To Pine Script™ version 5

Introduction

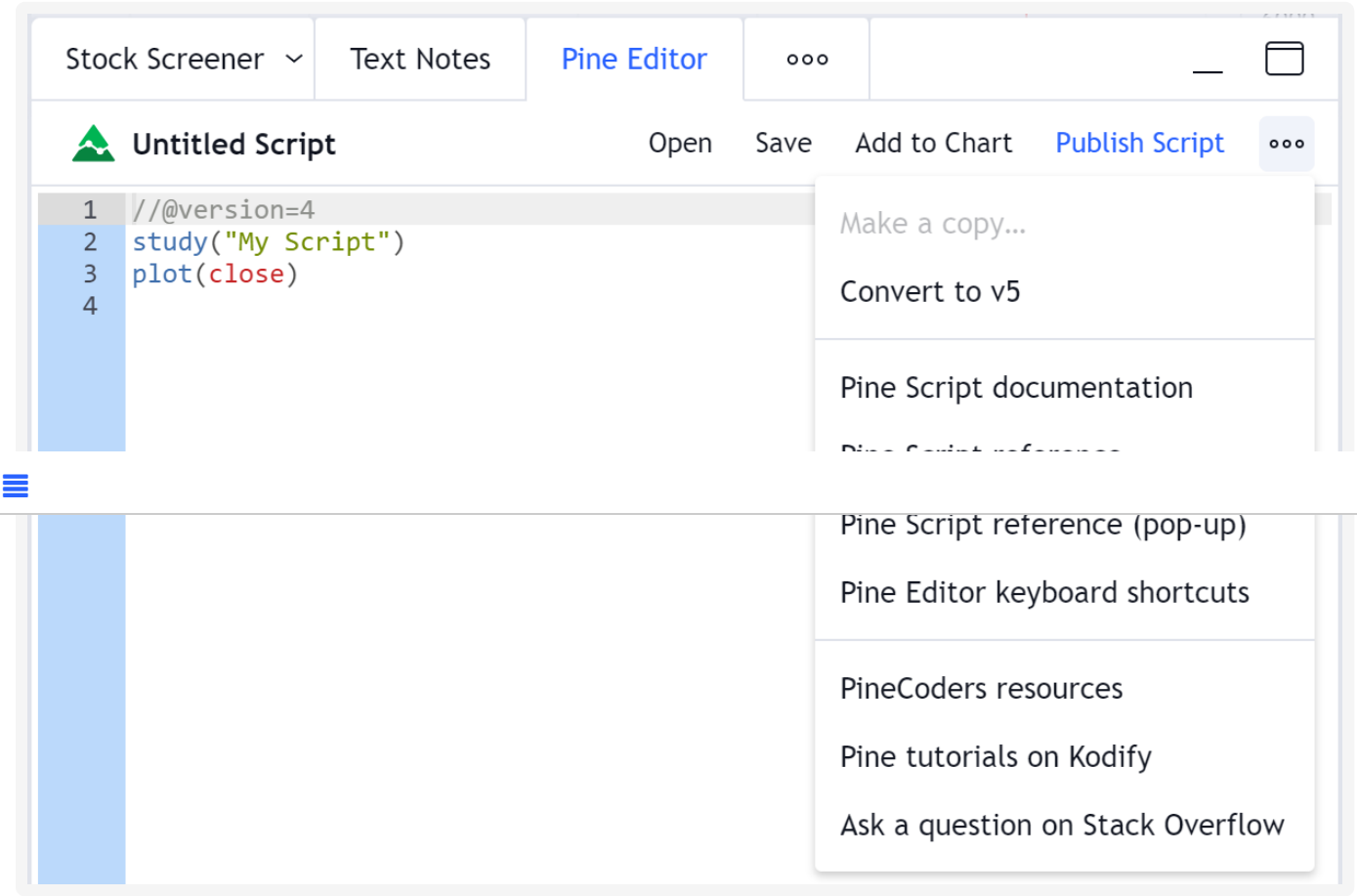
This guide documents the **changes** made to Pine Script™ from v4 to v5. It will guide you in the adaptation of existing Pine scripts to Pine Script™ v5. See our Release notes for a list of the **new** features in Pine Script™ v5.

