ChatGPT

Pine Script® v6 Reference Manual

Introduction to Pine Script v6

Pine Script® is TradingView's domain-specific language for creating custom technical indicators and trading strategies [1]. It is designed to be lightweight yet powerful, allowing traders to develop and backtest strategies directly on TradingView charts. Pine Script code is executed server-side, meaning scripts run in the cloud with certain limits on data requests, execution time, memory, and script size to ensure fair resource usage [2]. Pine Script v6 is the latest version, introducing new features and improvements that enhance script flexibility and performance.

Every Pine Script starts by declaring its version and script type. For example, the first line of a Pine v6 script must be `//@version=6` to use version 6 syntax. After that, one of the declaration functions – `indicator()`, `strategy()`, or `library()` – is called to define the script's name and type (indicator or strategy) [3]. For instance:

```
//@version=6
indicator("My Custom Indicator", overlay=true)
```

The above code defines a Pine indicator script named "My Custom Indicator" that overlays on the main chart. Pine Script v6 maintains Pine's ease of use while adding advanced capabilities. In the following sections, we provide a comprehensive reference of Pine Script v6 syntax, keywords, built-in variables, and functions, organized by category for easy navigation.

Language Syntax and Keywords

Pine Script's syntax is designed for simplicity. It shares some similarities with languages like Python (for indentation and logical operators) and has specialized constructs for trading context. Key language elements include:

- **Annotations**: Annotations start with `//@` and provide meta instructions. The most important is `//@version=6` which sets the compiler to Pine v6. Others include `//@author`, `//@description`, or documentation tags like `//@param` and `//@return` in user-defined functions for clarity (these do not affect execution but document the code).

- **Script Declarations**: Use `indicator(title, options...)` for indicator scripts or `strategy(title, options...)` for strategy scripts as the first callable statement [3]. These functions set the script's title and properties (e.g. `overlay=true` if the indicator should plot on the price chart, or strategy-specific settings like `initial_capital` for strategies).

- **Variables and Assignment**: Variables are declared by simply assigning a value. By default, assignments produce **series** variables that have a value for each bar. For example: `myVar = close * 2`. To create a variable that persists its value across runs (static), Pine uses the `var` keyword (e.g. `var count = 0`) which initializes once and retains value between bars. Pine v6 also supports **mutable** variables with `:=` for reassignment [4]. Example:

```
var int counter = 0        // persists state
counter += 1               // same as counter := counter + 1
```

- **Data Types**: Pine v6 supports simple types like `int` (integer), `float` (floating-point number), `bool` (boolean), `color`, `string`, along with **series** variations of these (e.g. `series int`) which represent a sequence of values over bars. Pine also allows compound types:

- **Arrays**: dynamic lists of values (e.g. `array.new_float(10)` creates float array of length 10).
- **Matrices**: 2D arrays (tables of data).
- **Maps**: key-value mapping structure.
- **Tuples**: fixed-size ordered collections that can hold multiple values of possibly different types (useful for returning multiple values from a function).

- **User-Defined Types (UDT)**: custom structured types defined with the `type` keyword, grouping multiple fields.

- **Reserved Keywords**: Pine Script reserves certain keywords for language use:

- **Control structures**: `if`, `else`, `for`, `while`, `break`, `continue`.
- **Declaration**: `var`, `type` (for UDT), `func`/`function` (for defining functions if applicable; in Pine, functions are usually defined with the `=>` syntax, e.g. `myFunc(x) => ...`).
- **Logical literals**: `true`, `false`, `na` (the special *not available* value).
- **Operators**: Arithmetic operators `+ - * / %` for addition, subtraction, multiplication, division, modulo; Comparison operators `== != < <= > >=`; Logical operators `and`, `or`, `not` for boolean logic. Pine v6 uses **short-circuit evaluation** for `and`/`or`, meaning expressions are evaluated only as far as needed to determine the result [5] [6].

- **Other**: `import` (to include code from a published library script), `export` (to expose functions/ types from a library), `strategy` and `indicator` (also the names of built-in functions as noted).

- **Indentation and Blocks**: Pine Script uses indentation to define code blocks under `if`, `for`, `while`, etc., similar to Python. No braces are used; the scope is determined by indent level. Example:

```
if close > open
    barColor = color.green
else
    barColor = color.red
```

Here the assignments to `barColor` are within the if/else blocks due to indentation.

- **Ternary Operator**: Pine supports a ternary conditional operator of the form `condition ? valueIfTrue : valueIfFalse`. This provides a shorthand for simple conditional assignments. For example: `dir = close >= open ? 1 : -1` sets `dir` to 1 if the bar is bullish, else -1. (Note: In v6, the condition of a ternary must be boolean, since non-bool can no longer implicitly cast to bool [7].)

- **Loops**: Pine v6 supports `for` loops to iterate over a fixed range or array, and `while` loops for conditional repetition. For example:

  ```
  var float sum = 0
  for i = 0 to 9
      sum += close[i]  // sum of last 10 closing values
  ```

  This loop uses `i = 0 to 9` to iterate through indices. `break` can exit a loop early, and `continue` skips to the next iteration.

- **Functions**: User-defined functions are declared without a special keyword, typically using a concise syntax:

  ```
  myFunc(x, y) =>
      // function body
      x + y
  ```

  The `=>` introduces the function’s expression or block. Functions can return values (as above) or perform actions. In v6, functions can also return multiple values using tuples (e.g., `return [val1, val2]`). Use `export` in a library script to make a function available for import by others.

- **Comments**: Single-line comments start with `//` and extend to the end of line. Pine v6 also supports **documentation comments** that begin with `//@`. These are used to document functions, parameters, or types (they don’t affect runtime). There are no multi-line comment delimiters; use `//` on each line for multi-line comments.

Built-in Variables and Constants

Pine Script provides a variety of built-in variables (predefined series or constants) that supply important chart data or state information. Here is a breakdown of key built-in variables in Pine Script v6:

- **Price and Volume Series**: On any chart timeframe, the following series variables give you OHLCV data for each bar:
- `open`, `high`, `low`, `close` – the open, high, low, and close price of the current bar (series of type float).
- `volume` – the volume of the current bar (series float).
- `bar_index` – index of the current bar (0 for the first bar on the chart, increasing by 1 each bar, series int).

- **Aggregate price series**: Pine defines common aggregate prices:

  - `hl2` – (High+Low)/2, the average of high and low [8] [9].
  - `hlc3` – (High+Low+Close)/3, the typical price.
  - `ohlc4` – (Open+High+Low+Close)/4, the average price of the bar.

    - `hlcc4` – (High+Low+Close+Close)/4, a weighted typical price (close counted twice).

- **Time and Date**: Pine offers several ways to work with time:

- `time` – UNIX timestamp of the bar's open time (series int, in milliseconds since epoch).
- `timenow` – Current real-time UNIX timestamp (updates even on the last real-time bar).
- `time_close` – (If available) the closing time of the bar. If not directly provided, it can be derived (e.g., `time + interval`).
- **Date-part variables/functions**: You can get parts of the bar's date/time through built-ins such as `year()`, `month()`, `dayofweek()`, `dayofmonth()`, `hour()`, `minute()`, etc., which return integers for the current bar's timestamp components [10] [11]. For example, `year` gives the year, `month` the month (1-12), `dayofweek` the weekday (1-7), etc. *(In v6, these are often referenced as* `time.year`*,* `time.month`*, etc. in documentation, but they effectively provide the current bar's date components.)*

- **Time zone info**: `time.isdst` (daylight savings in effect or not), `time.timezone` (exchange timezone name), `time.tzoffset` (timezone offset in seconds) [12].

- **Bar State**: The `barstate` namespace provides boolean flags about the bar's context [13]:

- `barstate.isfirst` – true if this is the first historical bar in the dataset.
- `barstate.ishistory` – true if the script is currently on a historical (closed) bar.
- `barstate.isrealtime` – true if on the real-time (currently forming) bar.
- `barstate.islast` – true if this is the last bar on the chart (can be real-time or last historical if chart is not updating).
- `barstate.islastconfirmedhistory` – true if this bar is the last *confirmed* historical bar (i.e. the bar before the real-time bar).

- `barstate.isnew` – true if the current execution is on a new bar (just opened bar).

