ChartScript

Complete Documentation

TradingCharts · Scripting language for indicators, strategies and libraries

Compiled 26 October 2026

Table of Contents

ChartScript Overview 4

Part 1 – Concepts 5

- Inputs 5
- Alerts 7
- Bar States 8
- Chart Information 9
- Strings 11
- Time 12
- Timeframes 13
- Sessions 14
- Other Timeframes and Data 14
- Non-Standard Chart Data 16
- Strategies 17
- Libraries 19
- Repainting 20

Part 2 – Language 22

- Language Overview 22
- Script Structure 22
- Type System 22
- Identifiers 22
- Operators 22
- Variable Declarations 22
- Declaration Statements 22
- Execution Model 22
- Conditional Structures 22
- Loops 22
- User-Defined Functions 22
- Methods 22
- Objects 22
- Enums 22
- Arrays 22
- Maps 22
- Matrices 22
- Built-Ins 22
- Built-In Reference 23

Part 3 – Visuals 56

- Visuals Overview 56
- Plots 56
- Colors 56
- Fills 56
- Levels 56
- Lines and Boxes 56
- Backgrounds 56
- Bar Coloring 56
- Bar Plotting 56
- Text and Shapes 56
- Tables 56

Part 4 – Writing Scripts 67

- Writing Scripts Overview 67
- Debugging 67
- Limitations 67
- Profiling and Optimization 67

Publishing 67
Style Guide 67
What Is Not Supported 67
Part 5 – Errors **76**
Errors Overview 76
Undeclared Identifier 76
Expected End of Line 76
Expression Has No Effect 77
Drawing-Object Caps 78
Compute Budget Exceeded 79

ChartScript Overview

ChartScript lets you write your own indicators and strategies in its own scripting language, check them with Verify and run them on any chart.

The documentation is organised in five parts:

- **Concepts** – Inputs, Alerts, Bar States, Chart Information, Strings, Time, Timeframes, Sessions, Other Timeframes and Data, Non-Standard Chart Data, Strategies, Libraries, Repainting.
- **Language** – Script Structure, Type System, Identifiers, Operators, Variable Declarations, Declaration Statements, Execution Model, Conditional Structures, Loops, User-Defined Functions, Methods, Objects, Enums, Arrays, Maps, Matrices, Built-Ins, Built-In Reference.
- **Visuals** – Plots, Colors, Fills, Levels, Lines and Boxes, Backgrounds, Bar Coloring, Bar Plotting, Text and Shapes, Tables.
- **Writing Scripts** – Debugging, Limitations, Profiling and Optimization, Publishing, Style Guide, What Is Not Supported.
- **Errors** – Undeclared Identifier, Expected End of Line, Expression Statement Has No Effect, Drawing-Object Caps, Compute Budget Exceeded.

Part 1 – Concepts

Platform concepts every script author needs: inputs, alerts, bar states, time and sessions, cross-timeframe data, strategies, libraries, and repainting.

Inputs

Inputs are the values your users edit from the chart's Settings dialog. Declaring an input returns its current value; your script never cares whether the user has opened the dialog.

The basic form

```
length = input.int(20, "Length", minval=1)
source = input.source(close, "Source")
ma     = ta.sma(source, length)
plot(ma)
```

- Every `input.*()` call registers a row in the Inputs tab and returns the current value.
- The first argument is the **default**; the second is the **title** shown in the dialog.
- Omitting the title defaults to an auto-generated id (`in_0`, `in_1`, ...). Always provide a title.

Typed input functions

`input.int(defval, title, minval?, maxval?, step?, tooltip?, inline?, group?, confirm?, display?, options?, active?)`
Integer slider / stepper. `options=[a, b, c]` turns it into a dropdown of fixed values.

```
len   = input.int(14, "Length", minval=1, maxval=200)
mult  = input.int(2,  "Multiplier", options=[1, 2, 3, 4, 5])
```

`input.float(defval, title, minval?, maxval?, step?, tooltip?, inline?, group?, confirm?, display?, options?, active?)`
Floating-point input.

```
threshold = input.float(0.5, "Threshold", minval=0.01, step=0.05)
```

`input.bool(defval, title, ..., active?)` Checkbox. Pair `active=` with another `input.bool` or `input.int` to gate the control (greyed-out unless the condition holds).

```
useFilter = input.bool(true,  "Use filter")
strict    = input.bool(false, "Strict mode", active=useFilter)
```

`input.string(defval, title, options?, ...)` Text or dropdown:

```
mode = input.string("Fast", "Mode", options=["Fast", "Slow", "Custom"])
```

`input.color(defval, title, ...)` Swatch with a colour picker:

```
bull = input.color(color.lime, "Bull colour")
bear = input.color(color.red,  "Bear colour")
```

`input.source(defval, title, ...)` Picks a bar series: `close`, `open`, `hl2`, `hlc3`, `ohlc4`, or another indicator's output.

```
src = input.source(close, "Source")
```

`input.symbol(defval, title, ...)` Opens a symbol search dialog. The return is a string like `"NSE:TCS"` or a canonical secmaster key such as `"NSEE11536:TCS"`.

```
symA = input.symbol("NSE:NIFTY", "Benchmark")
```

`input.timeframe(defval, title, ...)` Picks a timeframe (`"5"`, `"60"`, `"D"`, ...):

```
htf = input.timeframe("D", "Higher timeframe")
```

`input.session(defval, title, ...)` Session string (`"0915-1530"` or `"0915-1530:12345"`):

```
sess = input.session("0915-1530", "Trading session")
```

`input.time(defval, title, ...)` Date-time picker. Default is an epoch ms (use `timestamp(...)`):

```
startT = input.time(timestamp("2020-01-01 00:00"), "Start")
```

`input.price(defval, title, ...)` Price picker that lets the user click on the chart to set a value:

```
level = input.price(100.0, "Level")
```

`input.textarea(defval, title, ...)` Multiline string:

```
notes = input.textarea("", "Notes")
```

`input.enum(defval, title, ...)` Dropdown driven by a user-defined enum:

```
enum Trend
    up    = "Bullish"
    down  = "Bearish"
    flat  = "Neutral"

t = input.enum(Trend.up, "Trend mode")
```

`input(defval, title, ...)` – type-inferred The generic form infers its type from `defval`. Prefer the typed form; this form is accepted for convenience.

```
length = input(20,         "Length")      // int
src    = input(close,      "Source")      // source
colour = input(color.lime, "Colour")      // color
```

Grouping and inline layout

`group=` collects related inputs under a section header. The dialog renders one heading per unique group name (in first-appearance order; repeated groups fold into their first section):

```
group_zone = "Zone Marking"
group_ma   = "MA Settings"

showDemand  = input.bool(true,   "Demand",   group=group_zone)
showSupply  = input.bool(true,   "Supply",   group=group_zone)
maLen       = input.int (20,     "Length",   group=group_ma, minval=1)
```

`inline=`: inputs sharing an `inline=` string **and** the same `group=` render on one horizontal row. Common pattern: a boolean toggle plus a colour swatch.

```
showMa1    = input.bool(true,        "Show MA", group="MA Settings", inline="ma1")
colMa1Up   = input.color(color.lime, "",        group="MA Settings", inline="ma1")
colMa1Dn   = input.color(color.red,  "",        group="MA Settings", inline="ma1")
```

The dialog caps **labelled** members per inline row at two to prevent long labels from colliding; label-less members (the swatches) ride along without counting.

tooltip=

Adds a hover help icon next to the control:

```
length = input.int(14, "Length", tooltip="Lookback window. Lower = more sensitive.")
```

active=

Grey out a control when another input is in a given state:

```
useFilter = input.bool(true, "Use filter")
strict    = input.bool(false, "Strict mode", active=useFilter)        // only if useFilter == true
negatedEx = input.bool(false, "Fallback",    active=not useFilter)    // only if useFilter == false
```

The reference supports `active = someBoolVar` or `active = not someBoolVar`.

confirm=

Mark an input so the dialog opens automatically the first time the script is added, so the user can override important defaults:

```
level = input.price(100.0, "Trigger level", confirm=true)
```

display=

Where the input's current value appears outside the Settings dialog:

```
length = input.int(20, "Length", display=display.data_window + display.status_line)
```

Inputs vs constants

- A `const` (`var int LEN = 20`) is baked into the script.
- An `input` becomes part of the script's wire **contract**; a version with different inputs renders a different dialog.

Reading inputs inside strategies

Inputs feed strategy sizing and risk literally the same way they feed indicators:

```
riskPct = input.float(1.0, "Risk per trade (%)")
strategy.entry("Long", strategy.long,
               qty = strategy.equity * riskPct / 100 / close)
```

Alerts

A ChartScript script talks to the alert system in two ways: the declarative `alertcondition()` and the dynamic `alert()`.

alertcondition(condition, title, message?)

```
rsi = ta.rsi(close, 14)
alertcondition(ta.crossover(rsi, 70),  "RSI overbought",  "{{ticker}} RSI crossed above 70")
alertcondition(ta.crossunder(rsi, 30), "RSI oversold",    "{{ticker}} RSI crossed below 30")
```

- `title` is what the user sees in the Alerts tab when adding an alert.
- `message` is the default text; the user can override it in the dialog. It accepts placeholders (below).
- Called from script scope only. Only const strings for `title`/`message`.

alert(message, freq?)

A dynamic call fires an alert directly from the body:

```
if volume > ta.sma(volume, 20) * 3
    alert("Volume spike on " + syminfo.ticker + " @ " + str.tostring(close),
          alert.freq_once_per_bar_close)
```

Freq	Fires
`alert.freq_all`	Every tick on the forming bar
`alert.freq_once_per_bar`	At most once per bar (on the first evaluation that matches)
`alert.freq_once_per_bar_close`	Once per bar, only on the confirmed (closing) evaluation

Omitting `freq` defaults to `alert.freq_once_per_bar` (the reference default). Use `freq_once_per_bar_close` when a signal must not fire on a forming bar that later reverses.

Placeholders

`alertcondition` default messages and the message typed into the alert dialog support these `{{...}}` placeholders. They are substituted by the alert service at delivery time:

Placeholder	Replaced with
`{{ticker}}`	Symbol ticker (TCS)
`{{exchange}}`	Exchange (NSE)
`{{interval}}`	Chart interval (5, 1D)
`{{time}}`	Time of the bar that fired (ISO 8601)
`{{timenow}}`	Wall-clock time at delivery (ISO 8601)
`{{plot("Title")}}`	Current value of the plot with that `title=` (`na` when the plot has no value)

Anything else inside `{{...}}` is left verbatim. Bar values (`close`, `volume`, ...), `strategy.*` account values and the `{{plot_0}}` / `{{strategy.order.*}}` forms are **not** substituted. Build such text in the script with `str.tostring()` and pass it to `alert()` or `alert_message=` instead:

```
if ta.crossover(close, ta.sma(close, 20))
    alert("Long " + syminfo.ticker + " @ " + str.tostring(close, format.mintick), alert.freq_once_per_bar_close)
```

Example: 3Commas-style strategy alert

```
BUY_JSON  = '{"action":"buy","bot_id":4635591,"token":"***"}'
SELL_JSON = '{"action":"sell","bot_id":4635591,"token":"***"}'

if ta.crossover(close, ta.sma(close, 20))
    strategy.entry("Long",  strategy.long,  alert_message = BUY_JSON)
if ta.crossunder(close, ta.sma(close, 20))
    strategy.close("Long",                  alert_message = SELL_JSON)
```

The alert panel can arm a durable rule with `trigger = order_fill` and a `webhookUrl`; every fill's `alert_message` is POSTed verbatim to the webhook.