The most frequent adaptations required to convert older scripts to v5 are:

- Changing study() for indicator() (the function's signature has not changed).
- Renaming built-in function calls to include their new namespace (e.g., highest() in v4 becomes ta.highest() in v5).
- Restructuring inputs to use the more specialized `input.*()` functions.
- Eliminating uses of the deprecated `transp` parameter by using color.new() to simultaneously define color and transparency for use with the `color` parameter.
- If you used the `resolution` and `resolution_gaps` parameters in v4's study(), they will require changing to `timeframe` and `timeframe_gaps` in v5's indicator().

v4 to v5 converter

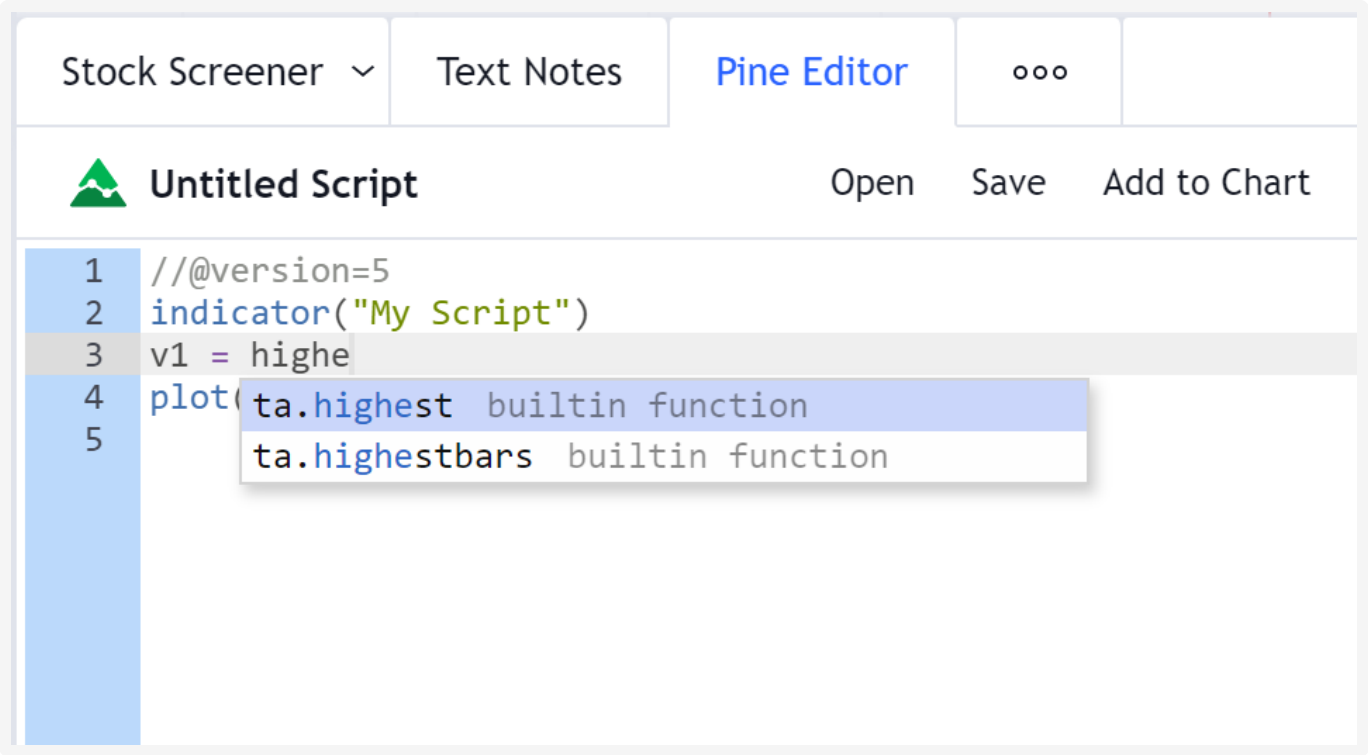
The Pine Script™ Editor includes a utility to automatically convert v4 scripts to v5. To access it, open a script with `//@version=4` in it and select the "Convert to v5" option in the "More" menu identified by three dots at the top-right of the Editor's pane:



Not all scripts can be automatically converted from v4 to v5. If you want to convert the script manually or if your indicator returns a compilation error after conversion, use the following sections to determine how to complete the conversion. A list of some errors you can encounter during the automatic conversion and how to fix them can be found in the Common script conversion errors section of this guide.

Renamed functions and variables

For clarity and consistency, many built-in functions and variables were renamed in v5. The inclusion of v4 function names in a new namespace is the cause of most changes. For example, the sma() function in v4 is moved to the `ta.` namespace in v5: ta.sma(). Remembering the new namespaces is not necessary; if you type the older name of a function without its namespace in the Editor and press the 'Auto-complete' hotkey (`Ctrl` + `Space`, or `Cmd` + `Space` on MacOS), a popup showing matching suggestions appears:



Not counting functions moved to new namespaces, only two functions have been renamed:

- `study()` is now indicator().
- `tickerid()` is now ticker.new().

The full list of renamed functions and variables can be found in the All variable, function, and parameter name changes section of this guide.

Renamed function parameters

The parameter names of some built-in functions were changed to improve the nomenclature. This has no bearing on most scripts, but if you used these parameter names when calling functions, they will require adaptation. For example, we have standardized all mentions:

```
// Valid in v4. Not valid in v5.
timev4 = time(resolution = "1D")
// Valid in v5.
timev5 = time(timeframe = "1D")
// Valid in v4 and v5.
timeBoth = time("1D")
```

The full list of renamed function parameters can be found in the All variable, function, and parameter name changes section of this guide.

Removed an `rsi()` overload

In v4, the rsi() function had two different overloads:

- `rsi(series float, simple int)` for the normal RSI calculation, and
- `rsi(series float, series float)` for an overload used in the MFI indicator, which did a calculation equivalent to `100.0 - (100.0 / (1.0 + arg1 / arg2))`.

This caused a single built-in function to behave in two very different ways, and it was difficult to distinguish which one applied because it depended on the type of the second argument. As a result, a number of indicators misused the

function and were displaying incorrect results. To avoid this, the second overload was removed in v5.

The ta.rsi() function in v5 only accepts a “simple int” argument for its `length` parameter. If your v4 code used the now deprecated overload of the function with a `float` second argument, you can replace the whole `rsi()` call with the following formula, which is equivalent:

```
100.0 - (100.0 / (1.0 + arg1 / arg2))
```

Note that when your v4 code used a “series int” value as the second argument to rsi(), it was automatically cast to “series float” and the second overload of the function was used. While this was syntactically correct, it most probably did **not** yield the result you expected. In v5, ta.rsi() requires a “simple int” for the argument to `length`, which precludes dynamic (or “series”) lengths. The reason for this is that RSI calculations use the ta.rma() moving average, which is similar to ta.ema() in that it relies on a length-dependent recursive process using the values of previous bars. This makes it impossible to achieve correct results with a “series” length that could vary bar to bar.

If your v4 code used a length that was “const int”, “input int” or “simple int”, no changes are required.

Reserved keywords

A number of words are reserved and cannot be used for variable or function names. They are: `catch`, `class`, `do`, `ellipse`, `in`, `is`, `polygon`, `range`, `return`, `struct`, `text`, `throw`, `try`. If your v4 indicator uses any of these, rename your variable or function for the script to work in v5.