- **Chart and Symbol Information**:

- **Chart attributes** (`chart` namespace):
    - `chart.left_visible_bar_time` – the UNIX time of the first visible bar on the chart's left side [14]. This can be used to find the earliest visible bar's timestamp.
    - `chart.is_standard` – true if the chart type is standard (regular candles/bars) vs. a non-standard chart type (like Renko, Kagi, etc.) [15].
    - `chart.fg_color` and `chart.bg_color` – the chart's foreground and background colors (color constants).

- **Symbol info** (`syminfo` namespace):

    - `syminfo.tickerid` – the full identifier of the current symbol (including exchange). For example `"NASDAQ:AAPL"` [16].
    - `syminfo.ticker` – the short ticker name of the current symbol (e.g. `"AAPL"`).
    - `syminfo.basecurrency` – base currency of the symbol (e.g. `"USD"` for a stock priced in USD).
    - `syminfo.pointvalue` – the dollar (or base currency) value of one whole unit move of price (important for futures).
    - `syminfo.mintick` – minimum price movement (tick size) of the symbol.

    - `syminfo.session` – the trading hours session string for the symbol (e.g. `"0930-1600"`).
    - `syminfo.type` – the type of security (stock, forex, crypto, etc.).
    - *New in v6*: `syminfo.main_tickerid` – the main chart's ticker ID, especially useful in multi-symbol contexts or libraries to always reference the primary chart symbol [17].
    - `syminfo.mincontract` – smallest tradable contract size for the symbol (introduced in v6 for futures/options) [17].
    - `timeframe.period` – the chart's current resolution as a string (e.g. `"D"` for daily, `"60"` for 60-minute) [17].
    - `timeframe.isintraday` – true if the chart timeframe is intraday.
    - *New in v6*: `timeframe.main_period` – the main chart's timeframe, regardless of where the code is executed (helpful in multi-timeframe scripts) [17].

- **Strategy Performance Variables** (available in strategy scripts): Pine provides a range of read-only variables in the `strategy` namespace that give information about backtest performance and current strategy state [18]:

- `strategy.equity` – current equity of the strategy (starting balance plus net profit/loss of closed trades plus open trade profit).
- `strategy.openprofit` – current unrealized profit of open positions.
- `strategy.closedprofit` – total profit from closed trades.
- `strategy.initial_capital` – the starting capital specified for the strategy [19].
- `strategy.netprofit`, `strategy.grossprofit`, `strategy.grossloss` – net profit, gross profit, gross loss from trades.
- `strategy.totaltrades` – total number of trades taken.
- `strategy.wintrades`, `strategy.losstrades` – count of winning and losing trades [19].
- `strategy.max_drawdown` – maximum drawdown observed.
- `strategy.position_size` – current open position size (number of contracts or shares, positive for long, negative for short) [20].
- `strategy.position_avg_price` – average entry price of the current open position [20].
- `strategy.market_position` – the current market position ("long", "short", or "flat").

- These variables update as trades are executed in backtesting. They are useful for position sizing logic, money management, or to display performance metrics.

- **Constants**: Pine defines various constants (often as enumerations) to use as parameter values:

- **Color constants**: e.g. `color.black`, `color.blue`, `color.red`, `color.green`, etc., representing common colors. These have predefined RGB values. *(Note: In v6 some color constants were adjusted to better match the TradingView palette [21].)* There are also `color.white`, `color.yellow`, etc., and semi-transparent versions like `color.new(color.red, 50)` to adjust transparency.
- **Plot style constants**: for `plot()`'s `style` parameter, e.g. `plot.style_line`, `plot.style_histogram`, `plot.style_area`, `plot.style_cross`, etc., to control how a series is plotted (line, histogram, area under curve, cross markers, etc.).
- **Shape/style constants**: for drawing functions, e.g. `label.style_label_up`, `label.style_label_down` (for label shapes), or `label.style_circle`, `label.style_none` etc.; `line.style_solid`, `line.style_dashed` for line styles; `shape.triangleup`, `shape.triangledown` for `plotshape()` shapes; `location.top`, `location.bottom` for positioning plot shapes or chars; `size.small`, `size.large`, etc. for sizes of text and shapes.

- **Session constants**: e.g. `session.regular` might define the regular trading hours vs `session.extended` for after-hours (used with inputs or `input.session`).
- **Resolution constants**: e.g. `resolution.minute`, `resolution.hour`, etc., though typically resolution is given as string like `"60"` or `"D"`.
- These constants make code more readable when configuring function parameters.

With variables and constants available, scripts can reference chart data and state directly, e.g. `close` for the latest closing price, `year()` for the current bar's year, or `syminfo.tickerid` for the symbol.

Built-in Functions by Category

Pine Script v6 provides hundreds of built-in functions that perform calculations or actions so you don't have to code them from scratch [22] [23]. They range from technical indicator calculations to utility functions, plotting, and trade execution. Below is a comprehensive list of Pine v6 built-in functions, grouped by purpose, with descriptions and key parameters:

Technical Analysis Functions (`ta.` namespace)

These functions compute popular technical indicators and related values. Most are in the `ta` namespace (short for "technical analysis"). **Indicator Functions** return indicator values (often series floats), while **Helper Functions** assist in analyzing series (crossovers, highest/lowest, etc.). Common technical analysis functions include [23]:

- **Moving Averages**:
- `ta.sma(source, length)` – Simple Moving Average over `length` bars [24]. Returns the average of the `source` series for the last `length` periods.
- `ta.ema(source, length)` – Exponential Moving Average. Places higher weight on recent values.
- `ta.wma(source, length)` – Weighted Moving Average (linearly weighted).
- `ta.vwma(source, length)` – Volume-Weighted Moving Average.
- `ta.hma(source, length)` – Hull Moving Average (less lag).
- `ta.rma(source, length)` – Rolling Moving Average (Wilders smoothing, used in RSI).

- *Parameters:* Typically these take a **series float** `source` (e.g. closing prices) and an **int** `length` (period). They return a **series float** (the MA value each bar).

- **Trend and Oscillators**:

- `ta.rsi(source, length)` – Relative Strength Index, a momentum oscillator showing overbought/oversold conditions [24]. *Params:* `source` series (often `close`), `length` (period, e.g. 14). Returns a series float (0-100 range typically).
- `ta.stoch(kLength, dLength, smooth)` – Stochastic Oscillator (with optional smoothing). Returns %K or %D lines.
- `ta.macd(source, fastLength, slowLength, signalLength)` – Moving Average Convergence/Divergence. Returns a tuple or series of MACD line (fast EMA - slow EMA) and signal line (EMA of MACD) – in Pine, use `ta.macd()` and possibly get components via separate functions or by capturing the tuple if provided.
- `ta.supertrend(factor, atrLength)` – Supertrend indicator (trend-following overlay) [25]. Returns a series float indicating the Supertrend value.

- `ta.bb(source, length, mult)` – Bollinger Bands. Typically returns a tuple or separate functions for upper/lower bands.
- `ta.kc(source, length, mult)` – Keltner Channels (uses ATR for channel distance).
- `ta.atr(length)` – Average True Range (volatility measure).
- `ta.adx(length)` – Average Directional Index (trend strength).
- `ta.dmi(length)` – Directional Movement (returns +DI, -DI, possibly ADX).
- `ta.obv(source)` – On-Balance Volume [26], cumulative volume indicator.
- `ta.mfi(length)` – Money Flow Index [27], volume-weighted RSI variant.
- `ta.cci(source, length)` – Commodity Channel Index.
- `ta.mom(source, length)` – Momentum (price change over `length` bars).
- `ta.roc(source, length)` – Rate of Change (percentage change).
- `ta.ppo(source, fastLength, slowLength)` – Percentage Price Oscillator (like MACD but expressed as percentage).
- `ta.stochrsi(source, length)` – Stochastic RSI.
- `ta.uo(fastLength, mediumLength, slowLength)` – Ultimate Oscillator (combines multiple lengths).

- **Note**: Many indicator functions have default parameters or multiple return values. For example, `ta.macd()` in Pine might actually return a tuple of three series: MACD, signal, histogram. Always check the function's reference signature for details (the Reference Manual shows parameter lists and return types [28]).

