float volUp = na, var float volDn = na if upState // For every bar that we are in the up state, volUp += volume // sum the up volume. if dnState volDn += volume if upEvent // If we change state to up, volDn := 0 // reset the down volume. if dnEvent volUp := 0 plot(+volUp, "Up Volume", color.green, 4, plot.style_columns) plot(-volDn, "Dn Volume", color.maroon, 4, plot.style_columns) plotchar(upEvent, "Up Event", "▲", location.bottom, color.green, size = size.tiny) plotchar(dnEvent, "Dn Event", "▼", location.top, color.maroon, size = size.tiny) bgcolor(upState ? color.new(color.green, 90) : dnState ? color.new(color.red, 90) : na)

Note that:

- Equivalent logic using ternary conditions is smaller and potentially more efficient, but not as easy to read, extend, or debug. This more verbose logic illustrates the concepts of events and states, which can apply to many types of scripting problems. This logic is an extension of the on-off switch in the FAQ entry “How can I implement an on/off switch?“.
- When using states, it is important to make the conditions for resetting states explicit, to avoid unforeseen problems.

- Displaying all events and states during script development, either on the chart or in the Data Window, helps debugging.

How can I organize my script's inputs in the Settings/Inputs tab?

A script's plots and inputs constitute its user interface. The following example script uses the following techniques to organize inputs for greater clarity:

- **Grouping inputs:** Create a section header for a group of inputs by using the `group` parameter in the input() functions. Use constants for group names to simplify any potential name changes.
- **Visual boundaries:** Use ASCII characters to create separators, establishing visual boundaries for distinct group sections. For continuous separator lines, reference group headers 1 and 2 in our script below, which use ASCII characters 205 or 196. Conversely, the dash (ASCII 45) and Em dash (ASCII 151), shown in group headers 3 and 4, do not join continuously, resulting in a less visually appealing distinction. Note that Unicode characters might display differently across different machines and browsers, potentially altering their appearance or spacing for various users.
- **Indentation of sub-sections:** For a hierarchical representation, use Unicode whitespace characters to indent input sub-sections. Group 3 in our script uses the Em space ( ) 8195 (0x2003) to give a tab-like spacing.
- **Vertical alignment of inlined inputs:** In our script, Group 1 shows how vertical alignment is difficult when inline inputs have varied `title` lengths. To counteract this misalignment, Group 2 uses the Unicode EN space ( ): 8194 (0x2002) for padding, since regular spaces are stripped from the label. For precise alignment, use different quantities and types of Unicode spaces. See here for a list of Unicode spaces of different widths. Note that, much like the separator characters, the rendering of these spaces might differ across browsers and machines.

- **Placing inputs on one line:** Add multiple related inputs into a single line using the `inline` parameter. Group 4 in our script adds the title argument to just the first input and skips it for the others.

```
//@version=6 indicator("Inputs", overlay = true) // Defining options strings improves script readability. // It also enables the creation of boolean variables by comparing these constants with user input strings in a single line of code. string EQ1 = "On" string EQ2 = "Off" // The `GRP*` strings used for group headers demonstrate using ASCII characters to create a visual boundary, // making it easier for users to differentiate between different sections in the menu. // Group 1 demonstrates inline inputs that do not align vertically in the menu. string GRP1 = "═══════════ Settings ═══════════" // ASCII 205 float ao1SrcInput = input.source(close, "AO source", inline = "11", group = GRP1) int ao1LenInput = input.int(14, "Length", inline = "11", group = GRP1) float long1SrcInput = input.source(close, "Signal source", inline = "12", group = GRP1) int long1LenInput = input.int(3, "Length", inline = "12", group = GRP1) // In Group 2, the title of `ao2SrcInput` is padded with three Unicode EN spaces (U+2002) to compensate for the misalignment. string GRP2 = "─────────── Settings ───────────" // ASCII 196 float ao2SrcInput = input.source(close, "AO source   ", inline = "21", group = GRP2) int ao2LenInput = input.int(14, "Length", inline = "21", group = GRP2) float long2SrcInput = input.source(close, "Signal source", inline = "22", group = GRP2) int long2LenInput = input.int(3, "Length", inline = "22", group = GRP2) // This configuration uses Unicode white space characters to indent input sub-sections. We use Em space ( ): 8195 (0x2003). string GRP3 = "——————————————— Settings ———————————————" // ASCII 151 (Em dash) float level1Input = input.float(65., "First level", group = GRP3) float level2Input = input.float(65., "  Second Level", group = GRP3) bool level3Input = input.string(EQ1, "  Checkbox equivalent", group = GRP3, options = [EQ1, EQ2]) == EQ1 float level4Input = input.float(65., "Widest Legend                    ", group = GRP3) // These options demonstrate the use of the `inline` parameter to create structured blocks of inputs that are relevant to one another. string GRP4 = "---------- Settings ----------" // ASCII 45 (dash) bool showMa1Input = input(true, "MA №1", inline = "1", group =
```

GRP4) string ma1TypeInput = input.string("SMA", "", inline = "1", group = GRP4, options = ["SMA", "EMA", "SMMA (RMA)", "WMA", "VWMA"]) float ma1SourceInput = input(close, "", inline = "1", group = GRP4) int ma1LengthInput = input.int(20, "", inline = "1", group = GRP4, minval = 1) color ma1ColorInput = input(#f6c309, "", inline = "1", group = GRP4) bool showMa2Input = input(true, "MA №2", inline = "2", group = GRP4) string ma2TypeInput = input.string("SMA", "", inline = "2", group = GRP4, options = ["SMA", "EMA", "SMMA (RMA)", "WMA", "VWMA"]) float ma2SourceInput = input(close, "", inline = "2", group = GRP4) int ma2LengthInput = input.int(50, "", inline = "2", group = GRP4, minval = 1) color ma2ColorInput = input(#fb9800, "", inline = "2", group = GRP4) bool showMa3Input = input(true, "MA №3", inline = "3", group = GRP4) string ma3TypeInput = input.string("SMA", "", inline = "3", group = GRP4, options = ["SMA", "EMA", "SMMA (RMA)", "WMA", "VWMA"]) float ma3SourceInput = input(close, "", inline = "3", group = GRP4) int ma3LengthInput = input.int(100, "", inline = "3", group = GRP4, minval = 1) color ma3ColorInput = input(#fb6500, "", inline = "3", group = GRP4) // @function Calculates various types of moving averages for the `source` based on the specified `maType`. // @param series (series float) Series of values to process. // @param length (simple int) Number of bars (length). // @param maType (simple string) The type of moving average to calculate. // Options are "SMA", "EMA", "SMMA (RMA)", "WMA", and "VWMA". // @returns (float) The moving average of the `source` for `length` bars back. ma(series float source, simple int length, simple string maType) => switch maType "SMA" => ta.sma(source, length) "EMA" => ta.ema(source, length) "SMMA (RMA)" => ta.rma(source, length) "WMA" => ta.wma(source, length) "VWMA" => ta.vwma(source, length) => na // Calculate the moving averages with the user-defined settings. float ma1 = ma(ma1SourceInput, ma1LengthInput, ma1TypeInput) float ma2 = ma(ma2SourceInput, ma2LengthInput, ma2TypeInput) float ma3 = ma(ma3SourceInput, ma3LengthInput, ma3TypeInput) // Plot the moving averages, if each checkbox is enabled. plot(showMa1Input ? ma1 : na, "MA №1", ma1ColorInput) plot(showMa2Input ? ma2 : na, "MA №2", ma2ColorInput) plot(showMa3Input ? ma3 : na, "MA №3", ma3ColorInput)

Tips:

- Order the inputs to prioritize user convenience rather than to reflect the order used in the script's calculations.
- Never use two checkboxes for mutually exclusive selections. Use dropdown menus instead.
- Remember that dropdown menus can accommodate long strings.
- Provide adequate minimum and maximum values for numeric values, selecting the proper float or int type.
- Customize step values based on the specific needs of each input.
- Because checkboxes cannot be indented, use the input() function's `options` parameter to create dropdown selections so that the sections appear more organized compared to using checkboxes.
- Observe how the `level3Input` is calculated as a boolean variable by comparing the input with the `EQ1` "ON" constant. This method provides a visually appealing indented on-off switch in the menu without adding complexity to the code.
- For a consistent visual appearance, vertically center the separator titles across all inputs. Due to the proportional spacing of the font, achieving this might require some trial and error.
- To ensure that separators align just slightly to the left of the furthest edge of dropdowns, begin with the longest input title, because it sets the width of the window.
- To avoid adjusting separators if the longest input title is shorter than initially anticipated, extend its length using Unicode white space. Refer to the code example for input `level4Input` for a demonstration.

Previous

Next

Times, dates, and sessions

Times, dates, and sessions

How can I get the time of the first bar in the dataset?

The following example script initializes a variable using the var keyword on the first bar and then never updates it again. The variable stores the value of the time built-in, which represents the time of the bar open in UNIX format (milliseconds since 00:00:00 UTC on 1 January 1970).

//@version=6 indicator('Time at first bar') // Capture the time of the first bar in the dataset. var int t = time plot(t)

How can I convert a time to a date-time string?

The built-in function str.format_time() translates a UNIX timestamp into a string representation of date and time. In the following example script, we provide four distinct methods to obtain the date-time string, two of which offset the date and time into the future. To improve readability, we use a custom function, `timeToString()`, which calls str.format_time() and applies a consistent format, instead of specifying the format for every function call.

The format `"YYYY.MM.dd @ HH:mm:ss"` is similar to the ISO 8601 date and time representation, but with periods to separate parts of the date, and an `@` symbol between the date and time. For more formatting customization options, consult the str.format_time() documentation.

image

//@version=6 indicator("Time to string example") // Function converts a timestamp to a formatted string. timeToString(t) => str.format_time(t, format = "YYYY.MM.dd @ HH:mm:ss") var int t = time // Capture the time of the first bar in the dataset. var table tbl = table.new(position.middle_right, 1, 1) // Create table on the first bar only. // On the first bar, build the table cell. if barstate.isfirst table.cell(tbl, 0, 0, "", bgcolor = color.yellow, text_color = color.black, text_halign = text.align_left, text_font_family = font.family_monospace) // On the last bar, build display text and populate the table. else if barstate.islast string txt = str.format("Date/time at bar_index = 0 {0} \nCurrent Date/time {1} \nDate/time 4 days from current time {2} \nDate/time at beginning of last bar {3} \nDate/time 4 days after last bar''s start: {4}", timeToString(t), timeToString(timenow), timeToString(timestamp(year(timenow), month(timenow), dayofmonth(timenow) + 4, hour(timenow), minute(timenow), second(timenow))), timeToString(time), timeToString(timestamp(year, month, dayofmonth + 4))) table.cell_set_text(tbl, 0, 0, txt)

How can I find out how many days are in the current month?

The following example script calculates the number of days in the current month, including adjustments for leap years. By default, the `daysPerMonth()` function uses the current year and month, but any year or month number can be passed as arguments:

image

//@version=6 indicator("Days in month") // @function Calculates the number of days in a specified month, accounting for leap years. // @param yearNumber (int) The year of the `monthNumber` month. Optional. Default is the current year. // @param monthNumber (int, optional) The month for which to find the number of days. Optional. Default is the current month. // @returns (int) The number of days in the `monthNumber` month of the `yearNumber` year.

daysPerMonth(int yearNumber = year, int monthNumber = month) => bool leapYear = (yearNumber % 4 == 0 and yearNumber % 100 != 0) or (yearNumber % 400 == 0) int result = switch monthNumber == 2 => leapYear ? 29 : 28 => 31 - (monthNumber - 1) % 7 % 2 plot(daysPerMonth())

How can I detect the chart's last day?

Scripts can detect the last day on a chart in different ways. However, because the last_bar_time and timenow variables constantly update to mirror the time of the newest bar, each subsequent realtime bar is *also* interpreted as if it is the chart's last day.

Using `timeframe.change`

This example script checks for all of these conditions:

- The daily timeframe begins a new day.
- The difference between the time of the last chart bar (retrieved using last_bar_time) and the current bar's time (from time) is less than the time in one day.

The script colors the background red on the last day.