Removed `iff()` and `offset()`

The iff() and offset() functions have been removed. Code using the iff() function can be rewritten using the ternary operator:

```
// iff(<condition>, <return_when_true>, <return_when_false>)
// Valid in v4, not valid in v5
barColorIff = iff(close >= open, color.green, color.red)
// <condition> ? <return_when_true> : <return_when_false>
// Valid in v4 and v5
barColorTernary = close >= open ? color.green : color.red
```

Note that the ternary operator is evaluated “lazily”; only the required value is calculated (depending on the condition’s evaluation to `true` or `false`). This is different from iff(), which always evaluated both values but returned only the relevant one.

Some functions require evaluation on every bar to correctly calculate, so you will need to make special provisions for these by pre-evaluating them before the ternary:

```
// `iff()` in v4: `highest()` and `lowest()` are calculated on every bar
v1 = iff(close > open, highest(10), lowest(10))
plot(v1)
// In v5: forced evaluation on every bar prior to the ternary statement.
h1 = ta.highest(10)
l1 = ta.lowest(10)
v1 = close > open ? h1 : l1
plot(v1)
```

The offset() function was deprecated because the more readable [] operator is equivalent:

```
// Valid in v4. Not valid in v5.
prevClosev4 = offset(close, 1)
// Valid in v4 and v5.
prevClosev5 = close[1]
```

Split of `input()` into several functions

The v4 input() function was becoming crowded with a plethora of overloads and parameters. We split its functionality into different functions to clear that space and provide a more robust structure to accommodate the additions planned for inputs. Each new function uses the name of the `input.*` type of the v4 `input()` call it replaces. E.g., there is now a specialized input.float() function replacing the v4 `input(1.0, type = input.float)` call. Note that you can still use `input(1.0)` in v5, but because only input.float() allows for parameters such as `minval`, `maxval`, etc., it is more powerful. Also note that input.int() is the only specialized input function that does not use its equivalent v4 `input.integer` name. The `input.*` constants have been removed because they were used as arguments for the `type` parameter, which was deprecated.

To convert, for example, a v4 script using an input of type `input.symbol`, the input.symbol() function must be used in v5:

```
// Valid in v4. Not valid in v5.
aaplTicker = input("AAPL", type = input.symbol)
// Valid in v5
aaplTicker = input.symbol("AAPL")
```

The input() function persists in v5, but in a simpler form, with less parameters. It has the advantage of automatically detecting input types “bool/color/int/float/string/source” from the argument used for `defval`:

```
// Valid in v4 and v5.
// While "AAPL" is a valid symbol, it is only a string here because `input.symbol()` is
tickerString = input("AAPL", title = "Ticker string")
```

Some function parameters now require built-in arguments

In v4, built-in constants such as `plot.style_area` used as arguments when calling Pine Script™ functions corresponded to pre-defined values of a specific type. For example, the value of `barmerge.lookahead_on` was `true`, so you could use `true` instead of the named constant when supplying an argument to the `lookahead` parameter in a security() function call. We found this to be a common source of confusion, which caused unsuspecting programmers to produce code yielding unintended results.

In v5, the use of correct built-in named constants as arguments to function parameters requiring them is mandatory:

```
// Not valid in v5: `true` is used as an argument for `lookahead`.
request.security(syminfo.tickerid, "1D", close, lookahead = true)
// Valid in v5: uses a named constant instead of `true`.
request.security(syminfo.tickerid, "1D", close, lookahead = barmerge.lookahead_on)

// Would compile in v4 because `plot.style_columns` was equal to 5.
// Won't compile in v5.
a = 2 * plot.style_columns
plot(a)
```

To convert your script from v4 to v5, make sure you use the correct named built-in constants as function arguments.

Deprecated the `transp` parameter

The `transp=` parameter used in the signature of many v4 plotting functions was deprecated because it interfered with RGB functionality. Transparency must now be specified along with the color as an argument to parameters such as `color`, `textcolor`, etc. The color.new() or color.rgb() functions will be needed in those cases to join a color and its transparency.