Durable server-side alerts

A rule armed from the Alerts panel registers with the platform's alert service. The script continues executing with no chart open; `alertcondition()` fires are delivered through the configured channels (push / email / webhook).

Durable alert fields:

- `scriptId` / `version` – pinned to the version the chart ran
- `inputs` – the inputs snapshot the chart used
- `symbol` / `interval`
- `trigger` – what fires the rule:
    - `alert_condition` (default) – an `alertcondition()` / `alert()` call; optionally pinned to ONE `alertcondition` title (`conditionName`), otherwise any fires
    - `order_fill` – a strategy order filled; the fill's `alert_message` is the payload
    - `plot_condition` – a plot compared against a value, the price, or another script's plot (crossing / crossing up / crossing down / greater than / less than / entering, exiting, inside, outside channel / moving up, moving down, moving up %, moving down %), no code needed
- `freq` – `only_once` (fires then stops), `once_per_bar`, `once_per_bar_close`, `once_per_minute`, `all`
- `channels` – any combination of push, email, webhook
- `webhookUrl` – target for the webhook channel; must be a public http(s) host (private, loopback and link-local addresses are rejected, and re-checked on every delivery)
- `message` – overrides the script's default message; supports the placeholders above
- optional expiry – the rule deactivates itself at the chosen time

Marketplace scripts. A rule on a paid script is tied to your plan: when the subscription expires the alert service deactivates the rule (reason *subscription expired, renew it ... to re-arm this alert*) within ten minutes, or exactly at the plan end when it is known. Renewing does not re-arm it; open the Alerts panel and enable the rule again.

Safety rails: a rule that fires 15 times within 3 minutes is stopped automatically (runaway guard), and a webhook body is posted verbatim; a JSON `alert_message` arrives byte-for-byte, with `Content-Type: application/json` when it parses as JSON.

See the alerts tab in the Indicators hub for creating, pausing, deleting rules, and reading the delivery log.

Alerts and strategies

A strategy that uses `alert_message=` on orders can arm a `trigger = order_fill` rule; every filled order emits one alert carrying its `alert_message`. Combine with an `alertcondition()` or two for pre-fill signals that notify you before a bar closes.

Debugging alerts

- `log.info(message)` from inside the body prints to the Logs panel, mirroring exactly what would go into an alert.
- A test alert can be fired with `alert()` and `freq=alert.freq_all` to validate placeholder interpolation.

Bar States

`barstate.*` is a namespace of booleans that describe where the script currently is in the per-bar loop.

Core booleans

Name	True when
`barstate.isfirst`	The script is on the first bar of the chart's history.
`barstate.islast`	The script is on the current (newest) bar in the chart's data set.
`barstate.ishistory`	The current bar is a confirmed historical bar.
`barstate.isrealtime`	The current bar is being formed in real time.
`barstate.isconfirmed`	The current bar has closed (true on every historical bar; on the live bar only on its very last tick before close).
`barstate.isnew`	The runtime has just advanced to a new bar; true on the first tick of each new bar.
`barstate.islastconfirmedhistory`	The last historical bar BEFORE the realtime pass begins. Useful for one-shot setup that must survive to realtime.

Historical pass vs realtime pass

- Historical bars run once each. On every one `barstate.ishistory = true`, `barstate.isconfirmed = true`.
- The live bar re-runs on every tick. On most ticks `barstate.isrealtime = true`, `barstate.isconfirmed = false`. On its final tick (bar close) `barstate.isconfirmed = true` and the bar joins the historical pass.

Common patterns

One-time setup at the first bar

```
if barstate.isfirst
    log.info("Script loaded on " + syminfo.ticker)
```

Finalise only on confirmed bars

```
if barstate.isconfirmed and ta.crossover(close, ta.sma(close, 20))
    strategy.entry("Long", strategy.long)
```

Confirmed-only signals avoid intra-bar flapping; the entry fires once at bar close rather than every tick.

Draw once per bar, update on live

```
var label l = na
if barstate.isnew
    label.delete(l)
    l := label.new(bar_index, high, "")
if not barstate.isconfirmed
    label.set_text(l, str.tostring(close, "#.##"))
```

Snapshot the last historical bar

```
var float lastHistoricalClose = na
if barstate.islastconfirmedhistory
    lastHistoricalClose := close
```

Values written in `islastconfirmedhistory` survive into the realtime pass; the snapshot/restore cycle preserves `var` state.

Chart Information

The runtime exposes everything the script knows about the chart it is running on through three namespaces: `syminfo.*`, `chart.*`, and `timeframe.*`.

syminfo.*

Metadata about the chart symbol. Identity and contract fields come from the security master; company facts (sector, shareholders, analyst recommendations, shares outstanding) come from the scrip details the platform syncs per ISIN and are available for listed equities only. Any field the platform has no value for reads `""` (strings) or `na` (numbers); guard with `na()` or a string compare before using it.

Identity

Name	Type	Example value (TCS on NSE)
`syminfo.tickerid`	string	`"NSEE11536:TCS"` (exchange + segment + security id, then the ticker)
`syminfo.ticker`	string	`"TCS"`
`syminfo.symbol`	string	`"TCS"` (alias of ticker)
`syminfo.prefix`	string	`"NSEE11536"` (the part of tickerid before the colon)
`syminfo.root`	string	`"TCS"`
`syminfo.main_tickerid`	string	`"TCS"` (the security-master symbol; same for the equity and its derivatives)
`syminfo.description`	string	`"TATA CONSULTANCY SERV LT"`
`syminfo.type`	string	`"EQUITY"`, `"FUTIDX"`, `"FUTSTK"`, `"OPTIDX"`, `"OPTSTK"`, `"IDX"`, `"FUTCOM"`, … (security-master instrument name)
`syminfo.isin`	string	`"INE467B01029"`

Market

Name	Type	Value
`syminfo.currency`	string	`"INR"`
`syminfo.basecurrency`	string	`"INR"`
`syminfo.country`	string	`"IN"`
`syminfo.timezone`	string	`"Asia/Kolkata"`
`syminfo.session`	string	`"regular"` (the session type, comparable with `session.regular` / `session.extended`; it is not the trading hours string)

`syminfo.volumetype`	string	`"base"`

Contract

Name	Type	Example value
`syminfo.mintick`	float	0.05 (tick size from the security master)
`syminfo.minmove`	int	1
`syminfo.pricescale`	int	20 (`round(1 / mintick)`)
`syminfo.pointvalue`	float	1
`syminfo.mincontract`	int	lot size: 1 for equities, e.g. 25 for a NIFTY option
`syminfo.expiration_date`	int	contract expiry as a UNIX timestamp in milliseconds (IST); `na` for equities and indices; format with `str.format_time()`

Company facts (equities only)

Name	Type	Example value (TCS)
`syminfo.sector`	string	`"Information Technology"`
`syminfo.industry`	string	`"Computer Software & Consulting"`
`syminfo.shareholders`	int	2605180
`syminfo.shares_outstanding_total`	int	3618087518
`syminfo.shares_outstanding_float`	int	1021386106 (total shares x (1 - promoter holding %), a free-float estimate)
`syminfo.recommendations_buy`	int	21
`syminfo.recommendations_hold`	int	14
`syminfo.recommendations_sell`	int	5
`syminfo.recommendations_total`	int	40
`syminfo.employees`	int	always `na` (no data source yet)

Not available (compile error): `syminfo.current_contract`, `syminfo.recommendations_buy_strong`, `syminfo.recommendations_sell_strong`, `syminfo.recommendations_date`, and the six analyst targets `syminfo.target_price_average`, `_date`, `_estimates`, `_high`, `_low`, `_median`. The full platform list is on *What Is Not Supported*.

```
//@version=6
indicator("Symbol facts", overlay = true)
if barstate.islast
    buyShare = na(syminfo.recommendations_total) or syminfo.recommendations_total == 0 ? na : 100 *
syminfo.recommendations_buy / syminfo.recommendations_total
    txt = syminfo.description + "\n" + (syminfo.sector == "" ? "sector n/a" : syminfo.sector) + "\nBuy ratings: " +
(na(buyShare) ? "na" : str.tostring(buyShare, "#") + "%")
    label.new(bar_index, high, txt, style = label.style_label_down)
```

chart.*

Information about the chart's current state:

Name	Type	Meaning
`chart.left_visible_bar_time`	int	Epoch ms of the leftmost visible bar. Approximately the full-range start when the viewport is not yet reported.
`chart.right_visible_bar_time`	int	Epoch ms of the rightmost visible bar.
`chart.is_standard`	bool	true for candles / line / bar charts.
`chart.is_heikinashi`	bool	
`chart.is_renko`	bool	
`chart.is_kagi`	bool	
`chart.is_linebreak`	bool	
`chart.is_range`	bool	
`chart.is_pnf`	bool	Point-and-figure.
`chart.fg_color`	color	Theme foreground (adapts light/dark at render time).
`chart.bg_color`	color	Theme background.

timeframe.*

Name	Type	Meaning
`timeframe.period`	string	"5", "60", "D", "W", "M"
`timeframe.multiplier`	int	Numeric part (5 for "5", 1 for "D")
`timeframe.isintraday`	bool	

`timeframe.isminutes`	bool	
`timeframe.isseconds`	bool	
`timeframe.isdaily`	bool	
`timeframe.isweekly`	bool	
`timeframe.ismonthly`	bool	

Example: a status badge

```
var table t = table.new(position.top_right, 1, 1, bgcolor=color.new(color.black, 20))
if barstate.islast
    table.cell(t, 0, 0,
        syminfo.ticker + " · " + timeframe.period + " · " + str.tostring(close, "#.##"),
        text_color=color.white, text_halign=text.align_left, text_size=size.small)
```

Strings

Strings are 16-bit Unicode. Both `'single'` and `"double"` quote delimiters work; both support the same escape sequences.

Concatenation

`+` joins two strings. Mixed types do not concatenate automatically:

```
msg = "Close: " + str.tostring(close)
```

Conversion

Call	Result
`str.tostring(x)`	Decimal representation.
`str.tostring(x, format)`	Formatted. See Format strings.
`str.tonumber(s)`	float from a numeric string ("3.14" gives 3.14). Returns `na` on failure.
`int(x)` / `float(x)` / `bool(x)`	Type coercion.