//@version=6 indicator("Last day example", overlay = true) var bool isLastDay = false if timeframe.change("1D") and last_bar_time - time < timeframe.in_seconds("1D") * 1000 isLastDay := true bgcolor(isLastDay ? color.new(color.red, 90) : na)

Using `timenow`

This script uses the timenow variable to retrieve the current time — not the time of the bar on which the script is executing, but the time of the latest chart update. We pass this time as input to the built-in functions year(), month(), and dayofmonth() to calculate the year, month, and day of the month for that time. We compare these calculated values with the built-in variables year, month, and dayofmonth, which store the year, month, and day of the month of the current chart bar. When the year, day and month of the latest chart update match the values of

the current bar, then the script is executing on bars from the most recent day.

//@version=6 indicator("Detect today", overlay = true) isToday() => int currentYear = year(timenow) int currentMonth = month(timenow) int currentDay = dayofmonth(timenow) bool result = year == currentYear and month == currentMonth and dayofmonth == currentDay bgcolor(isToday() ? color.new(color.red, 90) : na)

How can I get bar’s trading date on symbols with an overnight session?

Some symbols trade overnight, which means that today’s trading day actually started yesterday, or even several days ago. Many futures and Forex pairs trade overnight. Trading hours are defined by the exchange. On timeframes of one day and above, the chart visually adjusts the dates of these bars to ensure the date corresponds to the bar’s trading day; on intraday timeframes the date and time are shown as they are.

In Pine Script, the opening time of a bar is represented by the variable time, which also returns the date and time without adjustment. Variables that return date-related values, like dayofmonth, weekofyear, etc., are calculated based on this opening time, which can lead to unexpected behavior on overnight symbols.

The example script below outputs the value of the built-in dayofmonth variable onto a chart as a label. The following screenshot shows the output of this script on two symbols: one stock and one Forex. We added a vertical line on the chart using the drawing tools to highlight the bar that represents May 22nd.

The stock symbol “NASDAQ:AAPL ” displays on the upper chart. Its trading hours open and close within a single day with no overnight session. Therefore, the label displays the same day as the vertical line.

In contrast, on the lower chart, which displays the “FX:EURUSD ” symbol, the label shows 21 on the bar marked May 22nd. This is because the trading for May 22 actually starts on May 21, at 17:00 in

the exchange’s time zone, and the dayofmonth variable uses this time to determine the day of the month for this bar.

image

//@version=6 indicator("Day number", overlay = true) currTradingDay = dayofmonth labelStr = "`time`: " + str.format_time(time, "dd MMM HH:mm 'ET'") labelStr += "\n`dayofmonth`: " + str.tostring(currTradingDay) label.new(bar_index, high, labelStr)

There is a way to avoid this issue. Each date-related variable like dayofmonth has a corresponding function with the same name, e.g., dayofmonth(). The function takes two parameters, `time` and `timezone`, which together allow you to specify the exact timestamp to convert to a date.

In addition, the time_tradingday variable returns the timestamp of 00:00 UTC of the *trading day* the bar belongs to, regardless of the bar’s actual opening time. You can pass this timestamp to the dayofmonth() function along with the `"UTC"` time zone to extract the date from the trading day of the bar, instead of its opening time. Below, we update our example script to use this method:

image

//@version=6 indicator("Day number", overlay = true) currTradingDay = dayofmonth(time_tradingday, "UTC") labelStr = "`time_tradingday`: " + str.format_time(time_tradingday, "dd MMM HH:mm 'UTC'", "UTC") labelStr += "\n`dayofmonth(time_tradingday)`: " + str.tostring(currTradingDay) label.new(bar_index, high, labelStr)

Note that:

- Here, we add `"UTC"` as the time zone for the dayofmonth() calculation, because time_tradingday always returns the trading day as 00:00 in the UTC time zone. The variable returns 00:00 because its only purpose is to specify a year, month, and day, and its hour and minute components carry no specific meaning.

How can I plot a value starting n months or years back?

The timestamp() function can accept negative argument values and convert them to accurate dates. For instance, a negative month value deducts the corresponding number of months from the outcome.

Our example script calculates the date three months prior to the current time, using the timestamp() function to create a timestamp that is the configured amount of time prior to the time now, as measured using timenow. As the script executes on each bar, it compares the bar time with the previously calculated timestamp. When the time of a bar matches our specified target date, we assign the high of that bar as our value to plot and color the background red for demonstration.

image

//@version=6 indicator("Plot value starting n months/years back", overlay = true) int monthsBackInput = input.int(3, minval = 0) int yearsBackInput = input.int(0, minval = 0) bool calcFromNowInput = input(false, "Calculate from current date/time instead of first of the month") bool isTargetDate = time >= timestamp(year(timenow) - yearsBackInput, month(timenow) - monthsBackInput, calcFromNowInput ? dayofmonth(timenow) : 1, calcFromNowInput ? hour(timenow) : 0, calcFromNowInput ? minute(timenow) : 0, calcFromNowInput ? second(timenow) : 0) bool isBeginMonth = not isTargetDate[1] and isTargetDate var float valueToPlot = na if isBeginMonth valueToPlot := high plot(valueToPlot) bgcolor(isBeginMonth ? color.new(color.red, 80) : na)

How can I track highs and lows for a specific timeframe?

The following example script tracks the high and low values of a timeframe that the user selects. Our script avoids using request.security() calls, which are resource-intensive.

image

//@version=6 indicator("Periodic high/low example", overlay = true) // Inputs bool showHiInput = input.bool(true, "Show highs") bool showLoInput = input.bool(true, "Show lows") string periodInput = input.timeframe("1D", "Period after which high/low is reset") // Declare with `var` to retain values bar to bar. var float hi = na var float lo = na // When a new period begins, reset hi/lo; otherwise, trail them. bool isNewPeriod = timeframe.change(periodInput) hi := isNewPeriod ? high : math.max(high, hi) lo := isNewPeriod ? low : math.min(low, lo) // Plot the hi, lo, and an invisible mid value for area fill. p1 = plot(hi, "Highs", isNewPeriod ? na : color.new(color.lime, 60), display = showHiInput ? display.all : display.none) p2 = plot(lo, "Lows", isNewPeriod ? na : color.new(color.fuchsia, 60), display = showLoInput ? display.all : display.none) p3 = plot(hl2, editable = false, display = display.none) // Create fills between the current mid price and the highest and lowest price. fill(p1, p3, color = isNewPeriod ? na : color.new(color.lime, 90)) fill(p2, p3, color = isNewPeriod ? na : color.new(color.fuchsia, 90)) bgcolor(isNewPeriod ? color.new(color.gray, 90) : na) // Highlight the background when a new period begins.

Note that:

- We detect the beginning of a new timeframe by passing the period input to the timeframe.change() function.
- The script resets the `hi` and `lo` variables at the beginning of a new timeframe and then trails `hi` up and `lo` down.
- We change the color of plots to na every time a new timeframe begins, to visually distinguish between periods.

How can I track the highs and lows within a specific session or time of day?

To find if a time is within a specific session, pass a session string to the time() function, which retrieves the UNIX time of the current bar based on a given timeframe and session. If the time point falls outside the session, the function returns na.

In the following example script, on the first bar that the time is in session, we set the `hi` and `lo` variables to the source inputs provided by the user (high and low by default). During the session we trail the `hi` value up and the `lo` value down. The script also highlights the background red when a bar's time falls outside the session time.

image

```
//@version=6 indicator("Session high/low", overlay = true) const string DEFAULT = "Default" const string EQ1 = "On" const string EQ2 = "Off" // Inputs bool showHiInput = input.string(EQ1, "Show highs", options = [EQ1, EQ2]) == EQ1 bool showLoInput = input.string(EQ1, "Show lows", options = [EQ1, EQ2]) == EQ1 float srcHiInput = input.source(high, "Source for Highs") float srcLoInput = input.source(low, "Source for Lows") string sessionInput = input.session("1200-1500", "Allowed hours") string timezoneInput = input.string(DEFAULT, "Time zone", options = [DEFAULT, "GMT-12", "GMT-11", "GMT-10", "GMT-9", "GMT-8", "GMT-7", "GMT-6", "GMT-5", "GMT-4", "GMT-3", "GMT-2", "GMT-1", "GMT-0", "GMT+1", "GMT+2", "GMT+3", "GMT+4", "GMT+5", "GMT+6", "GMT+7", "GMT+8", "GMT+9", "GMT+10", "GMT+11", "GMT+12", "GMT+13", "GMT+14"]) // Check to see if we are in allowed hours using session info. int timeInSession = time(timeframe.period, sessionInput, timezoneInput == DEFAULT ? syminfo.timezone : timezoneInput) bool timeIsAllowed = not na(timeInSession) var float hi = na var float lo = na if timeIsAllowed // We are entering allowed hours; reset hi/lo. if not timeIsAllowed[1] hi := srcHiInput lo := srcLoInput else // We are in allowed hours; track hi/lo. hi := math.max(srcHiInput, hi) lo := math.min(srcLoInput, lo) // Plot hi/lo within allowed hours. plot(showHiInput and timeIsAllowed ? hi : na, "Highs", color.lime, 3, plot.style_circles) plot(showLoInput and timeIsAllowed ? lo : na, "Lows", color.fuchsia, 3, plot.style_circles) bgcolor(not timeIsAllowed ? color.new(color.red, 90) : na)
```

Note that:

- We use the 3-parameter version of the time() function to ascertain whether we are within user-specified hours and time zone for tracking highs and lows.

- We pass a session input and an optional time zone to the time() function.

How can I track highs and lows for periods within a bar?

Historical bars contain one set of OHLC data for each bar. To identify values and times that start or end *within* a chart bar, intrabar inspection is necessary, using data from lower timeframes.

For instance, on an hourly chart, the market open might occur mid-bar, making it hard to determine pre-market highs and lows. Pulling data from a lower timeframe, such as a 15-minute chart, provides a clearer view of that hour's events.

The following example script calculates pre-market highs and lows from 15-minute chart data.

- First, we define a function called `hiLoBetweenTime()` to track the highs and lows within a session.
- Next, we modify the symbol's ticker using ticker.modify() to include extended session data, ensuring that the script identifies premarket highs and lows even when the chart uses regular trading hours.
- Finally, we pass the `hiLoBetweenTime()` function as the `expression` of the request.security() function, with the modified ticker as the `symbol`. The request.security() function evaluates the `hiLoBetweenTime()` function *within the context of the lower timeframe* to calculate the session's highs and lows.

image

//@version=6 indicator("Pre-market high/low", overlay = true) // Inputs string timeAllowedInput = input.session("0700-0930", "Allowed Hours") string lowerTfInput = input.timeframe("15", "Intrabar Resolution") string timezoneInput = input.string("Default", "Time zone", options = ["Default", "GMT-12", "GMT-11", "GMT-10", "GMT-9", "GMT-8", "GMT-7", "GMT-6", "GMT-5", "GMT-4", "GMT-3", "GMT-2", "GMT-1", "GMT-0", "GMT+1", "GMT+2", "GMT+3", "GMT+4", "GMT+5", "GMT+6",

"GMT+7", "GMT+8", "GMT+9", "GMT+10", "GMT+11", "GMT+12", "GMT+13", "GMT+14"]) // @function Tracks the highest high and lowest low between specified session times. // @param sess (simple string) Session duration in the format "start time - end time". Example: "0930-1600" // @param timeZone (simple string) Time zone of the session in "GMT-0" format. Optional. Default is the symbol's time zone. // @returns ([float, float]) A tuple of the highest high and lowest low between the specified session times. hiLoBetweenTime(simple string sess, simple string timeZone = "Default") => var float hi = na, var float lo = na // Check to see if we are in allowed hours using session and time zone information. bool inSession = not na(time("", sess, timeZone == "Default" ? syminfo.timezone : timeZone)) if inSession if not inSession[1] // We are entering allowed hours; reset hi/lo. hi := high, lo := low else // We are in allowed hours; track high and low. hi := math.max(hi, high), lo := math.min(lo, low) [hi, lo] // Request data from lower timeframe using the `hiLoBetweenTime()` function. [highAtTime, lowAtTime] = request.security(ticker.modify(syminfo.tickerid, session.extended), lowerTfInput, hiLoBetweenTime(timeAllowedInput, timezoneInput)) // Plot the most recent value. plot(highAtTime, "High", color.green) plot(lowAtTime, "Low", color.red) // Raise error if lower tf is the same or greater than chart's tf. if timeframe.in_seconds() <= timeframe.in_seconds(lowerTfInput) runtime.error("The lower timeframe for intrabar inspection must be lower than the chart's timeframe.")