- **Highest/Lowest and Range**:

- `ta.highest(source, length)` – Highest value of `source` over the last `length` bars [29]. Returns the series float of the highest value.
- `ta.lowest(source, length)` – Lowest value over last `length` bars.
- `ta.highestbars(source, length)` – The number of bars since the highest value of `source` over `length` bars (returns int index offset).
- `ta.lowestbars(source, length)` – Bars since lowest value.
- `ta.barssince(condition)` – Number of bars since `condition` was true [29]. If condition never true in lookback, returns a large number or na.
- `ta.range(source, length)` – Range (max-min) of `source` over `length` period.
- `ta.dev(source, length)` or `ta.stdev(source, length)` – Standard deviation of `source` over period.

- `ta.var(source, length)` – Variance.

- **Crossover Functions**:

- `ta.cross(source1, source2)` – True (bool) if `source1` crosses `source2` *this bar* (either upward or downward) [30].
- `ta.crossover(source1, source2)` – True if `source1` crossed above `source2` from below this bar [31].
- `ta.crossunder(source1, source2)` – True if `source1` crossed below `source2` from above [32].

- These functions are *series bool* outputs useful for signals (they are true only on the bar a crossover event occurs).

- **Value When**:

- `ta.valuewhen(condition, expression, occurrence)` – Returns the value of `expression` at the bar where `condition` was true *n* occurrences ago (occurrence=1 for most recent, 2 for second most recent, etc). This is useful to retrieve past values that met a condition (for example, the price at the last crossover) [33] [34]. In v6, this is in the `ta` namespace (`ta.valuewhen`).

- **Change and Momentum**:

- `ta.change(source)` – Change in value from prior bar (same as `source - source[1]`).
- `ta.percentchange(source)` – Percentage change from previous bar.
- `ta.cum(source)` – Cumulative sum of `source` over time (adds up values each bar) [35].
- `ta.running(total)` – Possibly running total (if exists).
- `ta.tr` – True Range (difference considering previous close).

- `ta.tr(true)` – True Range with a `true` argument uses high-low if previous close is na [36].

- **Statistical**:

- `ta.correlation(source1, source2, length)` – Pearson correlation coefficient between two series over given length [37].
- `ta.covariance(source1, source2, length)` – Covariance of two series.
- `ta.linreg(source, length, offset)` – Linear regression (least squares) of series (slope or intercept depending on usage).
- `ta.percentrank(source, length)` – Percentile rank of the latest value within the last `length` values.
- `ta.percentile_linear_interpolation(source, length, percentile)` – Percentile value with linear interpolation [38].

- `ta.percentile_nearest_rank(source, length, percentile)` – Percentile using nearest rank method.

- **Other Indicators**: Pine covers virtually all classical indicators:

- `ta.ad` (Accumulation/Distribution line), `ta.accdist()` might be the same.
- `ta.obv()` – On Balance Volume [26].
- `ta.mfi()` – Money Flow Index [27].
- `ta.tsi()` – True Strength Index (momentum oscillator) [39].
- `ta.cci()` – Commodity Channel Index.
- `ta.wpr()` – Williams %R (momentum oscillator) [40].
- `ta.sar()` – Parabolic SAR (stop-and-reverse).
- `ta.ichimoku()` – Ichimoku Cloud (often returns multiple outputs: tenkan, kijun, senkou spans).
- `ta.pivotlow()` / `ta.pivothigh()` – Identify pivot points (local minima/maxima) [41].
- and many more.

Usage Example: To calculate a 14-period RSI of closing prices and a 9-period EMA of RSI:

```
rsi14 = ta.rsi(close, 14)
rsiSignal = ta.ema(rsi14, 9)
```

```
plot(rsi14, color=color.blue)
plot(rsiSignal, color=color.orange)
```

This uses `ta.rsi` and `ta.ema` and plots them.

Note: Many `ta.` functions expect a **series float** input and produce a series float output (or series bool for crossovers). If you conditionally call them inside an `if`, be aware that some functions that rely on historical data should be called on every bar for consistency [42] [4]. If they're not, the Pine compiler may warn you to move them outside the conditional (to global scope) to avoid using incomplete history. For example, always call moving averages every bar, or store them in a variable at global scope.

Math and Numerical Functions (`math.` namespace)

The `math` namespace provides common mathematical functions, eliminating the need to implement basic math routines. These functions typically take numeric arguments (int or float, often can be series or constants) and return a numeric result. Some key math functions include [43] [44]:

- **Basic Math**:
- `math.abs(x)` – Absolute value of `x` (returns positive value) [44].
- `math.min(a, b)` / `math.max(a, b)` – Minimum or maximum of two numbers (there are overloads to handle more than two inputs as well) [45].
- `math.round(x)` – Round to nearest integer. Also variants: `math.floor(x)` (round down), `math.ceil(x)` (round up) [46].
- `math.sqrt(x)` – Square root of `x` [47].
- `math.log(x)` – Natural logarithm of `x` [47].
- `math.log10(x)` – Base-10 logarithm.
- `math.exp(x)` – e^x (exponential function) [48].
- `math.pow(base, exponent)` – Power function (base^exponent) [49].

- `math.sign(x)` – Sign of x: returns 1 if x > 0, -1 if x < 0, 0 if x == 0.

- **Trigonometry**:

- `math.sin(angle)` – Sine of angle (angle in radians).
- `math.cos(angle)` – Cosine [48].
- `math.tan(angle)` – Tangent.

- Inverse trig: `math.asin(x)`, `math.acos(x)`, `math.atan(x)` – arc-sine, arc-cosine, arc-tangent of x (returns angle in radians) [50].

- **Other**:

- `math.round_to_mintick(price)` – Rounds `price` to the nearest valid price increment (uses symbol's tick size) [51]. Useful for rounding calculated prices to actual tradable values.
- `math.random(min, max)` – Random number generator [52]. It might return a float between 0 and 1 if used with no parameters, or allow specifying a range. (Be cautious: using random in a script can make results non-deterministic in backtesting).
- `math.avg(x, y)` – Average of two numbers ((x+y)/2).
- `math.pi` – Constant for π (3.14159...).
- `math.e` – Constant for e (2.71828...).

Most math functions accept both **simple** (literal) and **series** arguments, and the output will be series if any input is series. They return a **float** (for trig and logs) or the same type/precision as inputs for basic ops. These functions **do not have side effects**; they purely compute values and can be used in expressions.

Example: To normalize a price change by ATR:

```
change = close - close[1]
normChange = change / math.max(ta.atr(14), 1)
```

Here we use `math.max` to avoid division by zero, comparing ATR to 1 (assuming ATR won't be below zero).

String Functions (`str.` namespace)

The `str` namespace contains functions for working with text strings, helpful for labels, tables, or alert messages [53]. Key string functions:

- `str.tostring(x, format)` – Convert value `x` to string. Accepts optional `format` to format numbers [53]. For example, `str.tostring(close, "#.##")` might format the close price with two decimals. `x` can be int, float, bool, color, etc., and the function returns a string representation.
- `str.format(format_string, value1, value2, ...)` – C-style string formatting [53]. For example: `str.format("Hello {} - {}", name, value)` replaces placeholders with provided values.
- `str.length(text)` – Returns the length of string `text` [53].
- `str.substring(text, offset, length)` – Extract substring from `text` starting at `offset` for `length` characters.
- `str.indexof(text, search)` – Find the index of substring `search` in `text` (returns -1 if not found).
- `str.contains(text, search)` – Returns true if `search` substring is found in `text`.
- `str.replace(text, target, replacement)` – Replace occurrences of `target` in `text` with `replacement`.
- `str.split(text, delimiter)` – Splits a string into an array of substrings around a `delimiter`.
- `str.tonumber(text)` – Convert numeric string to a number (float or int) [53]. If the text isn't purely numeric, result is `na`.
- `str.char(code)` – Returns a one-character string with the given Unicode code point (useful to get special characters by code).

These functions are especially useful in constructing dynamic labels or alerts. **Example**:

```
label.new(bar_index, high, "Close: " + str.tostring(close, "#.00"),
yloc=yloc.abovebar)
```

This will create a label above the bar showing the close price formatted with two decimals. Here `str.tostring` is used to format the number as text.