Format strings

`str.tostring(value, format)` and `str.format(template, ...)` accept both the modern `format.*` constants and `#.##` patterns:

```
str.tostring(123.456,  format.price)       // "123.46"
str.tostring(123.456,  format.volume)      // "123.456"
str.tostring(0.1234,   format.percent)     // "12.34%"
str.tostring(123.4,    format.mintick)     // rounded to syminfo.mintick (0.05 -> "123.40")

str.tostring(123.456,  "#.##")             // "123.46"
str.tostring(123.456,  "0.0000")           // "123.4560"
str.tostring(1234567,  "#,##0")            // "1,234,567"
```

str.format(template, ...)

```
msg = str.format("Price {0} at {1}", close, syminfo.ticker)
```

Positional substitutions use `{0}`, `{1}`, ...; a width/precision suffix can be added: `"{0,.2f}"`, `"{0,8}"`.

str.format_time(epochMs, format?, timezone?)

```
when = str.format_time(time, "yyyy-MM-dd HH:mm", "Asia/Kolkata")
```

Default format is `"yyyy-MM-dd'T'HH:mm:ssZ"` and, like every time function on the platform, it formats in IST (Asia/Kolkata) unless a timezone is given.

Query functions

Call	Returns
`str.length(s)`	character count
`str.contains(s, needle)`	bool
`str.startswith(s, prefix)`	bool
`str.endswith(s, suffix)`	bool
`str.pos(s, needle)`	0-based index, or -1

`str.lower(s)` / `str.upper(s)`	case conversion
`str.split(s, sep)`	array
`str.replace(s, needle, repl, nth?)`	replaces the nth occurrence (default: first)
`str.replace_all(s, needle, repl)`	all-occurrences replace
`str.trim(s)`	strips leading / trailing whitespace
`str.match(s, regex)`	first match of a regular expression, or `""`
`str.repeat(s, n, separator?)`	n copies, joined by separator (default `""`)
`str.substring(s, from, to?)`	half-open slice

Method form

Every function above is also callable as a method on the string:

```
"Hello world".upper()           // "HELLO WORLD"
"a,b,c".split(",")              // ["a","b","c"]
"abcabc".replace_all("a", "X")  // "XbcXbc"
```

Concatenation with alerts and tables

```
alert("Breach on " + syminfo.ticker + " · " + str.tostring(close, format.price),
      alert.freq_once_per_bar_close)
```

Time

All times inside ChartScript are expressed as epoch milliseconds, UTC: a 64-bit integer. Chart display formats are a presentation concern; the script works with raw numbers.

Built-in time series

Name	Meaning
`time`	The bar's open time (ms UTC).
`time_close`	The bar's close time (ms UTC).
`timenow`	Wall-clock time (ms UTC): the executor's current moment.
`last_bar_time`	Open time of the chart's last loaded bar.

Historical access works as expected: `time[1]`, `time_close[5]`.

time(timeframe, session?, timezone?)

Returns the start time of the bar's **bucket** at the requested timeframe. Reset points: `ta.change(time("D")) != 0` is true exactly once per calendar day.

```
isNewDay   = ta.change(time("D")) != 0
isNewWeek  = ta.change(time("W")) != 0
isNewMonth = ta.change(time("M")) != 0

if isNewDay
    log.info("New session: " + str.format_time(time, "yyyy-MM-dd"))
```

With a session argument, the result is `na` outside the session window:

```
inRth = not na(time("", "0915-1530"))
```

With a timezone argument, the session is interpreted in that zone:

```
inLondon = not na(time("", "0800-1700", "Europe/London"))
```

time_close(timeframe, session?, timezone?)

Mirror of `time(timeframe, ...)` but returns the bucket's close time.

```
daysLeft = (time_close("D") - timenow) / (60 * 1000)   // minutes until EOD
```

timestamp(): build a time

```
timestamp(year, month, day, hour?, minute?, second?)        // IST
timestamp("tz", year, month, day, hour, minute, second)     // explicit timezone
timestamp("01 Jan 2020 09:15 +0530")                        // free-form string
```

The string form is handy for input defaults:

```
startTime = input.time(timestamp("01 Jan 2020 00:00 +0530"), "Start")
```

Decomposition

Function	Returns
`year(t?)`	int (defaults to the current bar)
`month(t?)`	int (1-12)
`dayofmonth(t?)`	int (1-31)
`dayofweek(t?)`	int (1=Sunday ... 7=Saturday); members `dayofweek.sunday` / `.monday` / ...
`hour(t?)`	int (0-23)
`minute(t?)`	int (0-59)
`second(t?)`	int (0-59)
`weekofyear(t?)`	ISO week number

An optional second argument lets you request a specific timezone:

```
hourIST = hour(time, "Asia/Kolkata")   // same as plain hour(time): IST is the default
```

Formatting

See *Strings*: `str.format_time(epochMs, format, timezone?)`.

```
str.format_time(time, "yyyy-MM-dd HH:mm:ss", "Asia/Kolkata")
```

Caveats

- `time` on a historical bar is immutable. `time` on the live forming bar updates on every tick (until the bar closes).
- `timenow` ticks faster than `time` during the live pass; it is the wall clock, not the bar's clock.

Timeframes

A timeframe names the bar size: `"5"` (5 minutes), `"60"` (1 hour), `"D"` (1 day), `"W"` (1 week), `"M"` (1 month), or seconds (`"1S"`, `"30S"`).

The chart's own timeframe

```
timeframe.period        // string: "5", "60", "D", ...
timeframe.multiplier    // int:    5,   60, 1, 1, 1
```

Boolean helpers:

```
timeframe.isseconds   timeframe.isminutes
timeframe.isintraday  timeframe.isdaily
timeframe.isweekly    timeframe.ismonthly
```

Comparing timeframes

```
timeframe.in_seconds("5")  == 300
timeframe.in_seconds("D")  == 86400
```

`timeframe.in_seconds(tf)` converts a timeframe string to seconds, letting you build arithmetic comparisons:

```
higherTf  = "D"
isHigher  = timeframe.in_seconds(higherTf) > timeframe.in_seconds(timeframe.period)
```

Buckets

- `time(tf)` returns the bar's bucket **start** at `tf`.
- `time_close(tf)` returns the bar's bucket **close** at `tf`.

```
newDay   = ta.change(time("D")) != 0
secondsLeftInHour = (time_close("60") - time) / 1000
```

See *Time*.

Chart-timeframe override: indicator(timeframe=...)

Force the whole script to compute at a different resolution regardless of the chart's own:

```
indicator("Weekly RSI (always)", overlay=false, timeframe="W")
plot(ta.rsi(close, 14), "RSI 14")
```

- The runtime fetches the chart's higher-timeframe bars once and runs the script on those; the chart's native bars are not seen.
- Pair with `timeframe_gaps=true` to carry the previous value forward on bars where the higher-timeframe hasn't produced a new bar yet.

Fetching from a different timeframe mid-script

Use `request.security()` when only part of the script needs a different timeframe (e.g. a daily RSI shown on a 5-minute chart).

Sessions

A session is a time window within a trading day. ChartScript represents a session as a string in `HHMM-HHMM` form, optionally followed by a day-of-week filter `:1234567` (1 = Sunday, 7 = Saturday).

session.* booleans

Name	Value
`session.ismarket`	Always `true`; executions run on regular-session data only.
`session.ispremarket`	Always `false` (no extended-hours bars are served).
`session.ispostmarket`	Always `false`.
`session.isfirstbar`	The first bar of a session.
`session.islastbar`	The last bar of a session.
`session.isfirstbar_regular`	First bar of the regular (non-extended) session.
`session.islastbar_regular`	Last bar of the regular session.

time(timeframe, session) for session membership

```
inRth = not na(time(timeframe.period, "0915-1530:12345", "Asia/Kolkata"))
```

- Returns the bar's time inside the session, `na` outside.
- Pair with a timezone argument when the session is defined in a non-exchange zone.

Example: tint the opening range

Because there are no extended-hours bars, session tinting is done with `time(tf, session)`:

```
inOpening = not na(time(timeframe.period, "0915-0930:23456"))
bgcolor(inOpening ? color.new(color.orange, 92) : na)
```

Example: reset at session open

```
openPrice = na
if session.isfirstbar
    openPrice := open
plot(openPrice, "RTH open", style=plot.style_stepline, color=color.gray)
```

Other Timeframes and Data

The `request.*` family lets a script reach outside its own symbol or timeframe: a higher-timeframe RSI on an intraday chart, a benchmark index's return, a stock's dividend / earnings history.

request.security(symbol, timeframe, expression, gaps?, lookahead?, ignore_invalid_symbol?, currency?)

```
dClose = request.security(syminfo.tickerid, "D", close)
plot(dClose, "Daily close on intraday chart", style=plot.style_stepline)
```

Parameter	Type / values	Default	Meaning
`symbol`	string	required	`""` means the current chart symbol; otherwise a ticker or secmaster key.
`timeframe`	string	required	`""` means the chart's own; or `"5"`, `"D"`, ...
`expression`	series / tuple	required	Evaluated on the requested series' bars.

`gaps`	`barmerge.gaps_on` / `barmerge.gaps_off`	`gaps_off`	`_on` returns `na` where no new HTF bar exists; `_off` carries the last value forward.
`lookahead`	`barmerge.lookahead_off` / `barmerge.lookahead_on`	`lookahead_off`	`_on` lets the chart bar see the HTF bar's close before it has formed; repaints, used only intentionally.
`ignore_invalid_symbol`	const bool	`false`	`true` returns `na` when the symbol has no data instead of raising.
`currency`	const string	ignored	Accepted but ignored; prices stay in the symbol's native currency.

Where the expression runs. The `expression` is compiled into a small program that executes **on the requested series' bars**, not on the chart's. That includes any script variable it refers to: a variable defined earlier in your script and used inside the expression is *re-evaluated in the requested context* (with the requested symbol's `close`, `high`, ... and `syminfo.*` / `timeframe.*` of the requested series), so

```
len   = input.int(14)
basis = ta.sma(close, len)                                  // chart context
dBasis = request.security(syminfo.tickerid, "D", basis)     // ta.sma over DAILY closes, not the chart's basis
sampled daily
```

returns the daily SMA, as expected. Only values that cannot be replayed there (drawings, `log.*`, `strategy.*`, `alert*`) are rejected inside an expression. User functions are allowed, and a `request.security` **nested inside another request's expression** is allowed too (the inner one runs in the outer request's context).

Inside the requested context `syminfo.*`, `timeframe.*` and `bar_index` describe the requested series, and `barstate.*` the requested bar.