Note that:

- The script raises an error using runtime.error() if the chosen lower timeframe for intrabar inspection is not shorter than the main chart’s timeframe. Including error-checking such as this when working with timeframes makes the script more robust.

How can I detect a specific date and time?

The simplest way to detect a specific date and time is to check the time variable, which returns the timestamp of the current bar’s open. If

this matches a certain timestamp, then the script has detected that specific date and time.

The following example script colors the background green when the chart time exactly matches a date and time that the user inputs.

//@version=6 indicator("Exact date/time detector", overlay = true) bgcolor(time == input.time(timestamp("2025-03-24 10:00"), "Target Date/Time") ? color.new(color.green, 90) : na)

However, matching an exact date and time is not very useful. If the target time does not coincide with the opening of a candle on the chart timeframe, then that exact time never matches. For example, if the user specifies “2025-03-24 10:01” as the target time for the script above, then it never matches on any timeframe above one minute.

The following script behaves more intuitively, by coloring the background if a target time input by the user falls anywhere within a particular chart bar. The script colors one and only one bar even if the time does not match the open of a bar. The days, hours, and minutes are all tested separately to see if they match the target time, and these boolean conditions must all be true in order for the background to be colored. If the chart timeframe is too high, we set the corresponding lower time conditions to true so that they do not prevent the overall condition from firing. For example, if the timeframe is one day or above, we bypass the `isTargetHour` condition by setting it to true without evaluating the number of hours.

image

//@version=6 indicator("Date/Time detector", overlay = true) int targetInput = input.time(defval = timestamp("2025-03-24 10:00"), title = "Target Date/Time", confirm = true) bool isTargetYear = year == year(targetInput) bool isTargetMonth = month == month(targetInput) int targetDay = dayofmonth(targetInput) int dayOpen = dayofmonth // The built-in `dayofmonth` variable holds the day of the month of the bar open. int dayClose = dayofmonth(time_close) bool day_isBetweenOpenAndClose = (targetDay >= dayOpen) and (targetDay < dayClose) bool isTargetDay = timeframe.ismonthly or targetDay == dayofmonth or day_isBetweenOpenAndClose bool

```
TF_isOneHourOrLess = timeframe.in_seconds() <= (60 * 60)
bool TF_isAboveOneHour = timeframe.in_seconds() > (60 * 60)
bool TF_isAboveOrEqualToOneHour = timeframe.in_seconds() >= (60 * 60)
bool TF_isDayOrAbove = timeframe.isdwm
int targetHour = hour(targetInput)
int hourOpen = hour // The built-in `hour` variable holds the hour of the bar open.
int hourClose = hour(time_close)
bool hour_isBetweenOpenAndClose = (targetHour >= hourOpen) and (targetHour < hourClose)
bool isTargetHour = TF_isDayOrAbove or (TF_isOneHourOrLess and (targetHour == hourOpen)) or (TF_isAboveOneHour and hour_isBetweenOpenAndClose)
int minuteOpen = minute // The built-in `minute` variable holds the minute of the bar open.
int minuteClose = minute(time_close)
int targetMinute = minute(targetInput)
bool minute_isBetweenOpenAndClose = (targetMinute >= minuteOpen) and (targetMinute < minuteClose)
bool isTargetMinute = TF_isAboveOrEqualToOneHour or minute_isBetweenOpenAndClose
bool isTargetTime = isTargetYear and isTargetMonth and isTargetDay and isTargetHour and isTargetMinute
bgcolor(isTargetTime ? color.new(color.green, 70) : na)
```

Note that:

- The script does not work on timeframes less than one minute, because the time input does not use seconds. The script could be extended to work on timeframes of one month and higher.
- We use `confirm = true` for the time input, which prompts the script user to select a date and time when the script first loads on the chart. The time selected in this way is always equal to the time of the selected bar's open, but the user can change it in the "Settings" window.
- We use the built-in variables year, month, dayofmonth, hour, and minute, which store the time of the current bar's open.
- We use the built-in functions year(), month(), dayofmonth(), hour(), and minute() to retrieve the time of the current bar's close, and to evaluate the time that is input by the user.

How can I know the date when the highest value was found?

To determine the date on which the highest value occurred, the following example script uses the ta.highest() function to find the highest value within a certain lookback period, and the ta.highestbars() function to find how many bars back this value occurred. If the highest point occurs on the current bar, the number of bars back, or *offset*, is zero. In our example script, when this condition is met, we update labels showing the time at the respective highest or lowest price.

image

//@version=6 indicator("Date of High/Low", overlay = true) int lengthInput = input.int(20) // Create labels. They do not display until the na values for the time and price are updated with new values. var label hiLabel = label.new(na, na, "", color = na, textcolor = color.lime) var label loLabel = label.new(na, na, "", color = na, textcolor = color.fuchsia) // Find the highest and lowest values over the input lookback and the bars on which they occurred. float hiValue = ta.highest(lengthInput), float loValue = ta.lowest(lengthInput) float hiOffset = ta.highestbars(lengthInput), float loOffset = ta.lowestbars(lengthInput) // If the high or low occur on the current bar, update the label and time variables. if hiOffset == 0 label.set_xy(hiLabel, bar_index, hiValue) label.set_text(hiLabel, str.format_time(time, "YYYY.MM.dd @ HH:mm:ss")) if loOffset == 0 label.set_xy(loLabel, bar_index, loValue) label.set_text(loLabel, str.format_time(time, "YYYY.MM.dd @ HH:mm:ss")) // Plot the highest and lowest values. plot(hiValue, "High", color.lime) plot(loValue, "Low", color.fuchsia)

Note that:

- We declare the high and low labels with var but supply na values for the x and y coordinates, so that the labels do not appear until these parameters are defined.
- When the bar offset functions return zero, we modify the respective label to display the bar time.

- The script plots the highest and lowest level for visual reference.

How can I detect bars opening at a specific hour?

The following example script demonstrates three methods for detecting bars that open at 18:00 hours:

Check the current bar's time

We check whether the built-in hour variable is equal to eighteen and the minute variable is equal to zero.

Detect the start of the session

Using the time() function with a session string returns the time of the current bar if it is within the session time. If it's outside the session time, the function returns na. We determine the first bar that falls within the session by verifying that the current returned time is not na but was na one bar earlier.

Check current time against a timestamp

We generate a timestamp using the timestamp() function by specifying 18 hours and 0 minutes, as well as the current day, month, and year. We then verify whether the bar's time matches this timestamp.

//@version=6 indicator("18:00 hours", overlay = true) int timeInSession = time(timeframe = timeframe.period, session = "1800-1900") int timestamp1800 = timestamp(year, month, dayofmonth, 18, 00, 00) bool method1 = hour == 18 and minute == 00 bool method2 = not na(timeInSession) and na(timeInSession[1]) bool method3 = timestamp1800 == time plotchar(method1 ? 1 : na, "method1", "", location.abovebar, color.red, text="1", size=size.tiny) plotchar(method2 ? 2 : na, "method2", "", location.abovebar, color.orange, text="2\n", size=size.tiny) plotchar(method3 ? 3 : na, "method3", "", location.abovebar, color.yellow, text="3\n\n", size=size.tiny)

Note that:

- To ensure that the numbers do not plot on top of one another, we use the technique described in the FAQ entry How can I stack plotshape text?

Can I time the duration of a condition?

Normally, variables reset every time the script performs a new iteration. This process is called *rollback* and is described in the Execution model page of the User Manual.

By contrast, variables declared with varip *preserve* their values between realtime updates. This behavior allows scripts to track conditions during bars.

The following example script uses the `secondsSince()` function from the PineCoders' time library. This function uses variables declared with varip to track the number of seconds that elapse within a realtime bar while a condition remains true. Our script waits for price to move a user-defined number of ticks from the open, and then starts a timer. When the appropriate number of seconds elapses, it triggers an alert. A table displays these conditions in real time.

image

//@version=6 indicator("Condition timer", overlay = true) import PineCoders/Time/4 as PCtime string TICKS_TT = "The number of ticks price must move above the open to trigger the alert." string SEC_TT = "Seconds for which the condition must be continuously true before the alert triggers." string RESET_TT = "When checked, the duration resets every time a new realtime bar begins." int ticksInput = input.int(2, minval = 1, title = "Number Of Ticks From Open", tooltip = TICKS_TT) float secondsInput = input.int(20, minval = 10, title = "Seconds condition must last", tooltip = SEC_TT) bool resetInput = input.bool(true, title = "Reset timing on new bar", tooltip = RESET_TT) float targetTicks = open + (syminfo.mintick * ticksInput) bool targetTicksReached = close >= targetTicks // Calculate seconds elapsed since price reached the target. int secondsSinceTarget =

PCtime.secondsSince(targetTicksReached, resetInput and barstate.isnew) bool timeAlert = secondsSinceTarget > secondsInput // Has the timer expired? string alertTime = str.format_time(secondsSinceTarget * 1000, "mm:ss") // Format a time string for the timer label. // Set the contents for the label depending on the stage of the alert timer. string alertString = "Waiting for price to reach " + str.tostring(targetTicks) + "(" + str.tostring(ticksInput) + " ticks from " + str.tostring(open, format.mintick) + ")\nCurrent price: " + str.tostring(close, format.mintick) alertString := timeAlert ? "Timed Alert Triggered\n\n" + alertTime : targetTicksReached ? "Condition Detected...\n\nTimer count\n" + alertTime : alertString if barstate.islast var table statusTable = table.new(position.top_right, 1, 2, bgcolor = color.new(color.black, 70)) // Row 1: Combined tick conditions and current status string row1Text = "Target price: " + str.tostring(targetTicks, format.mintick) + " (+" + str.tostring(ticksInput) + " ticks from " + str.tostring(open, format.mintick) + ")\n" + (targetTicksReached ? "Price condition reached" : "Current price: " + str.tostring(close, format.mintick)) table.cell(statusTable, 0, 0, row1Text, text_color = chart.fg_color, bgcolor = targetTicksReached ? color.new(color.blue, 50) : color.new(color.black, 70)) if targetTicksReached // Only show the timer on row 2 when the price condition is reached. string row2Text = "Time required: " + str.tostring(secondsInput) + " seconds\n" + (timeAlert ? "TIMED ALERT TRIGGERED" : "Timer: " + alertTime) table.cell(statusTable, 0, 1, row2Text, text_color = chart.fg_color, bgcolor = timeAlert ? color.new(color.green, 50) : color.new(color.blue, 50)) if timeAlert // Fire alert if timer is triggered. alert("Timed Alert Triggered")

Note that:

- The `secondsSince()` function from the PineCoders' time library resets the timer if the supplied condition (in this case, the price condition) becomes false at any time.
- The `secondsSince()` function only works in real time, because it relies on intrabar updates, which are not present on historical bars. Therefore, it is possible for the timer to exceed a given specified time, especially on symbols with low liquidity, if a bar

update does not occur around the time that the timer should finish.
- We include the option to reset the timer on every bar or not, for demonstration purposes. The price condition in this example script resets on each bar, but other implementations might use different conditions.

For more information on the `secondsSince()` function and the use of varip variables, consult the PineCoders' Using `varip` variables publication.

How can I identify the nth occurrence of a weekday in the month?