Input Functions (`input` namespace)

Input functions allow you to create user-adjustable settings in your script's interface (the *Settings/Inputs* dialog). The primary function is `input()` with different type signatures based on the `defval` (default value) you provide [54]. These functions do not plot or calculate values per se; they define adjustable parameters. Common usage patterns:

- **Generic input**: `myInput = input(defval, title="Description", minval=..., maxval=..., step=..., options=..., inline="group name")`. The return type of `input()` matches the type of `defval`:
- If `defval` is a boolean (`true/false`), it creates a checkbox input and returns an **input bool** [55].
- If `defval` is an integer, returns **input int**.
- If float, **input float**.
- If string, **input string**.
- If a color (e.g. `color.red`), returns **input color**.
- If an expression like `close` (series), the input will be of type series (allowing a user to pick a series from dropdown, like open/high/low/close) [56].

Example:

```
length = input(14, "Length", minval=1)
showSignal = input(true, "Show Signals")
src = input(close, "Source")
```

This creates an integer input for length (default 14), a boolean checkbox "Show Signals", and a drop-down to choose a source series (default close) [57] [58].

- **Type-specific input functions**: Pine v6 also provides convenience functions for certain types (these are essentially the same as using `input(defval)` with a given type, but named clearly):
- `input.bool(defval, title="...")` – Boolean input [54].
- `input.int(defval, title="...")` – Integer input.
- `input.float(defval, title="...")` – Float input.
- `input.string(defval, title="...")` – String input (text field or dropdown if `options` provided).
- `input.color(defval, title="...")` – Color picker input [54].
- `input.symbol(defval, title="...")` – Symbol picker (lets user choose a ticker).
- `input.timeframe(defval, title="...")` – Timeframe selector (e.g. default `"D"` for Daily).
- `input.session(defval, title="...")` – Session selector (trading session times).
- `input.source(defval, title="...")` – Similar to input of type series (choose from open, high, low, etc. or another indicator's output).

All `input` functions return a special *input type* which behaves like a constant (it will have the same value on all bars until changed by user). Internally, Pine treats inputs with an `input` qualifier and you use them as you would constants in calculations.

Example usage in code:

```
length = input.int(20, "MA Length", minval=1)
price = input.source(close, "Source")
ma = ta.sma(price, length)
plot(ma, "Moving Average", color=color.blue)
```

This script will plot a moving average where the user can adjust the length and the price source via the settings dialog.

Color Functions (`color.` namespace)

While colors in Pine are often handled as constants or by directly specifying (e.g., `color.red`), the `color` namespace has a few functions to create or manipulate colors [59] :

- `color.new(baseColor, transparency)` – Creates a new color based on `baseColor` but with adjusted transparency [59] . `transparency` is an integer 0-100 (0 = opaque, 100 = fully transparent). For example, `color.new(color.red, 50)` gives a semi-transparent red.
- `color.rgb(r, g, b, transparency)` – Create a color from RGB components (0-255 each) and optional transparency. This allows custom colors not pre-defined.
- `color.from_gradient(val, val_min, val_max, color_min, color_max)` – Produces a color by linearly interpolating between `color_min` and `color_max` based on `val` in the range [val_min, val_max] [59] . Useful for heatmaps or conditional coloring.
- `color.hsv(h, s, v, transparency)` – If available, create color from HSV model.

Additionally, you have built-in color constants as mentioned. Many plotting functions also allow specifying color by conditional expressions. The above functions, especially `color.new` and `color.from_gradient`, are handy for dynamic coloring (e.g., making a color gradually change from green to red based on a value).

Example:

```
// Fade the plot color based on volume intensity
volPercent = volume / ta.highest(volume, 100)
plot(close, color = color.from_gradient(volPercent, 0, 1, color.blue,
color.orange))
```

This will color the price plot between blue and orange depending on current volume relative to max volume of last 100 bars.

Plotting and Chart Visualization Functions

These functions produce graphical output on the chart (lines, shapes, colors). They generally don't return meaningful values (or you typically ignore their return) – they are used for their **side effects** (drawing on the chart) [60] [61] . The main plotting functions include:

- **Primary Plot Functions**:
- `plot(series, title="", color=..., linewidth=..., style=..., join=..., trackprice=..., display=...)` – Plots a line (or other style) on the chart for the given

series of values. This is the most common function to draw indicator lines. [62] It returns a `plot` ID (which can be captured for advanced use, but usually not needed).

  - *Parameters:* `series` (float or bool series; bool will plot 1/0); `title` (legend name), `color`, `linewidth`, `style` (e.g. line, histogram, area, circles), `transp` (transparency), `offset` (bars shift), `join` (whether to join plots with same name to create gaps), `trackprice` (extend last value to right scale), `display` (control which pane – e.g. `display.none` to hide, or default main pane).
  - **Example**: `plot(ta.sma(close, 50), "SMA50", color=color.yellow)`

- `plotshape(series, title="", style=..., location=..., text=..., textcolor=..., size=..., color=...)` – Plots a shape (like an icon) on bars where `series` is true (or non-na) [63] . Use for marking signals.
  - *Parameters:* `series` (bool or condition for where to plot), `style` (shape type: e.g. `shape.triangleup`, `shape.cross`, etc.), `location` (above or below bar, top or bottom of pane), `text` (string to display), `textcolor`, `size` (e.g. `size.tiny`, `size.small`), `color` (fill color of the shape).
  - **Example**: `plotshape(crossover(maFast, maSlow), title="Buy Signal", style=shape.triangleup, location=location.belowbar, color=color.green, size=size.small)`
- `plotchar(series, title="", char=..., location=..., text=..., color=...)` – Similar to plotshape but allows plotting a custom character or symbol (specified by the `char` parameter, e.g. `" "` or `"*"`) at specified locations. Used for more textual markers on chart.
- `plotbar(open, high, low, close, title="", color=...)` – Plots OHLC bars (vertical line with ticks) given four series. Useful if you want to display synthetic bars (e.g. reconstructed from higher timeframe data).
- `plotcandle(open, high, low, close, title="", color=..., wickcolor=..., bordercolor=...)` – Plots candle sticks given OHLC series. Color parameters for body and optionally wicks/borders.

- `plotarrow(series, title="", colorUp=..., colorDown=...)` – Draws up/down arrows based on the sign of `series` (positive values = up arrow, negative = down arrow). If series is zero or na, no arrow. Often used to mark signals in a simpler way (e.g., `plotarrow(close - open)` to mark up days vs down days with arrows).

- **Fills and Backgrounds**:

- `fill(plot1, plot2, color=..., title="")` – Fills the area between two plot lines (identified by their plot ID or by calling `plot` and storing the return) [64] . Used to create shaded regions (e.g., highlight between upper and lower bands).
  - *Parameters:* Two plot references, and `color` (which can have transparency).
  - **Example**:

```
p1 = plot(ta.sma(close, 20), color=color.green)
p2 = plot(ta.sma(close, 50), color=color.red)
fill(p1, p2, color=color.new(color.blue, 90))
```

  This shades the area between the 20 and 50 period SMA with a translucent blue.
- `bgcolor(color, title="", editable=...)` – Fills the chart's background for the current bar with the given color [64] . If used on a series of colors (where some bars might be `na` color), it conditionally colors the background. This is good for highlighting regions (e.g., different background when RSI is overbought).

- `barcolor(color)` – Colors the candlesticks or bars themselves conditionally [65]. For example, `barcolor(color.green)` will make all bars green; typically used with conditions: `barcolor(close > open ? color.green : color.red)`.

- **Horizontal Lines**:

- `hline(price, title="", color=..., linestyle=..., linewidth=...)` – Draws a horizontal line at the given price level. Returns an `hline` object (which generally isn't used further).
- You can create constant reference lines (like pivot or threshold levels) using hline.

These plotting functions allow creating the visual output of indicators. Multiple plots can be added; each will appear either in the same pane or in separate sub-panels depending on script type and `overlay` setting.

Example – Plotting in action:

```
plot(close, "Price", color=color.white)  // plot price
plot(ta.ema(close, 9), "EMA9", color=color.orange)  // plot a fast EMA
plotshape(crossover(ta.ema(close,9), ta.ema(close,21)),
          title="Bullish X", style=shape.triangleup,
location=location.belowbar, color=color.lime,
          text="Buy")
bgcolor(rsi14 > 70 ? color.new(color.red, 80) : na)  // highlight overbought
zones
```

This will draw the price and EMA lines, mark bullish crossovers with an up triangle and label "Buy", and shade the background when RSI > 70.