Note that in v4, the bgcolor() and fill() functions had an optional `transp` parameter that used a default value of 90. This meant that the code below could display Bollinger Bands with a semi-transparent fill between two bands and a semi-transparent backround color where bands cross price, even though no argument is used for the `transp` parameter in its bgcolor() and fill() calls:

```
//@version=4
study("Bollinger Bands", overlay = true)
[middle, upper, lower] = bb(close, 5, 4)
plot(middle, color=color.blue)
p1PlotID = plot(upper, color=color.green)
p2PlotID = plot(lower, color=color.green)
crossUp = crossover(high, upper)
crossDn = crossunder(low, lower)
// Both `fill()` and `bgcolor()` have a default `transp` of 90
fill(p1PlotID, p2PlotID, color = color.green)
bgcolor(crossUp ? color.green : crossDn ? color.red : na)
```

In v5 we need to explictly mention the 90 transparency with the color, yielding:

```
//@version=5
indicator("Bollinger Bands", overlay = true)
[middle, upper, lower] = ta.bb(close, 5, 4)
plot(middle, color=color.blue)
p1PlotID = plot(upper, color=color.green)
p2PlotID = plot(lower, color=color.green)
crossUp = ta.crossover(high, upper)
crossDn = ta.crossunder(low, lower)
var TRANSP = 90
// We use `color.new()` to explicitly pass transparency to both functions
fill(p1PlotID, p2PlotID, color = color.new(color.green, TRANSP))
bgcolor(crossUp ? color.new(color.green, TRANSP) : crossDn ? color.new(color.red, TRANS
```

Changed the default session days for `time()` and `time_close()`

The default set of days for `session` strings used in the time() and time_close() functions, and returned by input.session(), has changed from `"23456"` (Monday to Friday) to `"1234567"` (Sunday to Saturday):

```
// On symbols that are traded during weekends, this will behave differently in v4 and v
t0 = time("1D", "1000-1200")
// v5 equivalent of the behavior of `t0` in v4.
t1 = time("1D", "1000-1200:23456")
// v5 equivalent of the behavior of `t0` in v5.
t2 = time("1D", "1000-1200:1234567")
```

This change in behavior should not have much impact on scripts running on conventional markets that are closed during weekends. If it is important for you to ensure your session definitions preserve their v4 behavior in v5 code, add `":23456"` to your session strings. See this manual's page on Sessions for more information.

`strategy.exit()` now must do something

Gone are the days when the strategy.exit() function was allowed to loiter. Now it must actually have an effect on the strategy by using at least one of the following parameters: `profit`, `limit`, `loss`, `stop`, or one of the following pairs: `trail_offset` combined with either `trail_price` or `trail_points`. When uses of strategy.exit() not meeting these criteria trigger an error while converting a strategy to v5, you can safely eliminate these lines, as they didn't do anything in your code anyway.

Common script conversion errors

Invalid argument 'style'/'linestyle' in 'plot'/'hline' call

To make this work, you need to change the "int" arguments used for the `style` and `linestyle` arguments in plot() and hline() for built-in constants:

```
// Will cause an error during conversion
plotStyle = input(1)
hlineStyle = input(1)
plot(close, style = plotStyle)
hline(100, linestyle = hlineStyle)

// Will work in v5
//@version=5
indicator("")
plotStyleInput = input.string("Line", options = ["Line", "Stepline", "Histogram", "Cros
hlineStyleInput = input.string("Solid", options = ["Solid", "Dashed", "Dotted"])

plotStyle = plotStyleInput == "Line" ? plot.style_line :
             plotStyleInput == "Stepline" ? plot.style_stepline :
             plotStyleInput == "Histogram" ? plot.style_histogram :
             plotStyleInput == "Cross" ? plot.style_cross :
             plotStyleInput == "Area" ? plot.style_area :
             plotStyleInput == "Columns" ? plot.style_columns :
             plot.style_circles

hlineStyle = hlineStyleInput == "Solid" ? hline.style_solid :
             hlineStyleInput == "Dashed" ? hline.style_dashed :
             hline.style_dotted

plot(close, style = plotStyle)
hline(100, linestyle = hlineStyle)
```

See the Some function parameters now require built-in arguments section of this guide for more information.