The following example script colors the background of the nth occurence of a user-configurable weekday. This identification can be useful for scheduled events that occur on specific weekdays, such as certain options expiry days.

image

// @version=6 indicator("N-th weekday of the month", overlay = true) int occurrenceInput = input.int(3, "Occurrence", 1, 5) // The occurrence of the weekday to check for in the current month. enum Wday Mon = "Monday" Tue = "Tuesday" Wed = "Wednesday" Thu = "Thursday" Fri = "Friday" wdInput = input.enum(Wday.Mon, "Weekday") int weekdayInt = wdInput == Wday.Mon ? 1 : wdInput == Wday.Tue ? 2 : wdInput == Wday.Wed ? 3 : wdInput == Wday.Thu ? 4 : wdInput == Wday.Fri ? 5 : na // @function Calculates the number of days in a specified month, accounting for leap years. // @param yearNumber (int) The year of the `monthNumber` month. Optional. Default is the current year. // @param monthNumber (int, optional) The month for which to find the number of days. Optional. Default is the current month. // @returns (int) The number of days in the `monthNumber` month of the `yearNumber` year. daysPerMonth(int yearNumber = year, int monthNumber = month) => bool isLeapYear = (yearNumber % 4 == 0 and yearNumber % 100 != 0) or (yearNumber % 400 == 0) int result = switch monthNumber == 2 => isLeapYear ? 29 : 28 => 31 -

(monthNumber - 1) % 7 % 2 //@function Creates a timestamp representing the N-th occurrence of a specified weekday within a given month. //@param yearNumber (int) The year of the timestamp. //@param monthNumber (int) The month of the timestamp. //@param weekdayNumber (int) The weekday of the timestamp. Can be a value between 1 and 7, where 1 is a Monday. //@param occurrence (int) The occurrence of the `weekdayNumber` to check for. //@returns (int) The timestamp at the N-th `occurrence` of the `weekdayNumber` in the month. weekdayOfMonth(int yearNumber, int monthNumber, int weekdayNumber, int occurrence) => int startTime = timestamp(yearNumber, monthNumber, 1) int daysInTheMonth = daysPerMonth(yearNumber, monthNumber) int endTime = timestamp(yearNumber, monthNumber, daysInTheMonth) int weekday = dayofweek(startTime) - 1 if syminfo.timezone == "Etc/UTC" and not timeframe.isintraday weekday -= 1 if weekday == 0 weekday := 7 int offset = weekdayNumber - weekday if offset < 0 offset := 7 + offset int result = startTime + (offset + 7 * (occurrence - 1)) * 86400000 if result > endTime result := na result int occurrenceTime = weekdayOfMonth(year(time_close), month(time_close), weekdayInt, occurrenceInput) bool isAtOccurrence = time_close[1] < occurrenceTime and time_close >= occurrenceTime plot(occurrenceTime, "Time of the N-th weekday occurrence", color.orange, display = display.data_window) plot(time_close, "Bar close time", display = display.data_window) bgcolor(isAtOccurrence ? color.purple : na, title = "Time condition highlight")

Note that:

- We use an enum input to specify the target weekday, and convert it to an integer using a switch structure.
- The custom function `daysPerMonth()` calculates the number of days in any given month, considering leap years.
- The `weekdayOfMonth()` function determines the timestamp for the nth occurrence of a selected weekday within a specific month. This function first calculates the start time of the month and assesses the total number of days it contains. The function accounts for the unique characteristic of futures symbols, which

begin trading on Sunday night, potentially causing a one-day discrepancy in the calculation.
- We plot the time of the bar close and of the occurence to the Data Window.

How can I count down the remaining time in a bar?

Users can display a countdown on the price scale of the chart that shows the time remaining in each bar, by enabling the “Countdown to bar close” option in the “Scales and lines” section of the chart “Settings” menu. Our example script below displays a similar countdown timer. This script functions on intraday and “1D” timeframes only. For timeframes longer than “1D”, more complex logic is necessary. The script throws a runtime error if the chart timefrmae is greater than one day.

We subtract timenow from time_close to calculate the time remaining in the current bar, and then display the result in a table.

Because Pine scripts run only when there is a chart update, countdown timers do not usually update every second. The script refreshes more often on symbols with higher liquidity.

image

//@version=6 indicator("Countdown timer", overlay = true) if not ((timeframe.isdaily and timeframe.multiplier == 1) or timeframe.isintraday) runtime.error("This script functions only on daily or intraday timeframes.") // Inputs for bullish and bearish candle colors color bullishCandleColor = input.color(color.green, "Bullish Candle Color") color bearishCandleColor = input.color(color.red, "Bearish Candle Color") // Analyse the candle colors to see if black or white text has better contrast f_contrastColor(bgColor) => // Calculate luminance (relative brightness) using standard formula luminance = 0.2126 * color.r(bgColor) + 0.7152 * color.g(bgColor) + 0.0722 * color.b(bgColor) contrastColor = luminance > 127.5 ? color.black : color.white color bullishContrastColor = f_contrastColor(bullishCandleColor) color

bearishContrastColor = f_contrastColor(bearishCandleColor) int timeLeftInBar = time_close - math.min(timenow, time_close) var table timer = table.new(position = position.middle_right, columns = 1, rows = 1) if barstate.isfirst table.cell(timer, 0, 0, text_color = chart.fg_color, text_size = 11) else if barstate.islast string timeFormat = timeLeftInBar >= 60 * 60 * 1000 ? "HH:mm:ss" : "mm:ss" string countDown = str.format_time(timeLeftInBar, timeFormat, "UTC-0") table.cell_set_text(timer, 0, 0, countDown) bool isUpCandle = close >= open table.cell_set_text_color(timer, 0, 0, isUpCandle ? bullishContrastColor : bearishContrastColor) color bgcolor = isUpCandle ? bullishCandleColor : bearishCandleColor table.cell_set_bgcolor(timer, 0, 0, bgcolor)

Note that:

- We use the str.format_time() function to present the timestamp in a format similar to the price scale countdown timer. The function uses a conditional format so that we display hours only if the time remaining is one hour or greater.
- The built-in bar countdown uses a background the same color as the current candle. Pine scripts do not have access to the chart candle color settings, but if the user sets the correct colors in the script settings, our countdown displays in the same way.
- We use a custom function to analyze the luminance of the background colors and decide whether black or white text has the better contrast.

How can I get the week of the month?

Counting the week of the month in which the current bar occurs is not always straightforward. Some symbols trade overnight, so the time of the bar open is technically the previous day to the bar close, and sometimes the previous week. Because weekofyear, month, and similar variables use time (the time of the bar open) in their calculations, they can give unexpected results. The following example script, which uses these variables, gives unexpected results on some symbols such as US futures charts. For example, it increments the number of weeks on a Tuesday if the first trading day of the month is a Monday.

// @version=6 indicator("Week of month demo 1") bool isNewWeek = ta.change(weekofyear) != 0 bool isNewMonth = ta.change(month) != 0 var int weekCount = na weekCount := timeframe.ismonthly ? na : isNewMonth ? 1 : isNewWeek ? weekCount + 1 : weekCount plot(weekCount, "Week of month", chart.fg_color) bgcolor(isNewWeek ? color.new(chart.fg_color, 90) : na) bgcolor(isNewMonth ? color.new(color.lime, 80) : na)

The following amended script version uses the functions weekofyear() and month() instead of the equivalent variables, and evaluates the time_tradingday variable, which, unlike time, returns the beginning time *of the trading day* that the bar belongs to. Using this variable, and fixing the time zone to “UTC”, avoids problems with sessions that span days.

// @version=6 indicator("Week of month demo 2") bool isNewWeek = ta.change(weekofyear(time_tradingday, "UTC")) != 0 bool isNewMonth = ta.change(month(time_tradingday, "UTC")) != 0 var int weekCount = na weekCount := timeframe.ismonthly ? na : isNewMonth ? 1 : isNewWeek ? weekCount + 1 : weekCount plot(weekCount, "Week of month", chart.fg_color) bgcolor(isNewWeek ? color.new(chart.fg_color, 90) : na) bgcolor(isNewMonth ? color.new(color.lime, 80) : na)

While the preceding examples are useful for understanding the difference between time and time_tradingday, the simplest solution is to use the timeframe.change() function, which detects changes in a given timeframe. We can rewrite the boolean conditions in our example script as follows:

bool isNewWeek = timeframe.change("1W") bool isNewMonth = timeframe.change("1M")

Note that:

- All versions of the script reset the number of weeks at the beginning of the month, then increment the count each new week.
- These scripts do not plot anything on timeframes of one month or greater.
- In Pine V5 and below, scripts can assign values of type “float” and “int” to a boolean variable. A value of zero is equivalent to false,

and any other value to true. In later versions of Pine, scripts can assign only boolean values to boolean variables. So in V5, this code works: `bool isNewMonth = ta.change(month)` but in later versions this equivalent code is needed: `bool isNewMonth = ta.change(month) != 0`.

Previous

Next

Variables and operators

Variables and operators

What is the variable name for the current price?

In Pine Script®, the close variable represents the current price. It provides the *closing price* of each historical bar, and, for indicator scripts, the *current price* of the most recent realtime bar. The close value of an open bar can change on each tick to reflect the latest price.

Strategy scripts usually execute only once on each historical *and* realtime bar, at the bar close. Consequently, during a realtime bar, the close variable holds the *previous* bar's closing price. However, if a script sets the `calc_on_every_tick` parameter of the strategy() declaration statement to `true`, the strategy executes with each price change of the realtime bar, like indicators do. As a result, close holds the latest realtime price update.

To reference the closing price of the previous bar, use `close[1]`. Learn more about using square brackets to reference previous values in the history-referencing operator section.

Why declare variables with the var keyword?

The var keyword is useful for storing data across multiple bars. By default, the value assigned to a variable is *reset* and calculated again on each new bar. This process is called rollback.

If a script declares a variable with the var keyword, this *persistent* variable is initialized only *once*. Variables declared in the global scope are initialized on the first bar. Variables declared in a local block are initialized the first time that the local block executes. After its initial assignment, a persistent variable maintains its last value on subsequent bars until it is reassigned a new value.

In the example below, we demonstrate how to accumulate volume across bars, comparing an ordinary and a persistent “float” variable.

//@version=6 indicator("Var keyword example") // Declare and initialize a persistent variable by using `var`. var float a = 0 // Declare and initialize a normal float variable. float b = 0 // Reset the values of a and b whenever a new day begins. if timeframe.change("D") a := 0 b := 0 // Add the current volume to both a and b. a += volume b += volume // Plot the values of `a` and `b`. The value of `a` accumulates over time; `b` is reinitialized at every bar. plot(a, "a", close > open ? #089981 : #f23645, style = plot.style_columns) plot(b, "b", color.yellow)

What is `varip` used for?

The varip keyword declares variables whose values persist *within the same realtime bar*. This contrasts with the typical mode of Pine’s execution model, where variables are reset to their last committed value with *each realtime script execution*, potentially many times in each bar.

Recall that the var keyword allows a variable to retain its value from bar to bar — however, the value still resets on each script execution *within* a bar. The varip keyword takes this persistence a step further and escapes the rollback process, or re-initialization, on *each price update* within the same realtime bar.

As a result, varip (which stands for “variable intrabar persist”) variables can perform calculations that span *across executions* in the same bar. For example, they can track the number of realtime updates that occur within a realtime bar.

It’s important to note that varip only affects the behavior of code on realtime bars, not historical ones. Therefore, backtest results on strategies based on varip variables might not accurately reflect the behavior of those historical bars. Similarly, calculations on historical bars won’t reproduce the script’s realtime behavior.

To distinguish between var and varip, add the following script to a live market symbol. With realtime updates, the varip plot increments within a bar on each price update, whereas the var plot stays constant within a bar:

image

//@version=6 indicator("varip vs var demo") // `var` : Retains value across bars, resets on intrabar price updates. // 'varip': Retains value across bars and across intrabar price updates within a realtime bar. var int varCount = -1 varip int varipCount = -1 // Increment `varCount` on each bar and `varipCount` on each intrabar price update. varCount += 1 varipCount += 1 // Plot values for comparison. plot(varCount, "var counter", color.fuchsia, 4) plot(varipCount, "varip counter", color.lime)

Note that:

- Both plots in the above script are the *same* for historical bars, because there are no intrabar updates on historical bars.

What’s the difference between ==, =, and :=?

The = operator declares and initializes variables, assigning a specific value to a named variable. For example, `a = 0` sets the variable a to hold the value 0.

The := property reassigns values to existing variables. For instance, if a script declared `int a = 1`, a subsequent line `a := 2` updates the value of a to 2, which is possible because integer variables are *mutable*, or changeable.

Finally, the == operator is a comparison operator. It checks the equality between two values, returning a Boolean (true/false) result. For instance, a == b is true if a and b hold the same value. The opposite operator is !=, which is true if the two variables are *not* equal.

The following script initializes two variables a and b, reassigs a, and then performs and plots equality comparisons.