Tuple returns. Return more than one series from a single `request.security` call:

```
[dOpen, dHigh, dLow, dClose] = request.security(syminfo.tickerid, "D", [open, high, low, close])
```

Dynamic requests. The `symbol` and `timeframe` arguments can be series values (an `input.symbol` + `input.timeframe` pair, say). The declaration must either leave `dynamic_requests` at its default `true` or set it explicitly:

```
sym = input.symbol("NSE:NIFTY", "Symbol")
tf  = input.timeframe("D",      "Timeframe")
val = request.security(sym, tf, close)
```

The executor fetches each unique (symbol, timeframe) pair once per computation (up to 512 pairs per script). What is supported: a symbol or timeframe that is constant for the run (inputs, constants) or a **loop variable over a fixed list** of symbols. Not supported: a *timeframe* that changes from bar to bar (rejected at compile time), and symbol chains whose value depends on another request's data deeper than a few levels.

request.security_lower_tf(symbol, timeframe, expression)

Lower-timeframe aggregation: for each chart bar, the expression is evaluated on **every lower-timeframe bar** that falls inside it, and the results are returned as an array:

```
m1Closes = request.security_lower_tf(syminfo.tickerid, "1", close)
mean     = m1Closes.avg()
plot(mean, "Mean 1-minute close within this bar")
```

- The returned array is ordered intrabar-first-to-last.
- Empty-array semantics: when the requested timeframe is **not** lower than the chart's, every bar returns `[]`. The executor treats "no data" as `[]` rather than raising.
- Tuple form:

```
[closes, volumes] = request.security_lower_tf(syminfo.tickerid, "1", [close, volume])
```

request.dividends(ticker, field, gaps?, lookahead?, currency?)

Dividend series.

```
dps = request.dividends(syminfo.tickerid, dividends.gross)
plot(dps, "Dividend per share")
```

`field`: `dividends.gross`, `dividends.net`, `dividends.payout_ratio`. Any other member is a compile error. `payout_ratio` is accepted but is **always `na`**; the payout ratio is not in the source data. Dividends land on the ex-date bar.

request.splits(ticker, field, ...)

Split and bonus events.

```
splitRatio = request.splits(syminfo.tickerid, splits.numerator)
```

`field`: `splits.numerator`, `splits.denominator` (a 1:2 bonus gives `1` / `2`; divide for the ratio).

request.earnings(ticker, field, ...)

Reported earnings.

```
eps = request.earnings(syminfo.tickerid, earnings.actual)
```

`field`: `earnings.actual`, `earnings.estimate`, `earnings.standardized` (quarterly EPS). `estimate` is accepted but is **always `na`**; analyst estimates are not in the source data. Earnings land on the report date.

request.financial(ticker, financial_id, period)

Fundamentals. `period` is `"FQ"` (fiscal quarter), `"FH"` (half year), `"FY"` (fiscal year), `"TTM"` (trailing twelve months) or `"D"` (daily series such as market cap). `financial_id` is a reference identifier like `"TOTAL_REVENUE"`, `"NET_INCOME"`, `"TOTAL_ASSETS"`, `"CASH_AND_SHORT_TERM_INVESTMENTS"`, `"TOTAL_LIABILITIES"`.

```
rev  = request.financial(syminfo.tickerid, "TOTAL_REVENUE", "FY")
yoy  = (rev - rev[1]) / rev[1]
```

Values land on the bar that opens the **next** fiscal period (the point when the figure first becomes known). An id that is not available for a given symbol injects a script warning into the Logs panel instead of returning `na` silently.

request.economic(country, series, ...)

Compiles with the warning *request.economic has no data source yet; the series is na*; every bar returns `na`. `request.quandl`, `request.seed`, `request.currency_rate` and `request.footprint` are not available (*Unknown function*).

Repainting

- `lookahead = barmerge.lookahead_off` (default) uses the HTF bar's **confirmed** close. The confirmed value becomes visible on the **last chart bar inside the HTF bar** (the chart bar whose successor opens at or after the next HTF bar; on a weekly request that is Friday's last bar, not Monday's first), and every earlier chart bar inside that HTF bar still reads the previous HTF value. The live chart bar reads the HTF bar's current forming value. This is the safe default.
- `lookahead = barmerge.lookahead_on` reads the forming HTF close on historical bars too; the HTF value *has not actually formed yet* at that chart-bar time. Signals derived from `lookahead_on` will repaint. Use only when the forward peek is intentional.

Non-Standard Chart Data

ChartScript indicators run on whatever chart type the user has selected. On candle / bar / line charts the series values match the raw symbol data. On non-standard chart types the series are transformed first.

chart.is_* booleans

Inspect the chart type at runtime:

```
chart.is_standard     // candles / bar / line / area
chart.is_heikinashi
chart.is_renko
chart.is_kagi
chart.is_linebreak
chart.is_pnf
chart.is_range
```

What changes on each type

Heikin-Ashi. `open`, `high`, `low`, `close` are **HA values**. The raw open/close must be fetched via `request.security()` with a `tickerid` that drops the chart style, or inspected via `request.security(ticker.standard(), ...)` which returns the raw candle data.

Renko / Line Break / Kagi / Point & Figure.

- `time` of a brick/line/leg is the *build time* of the brick, not a clock-aligned bar time.
- `volume` is the summed volume for the raw bars that built the brick.
- Multiple raw bars can produce zero bricks (consolidation) or one brick (breakout), so the index is not clock-regular.

The runtime assigns each brick a unique ordinal so trade markers and shape plots stay aligned when a chart bar can span zero or many raw bars.

Range bars. Each bar has a fixed price range (volatility-based). Volumes aggregate the raw bars that filled the range.

Fetching raw bars on a non-standard chart

```
[rawOpen, rawHigh, rawLow, rawClose] =
    request.security(ticker.standard(syminfo.tickerid), timeframe.period,
                     [open, high, low, close])
```

`ticker.standard(tickerid)` returns a canonical secmaster key that forces the fetch to use raw candles regardless of the chart style.

Compatibility

- `plot()`, `plotshape()`, `plotcandle()`, `plotbar()`, `line.new()`, `label.new()`, `box.new()` all work on every chart type. The host interprets the x coordinate as the brick/leg index, not a clock time.
- `strategy.*` entries fill at the rendered bar's close (or next bar's open under `process_orders_on_close = false`). Make sure the brick size is coarse enough that your signal isn't painting one entry per raw tick.

How it works, and the limits

- The chart sends its chart type and settings with the subscription, and the **executor transforms the raw bars server-side** before the script runs; the script never sees the raw candles on these chart types (use `ticker.standard` in a `request.security` call when you need them).
- **Renko box size follows the chart.** The chart resolves its box (ATR-based or fixed, after its own rounding) and sends the resolved value, so the server bricks match the ones you see. A re-resolved ATR (after scrolling far back) starts a fresh execution.
- Not honoured server-side yet: a Renko `reversal` amount greater than 1, and `source = high/low` (bricks are built from close). Point & Figure and Kagi follow the chart's reversal settings.
- The live bar on these chart types is the forming brick/column; `barstate.isconfirmed` turns true when the chart closes it.

Strategies

A strategy is an indicator that also places orders. The runtime maintains a position, matches entries against exits, applies commission and slippage, and feeds a backtest report.

Declaration

```
strategy("My strategy", overlay=true,
         initial_capital           = 100_000,
         default_qty_type          = strategy.cash,
         default_qty_value         = 10_000,
         commission_type           = strategy.commission.percent,
         commission_value          = 0.05,
         slippage                  = 1,
         pyramiding                = 1,
         process_orders_on_close   = false)
```

- Every argument to `strategy()` is `const`. See *Declaration Statements* for the full parameter list.
- `strategy()` scripts can also use everything an `indicator()` can: plots, shapes, drawings, tables, alerts.

Placing orders

`strategy.entry(id, direction, qty?, when?, limit?, stop?, oca_name?, oca_type?, alert_message?)`

```
if ta.crossover(close, ta.sma(close, 20))
    strategy.entry("Long", strategy.long)

if ta.crossunder(close, ta.sma(close, 20))
    strategy.close("Long")
```

- `direction`: `strategy.long` / `strategy.short`.
- `qty` omitted: the declaration's `default_qty_value`/`type` apply.
- `limit`=/`stop`= turn the entry into a *working order* that fills intrabar when the price touches the level.
- `oca_name`= groups working orders one-cancels-all; `oca_type`= is `strategy.oca.cancel`, `strategy.oca.reduce`, or `strategy.oca.none`.
- `alert_message`= is forwarded verbatim when the fill triggers an alert.

`strategy.exit(id, from_entry, stop?, limit?, trail_points?, trail_price?, trail_offset?, profit?, loss?, qty?, qty_percent?, alert_message?)`

Attach exit orders to an entry. All *tick-count* parameters (`profit`, `loss`, `trail_points`, `trail_offset`) are converted to price using the symbol's `mintick`.

```
strategy.exit("Long Exit", from_entry="Long",
              stop  = strategy.position_avg_price - 2 * ta.atr(14),
              limit = strategy.position_avg_price + 3 * ta.atr(14),
              trail_points = 10, trail_offset = 5)
```

- `stop`/`limit` are **prices**.
- `profit`/`loss` are **ticks from entry**.
- `trail_points`/`trail_offset` are **ticks**. A trailing stop requires BOTH `trail_points`/`trail_price` *and* `trail_offset`; omitting the offset drops the trail (a trailing stop needs both).

`strategy.close(id, when?, qty?, qty_percent?, alert_message?)` market-closes the named entry.

`strategy.close_all(alert_message?)` market-closes every open position.

`strategy.order(id, direction, qty?, when?, limit?, stop?, oca_name?, ...)` is lower-level: directly places an order that **nets**

FIFO against opposite open positions. Use `strategy.order` when `strategy.entry`'s reversal + pyramiding semantics are not what you want.

`strategy.cancel(id)` / `strategy.cancel_all()` remove working orders.

Fill model

By default a strategy calculates **once per bar** and its market orders fill at the **next bar's open**; stop / limit / trailing orders are evaluated against each bar's OHLC. Five declaration flags refine this; all of them are enforced by the backtester and the live engine:

Flag	Effect
`calc_on_every_tick=true`	The forming live bar recalculates on every price update, so orders can be placed and filled intrabar. Historical bars still calculate once per bar; add `calc_on_every_history_tick=true` (ChartScript extension) to replay history tick-by-tick too, so the backtest matches live behaviour.
`calc_on_order_fills=true`	After a fill, the script runs once more on the same bar. Use it when an exit must be placed from the updated `strategy.position_size` / `strategy.position_avg_price` without waiting for the next bar.
`use_bar_magnifier=true`	Fills of working orders are resolved against the **lower-timeframe bars** inside each chart bar (the actual price path) instead of the assumed open-high-low-close path. The backtester uses the finest timeframe it has data for and falls back to coarser ones; the report's header shows the timeframe(s) used, the number of intrabars, and how many chart bars they covered. Capped at 200,000 intrabars per run.
`fill_orders_on_standard_ohlc=true`	On non-standard charts (Heikin Ashi, Renko, Kagi, Line Break, Point & Figure, Range) orders fill at the underlying real OHLC prices rather than the synthetic bar's.
`margin_long` / `margin_short`	Percent of a position's value that equity must cover (`100` = cash only, `50` = 2x leverage). An entry that cannot be funded is rejected (counted in the report); when equity no longer covers the requirement a **margin call** liquidates enough of the position to restore it (shown as `Margin call` exits, with the smallest account size that would have avoided every call).

`slippage` (ticks) worsens every market and stop fill; `commission_type` / `commission_value` apply the commission model to each fill and to all `strategy.*` account values.