Drawing Objects: Labels, Lines, Boxes, and Tables

Pine Script lets you draw custom graphics on the chart via objects. Unlike basic plot functions, these objects persist and can be created or modified dynamically. They include **labels**, **lines**, **boxes**, and **tables**. Each of these is referenced by an ID after creation (so you can update or delete them). In Pine v6, object functions can be called in two ways: via the namespace (e.g. `label.new()`) or using **method syntax** on the object ID (a new convenience in v6, allowing code like `myLabel.set_text("New")` where `myLabel` is an ID) [66]. Key functions:

- **Labels (`label` namespace)**: Labels are text boxes that can be placed at specified coordinates (bar index and price level) on the chart. Common functions:
- `label.new(x, y, text="", xloc=..., yloc=..., color=..., style=..., textcolor=..., size=..., textalign=..., tooltip=...)` – Creates a new label [67]. Returns a **label ID** which can be stored in a variable for later reference.
    - *Parameters:*
    - `x` (int/float) – x-coordinate (bar index or timestamp depending on `xloc`).
    - `y` (float) – y-coordinate (price or pixel offset depending on `yloc`).
    - `text` – string to display.
    - `xloc` – either `xloc.bar_index` (treat x as bar index) or `xloc.timestamp` (treat x as time in milliseconds).

    - `yloc` – position relative to price: `yloc.abovebar`, `yloc.belowbar`, `yloc.price` (exact price), or `yloc.panel` (for fixed pixel positioning in the pane).
    - `color` – background color of label.
    - `style` – shape of the label background (rectangle, circle, label_up, label_down, etc).
    - `textcolor` – color of the text.
    - `size` – text size (`size.tiny` to `size.large` or even actual point sizes in v6 with new text formatting).
    - `textalign` – alignment of text within the label (e.g. `text.align_left`, `text.align_center`).
    - `tooltip` – hover text to show when cursor is on the label.
    - **Example**: `lbl = label.new(bar_index, high, text="Swing High", yloc=yloc.abovebar, color=color.black, textcolor=color.white)`
- `label.delete(label_id)` – Removes the specified label from the chart.
- `label.set_text(label_id, text)` – Change the text of an existing label.
- `label.set_xy(label_id, x, y)` – Move the label to a new position [68].
- `label.set_color(label_id, color)` – Change background color.
- `label.set_textcolor(label_id, color)` – Change text color.
- `label.set_size(label_id, size)` – Change text size.
- `label.set_style(label_id, style)` – Change the style (shape) of the label.
- `label.get_text(label_id)` – Retrieve current text (if needed).
- *Method syntax:* If `lbl` is a label id variable, you can call e.g. `lbl.set_text("New")` as shorthand for `label.set_text(lbl, "New")` in Pine v6.

Labels are great for annotating specific bars or points with text. Remember to limit the number of labels via `max_labels_count` in the indicator declaration if you create many (the default max is limited).

- **Lines (`line` namespace)**: Lines connect two points on the chart.
- `line.new(x1, y1, x2, y2, xloc=..., extend=..., color=..., style=..., width=...)` – Creates a new line from `(x1, y1)` to `(x2, y2)`. Returns a **line ID**.
    - `extend` can be `extend.none`, `extend.left`, `extend.right`, or `extend.both` to extend the line infinitely in one or both directions.
    - Other params are similar to label: coordinate reference (bar index or timestamp, price coordinates), color, style (solid, dashed, dotted), width (thickness).
    - Example: `ln = line.new(bar_index, low, bar_index+5, low+10, extend=extend.right, color=color.yellow)`.
- `line.delete(line_id)` – Remove the line.
- `line.set_xy1(line_id, x1, y1)` / `line.set_xy2(line_id, x2, y2)` – Move line endpoints.
- `line.set_color(line_id, color)`, `line.set_width(line_id, width)`, `line.set_style(line_id, style)` – Adjust appearance.

- Lines are useful for trend lines, connecting highs/lows, marking custom support/resistance, etc.

- **Boxes (`box` namespace)**: Boxes draw rectangles.

- `box.new(x1, y1, x2, y2, xloc=..., extend=..., color=..., border_color=..., border_width=..., border_style=...)` – Creates a box defined by top-left `(x1,y1)` and bottom-right `(x2,y2)` corners. Returns a **box ID**.
    - Can extend similarly, e.g. extend to infinity in one direction if desired.
    - Colors: fill color and border color/width/style.

- `box.delete(box_id)`.
- `box.set_position(box_id, x1, y1, x2, y2)` – Resize/move the box in one go.
- `box.set_color`, `box.set_border_color` etc. – Change styling.

- Use boxes to highlight regions, such as a consolidation area or time window on the chart.

- **Tables (`table` namespace)**: Tables allow placing a grid of text (like a small spreadsheet or dashboard) on the chart, introduced in Pine v4/v5.

- `table.new(position, columns, rows, border_width=..., border_color=..., frame_color=..., bgcolor=...)` – Creates a table at a given fixed position on the chart (not anchored to bars). The `position` is a constant like `position.top_left`, `position.bottom_right` of the pane. You specify number of columns and rows. Returns a **table ID**.
- `table.cell(table_id, column, row, text, text_color=..., text_size=..., text_align=..., bgcolor=...)` – Sets the content of a cell in the table (by indices). This is how you populate or update the table cells.
- Tables remain until updated or cleared; they are great for showing multiple values in a structured way (e.g. custom metrics or multi-symbol quotes).

Note: All these objects (label, line, box, table) count towards resource limits (max labels, max lines, etc.). Use them judiciously and delete ones you no longer need (e.g., when a new bar comes in, you might delete older objects to avoid overflow).

Array Functions (`array.` namespace)

Arrays are dynamic lists that can store sequences of values (of a specified type). Pine arrays are 0-indexed. The `array` namespace provides functions to create and manipulate arrays. Key functions include [69]:

- **Creation**:
- `array.new_<type>(size, initial_value)` – Creates a new array of given type and size. `<type>` can be `float`, `int`, `bool`, `color`, `string`. For example, `array.new_float(5, 0)` makes a float array of length 5 initialized with 0s. It returns an **array ID**.
- `array.new` – a generic constructor that might infer type from the initial values provided if any.

- Literal array creation: Pine also allows declaring an array with literal syntax: e.g. `myArr = [1, 2, 3]` creates an array of 3 ints.

- **Array Properties**:

- `array.size(arr)` – Get current length of array.

- `array.empty(arr)` – Returns true if array is empty (size 0).

- **Adding/Removing Elements**:

- `array.push(arr, value)` – Append a value to end of array (increases length by 1) [70].
- `array.unshift(arr, value)` – Prepend value at start of array.
- `array.pop(arr)` – Remove the last element and **return it** [61]. If array is empty, returns `na`.

- `array.shift(arr)` – Remove the first element and return it.
- These allow using arrays as stacks or queues.

- **Accessing/Setting Elements**:

- `array.get(arr, index)` – Retrieve the element at `index`. If index out of bounds, returns `na`.
- `array.set(arr, index, value)` – Set the element at `index` to `value` (index must exist already).

- In Pine v6, **negative indices** are allowed: e.g. `array.get(arr, -1)` gets the last element, `-2` gets second-last, etc [71]. This is new in v6 and makes it easier to reference end of array without knowing its length.

- **Utility**:

- `array.indexof(arr, value)` – Find the index of `value` in the array (returns -1 if not found) [72].
- `array.sort(arr, ascending=true)` – Sort the array in ascending or descending order [73]. The array's content is rearranged.
- `array.reverse(arr)` – Reverse the order of elements.
- `array.copy(arr)` – Creates a shallow copy of the array.
- `array.fill(arr, value, index_from, index_to)` – Fill a range of indices with a value (like Python's slice fill).
- `array.slice(arr, index_from, index_to)` – Get a subarray from index_from to index_to (exclusive).
- `array.join(arr, delimiter)` – Concatenate array elements into a string separated by delimiter (works if array is of strings or will tostring them).
- `array.concat(arr1, arr2)` – Append arr2 elements to arr1.
- `array.remove(arr, index)` – Remove element at specific index (shifting following elements left).
- `array.insert(arr, index, value)` – Insert value at index (shifting others right).