//@version=6 indicator("Variable operators demo", overlay = true) // Define two variables `a` and `b` using `=`, representing the high and low of each bar. float a = high float b = low // Define the initial line color as lime color lineColor = color.lime // When there are fewer than 10 bars left on the chart, use `:=` to update `a` to `b` and change the line color. if last_bar_index - bar_index < 10 a := b lineColor := color.fuchsia // Plot the variable 'a' to visualize its change in value. // Initially, 'a' represents the 'high' of each bar. // If there are fewer than 10 bars remaining in the chart, 'a' is updated to represent the 'low' of each bar. plot(a, "Our line", lineColor, 2) // Plot a checkmark character whenever `a` is equal to `b`. plotchar(a == b, "a equals b", "✅", location.bottom) // Plot a cross character whenever `a` is not equal to `b`. plotchar(a != b, "a does not equal b", "❌", location.bottom)

Can I use the := operator to assign values to past values of a series?

Historical values are fixed and cannot be changed. Just as we can’t alter the past in real life, scripts are unable to modify historical values in a series, because they are read-only. For example, the following script generates an error:

//@version=6 indicator("Changing historical values demo", overlay = true) // Initialize a variable to hold the value of the current bar's high. series float a = high // Reassign the *previous* value of the series `a` to hold the low of the current bar. a[1] := low // This line causes a compilation error. plot(a, color = chart.fg_color, linewidth = 3)

However, scripts *can* assign or reassign a value to the current instance of a series, and assign a historical value of a series to a variable. The following version of our script works without error.

//@version=6 indicator("Changing historical values demo", overlay = true) // Initialize a variable to hold the value of the current bar's high. series float a = high // Reassign the *current* value of the series `a` to hold the high of the *previous* bar. a := high[1] plot(a, color = chart.fg_color, linewidth = 3)

Why do the OHLC built-ins sometimes return different values than the ones shown on the chart?

The OHLC (Open, High, Low, Close) values displayed on the chart and the values returned by the built-in OHLC variables open, high, low, close can differ. This is because data feeds can contain price points that exceed a symbol's defined *tick precision*. While visually, chart prices are always rounded to tick precision, the built-in variables maintain their original, unrounded values.

For instance, if an exchange feed provides a closing price of 30181.07, which is more precise than the symbol's 0.1 tick size, the chart displays a rounded value of 30181.1, whereas the built-in variable holds the unrounded value of 30181.07.

Subtle differences, while not immediately obvious, can lead to significant outcomes, especially in scripts requiring precise calculations or when diagnosing unexpected behaviors in scripts. An example of this is in detecting crossover events. Discrepancies between unrounded and rounded values can cause scripts to identify crossover events in one scenario but not in the other.

One way to mitigate this issue is to round the OHLC built-in variables to the nearest tick size before using them in calculations. The script below highlights discrepancies between actual OHLC values and their rounded counterparts, visually indicating any differences by coloring the background red:

image

```
//@version=6 indicator("Different tick values example", overlay = true, precision = 10) // @function Rounds each OHLC value to the nearest minimum tick size. // @returns A tuple containing the rounded values. OHLCToMinTick() => [math.round_to_mintick(open), math.round_to_mintick(high), math.round_to_mintick(low), math.round_to_mintick(close)] //@function Checks whether two float values are equal or not. //@param v1 (series float) The first value to compare. //@param v2 (series float) The second value to compare. // @returns The color blue if the values are equal or red otherwise. getTickColor(series float v1, series float v2) => color result = v1 != v2 ? color.red : color.blue // Round each OHLC value to the nearest mintick size. [o, h, l, c] = OHLCToMinTick() // Plot the original and rounded values of each OHLC component in the data window. // If a value and its rounded counterpart are not equal, color the plot red. Otherwise, color it blue. plot(o, "o", getTickColor(o, open), display = display.data_window) plot(open, "open", getTickColor(o, open), display = display.data_window) plot(h, "h", getTickColor(h, high), display = display.data_window) plot(high, "high", getTickColor(h, high), display = display.data_window) plot(l, "l", getTickColor(l, low), display = display.data_window) plot(low, "low", getTickColor(l, low), display = display.data_window) plot(c, "c", getTickColor(c, close), display = display.data_window) plot(close, "close", getTickColor(c, close), display = display.data_window) // If any of the original and rounded values of OHLC components are not equal, set the background color to red. bgcolor(o != open or h != high or l != low or c != close ? color.new(color.red, 90) : na)
```

Why do some logical expressions not evaluate as expected when na values are involved?

In Pine Script, every type of variable can take an na value — *except* Boolean variables, which can only be true or false. Here, na stands for “not available”, and signifies the absence of a value, similar to NULL in other programming languages.

Although Boolean values themselves cannot be na, logical expressions that evaluate to true or false can depend on variables of other types that *can* be na.

This behavior can cause unexpected outcomes because any valid logical comparison that includes na values *always returns false*.

The following example script evaluates a single comparison where one value is always na. The user can choose which comparison to evaluate from a set. A label displays the chosen comparison and its result.

//@version=6 indicator("`na` example") string conditionInput = input.string("a != b", "Condition", options=["a != b","a == b", "a > b", "a < b"]) int a = 1 int b = na bool condition = switch conditionInput "a != b" => a != b "a == b" => a == b "a > b" => a > b "a < b" => a < b if barstate.islastconfirmedhistory string conditionText = condition ? "true" : "false" label.new(x = bar_index, y = high, text = "a = 1\nb = na\n" + conditionInput + ": " + conditionText, color = condition ? color.green : color.red, textcolor = color.new(chart.fg_color, 0), style = label.style_label_down, size = size.large)

To avoid unwanted false negatives, write code that checks for na values and, if necessary, replaces them. For a discussion of na values and how to manage them, see the na value section of the User Manual.

Previous

Next

Visuals

Pine Script® v6 Documentation

User Manual / FAQ / Visuals

Visuals

Why can’t I use a plot in an `if` or `for` statement?

In Pine Script®, scripts cannot place plot() calls directly within if or for statements — or in any other local scopes. The compiler needs to know about all plots during script compilation.

However, scripts *can* plot values conditionally, by changing the series or color of the plot.

Our example script plots two ALMA moving averages only when the shorter average is below the longer one. It fills between the two averages on every bar, but the fill color is na unless the shorter average is above or equal to the longer one.

image

//@version=6 indicator("Conditional plot example", "", true) // Calculate two ALMAs. float ma1 = ta.alma(close, 21, 0.85, 6) float ma2 = ta.alma(close, 50, 0.85, 6) // Calculate whether the MAs are in bullish or bearish order. var bool areBullCrossed = false if ma1 > ma2 areBullCrossed := true else areBullCrossed := false // Plot the MAs to the Data Window for use in the fill. p1 = plot(ma1, "MA 1: Conditional fill", display = display.data_window, editable = false) p2 = plot(ma2, "MA 2: Conditional fill", display = display.data_window, editable = false) // Fill the MAs with color only when the MAs are in bullish order. fill(p1, p2, areBullCrossed ? color.aqua : na) // Plot the MAs only when they are in bearish order. plot(not areBullCrossed ? ma1 : na, "MA 1:

Conditional plot", color.fuchsia, 3, plot.style_linebr) plot(not areBullCrossed ? ma2 : na, "MA 2: Conditional plot", color.new(color.fuchsia, 70), 5, plot.style_linebr)

Note that:

- The script updates the Boolean variable that describes whether the moving averages are in bullish order — and thereby controls the plots and fill — in a *local scope*. However, the script declares it in the *global scope* so that the fill() and plot() calls can use it.
- The moving averages are each plotted twice. The plots that display in the Data Window must be assigned to variables so that the fill() function call can reference them. The fill() function cannot use the later plots, which display on the chart, because they have na values when the fill color is not na.
- We use `plot.style_linebr` as the argument for the `style` parameter of the plot() function so that the plot does not span bars with an na plot value.

Can I plot diagonals between two points on the chart?

Scripts can plot diagonal lines between two points on a chart by using plots or line objects.

Using plots

The plot() function connects consecutive data points with straight lines.

The default value of the `style` argument for the plot() function is `plot.style_line`. This style of plot connects the plotted points on either side of bars that have na series values to each other with a line. If the points are at different heights, the lines are diagonal.

The functions ta.pivotlow() and ta.pivothigh() return na for all bars except those with identified pivots. The following example script draws diagonal lines joining pivot highs and lows using plot().

image

```
//@version=6 indicator("Diagonal plots", overlay = true) // User input to define the number of bars to look back and forwards to define pivots. int pivotBarsInput = input.int(5, "Pivot Bars") // Find the value of pivot lows and highs. The value is `na` if the current bar is not a pivot. float pivotLow = ta.pivotlow(pivotBarsInput, pivotBarsInput) float pivotHigh = ta.pivothigh(pivotBarsInput, pivotBarsInput) // Define the offset for the plot. Negative offset values plot on bars earlier than the current bar. pivotOffset = -1 * pivotBarsInput // Join the pivot values with plotted lines. plot(pivotLow, "Pivot Low", color.fuchsia, offset = pivotOffset) plot(pivotHigh, "Pivot High", color.teal, offset = pivotOffset) // Place a circle on each pivot to emphasize the points being connected, offsetting it back to the pivot bar. plot(pivotLow, "Pivot Low", color.fuchsia, 3, plot.style_circles, offset = pivotOffset) plot(pivotHigh, "Pivot High", color.teal, 3, plot.style_circles, offset = pivotOffset)
```

Note that:

- The ta.pivotlow() and ta.pivothigh() functions confirm pivot highs and lows only after a specified number of bars, so the script must offset the plotted lines by the same number of bars. Plot offsets cannot change during script execution.

Using lines

Line objects provide more flexibility than plots. The plot() function can plot a line, symbol, or area only at the bar on which the script is executing (or at a *fixed* offset from it). In contrast, line objects can be created on *any* bar. Similarly, whereas plots are fixed once the bar closes, line properties can be updated at any time.

The following example script demonstrates these advantages. Like the example script in the previous section, Using plots, this script draws lines between pivot highs and lows. In this case, however, the pivot highs and lows are confirmed an *unpredictable* number of bars afterwards, so plots are completely unsuitable for drawing lines between them. Line drawings are suitable because scripts can offset lines an arbitrary and dynamic number of bars into the past. The script below also retroactively *changes* the color of drawn lines, which are

initially white, depending on whether the *next* line of the same type slopes up or down. Such updates are not possible with plots.

image

//@version=6 indicator("Diagonal lines", overlay = true, max_lines_count = 500) int lookbackInput = input.int(defval=10, title="Lookback Bars", minval=1) enum PivotState // Stores the state of the pivot detection. none // Start from zero. hadPivot // Have we found a pivot? hadCounter // Have we had a down candle for a high or a up candle for a low? var PivotState highState = PivotState.none, var PivotState lowState = PivotState.none // Create state objects. updatePivotState(PivotState state, bool foundPivot, bool counterCandle, bool newExtreme, bool isConfirmedPivot) => if newExtreme // Start again if we make a new high/low. PivotState.none else switch state // Progress the state when we get the right events. PivotState.none => foundPivot ? PivotState.hadPivot : state PivotState.hadPivot => counterCandle ? PivotState.hadCounter : state PivotState.hadCounter => isConfirmedPivot ? PivotState.none : state => state // Default: persist. // Calculate the events. float highestHigh = ta.highest(high, lookbackInput) float lowestLow = ta.lowest(low, lookbackInput) bool foundHigh = high[1] == highestHigh[1] and high < highestHigh[1] bool foundLow = low[1] == lowestLow[1] and low > lowestLow[1] bool newHigh = high >= highestHigh bool newLow = low <= lowestLow // Update the state. bool isConfirmedHigh = highState == PivotState.hadCounter and close > open bool isConfirmedLow = lowState == PivotState.hadCounter and close < open highState := updatePivotState(highState, foundHigh, close < open, newHigh, isConfirmedHigh) lowState := updatePivotState(lowState, foundLow, close > open, newLow, isConfirmedLow) // Store the price and bar index of the pivots. var float pivotHighPrice = na, var float pivotLowPrice = na var int pivotHighBarIndex = na, var int pivotLowBarIndex = na if foundHigh pivotHighPrice := high[1] pivotHighBarIndex := bar_index[1] if foundLow pivotLowPrice := low[1] pivotLowBarIndex := bar_index[1] // Store the current and previous high and low points. var chart.point prevHighPivot = na, var chart.point highPivot = na var chart.point prevLowPivot = na, var chart.point lowPivot = na var line pivotHighLine = na, var line pivotLowLine = na // Draw the lines

drawPivotLine(chart.point prevPivot, line prevLine, int barIndex, float price) => if not na(prevPivot) line.set_color(prevLine, price < prevPivot.price ? color.red : color.green) newLine = line.new(x1=prevPivot.index, y1=prevPivot.price, x2=barIndex, y2=price, color=color.white, width=2) newLine if isConfirmedHigh prevHighPivot := highPivot // Shift current to previous highPivot := chart.point.from_index(pivotHighBarIndex, pivotHighPrice) pivotHighLine := drawPivotLine(prevHighPivot, pivotHighLine, pivotHighBarIndex, pivotHighPrice) if isConfirmedLow prevLowPivot := lowPivot // Shift current to previous lowPivot := chart.point.from_index(pivotLowBarIndex, pivotLowPrice) pivotLowLine := drawPivotLine(prevLowPivot, pivotLowLine, pivotLowBarIndex, pivotLowPrice)

Note that:

- This script creates custom pivots using a combination of a lookback similar to that used in Williams Fractals and price confirmation. Potential pivots require a candle in the counter-trend direction *followed by* a candle in the trend direction in order to confirm. This means that the candle on which pivots confirm cannot be predicted, unlike Williams pivots.
- Because the pivot is an arbitrary number of bars back from the confirming candle, we cannot plot circles to highlight the pivots, as we did for the script example in the previous section.
- We use an enum to store the state of the pivot confirmation process. This, together with a set of explicit rules for changing the state, forms a simple state machine. Such constructs can be easier to debug than maintaining and resetting multiple persistent global variables that depend on each other.
- Scripts can create only a certain number of lines. The limit is set by the `max_lines_count` parameter of the indicator() or strategy() declaration. The default is 50, and the maximum is 500 per script.