Undeclared identifier 'input.%input_name%'

To fix this issue, remove the `input.*` constants from your code:

```
// Will cause an error during conversion
_integer = input.integer
_bool = input.bool
i1 = input(1, "Integer", _integer)
i2 = input(true, "Boolean", _bool)

// Will work in v5
i1 = input.int(1, "Integer")
i2 = input.bool(true, "Boolean")
```

See the User Manual's page on Inputs, and the Some function parameters now require built-in arguments section of this guide for more information.

Invalid argument 'when' in 'strategy.close' call

This is caused by a confusion between strategy.entry() and strategy.close().

The second parameter of strategy.close() is `when`, which expects a "bool" argument. In v4, it was allowed to use `strategy.long` an argument because it was a "bool". With v5, however, named built-in constants must be used as arguments, so `strategy.long` is no longer allowed as an argument to the `when` parameter.

The `strategy.close("Short", strategy.long)` call in this code is equivalent to `strategy.close("Short")`, which is what must be used in v5:

```
// Will cause an error during conversion
if (longCondition)
    strategy.close("Short", strategy.long)
    strategy.entry("Long", strategy.long)

// Will work in v5:
if (longCondition)
    strategy.close("Short")
    strategy.entry("Long", strategy.long)
```

See the Some function parameters now require built-in arguments section of this guide for more information.

Cannot call 'input.int' with argument 'minval'='%value%'. An argument of 'literal float' type was used but a 'const int' is expected

In v4, it was possible to pass a "float" argument to `minval` when an "int" value was being input. This is no longer possible in v5; "int" values are required for "int" inputs:

```
// Works in v4, will break on conversion because minval is a 'float' value
int_input = input(1, "Integer", input.integer, minval = 1.0)

// Works in v5
int_input = input.int(1, "Integer", minval = 1)
```

See the User Manual's page on Inputs, and the Some function parameters now require built-in arguments section of this guide for more information.

All variable, function, and parameter name changes

Removed functions and variables

v4	v5
`input.bool` input	Replaced by `input.bool()`
`input.color` input	Replaced by `input.color()`
`input.float` input	Replaced by `input.float()`
`input.integer` input	Replaced by `input.int()`
`input.resolution` input	Replaced by `input.timeframe()`
`input.session` input	Replaced by `input.session()`
`input.source` input	Replaced by `input.source()`
`input.string` input	Replaced by `input.string()`
`input.symbol` input	Replaced by `input.symbol()`
`input.time` input	Replaced by `input.time()`
`iff()`	Use the `?:` operator instead
`offset()`	Use the `[]` operator instead

Renamed functions and parameters

No namespace change

v4	v5
`study(<...>, resolution, resolution_gaps, <...>)`	`indicator(<...>, timeframe, timeframe`
`strategy.entry(long)`	`strategy.entry(direction)`
`strategy.order(long)`	`strategy.order(direction)`
`time(resolution)`	`time(timeframe)`
`time_close(resolution)`	`time_close(timeframe)`
`nz(x, y)`	`nz(source, replacement)`