Order fill timing

Set by `process_orders_on_close`:

Setting	Fill
`process_orders_on_close = false` (default)	Market orders queue on the signal bar and fill at the **next bar's open** (platform default).
`process_orders_on_close = true`	Market orders fill at the **signal bar's close**.

`strategy.exit` orders (stop / limit / trailing) are always evaluated intrabar against each bar's OHLC, or against lower-timeframe bars with `use_bar_magnifier`. With `calc_on_every_tick`, market orders on the live bar fill at the current price of the tick that placed them.

strategy.risk.*

Risk limits are enforced per bar by the engine; a breach liquidates and may halt the strategy for the rest of the run or the trading day.

```
strategy.risk.allow_entry_in(strategy.direction.long)
strategy.risk.max_drawdown(10, strategy.percent_of_equity)
strategy.risk.max_position_size(5)
strategy.risk.max_intraday_loss(1000, strategy.cash)
strategy.risk.max_cons_loss_days(3)
strategy.risk.max_intraday_filled_orders(10)
```

Rule	Enforcement
`allow_entry_in(direction)`	Blocks disallowed-direction entries. Reversal closes are still permitted.
`max_position_size(n)`	Clamps entry size on every fill path.
`max_drawdown(v, type)`	`type=strategy.percent_of_equity`: percent of peak equity; `strategy.cash`: points. Breach liquidates + halts.
`max_intraday_loss(v, type)`	Day-scoped halt. Resumes next calendar day.
`max_cons_loss_days(n)`	Scored at UTC day rollover; halts after `n` consecutive losing days.
`max_intraday_filled_orders(n)`	Per-day cap on filled orders.

Both positional and reference named forms (`value=`, `type=`, `count=`, `entry_direction=`) are accepted.

Account values

`strategy.*` series expose the running account state (net of the declared commission model):

Name	Meaning
`strategy.position_size`	signed; positive = long, negative = short
`strategy.position_avg_price`	average entry price (or `na`)
`strategy.equity`	marked-to-market balance

`strategy.openprofit`	unrealised P&L on the current position
`strategy.netprofit`	cumulative realised P&L
`strategy.max_drawdown` / `max_runup`	running peak-to-trough / trough-to-peak
`strategy.wintrades` / `losstrades` / `eventrades`	counters
`strategy.closedtrades` / `opentrades`	counts

Trade accessors (`strategy.closedtrades.*` / `strategy.opentrades.*`): `entry_time`, `exit_time`, `entry_price`, `exit_price`, `profit`, `size`, `max_runup`, `max_drawdown`.

Backtest report

Running "Backtest" from the editor or the Strategy Tester runs the strategy over the chart's symbol, timeframe and chart type (Heikin Ashi etc. are honoured) in an isolated worker and shows the report in the **Account Manager** (bottom panel) as a strategy account tab:

- **Overview**: equity, drawdown and buy-and-hold curves over the tested range (deep histories are decimated to 6,000 points for drawing; metrics use every bar)
- **Performance**: net / gross profit, profit factor, max drawdown and run-up, Sharpe and Sortino (from monthly returns), CAGR, buy-and-hold return, commission paid, margin calls and liquidated quantity, entries rejected for insufficient funds, smallest account size without a margin call
- **Trades analysis**: counts, win rate, averages, largest win / loss, consecutive wins / losses, average bars held
- **List of trades**: every trade with entry / exit, size, profit, run-up / drawdown and margin; exportable to **XLSX** with a column chooser
- **Settings**: the effective declaration + overrides the run used (capital, sizing, slippage, commission, fill-model flags, magnifier tiers)

The Strategy Tester lets you re-run with a different period (default 5,000 bars, or a date window on deep history), initial capital and commission without editing the script; pyramiding and slippage can be overridden from Properties.

Alerts

See *Alerts*. `alert_message=` on orders rides the fill through to the alert channel; durable alert rules with `trigger = order_fill` POST the message to a webhook.

Libraries

A library is a reusable module of exported functions, types, and enums. Scripts import a library by owner, name, and version; version-pinned so a library update never silently breaks a consumer.

Creating a library

```
//@version=1
library("MyMa")

export smooth(float source, int length) =>
    ta.ema(source, length)

export method square(float this) =>
    this * this

export type Pivot
    int   bar
    float price
    bool  isHigh
```

Rules the compiler enforces

- The declaration must be `library("Name")`. The name must match `[A-Za-z0-9_]+`; library names ride import paths.
- At least one `export` function, type, or enum is required.
- Every exported function parameter needs an explicit type keyword (`float`, `int`, `bool`, `string`, `color`, a UDT, an enum, an array/map/matrix type).
- Exported function bodies cannot call `input.*()` or reference globals initialised from `input.*()`. Inputs belong to the consuming script.
- `alert()` *is* allowed inside exported bodies; the consumer surfaces the alerts.
- `request.*()` is allowed when the library's declaration keeps `dynamic_requests` at `true`.

Publishing

A library is a script, so publishing is the normal "Publish library" flow from the editor. The result lives at:

```
<ownerId>/<n>/<version>
```

Each save bumps the version; a library at `/1` cannot be edited in place; the next save becomes `/2`. Consumers pin a specific version (`import ownerId/Name/1 as m`), so your updates never silently reach them.

Importing a library

```
//@version=1
indicator("Uses MyMa")
import 1000000072/MyMa/1 as m

plot(m.smooth(close, 20),  "smoothed")
plot((5.0).square())
```

- The import path's **owner id** can be alphanumeric or fully numeric (your platform client id, e.g. `1000000072`).
- `as alias` is optional; omitted, the alias defaults to the library name.
- Imports are **top level only**. They cannot live inside a function or a conditional.
- Imports are **version-pinned**: `import x/MyMa/1` and `import x/MyMa/2` are separate libraries that can coexist in one script.

Libraries importing libraries

A library can import other libraries. The chain is resolved recursively; names imported inside a nested library are available to it under its own alias:

```
//@version=1
library("Outer")

import 1000000072/Signal/1 as sig

export wrap(float s) =>
    sig.smooth(s, 5)
```

The consumer only sees `Outer`'s exports; the inner `Signal` import is a private detail. Circular imports are detected and rejected with the full chain shown.

Visibility

- A library created in your account is private by default; only you can import it.
- Making a library public exposes it to the marketplace's `kind=library` listing.

Private libraries can still be imported from the *editor's* "Import library..." dialog by their owner.

Idiomatic style

- Keep library functions **pure**: a result depends only on its arguments plus the current bar's series values.
- Export every type a function signature references; the compiler enforces this for UDTs and enums.
- Group related helpers in one library rather than splitting fine-grained libraries.
- Document each export with a comment above the `export` line.

Example: a reusable zone helper

```
//@version=1
library("Zones")

export type Zone
    int   leftBar
    float top
    float bottom
    color col

export method draw(Zone this) =>
    box.new(this.leftBar, this.top, bar_index, this.bottom,
            bgcolor      = color.new(this.col, 85),
            border_color = this.col)

export of(bool isDemand, float px, float atrSize) =>
    isDemand ? Zone.new(bar_index, px, px - atrSize * 2, color.lime)
             : Zone.new(bar_index, px + atrSize * 2, px, color.red)
```

Consumer:

```
//@version=1
indicator("Zone user", overlay=true)
import 1000000072/Zones/1 as zones

zone = zones.of(close > open, close, ta.atr(14))
zone.draw()
```

Repainting

A script repaints when values displayed on a confirmed historical bar change *after* that bar has closed. Repainting is the main source of "my strategy was profitable in backtest but not live".

The three kinds of repainting

1. Last-value transfer (always safe). When a script runs on the forming bar, every bar-dependent calculation uses the current bar values. The moment the bar closes, the final values become the historical values. This is normal and not what we call repainting.

2. Series-mode functions in conditionals (bug). A series-mode function (`ta.sma`, `ta.rsi`, ...) must be called every bar for its state to stay in sync. Call it only inside an `if` branch and its state will diverge from the historical pass:

```
// WARNING: ta.rsi is skipped on bars where close == open; on those bars rsi reads stale
if close != open
    rsi = ta.rsi(close, 14)
    plot(rsi, "RSI")
```

The compiler warns in this case. The fix is to compute unconditionally and gate the display:

```
rsi = ta.rsi(close, 14)                          // computed every bar
plot(close != open ? rsi : na, "RSI")            // displayed selectively
```

3. `request.security(..., lookahead=lookahead_on)` (by design). `lookahead_on` reads the future close of a higher-timeframe bar before the chart's current bar has reached that time. The historical pass has access to that future value because history is complete; a live script never sees it.

```
// Repaints: the daily close that today's intraday bars read does not yet exist
dClose = request.security(syminfo.tickerid, "D", close,
                          lookahead = barmerge.lookahead_on)
```

Use `lookahead_off` (the default) when you need forward-safe behaviour. See *Other Timeframes and Data*.

Rules of thumb

- Prefer `request.security(..., lookahead=barmerge.lookahead_off)`.
- Keep series-mode `ta.*` calls at script scope, not inside conditionals.
- Confirm strategy entries with `barstate.isconfirmed` to avoid flapping on the live bar.
- A plot that uses `offset` to display a value "in the future" visually (lead indicators) is not repainting; it is relabelling what already happened. Document that in the plot title.

Debugging repaint claims

Add a `log.info` of the computed value at bar close:

```
if barstate.isconfirmed
    log.info("sig=" + str.tostring(signal) + " @ " + str.format_time(time))
```

Compare the log sequence between a historical replay and a live chart; any divergence surfaces repainting. Use `request.security(..., gaps=barmerge.gaps_on)` to see explicitly where the HTF value hasn't formed yet.

Strategy fill-timing

With `process_orders_on_close = false` (the default), a strategy's market order queues on the signal bar and fills at the next bar's open. The backtest fill is not repainting; the fill is a bar after the signal, matching live behaviour. Setting `process_orders_on_close = true` fills at the signal bar's close; that is still forward-safe for backtest reporting because the engine processes orders after the bar's close is known.

Part 2 – Language

Language Overview

Script Structure

A ChartScript source follows this general shape:

Type System

Every value in ChartScript has two tags: a type (what shape the value has) and a qualifier (how often it changes). The compiler uses both to decide where a value may be used.

Identifiers

An identifier is the name of a variable, constant, function, parameter, type, enum, library import, or input id.

Operators

ChartScript operators are grouped by purpose below. Precedence is listed in the final section.

Variable Declarations

ChartScript has three declaration forms. What differs between them is *when* the initializer runs and whether the value survives from bar to bar (and from tick to tick).

Declaration Statements

Every script contains exactly one declaration call. It identifies the kind of script and fixes properties that cannot change during a run.

Execution Model

A ChartScript script runs once per bar across the chart's visible history (and continues per-tick on the forming bar). Understanding this loop is essential — it is why every variable is...

Conditional Structures

ChartScript supports three forms of conditional selection: the ternary operator, if statements, and switch statements. All three can also be *expressions* that yield a value.

Loops

Loops iterate at the current bar. They are used for walks over arrays, index-based scans, or integer counters — not for reaching back through historical bars (use [n] for that).