How can I plot a line with gaps?

Scripts can plot lines with gaps by setting the argument of *either* the `series` parameter or `color` parameter of the plot() function to na for some bars. These two methods have different effects:

- An na *color* value omits the plotted line from the most recent bar with a non-na value to the first bar with an na value. The size of the gap is equal to the number of consecutive bars with na values.
- An na *series* value omits the plot point for each bar with an na value, effectively **increasing the size of the gap by one bar**.

The following example script shows the effect of these two methods. It plots two straight lines above the chart bars, one using each method. At a configurable interval, the `series` or `color` argument is na, and the background color changes for that bar.

image

//@version=6 indicator("Lines with gaps demo", "", true) var float lowerPlot = na, var float upperPlot = na bool change = bar_index % input.int(5, minval = 2, title = "Bar interval") == 0 float atr = ta.atr(14) if change lowerPlot := high + atr upperPlot := high + (1.5 * atr) plot(lowerPlot, "Plot with Varying Color", color = change ? na : color.red, linewidth = 2, style = plot.style_line) plot(change ? na : upperPlot, "Plot with Varying Series", color = color.blue, linewidth = 2, style = plot.style_linebr) bgcolor(change ? color.new(color.gray, 80) : na)

Note that:

- The style for the plot with na `series` values must be `plot.style_linebr` in order to show gaps. The default value of `plot.style_line` fills in the gaps.

How do I plot a line using start/stop criteria?

To plot a line based on start and stop criteria, consider the following structured approach:

1. Define start and stop conditions.
2. Control variables for *when* and *where* to plot.
3. Choose a plotting style. Use na for either the `series` or `color` parameter of the plot() function. See the section How can I plot a line with gaps? above for examples.
4. Optionally, use debug. For example, plot logical states using the plotchar() function.

In the following example script, the start condition is the detection of a new pivot. The start condition turns on the `doPlot` Boolean flag that controls when to plot, and captures the value to plot in the `savedValue` "float" variable. The stop condition is price closing above the pivot level. The script uses a bar timer as an extra stop condition. Either stop condition turns off the flag and *resets* the plot value to na. The script plots debug characters for all logical conditions.

image

//@version=6 indicator("Starting and stopping a plot", overlay = true) // Set the maximum number of bars for a line. int expiryBarsInput = input.int(50, "Maximum bars to plot line", minval = 0) var bool doPlot = false // Whether to plot the line. var float savedValue = na // The value to plot. var int timerStart = na // The bar on which the line started. // Define conditions for the start and end of the line plot. bool startCondition = not na(ta.pivothigh(close, 5, 2)) bool closeAboveLevel = ta.crossover(close, savedValue) bool hasExpired = not (bar_index < timerStart + expiryBarsInput) bool stopCondition = closeAboveLevel or hasExpired float atr = ta.atr(14) if startCondition and not doPlot // If this is the start of a *new* plot, savedValue := high + (atr /2) // set the value to plot, timerStart := bar_index // and start the timer. // Start, stop, or persist the state that controls the plot. doPlot := startCondition ? true : stopCondition ? false : doPlot // Plot the line if

`doPlot` is true. plot(doPlot ? savedValue : na, "Saved Value", color.white, style = plot.style_linebr) // Debug: Plot the logical events. plotchar(startCondition, "startCondition", "►", location.abovebar, color.green, size = size.tiny) plotchar(closeAboveLevel, "closeAboveLevel", "◄", location.belowbar, color.red, size = size.tiny) plotchar(hasExpired, "hasExpired", "×", location.abovebar, color.gray, size = size.tiny)

Note that:

- The order in which scripts turn conditions on and off is important when dealing with persistent states, both in terms of the order in the script and within ternary conditions.

How can I plot a support or trend line?

Support is a horizontal zone on a chart where analysts consider that a declining price is likely to turn upwards. Conversely, *resistance* is a horizontal area from which a rising price is likely to turn downwards. Trend lines are usually diagonal lines that function as support or resistance.

Different analysts — and different Pine scripts — understand and implement support, resistance, and trend lines differently. In the sections below, we provide some simple examples.

Plotting support and resistance

The following example script tracks levels of support and resistance until price action breaks them. The script uses the `ta.pivot*()` built-in functions to detect pivot highs and lows, then draws horizontal lines from these points. This script visualizes lows as support (green lines) and highs as resistance (red lines) for simplicity. If the close of a bar crosses a line, the script stops extending that line:

image

//@version=6 indicator("Support and resistance demo", "", true, max_lines_count = 500) color hiPivotColorInput =

input.color(color.fuchsia, "High pivot color") color loPivotColorInput = input.color(color.lime, "Low pivot color") int pivotSizeInput = input.int(5, "Pivot lookback/look forward") int maxLineLengthInput = input.int(100, "Maximum line length", minval = 2) // @function Extends every line in `lineArray` until price crosses it or it exceeds `maxLength` in length. // @param lineArray (array<line>) An array storing the lines to check. // @param maxLength (int) The maximum length a line can have. // @returns (void) The function has no explicit return. checkLinesForBreaches(array<line> lineArray, int maxLength) => // If there are no lines, the `from` value is `na` and we do not loop. int fromValue = lineArray.size() > 0 ? lineArray.size() - 1 : na // We loop the array in reverse to avoid errors even when we remove the first element in the array. for i = fromValue to 0 // Check each line crosses and length. line eachLine = lineArray.get(i) float linePrice = eachLine.get_price(bar_index) bool lineWasCrossed = math.sign(close[1] - linePrice) != math.sign(close - linePrice) bool lineIsTooLong = bar_index - eachLine.get_x1() > maxLength // Set the rightmost point of each line to the current bar. eachLine.set_x2(bar_index) // Set lines inactive if they have been crossed or are too long. if lineWasCrossed or lineIsTooLong // Stop the line from extending to the right. eachLine.set_extend(extend.none) // Remove the line from the array. The line stays on the chart but does not extend on further bars. lineArray.remove(i) // Arrays of active lines. var array<line> hiPivotLines = array.new<line>() var array<line> loPivotLines = array.new<line>() // Detect new pivots and record their values. float hiPivot = ta.pivothigh(pivotSizeInput, pivotSizeInput) float loPivot = ta.pivotlow(pivotSizeInput, pivotSizeInput) // Create new lines on new pivots and add to arrays. x1 = bar_index - pivotSizeInput if not na(hiPivot) hiPivotLines.push(line.new(x1, hiPivot, bar_index, hiPivot, extend = extend.right, color = hiPivotColorInput)) if not na(loPivot) loPivotLines.push(line.new(x1, loPivot, bar_index, loPivot, extend = extend.right, color = loPivotColorInput)) // Extend lines if they are still active. checkLinesForBreaches(hiPivotLines, maxLineLengthInput) checkLinesForBreaches(loPivotLines, maxLineLengthInput)

Note that:

- When price crosses a line, it becomes inactive. The script no longer updates inactive lines and does not extend them to the right.
- We store lines in one of two arrays, and remove lines from their array if they become inactive. Removing an object from an array does *not* delete the object.

Plotting trend lines

The following example script uses the `ta.pivot*()` built-in functions to detect pivot highs and lows, and then draws lines that connect the two most recent pivots of the same type. The lines extend indefinitely to the right. If the script draws a new line that causes the total number of lines to exceed a specified maximum number, it deletes the oldest line:

image

//@version=6 indicator("Simple trend lines demo", overlay = true, max_lines_count = 500) int pivotSizeInput = input.int(20, "Pivot lookback/look forward", minval = 1, maxval = 50) int maxLinesInput = input.int(2, "Quantity of lines to track per pivot", minval = 0, maxval = 50) color loTrendColorInput = input.color(color.lime, "Low pivot") color hiTrendColorInput = input.color(color.fuchsia, "High pivot") // @function Draws a line from the previous pivot when a new pivot of the same type occurs. // Keeps the number of lines under a specified number by deleting excess lines. // @param lineArray (array<line>) An array to store and manage the line objects. // @param pivotValue (float) The pivot price when a pivot is found. // @param pivotSize (simple int) The size of the pivot lookback/look forward. // @param maxLinesCount (simple int) The maximum number of lines to keep. // @param lineColor (color) The color for the line. // @returns (void) The function has no explicit return. queueLine(array<line> lineArray, float pivotValue, simple int pivotSize, simple int maxLinesCount, color lineColor) => var array<chart.point> pointArray = array.new<chart.point>(2, chart.point.new(na, na, na)) if not na(pivotValue) pointArray.push(chart.point.from_index(bar_index - pivotSize, pivotValue)) if pointArray.size() > 2 pointArray.shift() chart.point

firstPoint = pointArray.first() chart.point secondPoint = pointArray.last() line ln = line.new(firstPoint, secondPoint, extend = extend.right, color = lineColor, style = line.style_dotted) lineArray.push(ln) if lineArray.size() > maxLinesCount line.delete(lineArray.shift()) // Initialize two empty arrays for the high and low trend lines on the first bar. var array<line> hiLinesArray = array.new<line>() var array<line> loLinesArray = array.new<line>() // Detect new pivots. float hiPivot = ta.pivothigh(pivotSizeInput, pivotSizeInput) float loPivot = ta.pivotlow(pivotSizeInput, pivotSizeInput) // Draw new lines between the two most recent pivots when a pivot occurs, and add them to the line array. // The number of lines is limited to the specified number, after which lines are removed from the array *and* deleted. queueLine(hiLinesArray, hiPivot, pivotSizeInput, maxLinesInput, hiTrendColorInput) queueLine(loLinesArray, loPivot, pivotSizeInput, maxLinesInput, loTrendColorInput) // Visually highlight the pivot points with a dot. plot(hiPivot, "Pivot High", hiTrendColorInput, 3, plot.style_circles, offset = -pivotSizeInput) plot(loPivot, "Pivot Low", loTrendColorInput, 3, plot.style_circles, offset = -pivotSizeInput)

Note that:

- For simplicity, we do not deactivate lines when price crosses them. For an example of how to do this, see the example script from the previous section, Plotting support and resistance.
- We store the lines in arrays, which makes it easier to manage them. In this script, unlike in the example script in the previous section, we delete the line object at the same time as we remove it from the array.

How can I use colors in my indicator plots?

The strategic use of color in indicator plots helps comprehension, pattern and trend recognition, and differentiation of categories or values. For example, different hues can represent different data thresholds, and gradients can indicate increases or decreases in values. When a script transforms numbers into colors in an intelligent way, it enhances the user’s ability to spot anomalies, trends, and significant

data points. Here are some Pine features that script authors can use to work with colors:

Predefined colors

Without needing to specify hexadecimal or RGB (Red Green Blue) values, programmers can use predefined color constants, such as color.red. In the Pine Editor, clicking the automatically generated color swatch next to a color constant opens up a color picker; choosing a different color from the picker updates the code that defines the color.