“ta” namespace for technical analysis functions and variables

v4	v5
Indicator functions and variables	
`accdist`	`ta.accdist`
`alma()`	`ta.alma()`

atr()	ta.atr()
bb()	ta.bb()
bbw()	ta.bbw()
cci()	ta.cci()
cmo()	ta.cmo()
cog()	ta.cog()
dmi()	ta.dmi()
ema()	ta.ema()
hma()	ta.hma()
iii	ta.iii
kc()	ta.kc()
kcw()	ta.kcw()
linreg()	ta.linreg()
macd()	ta.macd()
mfi()	ta.mfi()
mom()	ta.mom()
nvi	ta.nvi
obv	ta.obv
pvi	ta.pvi
pvt	ta.pvt
rma()	ta.rma()
roc()	ta.roc()
rsi(x, y)	ta.rsi(source, length)
sar()	ta.sar()
sma()	ta.sma()
stoch()	ta.stoch()
supertrend()	ta.supertrend()
swma(x)	ta.swma(source)
tr	ta.tr
tr()	ta.tr()
tsi()	ta.tsi()
vwap	ta.vwap
vwap(x)	ta.vwap(source)

vwma()	ta.vwma()
wad	ta.wad
wma()	ta.wma()
wpr()	ta.wpr()
wvad	ta.wvad
Supporting functions	
barsince()	ta.barsince()
change()	ta.change()
correlation(source_a, source_b, length)	ta.correlation(source1, source2, length)
cross(x, y)	ta.cross(source1, source2)
crossover(x, y)	ta.crossover(source1, source2)
crossunder(x, y)	ta.crossunder(source1, source2)
cum(x)	ta.cum(source)
dev()	ta.dev()
falling()	ta.falling()
highest()	ta.highest()
highestbars()	ta.highestbars()
lowest()	ta.lowest()
lowestbars()	ta.lowestbars()
median()	ta.median()
mode()	ta.mode()
percentile_linear_interpolation()	ta.percentile_linear_interpolation()
percentile_nearest_rank()	ta.percentile_nearest_rank()
percentrank()	ta.percentrank()
pivothigh()	ta.pivothigh()
pivotlow()	ta.pivotlow()
range()	ta.range()
rising()	ta.rising()
stdev()	ta.stdev()
valuewhen()	ta.valuewhen()
variance()	ta.variance()

“math” namespace for math-related functions and variables

v4	v5
`abs(x)`	`math.abs(number)`
`acos(x)`	`math.acos(number)`
`asin(x)`	`math.asin(number)`
`atan(x)`	`math.atan(number)`
`avg()`	`math.avg()`
`ceil(x)`	`math.ceil(number)`
`cos(x)`	`math.cos(angle)`
`exp(x)`	`math.exp(number)`
`floor(x)`	`math.floor(number)`
`log(x)`	`math.log(number)`
`log10(x)`	`math.log10(number)`
`max()`	`math.max()`
`min()`	`math.min()`
`pow()`	`math.pow()`
`random()`	`math.random()`
`round(x, precision)`	`math.round(number, precision)`
`round_to_mintick(x)`	`math.round_to_mintick(number)`
`sign(x)`	`math.sign(number)`
`sin(x)`	`math.sin(angle)`
`sqrt(x)`	`math.sqrt(number)`
`sum()`	`math.sum()`
`tan(x)`	`math.tan(angle)`
`todegrees()`	`math.todegrees()`
`toradians()`	`math.toradians()`

“request” namespace for functions that request external data

v4	v5
`financial()`	`request.financial()`
`quandl()`	`request.quandl()`
`security(<...>, resolution, <...>)`	`request.security(<...>, timeframe, <...>)`
`splits()`	`request.splits()`
`dividends()`	`request.dividends()`
`earnings()`	`request.earnings()`

“ticker” namespace for functions that help create tickers

v4	v5
`heikinashi()`	`ticker.heikinashi()`
`kagi()`	`ticker.kagi()`
`linebreak()`	`ticker.linebreak()`
`pointfigure()`	`ticker.pointfigure()`
`renko()`	`ticker.renko()`
`tickerid()`	`ticker.new()`

“str” namespace for functions that manipulate strings

v4	v5
`tostring(x, y)`	`str.tostring(value, format)`
`tonumber(x)`	`str.tonumber(string)`