Example: Suppose we want to keep a rolling window of the last 20 closing prices manually (though Pine's history already provides this):

```
var myArray = array.new_float(0)  // dynamic array
if barstate.isnew
    if array.size(myArray) >= 20
        array.shift(myArray)      // remove oldest if >20 elements
    array.push(myArray, close)    // add latest close
```

This keeps the array length at most 20 by removing the first element when full, and adding the new close each new bar.

Arrays are powerful for custom data storage and algorithms that aren't straightforward with series alone. Remember that array operations in Pine are subject to runtime limits (very large loops or frequent inserts might hit performance limits).

Matrix Functions (`matrix.` namespace)

Matrices in Pine are essentially two-dimensional arrays with fixed size (rows and columns). They can store numeric data (float or int) in a grid form. Useful for more complex calculations or data storage beyond a simple list. Key matrix functions [74] :

- **Creation**:
- `matrix.new<float>(rows, cols, initial_value)` – Create a matrix of floats with given dimensions, initialized with a value (or 0 by default). Similarly `matrix.new<int>`. The `<>` specifies the type.
- There isn't a direct literal for matrix; you must use new().
- **Basic Operations**:
- `matrix.get(m, row, col)` – Get the value at specified row and column.
- `matrix.set(m, row, col, value)` – Set the value at [row, col] [75] .
- If indices are out of range, operation will likely fail or do nothing.
- **Analysis**:
- `matrix.sum(m)` – Sum of all elements in the matrix [75] .
- `matrix.avg(m)` – Average of all elements [75] .
- `matrix.transpose(m)` – Returns a transposed matrix (rows and columns swapped) [76] .
- `matrix.rows(m)` and `matrix.columns(m)` – get the dimensions.
- There is no extensive linear algebra library (like multiply, inverse) built-in beyond these basics.
- **Advanced**:
- `matrix.fill(m, value, row1, col1, row2, col2)` – Fill a submatrix region (from [row1,col1] to [row2,col2]) with a value.
- `matrix.resize(m, new_rows, new_cols)` – possibly to reshape matrix (if exists in v6).
- `matrix.add_row(m, array)` or `matrix.add_column` – if supported, to add a row/col from an array (some languages support adding).
- Given Pine's nature, matrices might primarily be used for organizing data for output (e.g., sending a matrix to a table or performing some calculations like correlation matrix across symbols if possible).

Example: Using a matrix to store recent OHLC:

```
// Create a 4xN matrix for O, H, L, C of last N bars
int N = 10
var prices = matrix.new<float>(4, N, na)
if barstate.isnew and bar_index >= N
    // shift older data left and append new on right
    for row = 0 to 3
        // shift each row values
        for col = 0 to N-2
            matrix.set(prices, row, col, matrix.get(prices, row, col+1))
    // set last column as current OHLC
    matrix.set(prices, 0, N-1, open)
    matrix.set(prices, 1, N-1, high)
    matrix.set(prices, 2, N-1, low)
    matrix.set(prices, 3, N-1, close)
```

This is a conceptual use; however, often simpler series or arrays suffice for most tasks. Matrices might shine for advanced computations like implementing certain algorithms or data transformations.

Map Functions (`map.` namespace)

Maps are key-value data structures introduced in Pine v5. A map holds pairs of keys and values (like a dictionary). They are useful for quick lookups by key. Key `map` functions:

- **Creation**:
- `map.new(key_type, value_type)` – Creates a new empty map that will use specified types for keys and values. For example, `map.new<string, float>()` creates a map from string keys to float values.
- `map.size(m)` – Number of entries in map.
- Keys can be `int`, `float`, `bool` or `string` (possibly color too if castable to int).
- **Setting and Getting**:
- `map.put(m, key, value)` – Insert or update the map: associates `key` with `value`.
- `map.get(m, key, default)` – Retrieve the value for `key`. If key not present, returns `default` (or na if default not provided).
- `map.remove(m, key)` – Remove entry by key.
- `map.clear(m)` – Remove all entries.
- `map.contains(m, key)` – Returns true if the map has an entry for `key`.
- `map.keys(m)` – Returns an array of all keys in the map (order not guaranteed).
- `map.values(m)` – Returns an array of all values.
- `map.copy(m)` – Shallow copy of the map.

Maps are particularly helpful in scripts where you need to categorize or store values associated with arbitrary keys (e.g., mapping ticker symbols to some calculated value). Since Pine doesn't easily allow dynamic variable names, a map can substitute by using string keys.

Example: If building a multi-symbol screener inside one script:

```
symbols = ["AAPL", "MSFT", "GOOGL"]
var symsMap = map.new<string, float>()
if barstate.isfirst
    // initialize with 0
    for sym in symbols
        map.put(symsMap, sym, 0.0)

// Each bar, update map values with close price of each symbol (simplified
example)
int idx = 0
```

```
for sym in symbols
    price = request.security(sym, timeframe.period, close)
    map.put(symsMap, sym, price)
```

Then you could use `map.get(symsMap, "AAPL")` to access AAPL's last price, etc.

Request Functions (`request.` namespace)

The `request` namespace contains functions to fetch data that is outside the current chart's series. This includes other symbols, other timeframes, as well as fundamental or economic data via built-in feeds. The most commonly used is `request.security()`, but v6 has several specialized request functions. Key functions [77] :

- `request.security(symbol, timeframe, expression, gaps=..., lookahead=...)` – Fetches data from another symbol or timeframe. This is fundamental for multi-timeframe or multi-symbol analysis.
- *Parameters:*
    - `symbol` – the ticker symbol or ticker id (e.g. `"NASDAQ:MSFT"` or a variable containing such a string).
    - `timeframe` – the resolution (e.g. `"60"` for 60min, `"D"` for daily, or a variable like `timeframe.period` for current resolution, or higher timeframe like `"W"` for weekly).
    - `expression` – the code to evaluate on that symbol/tf. Often just a built-in like `close` or a custom calculation. You can also pass a tuple or array of expressions to get multiple outputs [78] .
    - `gaps` – if set to `barmerge.gaps_on` or `barmerge.gaps_off`, controls whether to allow gaps when the target has fewer bars (defaults to no gaps).
    - `lookahead` – if `barmerge.lookahead_on`, will give future data on historical bars (usually keep off to avoid peeking).
- Returns the result of the expression from the other series, aligned to your chart bars (returns series of same type as expression).
- **Example**: `otherClose = request.security("NASDAQ:MSFT", "D", close)` – gets daily MSFT closing price on your chart's timeframe.

- Pine v6 allows **dynamic symbols**: the `symbol` argument can be a series string now, meaning you could vary the requested ticker per bar or based on conditions [79] . It also allows calling inside loops or conditions [80] (previously it had to be static/global).

- **Financials and Earnings**:

- `request.financial(symbol, report_type, financial_id)` – Retrieves fundamental financial data (from TradingView's financials) for a symbol. E.g., earnings, revenue, etc. `report_type` might be `"FY"` (annual) or `"Q"` (quarterly), and `financial_id` is a code for the specific metric (like `"NET_INCOME"`). Returns series of values (most recent value repeated until updated).
- `request.earnings(symbol, period, type)` – Gets EPS or earnings dates. For example, `request.earnings("AAPL", "annual", "actual")` could get annual earnings per share.
- `request.splits(symbol)` – Returns a series that indicates when a stock split occurred for the given symbol.
- `request.dividends(symbol)` – Returns the dividend value when a dividend event happens.

- `request.quandl(database_code, dataset_code, column)` – Fetch data from Quandl datasets by code (if available to your script).
- These allow incorporating fundamental data into indicators/strategies.

- **Economic data** (if any new in v6, not sure if introduced but possibly via `request.eco()` or similar, not documented here, but mention in passing if needed).

Example: to plot another symbol's price or an index:

```
appleClose = request.security("NASDAQ:AAPL", timeframe.period, close)
plot(appleClose, color=color.orange, title="AAPL Close")
```

This will overlay AAPL's closing price on whatever chart you attach the script to (if scales allow).

Be mindful of `request.security`: each unique call can greatly increase script load (fetching another series). Pine limits the number of securities that can be requested (max 40 in one script, as of writing). With v6's dynamic requests, you can request in loops but ensure not to exceed limits.