Custom colors

For a more personalized appearance, programmers can create custom colors, either by specifying a hexadecimal code, or by using the color.rgb() function to specify the RGB values. For the hexadecimal and RGB equivalents of the built-in color constants, see the table in the Constant colors section.

Transparency settings

The color.new() function can create colors with a specific transparency. The `transp` parameter can even take series values, meaning that a single color declaration can provide a color with dynamic transparency.

Conditional coloring

Scripts can assign different colors to variables based on logical conditions. Programmers can use conditional colors to color plots, fills, shapes, drawings, or the chart background differently on different bars.

Gradient transitions

The color.from_gradient() function creates color gradients that can highlight shifts in data values while ensuring a smooth transition between colors. For detailed guidance and innovative examples on implementing gradients, consult the Color Gradient Framework by PineCoders.

Fills

Scripts can create shaded areas between lines, plots, or hlines, as well as within boxes and polylines. Fills can be especially useful for highlighting ranges, zones, or contrasts.

In our example script below, we use various color display techniques such as conditional colors, fills, and gradients to depict areas of rising or falling values, as well as overbought and oversold levels. The script automatically recognizes whether the user’s chart background is light or dark, and adjusts the color scheme accordingly. For a light theme, it captures the ambiance of a day at the beach with coastal teal and coral hues, while for a dark theme, it reflects the vibrant neon hues of city nightlife:

image

image

//@version=6 indicator("Using colors in Pine", explicit_plot_zorder = true) float tsi = ta.tsi(close, 13, 23) * 100, float tsl = ta.ema(tsi, 13) // Calculate TSI and its EMA. // @function Determines if a given background color corresponds to a light theme based on its brightness level. // @param bgColor (color) The background color to check. Optional. Default value is the chart's background color. // @returns (bool) True if the background color is "light" (has a brightness greater than 0.5), otherwise false. isLightTheme(color bgColor = chart.bg_color) => float r = color.r(bgColor) float g = color.g(bgColor) float b = color.b(bgColor) float brightness = (r + g + b) / (3 * 255) bool isLight = brightness > 0.5 var bool isLightTheme = isLightTheme() // Define color schemes based on whether the theme is light or dark. color tsiDnUpColor = isLightTheme ? #1E90FF : #BA33FF, color tsiDnDnColor = isLightTheme ? #FF6B6B : #8100FF color tsiUpUpColor = isLightTheme ? #00CED1 : #1FE0F3, color tslBullColor = isLightTheme ? #00CED1 : #1FE0F3 color tslBearColor = isLightTheme ? #1E90FF : #0088A3, color tsiBullColor = isLightTheme ? #FFD700 : #33FF57 color tsiBearColor = isLightTheme ? #FF6B6B : #00940D, color tsiUpDnColor = isLightTheme ? #FFD700 : #FF21D4 color bullBgColor = isLightTheme ? #FFD700 : #80FFFF, color bearBgColor = isLightTheme ? #FF6B6B : #FF80FF color obFillColor = isLightTheme ? #FFD700 : #33FF57, color

osFillColor = isLightTheme ? #FF6B6B : #33FF57 // Find the direction of the TSI, signal line, and trend. Calculate the difference for histogram values. bool tsiIsBull = tsi >= tsi[1], bool tslIsBull = tsl >= tsl[1], bool trendIsBull = tsi >= tsl float diff = tsi - tsl // Get line and fill colors based on trend and plot direction. color tsiColor = tsiIsBull ? tsiBullColor : tsiBearColor, color tslColor = tslIsBull ? tslBullColor : tslBearColor color fillColor = trendIsBull ? tsiIsBull ? tsiUpUpColor : tsiUpDnColor : tsiIsBull ? tsiDnUpColor: tsiDnDnColor // Create invisible horizontal lines at +30 and -30 and fill areas between them with a gradient. h1 = hline(30, color = color(na)), h2 = hline(-30, color = color(na)) fill(h2, h1, 30, 5, color.new(obFillColor, 80), color(na)) fill(h2, h1, -5, -30, color(na), color.new(osFillColor, 80)) // Define colors and transparency for the histogram bar based on trend direction and theme and plot the bar. color barBgColor = isLightTheme ? trendIsBull ? bullBgColor : bearBgColor : chart.bg_color color barBdColor = trendIsBull ? bullBgColor : bearBgColor int barBdTransp = isLightTheme ? 50 : 80 int barBgTransp = isLightTheme ? 90 : 0 float barHi = math.max(0, 0 + diff) float barLo = math.min(0, 0 + diff) plotcandle(barLo, barLo, barHi, barHi, color = color.new(barBgColor, barBgTransp), bordercolor = color.new(barBdColor, barBdTransp), wickcolor = color(na), display = display.pane) // Plot the TSI and its EMA p1 = plot(tsi, "TSI", color.new(tsiColor, 30)) p2 = plot(tsl, "Signal", color.new(tslColor, 30)) p3 = plot(tsl > 30 ? 30 : na, display = display.none) p4 = plot(tsl < -30 ? -30 : na, display = display.none) // Fill between the signal line and overbought or oversold levels with a gradient. fill(p2, p3, 80, 30, bearBgColor, color.new(bearBgColor, 90)) fill(p2, p4, -30, -80, color.new(bullBgColor, 90), bullBgColor) // Fill the cloud between the TSI and signal line with a gradient. fill(p1, p2, tsi, tsl, fillColor, color.new(fillColor, 70))

Note that:

- TSI and signal plots use distinct colors for each plot, with different colors for rising and falling values.
- We implement horizontal lines to demarcate overbought and oversold zones and fill between them with a color gradient. This gives the areas an appearance of scaled severity as the plots travel through them.

- We fill between the signal line and the overbought and oversold levels with another gradient if the plot exceeds either, to further emphasize potential irregularities.
- The script uses the plotcandle() function to create bordered column bars painted in bullish or bearish hues based on the trend's direction, serving as a visually distinct histogram that depicts the distance between the TSI and signal line.
- We use a four-color fill between the TSI and signal line to clarify trend direction and changes in trend. For instance, an upward trend with a declining TSI features a distinct color from the general rising trend hue.

How do I make my indicator plot in the main chart pane?

By default, new scripts display in a separate pane. To make a script display in the main chart pane instead, use `overlay = true` in the strategy() or indicator() declaration statement:

```
indicator("My Script", overlay = true)
```

The default value of the `overlay` parameter is `false`. If a programmer changes the value of the `overlay` parameter *after* an indicator was already added to the chart, they need to remove the indicator and add it to the chart again for the change to take effect.

Note that users of an indicator can move scripts from a separate pane to the chart pane and vice-versa using the "Move to" option from the "More" script menu, regardless of the value of the `overlay` parameter.

How can I plot vertical lines on a chart?

Scripts can plot vertical lines in three main ways:

- Drawing lines
- Plotting histograms
- Coloring the background

By drawing lines

Advatages of drawing vertical lines include that lines can be drawn on past or future bars. Each script can draw up to a maximum of 500 lines. If a script draws a line that has both its x coordinates at the same bar, different y coordinates, and extends in both directions, the line is vertical. The following example script draws vertical lines every 10 bars:

//@version=6 indicator("Vertical line demo", overlay = true, max_lines_count = 500) float atr = ta.atr(5) if bar_index % 10 == 0 line.new(bar_index, open, bar_index, open + atr, extend = extend.both, color = color.silver)

By plotting histograms

Histogram plots that join a very large number and a very low number appear as vertical lines. Advantages of using histogram plots include that there is no limit on the number of lines. With this method, unlike using line drawings, the user must right-click the price scale and select “Scale price chart only” to avoid distorting the chart vertically. Scripts can plot histograms, like all types of plot, only on the bar where the script is executing or at a *fixed* offset.

The following example script plots a very large number, 10e20, every 10 bars, and sets the base of the histogram to its negative value:

//@version=6 indicator("Histogram demo", overlay = true) plot(bar_index % 10 == 0 ? 10e20 : na, "vLine", color.silver, 1, plot.style_histogram, histbase = -10e20)

By coloring the background

Coloring the background for a single bar displays as a vertical line. Advantages of this method include simplicity, no limit on the number of lines, and no need to adjust the price scale. Disadvantages include no control of the width of the line — it is always exactly one bar wide, and the width scales with the number of bars that display on the chart.

Scripts can change background color only on the bar on which the script is currently executing; offsetting the change is not possible.

//@version=6 indicator("Background color demo", overlay = true) bgcolor(bar_index % 10 == 0 ? chart.fg_color : na)

How can I toggle hline() levels on and off?

Scripts can toggle the display of horizontal levels plotted using hline() in several ways. Conditionally setting the argument of the `price` or `color` parameter to na shows or hides the level in a similar way that setting the `series` or `color` to na does for plotted lines, as described in the entry How do I plot a line using start/stop criteria? above.

Additionally, the `display` parameter of the hline() function can take an input value as its argument. This parameter controls where the line displays.

The following example script demonstrates all of these methods:

//@version=6 indicator("Toggle `hline()` using `na` values", overlay = false) bool showHlineInput1 = input.bool(true, "Show line 1") h1 = hline(price = showHlineInput1 ? 70 : na, color = chart.fg_color) bool showHlineInput2 = input.bool(true, "Show line 2") h2 = hline(price = 50, color = showHlineInput2 ? chart.fg_color : color(na)) bool showHlineInput3 = input.bool(true, "Show line 3") h3 = hline(price = 30, color = chart.fg_color, display = showHlineInput3 ? display.all : display.none)

How can I draw lines or labels into the future?

Individual *plotted* lines and shapes cannot be drawn into the future — only the entire series can be offset. By contrast, scripts can extend any *drawn* lines or boxes, or position drawn labels, at an arbitrary distance beyond the last data point. There are two ways to achieve this: using bar_index and using xloc.bar_time.

Using bar_index

The bar_index built-in variable represents the sequential number of the current bar, starting from zero for the first bar in the chart history and incrementing by 1 for each subsequent bar. Drawing objects with their `xloc` parameter set to xloc.bar_index can use a `bar_index` as their `x` coordinates. If the `xloc` parameter is not specified, it defaults to xloc.bar_index.

To project a certain number of bars into the future or past, simply add or subtract that number from the current bar_index. For instance, `bar_index + 20` positions the object 20 bars into the future.

The following example script draws lines and labels on the most recent pivot high and low, and extends the lines into the future beyond the last bar:

image

//@version=6 indicator("Draw into the future", overlay = true) int offsetInput = input.int(10, "Offset bars", maxval = 500) int pivotLengthInput = input.int(20, "Pivot lookback/look forward", minval = 2) // Initialize the drwaing objects. var line hiLine = line.new(na, na, na, na, color = chart.fg_color, style = line.style_dotted) var line loLine = line.new(na, na, na, na, color = chart.fg_color, style = line.style_dotted) var label hiLabel = label.new(na, na, na, color = color(na), textcolor = chart.fg_color, style = label.style_label_left) var label loLabel = label.new(na, na, na, color = color(na), textcolor = chart.fg_color, style = label.style_label_left) // Calculate the pivots on every bar. float pivotHighPrice = ta.pivothigh(pivotLengthInput, pivotLengthInput) float pivotLowPrice = ta.pivotlow(pivotLengthInput, pivotLengthInput) // Update the drawings when we get a pivot. if not na(pivotHighPrice) line.set_xy1(hiLine, bar_index - pivotLengthInput, pivotHighPrice), line.set_y2(hiLine, pivotHighPrice) label.set_y(hiLabel, pivotHighPrice) label.set_text(hiLabel, str.tostring(pivotHighPrice, format.mintick)) if not na(pivotLowPrice) line.set_xy1(loLine, bar_index - pivotLengthInput, pivotLowPrice), line.set_y2(loLine, pivotLowPrice) label.set_y(loLabel, pivotLowPrice) label.set_text(loLabel, str.tostring(pivotLowPrice, format.mintick)) // Update the position of the

labels and the right end of the lines on the last bar. if barstate.islast hiLine.set_x2(bar_index + offsetInput) loLine.set_x2(bar_index + offsetInput) hiLabel.set_x(bar_index + offsetInput) loLabel.set_x(bar_index + offsetInput)

Note that:

- We update the properties of the drawings each time we get a pivot. In this section, we call the setting functions for the lines and labels in the normal way.
- On the last bar of the dataset, we adjust the x2 point of both lines and the x point of the labels to extend into the future by the user-defined offset amount. In this section, for demonstration purposes, we call the same setting functions as methods by using dot notation syntax.