User-Defined Functions

Functions let you factor a repeated calculation behind a name.

Methods

A method is a user-defined function that can be called with the . syntax on a receiver type. Methods are not a new language feature — they are a call-site convenience that chains...

Objects

A user-defined type (UDT) groups related values under one name. UDTs are the structured-data primitive of ChartScript — any collection of fields that belong together should be a UDT...

Enums

An enum declares a small, closed set of named string values. Enums are the right tool for category inputs (Up, Down, Flat) and for mode flags you compare against by name.

Arrays

An array is an ordered, mutable, zero-indexed collection of values of one type.

Maps

A map is a mutable key/value container. Keys are unique, insertion order is preserved for iteration.

Matrices

A matrix is a rectangular, two-dimensional, mutable collection of values of one type.

Built-Ins

This page is a map, not a full reference — each sub-area has its own page with signatures and examples.

Built-In Reference

Every built-in function the ChartScript compiler accepts — 449 functions grouped by namespace, with their signatures. Generated from the compiler on 2026-10-16.

Script Structure

Version annotation

The first non-comment line of a script may be a version annotation:

```
//@version=1
```

Any integer is accepted. The compiler's behaviour does not change by version. The editor's templates write //@version=1.

When omitted, the compiler accepts the script without warning (version is a hint, not a gate).

The annotation is a comment and can only appear once; placing it at the very top is strongly recommended.

Declaration statement

Every script contains exactly one declaration statement. It identifies the kind of script and sets properties that cannot change at runtime (title, overlay-vs-pane, number of drawing objects, strategy capital, etc.):

```
indicator("My indicator", overlay=true)
strategy("My strategy", default_qty_type=strategy.cash, default_qty_value=10_000, initial_capital=100_000)
library("MyLibrary")
```

indicator() — a visual script. Renders plots, shapes, drawings, tables.

strategy() — an indicator with position-management (strategy.entry/exit/close) that produces trades and a backtest report.

library() — a bundle of exported functions, types, and enums that other scripts can import.

A short alias is accepted: study("...") compiles exactly like indicator("...").

Body

The body is a sequence of statements. A statement is a declaration, an assignment, an expression (whose value is discarded), or a block-opening keyword (if, for, while, switch, once).

```
//@version=1
indicator("Example", overlay=true)

length = input.int(14, "Length", minval=1)
src    = input.source(close, "Source")

ma     = ta.sma(src, length)
dir    = close > ma ? 1 : -1

plot(ma, "MA", color=color.aqua, linewidth=2)
```

One statement per line (continuations)

A statement normally runs to the end of its line. Two cases let you spread a statement over several lines:

1. Line-leading continuation operators. When the next line begins with one of these operators (followed by a space or quote), it continues the previous statement:

== != <= >= ? :

+ - * % /

and or

```
total = volume
      + volume[1]
      + volume[2]
```

2. Unclosed brackets. Lines inside ( ), [ ], or { } concatenate until the matching bracket is reached. This is the preferred way to format multi-argument calls:

```
plot(
    series = ta.sma(close, 20),
    title  = "SMA 20",
    color  = color.aqua,
    linewidth = 2
)
```

Indented blocks

Blocks are introduced by if/for/while/switch/once or => and are indented. Four-space indentation is idiomatic; tabs are accepted but must be consistent inside a single block.

```
if close > open
    bodyColor = color.lime
    bodyAlpha = 15
```

else

```
    bodyColor = color.red
    bodyAlpha = 15
```

A block statement that is assigned (x = if …) yields the value of its last executed statement.

Comments

```
// A single-line comment.
```

bar = 2 // Trailing comments are allowed.

Block comments (/* ... */) are not supported.

Compiler annotations

Only //@version= is parsed as an annotation. All other comments are ignored.

Idiomatic structure

Experienced authors arrange a script in the following order so readers can scan it top-to-bottom:

Header comment — one-line description, author, licence.

Version annotation.

Declaration statement (indicator()/strategy()/library()).

Constants (var color BULL = color.lime, named thresholds).

Inputs — grouped with group= and arranged with inline= where it makes sense.

Function definitions — pure helpers that take explicit parameters.

Calculations — intermediate series.

Strategy calls / Drawings — strategy.entry/exit, line.new, box.new, label.new.

Visualisation — plot, plotshape, bgcolor, fill, hline.

Alerts — alertcondition (indicators) or alert() (universal).

```
//@version=1
// Volume-weighted MACD divergence detector.
// © you · MIT.

indicator("VW-MACD divergence", overlay=false, max_lines_count=100)

// ─── Constants ──────────────────────────────────────────────────
```

color BULL = color.lime

color BEAR = color.red

```
// ── Inputs ──────────────────────────────────────────────────────────
fast  = input.int(12, "Fast length",  minval=1,  group="MACD")
slow  = input.int(26, "Slow length",  minval=1,  group="MACD")
sig   = input.int(9,  "Signal length", minval=1, group="MACD")
pivot = input.int(5,  "Pivot bars",   minval=1, group="Divergence")

// ── Helpers ─────────────────────────────────────────────────────────
vwMacd(fast, slow) =>
    fastV = ta.ema(close * volume, fast) / ta.ema(volume, fast)
    slowV = ta.ema(close * volume, slow) / ta.ema(volume, slow)
    fastV - slowV

// ── Calculations ────────────────────────────────────────────────────
m    = vwMacd(fast, slow)
s    = ta.ema(m, sig)
hist = m - s

// ── Visuals ─────────────────────────────────────────────────────────
plot(hist, "Histogram", style=plot.style_histogram, color=hist >= 0 ? BULL : BEAR)
plot(m,    "MACD",      color=color.aqua, linewidth=2)
plot(s,    "Signal",    color=color.orange, linewidth=1)

// ── Alerts ──────────────────────────────────────────────────────────
alertcondition(ta.crossover(m, s),  "MACD cross up",   "VW-MACD crossed up")
alertcondition(ta.crossunder(m, s), "MACD cross down", "VW-MACD crossed down")
```

Type System

Types

Fundamental types

TYPE	EXAMPLE LITERALS	NOTES
int	0, 42, -7, 1_000_000	Underscore separators allowed.
float	3.14, .5, 1e-4, math.pi	IEEE-754 double; a bare integer literal auto-promotes.
bool	true, false, na	na is accepted anywhere a bool is expected and reads as false.
string	"text", 'text'	Double- and single-quote delimited; same semantics.
color	#RRGGBB, #AARRGGBB, color.aqua	24-bit RGB or 32-bit ARGB literals; see Colors.

The special value na (not available) belongs to every type and signals "no value yet" — common on the first bars of a moving average, on days without a dividend, and when a request.* call returns no data for the current bar.

Series and simple values

The series/simple split is the heart of the language. A value is series when it changes bar to bar (close, volume, ta.sma(close, 20), anything derived from a bar). A value is simple when it is fixed over the entire run (a literal, a declared var constant, the result of input.*()).

int length = 20 // simple int (input-style constant)

float ma = ta.sma(close, length) // series float

bool up = close > open // series bool (changes every bar)

Series values are historical: close[0] is the current bar's close, close[1] the previous bar's, and so on back to bar 0 of the chart's history. Simple values have no [n] dimension (they never changed, so looking back has no meaning).

User-defined types (UDTs) and enums

Scripts can declare their own structured types with type and enumerated string sets with enum:

```
type Pivot
    int   bar
    float price
    bool  isHigh

enum Trend
    bull = "Bullish"
    bear = "Bearish"
    flat = "Neutral"
```

Collection types

COLLECTION	WRITTEN AS

array	array<float>
map	map<string, int>
matrix	matrix<float>

Collections are mutable; array.push(), map.put(), and matrix.set() modify the collection in place.

Qualifiers

A qualifier records how often a value changes:

QUALIFIER	CHANGES…	PRODUCED BY
const	never	literals, var initializers to literals
input	at most once, when the user edits an input	input.*() calls
simple	never during a run	const + input together
series	every bar	close, volume, time, any arithmetic involving a series

Qualifiers form a widening order: const < input < simple < series. A location that requires const may accept nothing less constant; a location that accepts series accepts everything.

Example — this fails because length is series int (it is read on every bar), but ta.sma's length parameter wants simple int:

```
length = bar_index                // series int
ma     = ta.sma(close, length)    // compile error: expected simple int
```

The fix is to make length simple:

```
length = input.int(20, "Length")  // input int (narrows to simple when used)
ma     = ta.sma(close, length)    // OK
```

Automatic conversions

The compiler inserts the following conversions silently:

int → float (20 where a float is expected).

const → wider qualifier when a wider is required.

Any type → string when the target position is a format string (str.format placeholders).

No other conversion is implicit. Explicit conversion uses the str.tostring/str.tonumber/int(x)/float(x)/bool(x) family.

na and type-tagged variables

A variable declared with only an initial na has an inferred type, which the compiler locks to the type of the first non-na value you assign:

```
var pivotPrice = na    // inferred later
if ta.pivothigh(5, 5) != na
    pivotPrice := high[5]      // locks to float
```

To declare the type up front (useful when the first assignment may be na), use an explicit type keyword:

```
var float pivotPrice = na
```

Explicit type keywords: int, float, bool, string, color, plus any UDT or array<…>/map<…>/matrix<…> form.

Checking for na

Use na(x) or nz(x, replacement):

```
previousClose = na(close[1]) ? close : close[1]       // fallback
smoothed      = nz(ta.sma(close, 20), close)          // 0-replace via nz
```

Comparison against na with ==/!= is accepted but discouraged — na(x) is clearer and always correct.

Identifiers

Allowed characters

[a-zA-Z_][a-zA-Z0-9_]*

Must start with a letter or underscore.

May then contain letters, digits, and underscores.

Case-sensitive: Close, close, and CLOSE are three different names.

No limit on length in practice (every identifier is interned once).

Examples: rsi, rsi14, _hidden, userName, MACDHist, kernelFunction.

Invalid: 20sma (starts with a digit), rsi-14 (hyphen), rsi.14 (dot).

Reserved keywords

The following names are reserved by the grammar and cannot be used as identifiers:

and break by continue else

```
enum        export      false       for         if
import      in          library     method      na
```

not once or strategy switch

true type var varip while

In addition the following names are reserved at top level because they are declaration keywords:

```
indicator   library   strategy   study
```

(Inside a function body they can appear as parameters — the restriction applies to top-level declaration targets.)

Built-in names

Many identifiers are owned by the standard library. Shadowing them with a = or var declaration is allowed but hides the built-in for the rest of the script:

```
sma(x, y) => x + y          // shadows ta.sma for bare `sma(…)` calls
plot(sma(close, 2))
```

The namespaced form (ta.sma) is always available regardless of shadowing. Common built-in namespaces:

Series — open, high, low, close, volume, time, bar_index

Namespaces — ta.*, math.*, str.*, array.*, map.*, matrix.*, input.*, request.*, strategy.*, color.*, shape.*, size.*, location.*, line.*, box.*, label.*, table.*, plot.*, hline.*, format.*, scale.*, display.*, extend.*, font.*, position.*, syminfo.*, timeframe.*, session.*, barstate.*, dayofweek.*, chart.*, currency.*, log.*

For an exhaustive list see Built-ins.