Alert Functions

These functions trigger TradingView alerts (when the script is added to chart and an alert condition is created). They don't impact backtesting or plotting, but are crucial for live usage:

- `alert(condition, message, freq)` – Triggers an alert when `condition` is true, with a custom `message`. `freq` determines how often (once per bar, once per minute, etc., using constants like `alert.freq_once_per_bar`). This function is used inside the script; however, to actually get alerts in TradingView, you must set up an alert and choose the script's alert() calls or alertcondition() as the trigger.
- `alertcondition(condition, title, message)` – Declares an alert condition for the script that can be enabled by the user in the Create Alert dialog [61]. Unlike `alert()`, which sends immediately when condition is true, `alertcondition` just flags that the given condition is an alert event. You provide a short `title` and default `message`.
- Example: `alertcondition(crossover(maFast, maSlow), title="MA Bullish Cross", message="Fast MA crossed above Slow MA")`. When the script is running, if you set up an alert on this condition, TradingView will notify when it occurs.

In Pine Script, typically you use `alertcondition` for defining meaningful events, and then the user can choose those when creating an alert. `alert()` is more dynamic (can include variable values in message using string concatenation) and can be triggered multiple times, but it will fire regardless of whether an alert is set (just needs the script running on a chart with an alert active for that script). Often, simple scripts prefer `alertcondition` and use the message templating in the alert dialog.

Strategy Trading Functions (`strategy.` namespace functions)

These functions are only available in strategy scripts (not in indicators). They execute trades in backtesting. They do **not** actually place real orders; they simulate orders on historical data to evaluate performance. Key strategy functions include:

- **Placing Orders**:
- `strategy.entry(id, long, qty, stop=..., limit=..., when=...)` – Place an entry order (either open a new trade or add to position).
  - `id` is a unique string for the order (used to identify/close it).
  - `long` is a bool or the literal strategy.long/strategy.short constant indicating direction.
  - `qty` is quantity (number of contracts/shares, can be fixed or a calculated value).
  - Optionally, `stop` or `limit` can be specified for stop-entry or limit-entry orders (if omitted, it's a market order).
  - In Pine v6, `when` **parameter is removed** (in v5 it was deprecated and now gone) – the order executes only if the `when` condition is true at the time of call [81]. So instead of passing `when`, simply call strategy.entry inside an if condition for clarity.
  - Example:

    ```
    if crossover(maFast, maSlow)
        strategy.entry("LongMA", strategy.long, 100)
    ```

    This enters 100 units long when fast MA crosses above slow MA.
- `strategy.exit(id, from_entry, profit=..., loss=..., trail_points=..., trail_offset=...)` – Place an exit order to close a trade (or position).
  - `id` is unique identifier for the exit order.
  - `from_entry` is the id of the entry order you want to exit (use `from_entry = ""` to apply to any position).
  - You can specify take-profit `profit` (in ticks or price, depending on if absolute or percentage) and stop-loss `loss`. There are three pairs of price targets (tp or sl) in Pine's strategy.exit – you can set one or multiple. In v6, **strategy.exit evaluates each pair even if one is na**, meaning you can dynamically choose between them (the relative/absolute pairs) [82].
  - `trail_points` and `trail_offset` define a trailing stop.
  - Example: `strategy.exit("ExitLongMA", "LongMA", profit=50, loss=20)` – will exit the trade from entry "LongMA" at +50 ticks profit or -20 ticks stop.
- `strategy.close(id, when=...)` – Close an open trade by entry id (market close). In v6, since `when` is removed, you'd call it conditionally if needed [81].
  - Example: `strategy.close("LongMA")` closes any open trade from "LongMA".
- `strategy.close_all()` – Closes all open positions.

Note: In backtest, orders are evaluated at bar close by default (except if you use intrabar order filling or specific conditions). There are nuances with how stop/limit are executed (immediate or on next bar). Pine's strategy engine handles filling logic – be sure to read TradingView's strategy execution model.

- **Order Management**:
- `strategy.cancel(id)` – Cancels a pending entry or exit order with given id [83].
- `strategy.cancel_all()` – Cancels all pending orders (doesn't close positions, just cancels orders not filled yet).

- These are useful if you place orders that might not fill and you want to retract them under some conditions.

- **Position Info**: While many position stats are accessible via the variables listed earlier (like strategy.position_size), there are also some functions:

- `strategy.opentrades` – Provides access to open trades (if multiple entries). In v5/6 this might be an array of trade objects with fields accessible via methods.
- `strategy.closedtrades` – Provides access to a list of past closed trades. In Pine v6, new methods allow retrieving specific details:
  - For example, `strategy.closedtrades.profit(entry_index)` could return profit of a particular closed trade, or as seen in documentation [84] :
  - `strategy.closedtrades.entry_bar_index(trade_index)` – Bar index where the trade was entered [84] .
  - `strategy.closedtrades.entry_price(trade_index)` – Entry price of that trade [84] .
  - `strategy.closedtrades.exit_bar_index(trade_index)`, `...exit_price(trade_index)` – Exit info [84] .
  - `strategy.closedtrades.profit(trade_index)` – Net profit of the trade.
  - `strategy.closedtrades.max_drawdown(trade_index)`, `...max_runup(trade_index)` – Worst adverse move and best favorable move during the trade.
  - These functions allow detailed analysis of past trades. The `trade_index` might be an index in the closed trades list (0 being the oldest or newest? Documentation needed – likely 0 for the most recent or first trade).
  - Pine v6 also introduced `strategy.closedtrades.first_index` – the index of the earliest remaining trade if older ones were trimmed due to limits [85] [86] .
- There are similar methods for `strategy.opentrades` to inspect currently open trade(s).

- Typically, one does not need these in the script unless doing some custom analysis of trade history.

- **Strategy Configuration**:

- `strategy.risk.max_positions_open` – (Annotation in strategy() call, not a function) to limit concurrent open positions.
- `strategy.risk.allow_entry_in` – e.g. `strategy.risk.allow_entry_in(strategy.direction.long)` to restrict entries to only longs or only one direction at a time.
- `strategy(title="...", initial_capital=..., default_qty=..., currency=..., pyramiding=..., commission_type=..., slippage=...)` – within the `strategy()` declaration you set initial balance, position sizing, pyramiding (multiple entries stacking), commission, etc.

Using strategy functions allows your script to simulate trades. Remember, **strategy scripts must contain at least one** `strategy.` **call that executes an order** (entry or order) or it will not run by default [87] .

Example – Simple strategy:

```
//@version=6
strategy("MA Crossover Strategy", overlay=true, initial_capital=10000,
default_qty=100)
fastMA = ta.sma(close, 10)
slowMA = ta.sma(close, 30)
bullCross = ta.crossover(fastMA, slowMA)
bearCross = ta.crossunder(fastMA, slowMA)
if bullCross
    strategy.entry("Long", strategy.long, comment="MA Cross Long")
if bearCross
    strategy.exit("ExitLong", from_entry="Long", comment="MA Cross Exit")
```

This will go long on a bullish crossover and exit on a bearish crossunder. (For a complete strategy, one might also handle short entries or use stop losses, etc., but this is a simple example.)

Miscellaneous and Utility Functions

A few other functions that don't neatly fall into above categories but are useful:

- **na handling**:
- `na(value)` – Returns true if `value` is not available (na) [88].
- `nz(value, repl)` – Returns `value` if it is non-na, otherwise returns `repl` (or 0 if no replacement given). "nz" stands for "no zero" or "non-zero" (treat na as zero).
- `ta.barssince(cond)` we covered (bars since cond true).
- `valuewhen(cond, expr, n)` we covered as `ta.valuewhen`.
- `falling(series, length)` – true if series has been continuously falling for `length` bars [89].
- `rising(series, length)` – true if series rising for length bars [90].

- These are in `ta.` now (e.g. `ta.falling`, `ta.rising`).

- **Type Conversion**:

- Casting between types can often be done by simply assigning (float to int will floor, etc.). But explicit cast functions/notations:
  - `float(x)` – might cast int to float (if allowed).
  - `int(x)` – cast float to int (truncates).
  - `bool(x)` – in v6, if you need to cast numeric to bool, you must explicitly compare or use a cast because implicit cast was removed [7]. Pine does not have a direct bool() function; instead you do, for example, `isNonZero = (x != 0)` to convert a number to boolean.

- `color(x)` – There is a `color()` constructor that might take an integer or components, but usually use `color.rgb` or constants.