Using time

Drawing objects can also be positioned based on UNIX time values, by using xloc.bar_time for an object's `xloc` parameter. The time values can be timestamps, or a bar's open time, or any other calculated time.

By using time to position objects, there is no limitation as to how far into the future objects can display. Any valid timestamp positions the object accordingly.

There are some challenges to this method, however:

- Project a specific number of bars into the future using time can be complex. Failing to account for weekends and market holidays can lead to discrepancies in non-24/7 markets. For instance, trying to project 10 hourly bars ahead on a Friday evening might inadvertently position the object during a non-trading weekend slot.
- Bar time is based on UNIX time. When using raw date values, it's important to consider both the exchange's time zone and the user's local settings. Discrepancies can arise due to time zone differences, leading to objects displaying in unexpected positions on the chart. This can be particularly confusing if users of the

script are located in various time zones or if the script is used on symbols from exchanges in different regions.

For help in displaying objects a specific number of time units into the future, refer to the Time Offset Calculation Framework by Pinecoders.

How can I keep only the last *n* number of drawings?

The two most robust and scalable ways to keep only the last *n* number of drawing objects are:

- Using the appropriate built-in `*.all` array.
- Adding drawings to an array and using the array as a queue.

Using a `*.all` array

The quickest and easiest method to limit the number of drawings displayed is to use the built-in `*.all` array for the drawing type. These arrays automatically contain all drawings of that type that currently display on the chart.

The `*.all` arrays are *read-only*. Scripts cannot change the arrays directly by pushing or shifting elements, but they can *update* or *delete* elements in the arrays. The following example script keeps a maximum of 10 labels on the chart. It gets the first label's ID via the array.first() function and then deletes that label with the label.delete function.

//@version=6 indicator("Limit labels using *.all", overlay = true) const int LABEL_LIMIT = 10 label.new(bar_index, high, str.tostring(high)) if label.all.size() > LABEL_LIMIT label.all.first().delete()

Note that:

- The items in the array are in the order that they were drawn, so if a script displays drawings in a different order than it creates them, using a `*.all` array gives unpredictable results.

- Only built-in drawing types have `*.all` arrays available, so if a script uses objects of user-defined types as containers for several drawings, using an array as a queue instead is the only option.
- When the script deletes a label, the label is *also* automatically removed from the label.all array. By contrast, if a script adds drawings to an array manually, it must both remove the object from the array *and* delete it. In this case, array.shift() is usually the best choice, because it removes the first value from an array and returns it for further manipulation.

Using an array as a queue

A more flexible method to keep only the last *n* number of drawings is to use an array as a queue. Each time the script adds a new element to the array, it removes the oldest element. This is possible because each element in an array has a unique *index*, and array indexes *always* begin at zero. If we remove element `0` from an array with several elements, the element that was at index `1` is now at index `0`, the element at index 2 moves down to index 1, and so on.

The following example script uses this behavior to keep a queue of labels and a queue of vertical lines to a defined user-configurable length. We demonstrate two different methods of managing the size of arrays:

- For labels, we create an array of the correct size, and always remove the same number of objects as we add. This method is simpler, but it requires programmers to ensure that objects are always added and removed in the same quantities (for example, by using a function to add and remove elements).
- For lines, we create an *empty* array, append lines to it, and only delete the oldest element if the array exceeds the maximum size. This more robust method is especially suitable for larger arrays, because the size of the array becomes meaningful: it corresponds to the number of — for example — lines drawn on the chart.

//@version=6 indicator("Limit drawings using arrays", overlay = true) int drawingQtyInput = input.int(5, "Quantity of drawings to show", minval = 0, maxval = 50) int drawingIntervalInput = input.int(4, "Draw

```
every n bars", minval = 1, maxval = 10) // Create a label array sized according to the user-selected quantity. var array<label> labelsArray = array.new<label>(drawingQtyInput) if bar_index % drawingIntervalInput == 0 // Every n bars // Draw a label and add it to the *end* of the array labelsArray.push(label.new(bar_index, high, str.tostring(high), style = label.style_label_down, color = color.new(chart.fg_color, 30), textcolor = chart.bg_color)) // Remove the oldest label and delete it. label.delete(labelsArray.shift()) // Create an *empty* line array. var array<line> linesArray = array.new<line>() // Every n bars (offset -1 from the labels), draw a vertical line and add it to the *end* of the array. if (bar_index + 1) % drawingIntervalInput == 0 linesArray.push(line.new(bar_index, high, bar_index, low, color = chart.fg_color, extend = extend.both, width = 3)) // If the array is larger than the number of drawings to display, remove the oldest line and delete it. if linesArray.size() > drawingQtyInput line.delete(linesArray.shift())
```

Is it possible to draw geometric shapes?

Geometric shapes can illustrate patterns, mark zones of interest, or create other visual aids for technical analysis. Scripts can use two main methods for drawing shapes: polylines and lines.

Drawing with polylines

Polylines offer an efficient method for constructing complex shapes on a chart. To draw a shape with polylines, follow these steps:

1. Create an array containing objects of type chart.point that define the vertices of your desired shape.
2. Pass this array to the polyline.new() function. This function draws sequential lines between the points in the order they appear in the array.
3. Choose between straight or curved lines, enabling the creation of a diverse range of shapes.

This method is particularly useful for joining many points with a single object. A single polyline object can join up to 10,000 points, and one script can contain up to 100 polyline objects.

The following example script creates five sets of *interactive points* by placing price and time inputs inline. When the user adds the script to the chart, the script prompts them to place five points on the chart. The script adds the points to an array and passes the array to the polyline.new() function, which connects the points with lines, constructing a shape.

image

//@version=6 indicator("Polylines example", overlay = true) //@variable If `true`, the connections in the polyline are curved lines. bool curvedInput = input.bool(false, "Curve Polyline") //@variable If `true`, connects the first point in the polyline to the last point. bool closedInput = input.bool(true, "Close Polyline") //@variable The color of the space filled by the polyline. color fillcolor = input.color(color.new(color.blue, 90), "Fill Color") // Time and price input pairs for the polyline's points. int p1x = input.time(0, "p1", confirm = true, inline = "p1") float p1y = input.price(0, " ", confirm = true, inline = "p1") int p2x = input.time(0, "p2", confirm = true, inline = "p2") float p2y = input.price(0, " ", confirm = true, inline = "p2") int p3x = input.time(0, "p3", confirm = true, inline = "p3") float p3y = input.price(0, " ", confirm = true, inline = "p3") int p4x = input.time(0, "p4", confirm = true, inline = "p4") float p4y = input.price(0, " ", confirm = true, inline = "p4") int p5x = input.time(0, "p5", confirm = true, inline = "p5") float p5y = input.price(0, " ", confirm = true, inline = "p5") if barstate.islastconfirmedhistory //@variable An array of `chart.point` objects for the new polyline. var array<chart.point> points = array.new<chart.point>() // Push new `chart.point` instances into the `points` array. points.push(chart.point.from_time(p1x, p1y)) points.push(chart.point.from_time(p2x, p2y)) points.push(chart.point.from_time(p3x, p3y)) points.push(chart.point.from_time(p4x, p4y)) points.push(chart.point.from_time(p5x, p5y)) // Add labels for each `chart.point` in `points`. label l1p1 = label.new(points.get(0), "p1", xloc.bar_time, color = na) label l1p2 = label.new(points.get(1), "p2",

xloc.bar_time, color = na) label l2p1 = label.new(points.get(2), "p3", xloc.bar_time, color = na) label l2p2 = label.new(points.get(3), "p4", xloc.bar_time, color = na) label l3p1 = label.new(points.get(4), "p5", xloc.bar_time, color = na) // Create a new polyline that connects each `chart.point` in the `points` array, starting from the first. polyline.new(points, curvedInput, closedInput, xloc.bar_time, fill_color = fillcolor)

Note that:

- The five sets of time and price inputs are *paired* to form interactive points because they share matching `inline` arguments.

Drawing with lines

Lines are simpler than polylines. A line in Pine Script is straight connection between two points. Here's how to use lines for drawing shapes:

- Determine the start and end points for each line, recognizing that every edge of your shape must be delineated separately.
- For each side of the shape, input the precise starting and ending coordinates to create individual line segments.
- Ensure that the end of one line meets the start of the next, to form a closed shape.

An advantage of using separate lines is that scripts can customize the style of each line separately. Additionally, arrays are optional for line management — as opposed to mandatory for polylines — which can result in simpler code in some cases. For large, complex shapes, however, polylines are more convenient.

The following example script creates six sets of *interactive points* by pairing price and time inputs, using the same `inline` arguments. When the script is added to the chart, it prompts the user to click six points on the chart (because the inputs use `confirm = true`). The script joins the points and creates two triangles.

image

//@version=6 indicator("Triangles", "", true) // Create interactive point inputs using pairs of inputs with the same `inline` argument, and the `confirm` parameter. string GRP1 = "Triangle A" int x1AInput = input.time(0, "Point 1", inline = "A1", group = GRP1, confirm = true) float y1AInput = input.price(0, "", inline = "A1", group = GRP1, confirm = true) int x2AInput = input.time(0, "Point 2", inline = "A2", group = GRP1, confirm = true) float y2AInput = input.price(0, "", inline = "A2", group = GRP1, confirm = true) int x3AInput = input.time(0, "Point 3", inline = "A3", group = GRP1, confirm = true) float y3AInput = input.price(0, "", inline = "A3", group = GRP1, confirm = true) string GRP2 = "Triangle B" int x1BInput = input.time(0, "Point 1", inline = "B1", group = GRP2, confirm = true) float y1BInput = input.price(0, "", inline = "B1", group = GRP2, confirm = true) int x2BInput = input.time(0, "Point 2", inline = "B2", group = GRP2, confirm = true) float y2BInput = input.price(0, "", inline = "B2", group = GRP2, confirm = true) int x3BInput = input.time(0, "Point 3", inline = "B3", group = GRP2, confirm = true) float y3BInput = input.price(0, "", inline = "B3", group = GRP2, confirm = true) // @function Constructs a triangle on the chart using three vertices and a specified line color. // @param vx1 (int) The bar time of the first vertex. // @param vy1 (float) The price of the first vertex. // @param vx2 (int) The bar time of the second vertex. // @param vy2 (float) The price of the second vertex. // @param vx3 (int) The bar time of the third vertex. // @param vy3 (float) The price of the third vertex. // @param lineColor (color) The color of the triangle's edges. // @param labelName (string) The text to display in the label at the triangle's peak. // @returns (label) The label to plot the text for the triangle. drawTriangle(int vx1, float vy1, int vx2, float vy2, int vx3, float vy3, color lineColor, string name) => line.new(vx1, vy1, vx2, vy2, xloc = xloc.bar_time, color = lineColor) line.new(vx2, vy2, vx3, vy3, xloc = xloc.bar_time, color = lineColor) line.new(vx1, vy1, vx3, vy3, xloc = xloc.bar_time, color = lineColor) array<int> xValues = array.from(vx1, vx2, vx3) array<float> yValues = array.from(vy1, vy2, vy3) float yMax = array.max(yValues) int xMax = array.get(xValues, array.indexof(yValues, yMax)) label.new(xMax, yMax, name, xloc = xloc.bar_time, color = color(na), textcolor = color.gray) // We draw the shapes only once for efficiency. if barstate.isfirst drawTriangle(x1AInput, y1AInput, x2AInput, y2AInput, x3AInput, y3AInput, color.lime, "A")

drawTriangle(x1BInput, y1BInput, x2BInput, y2BInput, x3BInput, y3BInput, color.fuchsia, "B")

How can I color the chart’s background on a condition detected on the last bar?

The usual way to color the chart background is by calling the bgcolor() function, which colors the chart background *for the bar on which it is called*. The background for a particular bar cannot be changed on later bars, and bars cannot be colored retroactively, using this method.

To color the entire chart background based on a condition detected on the last bar, one workaround is to use a table sized to the visible area of the chart, and color the background of the cell. See this example in the page on tables.

Previous