- **Printing/Debugging**: Pine has no direct print to console. The usual approach is plotting debug values or using `label`/`table` to display values, or using `alert()` with messages for debugging when running live.

- **Performance**: Pine v6 introduced profiling functions (e.g., `strategy.opentrades.performance`? Actually, the docs mention a profiling & optimization section [91] but not specific functions – likely means manual code adjustments, not a built-in function).

- **Misc**:

- `switch` statement: Not a function, but a new control structure in Pine v5 that allows multi-branch selection based on a value (similar to a switch-case in other languages).
- `for ... in ...`: Pine allows iterating over arrays with `for value in array` syntax as well.

This covers the broad landscape of Pine Script v6's built-in functionality. Each function above has more detailed documentation in the official reference manual, including edge cases and additional optional parameters [28].

What's New in Pine Script v6

Pine Script version 6 introduced several significant changes and additions compared to version 5. If you are migrating from an older version or want to leverage the latest features, here are the highlights of what's new in v6:

- **Dynamic Data Requests**: All `request.*()` functions (like `request.security`) can now accept series variables as arguments for symbol or timeframe [79] [92]. This means you can dynamically change the target of a security request in your script (even inside loops or conditionals), enabling more adaptive multi-symbol and multi-timeframe strategies. Previously, you had to hardcode or fix the symbol/timeframe at compile time.

- **Strict Boolean Handling**: In Pine v6, booleans are strictly true/false only – they **cannot be** `na` anymore [93] [94]. This removal of tri-state booleans simplifies logic. Additionally, implicit casting of numbers to bool is disallowed [7] – you must explicitly compare or cast. For example, if `myVar` is a float, you can't directly use `if myVar` (as was allowed before if nonzero); instead use `if myVar != 0` or similar. Logical operators `and`/`or` now use short-circuit evaluation [95] [6], which improves efficiency: the second operand is not evaluated if the first already determines the outcome (e.g., in `A and B`, if A is false, B is not evaluated).

- **Text Formatting Enhancements**: Pine v6 adds a `text_formatting` parameter for labels, table cells, and boxes, allowing **bold or italic** text styling [96] [97]. Constants like `text.format_bold`, `text.format_italic`, `text.format_none` can be used. Also, text sizes can now be specified as integer point sizes (for finer control over font size rather than just the `size.small/large` presets) [98]. This makes it possible to create more visually distinct and readable textual elements on charts.

- **Array Improvements**: Arrays now support **negative indexing** [99] [100]. Using `array.get(arr, -1)` will fetch the last element, `-2` the second last, and so on, which is more intuitive than calculating `array.size(arr)-1`. This applies to `array.get`, `array.set`, etc., and simplifies code that works with the tail of arrays.

- **Strategy Backtesting Changes**:

- The old 9,000 bar trade limit for strategies has been addressed. In v6, if a strategy generates more than the limit of closed trades, it will **trim older trades** instead of stopping the strategy

[101] [102]. This means backtests can continue running indefinitely, removing the earliest trades once the limit is exceeded, rather than halting simulation.
- A new variable `strategy.closedtrades.first_index` is provided to identify the index of the first remaining closed trade when trimming occurs (so you know how many were dropped) [85] [103].
- The `when` parameter for order functions (`strategy.entry`, etc.) was fully **removed** in v6 [81] (it was deprecated earlier). You now simply call orders conditionally, which clarifies code flow.

- **Fractional returns for integer division**: In const contexts, dividing two integers can produce a float result in v6 [104]. This means `5/2` might yield `2.5` if both are const literals, whereas previously it may have truncated. This could affect certain calculations or require explicit cast to int if needed.

- **New Built-in Variables**:

- `syminfo.main_tickerid` and `timeframe.main_period` were added to always reference the main chart's symbol and timeframe regardless of where the code runs (especially helpful in libraries or if you're using `security` calls with secondary series) [17].
- `syminfo.mincontract` was added to provide the minimum contract size for symbols (useful for futures or fractional instruments) [17].

- Under the `chart` namespace, `chart.left_visible_bar_time`, `chart.is_standard`, `chart.fg_color`, etc., were introduced for more context about the chart's current view and style [14] [16].

- **Performance Optimizations**: The Pine v6 engine includes under-the-hood improvements:

- **Lazy Evaluation**: With the boolean changes, you can now safely do things like `if i < array.size(arr) and array.get(arr, i) > 0` in one condition – the second part won't be evaluated if the first part is false, preventing out-of-bounds errors [5]. This lazy evaluation allows combining checks succinctly and efficiently.

- General runtime optimizations may make scripts run faster, especially in heavy logical operations.

- **Miscellaneous**:

- Some color constant values were updated for consistency [21].
- Unique-type function parameters (like the `plot()` function's `style` parameter which expects a specific enum type) can no longer be `na` [105]. You must supply a valid value.
- History-referencing operator `[]` can no longer be used on literal values or constants [106] – e.g. `5[1]` is invalid now (was meaningless anyway), and cannot be used on UDT fields directly [107].
- **Method Syntax**: As a convenience, v6 allows calling some functions as methods on objects (like the label/line/box IDs) as mentioned earlier. This doesn't add new functionality but can make code cleaner.

These changes mean Pine Script v6 is generally backward compatible with v5 except where noted (you may need to adjust bool logic or remove `when` parameters, etc.). TradingView provides a converter tool to automatically upgrade v5 scripts to v6, which can handle many of these changes [108].

Conclusion: Pine Script v6 offers a robust set of built-in functions, variables, and language features for developing custom trading tools on TradingView. This reference manual covered all major functions, syntax, and categories of the language. Whether you're calculating an RSI, plotting custom signals, or backtesting a strategy, Pine's built-ins provide a shortcut to implement complex logic with minimal code. Always refer to the official Pine Script reference for the exact signature and details of each function [28], and use this manual as a structured guide to Pine Script v6's capabilities. Happy coding and trading!

Sources: The information in this manual is based on the official Pine Script v6 documentation and reference [23] [77] [109] [14], Pine Script v6 release notes and community guides [7] [86], and other TradingView Pine Script resources.

---

[1] [2] [91] Welcome to Pine Script® v6
https://www.tradingview.com/pine-script-docs/welcome/

[3] [4] [13] [14] [15] [16] [18] [19] [20] [23] [25] [28] [29] [33] [34] [42] [43] [51] [53] [54] [59] [60] [61] [62] [63] [64] [67] [68] [77] [78] [83] [84] [87] [109] 1_Pine Script V6 User Manual PDF (1) | PDF | Scope (Computer Science) | Time Series
https://www.scribd.com/document/860957045/1-Pine-Script-V6-User-Manual-PDF-1

[5] [71] [80] [86] [95] [98] [100] [102] [103] Pine Script™ v6: An Exciting Update for Traders and Developers
https://crosstrade.io/blog/pine-script-v6-an-exciting-update-for-traders-and-developers/

[6] [17] [92] [94] [97] Pine Script™ v6: What's New and Why It Matters - TradersPost Blog
https://blog.traderspost.io/article/pine-script-tm-v6-whats-new-and-why-it-matters

[7] [21] [81] [82] [85] [88] [93] [101] [104] [105] [106] [107] Migration guides / To Pine Script® version 6
https://www.tradingview.com/pine-script-docs/migration-guides/to-pine-version-6/

[8] [9] [10] [11] [12] [30] [31] [32] [35] [37] [38] [39] [40] [41] [44] [45] [46] [48] [49] [50] [55] [56] [57] [58] [89] [90] A minimal reference to pine script v5 · GitHub
https://gist.github.com/kdkiss/731e6288e2314a7e6f36383888e5bc40

[22] [36] Language / Built-ins
https://www.tradingview.com/pine-script-docs/language/built-ins/

[24] [26] [27] [47] [52] [65] [69] [70] [72] [73] [74] [75] [76] Essential Pine Script v6 Cheat Sheet for Traders | Pineify Blog
https://pineify.app/resources/blog/best-pine-script-cheat-sheet

[66] NEW: Method syntax comes to Pine Script™ : r/TradingView - Reddit
https://www.reddit.com/r/TradingView/comments/11mwgzk/new_method_syntax_comes_to_pine_script/

[79] [96] [99] [108] Pine Script v6: Everything You Need to Know | Pineify Blog
https://pineify.app/resources/blog/pine-script-v6-everything-you-need-to-know