Short-form names

Many functions can be called without their namespace. ChartScript resolves bare names only after user-function lookup, so your own sma definition still wins:

TA globals routed to ta.*: sma, ema, rma, wma, vwma, swma, alma, hma, rsi, atr, tr, stdev, dev, variance, correlation, cci, cmo, cog, mfi, mom, roc, sar, stoch, tsi, wpr, linreg, cum, obv, accdist, highest, lowest, highestbars, lowestbars, barssince, valuewhen, crossover, crossunder, cross, rising, falling, change, pivothigh, pivotlow, percentrank, bbw, kcw, median, vwap

Math globals routed to math.*: avg, max, min, abs, ceil, floor, round, sqrt, pow, exp, log, log10, sign, sin, cos, tan, asin, acos, atan, sum, todegrees, toradians

String routed to str.*: tostring, tonumber

Flow: iff(cond, then, else) → cond ? then : else

Short color names also resolve as bare identifiers (aqua, silver, red, …) and fold to the same hex as color.aqua, color.silver, color.red; a local variable named red shadows the constant.

Suggested style

lowerCamelCase for variables and function parameters: kernelLength, isTrendingUp.

UPPER_SNAKE_CASE for script-level constants: MAX_ZONES, BULL_COLOR.

PascalCase for types and enums: type Pivot, enum Trend.

Prefix a variable with its unit when it carries one: len, lenBars, lenPct, lenPrice.

See Style guide for the full project conventions.

Operators

Arithmetic

OPERATOR	MEANING	EXAMPLE
+	add	a + b
-	subtract / negate	a - b, -a
*	multiply	a * b
/	divide	a / b
%	modulo	a % b

Division is always floating-point. / on two integer operands still produces a float — use math.floor(a / b) for truncation.

Comparison

OPERATOR	MEANING	RETURNS
==	equal	bool
!=	not equal	bool
<	less than	bool
>	greater than	bool
<=	less than or equal	bool
>=	greater than or equal	bool

NaN propagates: na == na is na (which reads as false), as is na == 0. Use na(x) to test for na.

Logical

and or not

Logical operators work on booleans (and on anything that converts to a boolean — na and 0 read as false, every other number reads as true). Short-circuit evaluation applies:

```
safe = volume > 0 and (high - low) / volume < 100
```

String concatenation

+ concatenates two strings. Mixed-type concatenation is not implicit — convert first:

```
msg = "Close is " + str.tostring(close)
```

Conditional expression

```
value = cond ? thenExpr : elseExpr
```

Nested ternaries are allowed and often read best when each branch is on its own line:

regime =

```
      close > ta.sma(close, 50) ? "up"
    : close < ta.sma(close, 50) ? "down"
    : "flat"
```

The two branches must share a type (both int, both float, both the same UDT, etc.). An int on one side and a float on the other auto-promotes to float.

Assignment and reassignment

```
rsi = ta.rsi(close, 14)        // declaration + initial assignment
rsi := nz(rsi, 50)             // reassignment (`var`-style target)
```

= is first-time declaration.

:= is reassignment of an existing var/varip target or of a local that lives inside the current function body.

A plain = cannot reassign; a plain := cannot declare.

Historical access ([])

```
prevClose = close[1]
rsiAgo    = ta.rsi(close, 14)[5]
```

x[n] returns x as it was n bars ago. n must be a non-negative integer that is itself simple or series. The implementation caches each referenced series' recent history (depth grows dynamically as scripts request further-back values).

Collection indexing

Arrays, maps, and matrices use method-call accessors (array.get, map.get, matrix.get) rather than square brackets. The square-bracket form is reserved for historical lookback.

Member access (.)

syminfo.ticker

```
color.new(color.red, 50)
```

pivot.price

. navigates namespaces, UDT fields, and enum members.

in (membership test)

Used only inside for ... in to iterate over an array:

```
for value in prices
    total += value
```

Precedence

From highest to lowest:

[] (historical access)

. (member access), function call (...)

Unary -, not

*, /, %

+, -

<, >, <=, >=

==, !=

and

or

? : (ternary)

=, := (assignment / reassignment — right-associative)

Parentheses override precedence as expected.

Variable Declarations

Default (name = expr)

```
rsi = ta.rsi(close, 14)
dir = close > open ? 1 : -1
```

The right side is re-evaluated every bar and the name holds the new value. This is what you want for anything that depends on the current bar (indicators, bar-state booleans, derived series).

```
var name = expr
var hitCount    = 0
var lastPrice   = na
var bullColor   = color.lime
```

The initializer runs once — the first time execution reaches the statement, which is bar 0 for a top-level declaration but can be much later for a var inside a block — and is skipped thereafter. The variable can be modified later with :=:

```
var hitCount = 0
if close > ta.sma(close, 20)
    hitCount := hitCount + 1
plot(hitCount)
```

var is the right tool whenever you need a value to carry across bars without being overwritten (counters, running totals, "last X" trackers, pivots).

```
varip name = expr
varip tickCount = 0
if barstate.isconfirmed
    tickCount := 0
```

else

```
    tickCount := tickCount + 1
```

varip ("variable in place") behaves like var on confirmed bars but is not rolled back on intra-bar updates. It is the right tool when you need to count ticks or accumulate between updates on the forming bar.

Semantics in detail:

On a historical bar, varip is identical to var.

On a live forming bar, each new tick continues updating the existing value instead of restoring the snapshot from the bar's close.

When the bar confirms, the final value becomes the new snapshot and the next bar's forming phase continues from there.

Explicit type keywords

A declaration may start with a type keyword to pin the variable's type up front. This is required when the initializer is na (otherwise the compiler cannot infer the type):

```
var float peakPrice   = na
var line  activeLine  = na
var array<float> buf  = array.new<float>()
```

Valid type keywords: int, float, bool, string, color, any UDT name, any enum name, array<T>, map<K, V>, matrix<T>, plus the special reference types line, box, label, table, polyline, linefill.

Tuple declarations

A call that returns multiple values can be destructured in a single declaration:

```
[middle, upper, lower] = ta.bb(close, 20, 2)
```

The number of names on the left must match the arity of the returned tuple.

Reassignment (:=)

Only var and varip variables, plus locals inside the current function body, can be reassigned:

```
var runningMax = na
if na(runningMax) or high > runningMax
    runningMax := high
```

Attempting to reassign a plain-= top-level variable is a compile error — those are intended as per-bar derived values and are overwritten by their initializer every bar instead.

Scope

Top-level declarations are visible everywhere below them in the script.

Declarations inside a block (if/for/while/function body) are visible only inside that block. A var inside a block still persists across bars but is only addressable from within its block.

```
if ta.crossover(close, ta.sma(close, 20))
    var crossCount = 0
    crossCount := crossCount + 1
    plot(crossCount, "crosses so far")
```

```
var inside a block
```

Because the initializer runs when the statement is reached, a var declared inside a conditional block initializes on the first bar that enters the block — a common pattern for a table that only needs to exist on the last bar:

```
if barstate.islast
    var table t = table.new(position.top_right, 2, 2)   // created once, on the first "last" bar
    table.cell(t, 0, 0, "close")
    table.cell(t, 1, 0, str.tostring(close))
```

A var int count = 0 inside such a block likewise starts counting on the first bar that reaches it and keeps its value from then on.

Declaration Statements

```
indicator()
indicator(title, shorttitle, overlay, format, precision, scale,
          max_bars_back, timeframe, timeframe_gaps, explicit_plot_zorder,
          max_lines_count, max_labels_count, max_boxes_count,
          max_polylines_count, dynamic_requests, behind_chart)
```

PARAMETER	TYPE	DEFAULT	MEANING
title	const string	required	Appears in the chart legend and the publish-to-marketplace flow.
shorttitle	const string	title	Used when space is tight (legend chip).
overlay	const bool	false	true renders on the price pane; false opens a separate pane.
format	const	format.inherit	Default label format for plots. format.price, format.volume, format.percent, format.mintick.
precision	const int	–	Default decimal places for plot labels. Overridden per plot().
scale	const	scale.right	scale.left, scale.right, scale.none (no own axis).
max_bars_back	const int	–	Accepted; the runtime always keeps the full loaded history (5,000 bars), so this is a no-op.
timeframe	const string	""	Forces the script to compute at this timeframe. See Timeframes.
timeframe_gaps	const bool	true	When timeframe= is set, fill gaps between upstream bars with the previous value.

explicit_plot_zorder	const bool	false	Honored implicitly — plots always draw in declaration order.
max_lines_count	const int	500	Cap on live line.new() objects; the oldest is dropped silently beyond it. Range 1–500.
max_labels_count	const int	500	Cap on live label.new() objects. Range 1–500.
max_boxes_count	const int	500	Cap on live box.new() objects. Range 1–500.
max_polylines_count	const int	100	Cap on live polyline.new() objects. Range 1–100.
dynamic_requests	const bool	true	Enables request.* calls whose arguments vary per bar.
behind_chart	const bool	false	true renders this indicator behind the price series.

```
//@version=1
indicator("Smoothed RSI", "sRSI", overlay=false, precision=2, max_labels_count=200)

strategy()
```

A strategy is an indicator that can place orders:

```
strategy(title, shorttitle, overlay, format, precision, scale,
         pyramiding, calc_on_order_fills, calc_on_every_tick,
         max_bars_back, process_orders_on_close, close_entries_rule,
         slippage, commission_type, commission_value,
         default_qty_type, default_qty_value, initial_capital,
         currency, use_bar_magnifier, fill_orders_on_standard_ohlc,
         calc_on_every_history_tick, margin_long, margin_short,
         max_lines_count, max_labels_count, max_boxes_count,
         max_polylines_count)
```

The visual-related parameters mirror indicator(). Strategy-specific parameters:

PARAMETER	TYPE / VALUES	DEFAULT	MEANING
pyramiding	const int	1	Max simultaneous same-direction entries.
slippage	const int (ticks)	0	Points worsened on every market/stop fill.
commission_type	strategy.commission.percent / cash_per_contract / cash_per_order	percent	How commission_value is interpreted.
commission_value	const float	0	Rate for the chosen model.
default_qty_type	strategy.fixed / strategy.cash / strategy.percent_of_equity	fixed	How default_qty_value sizes an entry when qty= is omitted.
default_qty_value	const float	1	Size in the chosen units.
initial_capital	const float	100000	Starting equity for the backtester.
currency	const	ignored	Accepted but ignored; backtest reports in the symbol's currency.
process_orders_on_close	const bool	false	false queues market orders to fill at the next bar's open (platform default); true fills at the signal bar's close.
calc_on_every_tick	const bool	false	Recalculate on every live price update of the forming bar (orders can fill intrabar), not only at bar close. Historical bars still calculate once per bar unless calc_on_every_history_tick is set.
calc_on_every_history_tick	const bool	false	ChartScript extension: replay historical bars tick-by-tick as well (bar OHLC path, or lower-timeframe bars with use_bar_magnifier) so backtests match the live every-tick behaviour. Slower.
calc_on_order_fills	const bool	false	After an order fills, run the script once more on the same bar so logic reading strategy.position_size reacts immediately.
use_bar_magnifier	const bool	false	Resolve limit / stop / trailing fills against lower-timeframe bars inside each chart bar instead of the assumed OHLC path. The backtester uses the finest timeframe available and falls back to coarser ones; the report lists the tiers used.
fill_orders_on_standard_ohlc	const bool	false	On Heikin Ashi, Renko and other non-standard charts, fill at the real OHLC prices rather than the synthetic bar's.
margin_long / margin_short	const float (%)	100	Percent of a position's value that equity must cover (100 = no leverage, 50 = 2×). Entries that cannot be funded are rejected; a margin call liquidates part of the position. The report shows margin calls and the smallest account size that avoids them.
close_entries_rule	"FIFO" / "ANY"	FIFO	Which entry a partial strategy.close / strategy.exit reduces first.
risk_free_rate, backtest_fill_limits_assumption	const	–	Accepted but ignored.

```
//@version=1
strategy("SMA cross", overlay=true,
         initial_capital=100000,
         default_qty_type=strategy.cash,
         default_qty_value=10000,
         commission_type=strategy.commission.percent, commission_value=0.05,
         slippage=1, process_orders_on_close=false)
```

See Strategies for the full picture, including strategy.risk.*.

```
library()
library(title, overlay, dynamic_requests)
```

PARAMETER	TYPE	DEFAULT	MEANING
title	const string	required	The import name. Must match [A-Za-z0-9_]+ because it rides the import path (owner/Name/version).
overlay	const bool	false	Advisory only; a library has no plots of its own.
dynamic_requests	const bool	true	Allow request.* calls inside exported functions.

A library must export at least one function, type, or enum:

```
//@version=1
library("MyMa")

export smooth(float source, int length) =>
    ta.ema(source, length)
```

See Libraries for publishing and importing.

Property restrictions

Every argument to a declaration statement must be a const value — a literal, a color.*/shape.*/strategy.* member, or a value computed only from other const operands.

No input.*() call may feed a declaration — the dialog does not exist yet when the declaration runs.

Only one declaration statement is allowed per script; the compiler rejects a second as "Script already declared".

Execution Model

The main loop

Conceptually the runtime does this:

```
for i = 0 … barCount - 1:
    makeBarState(i)
    runBody()                // every top-level statement in order
    collectOutput(i)
```

i = 0 is the oldest bar in the loaded history, i = barCount - 1 is the current (possibly forming) bar.

Every top-level statement runs on every iteration. A plain = assignment is therefore re-computed every bar.

var initializers run only on the first bar they are reached; var targets then survive across iterations.

Visual output (plot(), plotshape(), drawings) are collected into per-bar arrays that the host renders after the loop finishes.

Series access during the loop

When the body is at bar i, close returns the close of bar i. close[1] returns the close of bar i − 1, close[2] of bar i − 2, and so on. The runtime maintains a per-series ring buffer that grows automatically up to the deepest [n] the script ever requests.

```
rsi = ta.rsi(close, 14)
```

up = rsi > rsi[1] // true when RSI rose from the previous bar

On bar 0 the lookback is na (there is no previous bar) and bar-state booleans are false. All [n] accesses with n >= i + 1 evaluate to na.

Historical pass vs live pass

Two phases execute the same body:

Historical pass — bars 0 through the last confirmed bar are iterated once; barstate.isconfirmed is true on each.

Live pass — the forming bar re-runs on every tick with barstate.isconfirmed = false, until it closes and joins the historical pass.

During the live pass the runtime creates a snapshot of all state at the last confirmed bar. Each tick restores the snapshot, runs the body once, and collects the forming bar's output. This is what makes a strategy's "process_orders_on_close = false" market order

queue up on the signal bar and fill at the next bar's open — the queue sits in the strategy state carried across the snapshot/restore cycle.

varip variables (see Variable declarations) opt out of snapshot rollback and keep updating as ticks arrive.

Repainting

A script "repaints" when values displayed on confirmed bars change after the fact. In ChartScript this can happen for two reasons:

A built-in that reads from [1] or further back inside a function body — the historical pass computes the function with the historical values, but a live-pass re-run uses the forming bar's forming values. Confining request.security(..., lookahead=barmerge.lookahead_off) or using ..[1] eliminates the forward peek.

Explicit lookahead (request.security(..., lookahead=barmerge.lookahead_on)) — accepted, but documented as "repaints by design"; use it only when the forward peek is intentional.

See Repainting for the full catalogue.

Side effects and ordering

Statements inside a function body execute top-to-bottom just like at script scope. Function calls are evaluated when they appear — plot(), strategy.entry(), label.new(), and log.info() all fire at the point of the call, so order inside the body determines rendering z-order and strategy order sequence:

```
if ta.crossover(close, ta.sma(close, 20))
    strategy.entry("Long", strategy.long)
    label.new(bar_index, high, "Long")

if ta.crossunder(close, ta.sma(close, 20))
    strategy.close("Long")
    label.new(bar_index, low,  "Exit", color=color.red)
```

Series-mode functions

Many library functions carry state across bars (ta.sma, ta.rsi, ta.pivothigh, ...). They must be called from the top-level body on every bar — otherwise their internal buffer skips bars and goes out of sync. Calling such a function conditionally will trigger a compile warning:

```
if close > open
    rsi = ta.rsi(close, 14)   // ⚠ warning: series fn inside conditional
```

The idiomatic fix is to compute the series unconditionally and gate the consumption:

```
rsi = ta.rsi(close, 14)
if close > open
    plot(rsi, "RSI on green bars")
```

(There is no actual gating of plot behind a condition — the typical pattern is plot(cond ? rsi : na).)

Order of evaluation

Within a single statement, operands evaluate left-to-right; function arguments evaluate left-to-right before the call. Short-circuit and/or apply as expected.

Conditional Structures

Ternary operator

```
dir = close > open ? 1 : -1
color = close > open ? color.lime : color.red
```

Nested ternaries read best when each branch is on its own line:

regime =

```
      close > ta.sma(close, 200) ? "bull"
    : close < ta.sma(close, 200) ? "bear"
    : "neutral"
```

Both branches must share a type (see Operators).

```
if statement
if cond
    body
else if cond2
    body2
```

else

```
    body3
```

The if/else if/else chain can be as long as you like.
Blocks are indented; the dedent closes the block.
Any number of else if branches is allowed.

```
if close > ta.sma(close, 20)
    bgColor = color.new(color.lime, 90)
else if close < ta.sma(close, 20)
    bgColor = color.new(color.red,  90)
```

else

```
    bgColor = color.new(color.gray, 90)

if as an expression
```

Assign the result of an if block to a variable — the last statement of the taken branch becomes the value:
score =

```
    if close > open
        1
    else if close < open
        -1
    else
        0
```

Branches must share a type.

Nested if

Nest as deeply as needed:

```
if cond1
    if cond2
        innerBody
    else
        otherInner
```

else

```
    outer

switch statement
```

Two forms are supported.

Expression switch

```
tf = input.string("D", "Timeframe", options=["30", "60", "D", "W"])
```

bars =

```
    switch tf
        "30" => 48
        "60" => 24
        "D"  => 1
        "W"  => 5
        => 1               // default (no match)
```

The subject expression is evaluated once; its value is compared with each arm's match value with ==.
The right side of => is either a single expression or an indented block whose last statement is the value.
An arm with no match (=> …) is the default. Omitting it is allowed; a non-match returns na.

Boolean-chain switch

Omit the subject to use each arm's left side as a condition — the first truthy arm wins:
color =

```
    switch
        close > open and volume > ta.sma(volume, 20) * 2 => color.lime
        close < open and volume > ta.sma(volume, 20) * 2 => color.red
        close > open => color.new(color.lime, 60)
        close < open => color.new(color.red,  60)
        => color.gray
```

This is often clearer than a long if ... else if ... chain.

once

A compact form for "run this block the first time the condition is true, then never again":

once close > ta.sma(close, 200)

```
    label.new(bar_index, high, "First close above 200-SMA")
```

Equivalent to:

```
var fired = false
```

if not fired and close > ta.sma(close, 200)

```
    fired := true
    label.new(bar_index, high, "First close above 200-SMA")
```

Bare once (no condition) runs its block on the first bar of the chart's history.

Loops

```
for … to
total = 0.0
for i = 1 to length
    total := total + close[i]
```

for i = start to end iterates i from start to end inclusive.

When end < start, iteration counts down.

Both bounds are evaluated once at the start of the loop; a change inside the body does not affect iteration count.

Explicit step with by

```
for i = 0 to 100 by 10
    log.info(str.tostring(i))
```

by is the step magnitude (always positive).

Direction is still inferred from start/end.

```
for … in
```

Walks an array:

```
prices = array.from(100.0, 101.5, 102.0, 103.25)
sum = 0.0
for value in prices
    sum := sum + value
```

Walk with index as well:

```
for [index, value] in prices
    log.info(str.tostring(index) + " → " + str.tostring(value))
```

while

```
i = 0
remaining = volume
while remaining > 0 and i < 10
    remaining := remaining - 1000
    i := i + 1
```

Use while when the loop may terminate on a condition that is tested each iteration. Every loop carries a guard of 100,000 iterations per loop per bar: when it trips, that loop exits and the script continues with the next statement (no error is raised). Nested loops are guarded independently.

```
break and continue
```

Both work inside any of the three loop forms:

```
for i = 0 to 100
    if close[i] == 0 or na(close[i])
        continue              // skip this iteration
    if close[i] > close[0] * 2
```

break // stop the loop entirely

```
    log.info(str.tostring(close[i]))
```

Loops as expressions

A loop can produce a value — the result is the value of the last iteration's final statement:

```
maxGap = for i = 1 to 20
    math.abs(close[i] - close[i - 1])
```

If the loop never iterates (empty range), the result is na.

Compile-time safety

A loop whose body uses a series-mode function (ta.sma, ta.rsi, ...) does not turn that function series-mode-safe — the function still needs to be called on every bar from the top level. The compiler emits a warning.

for ... in over na is a no-op.

Modifying the loop array inside a for ... in is undefined behaviour; prefer building a new array and swapping references.

User-Defined Functions

Single-expression form

```
smooth(source, length) =>
    ta.ema(source, length)
```

The right side of => is a single expression; its value is what the function returns.

Parameters have no type annotations here — the compiler infers them from each call site.

Block form

A multi-line body is indented; the last statement is the return value:

```
rangeStretch(src, lookback) =>
    high  = ta.highest(src, lookback)
    low   = ta.lowest(src,  lookback)
    span  = high - low
    span == 0 ? 0.0 : (src - low) / span
```

Typed parameters (required for exported library functions)

Prefix a parameter with its type to pin it explicitly. This is mandatory for a library() script's exported functions:

```
export smoothTyped(float source, int length) =>
    ta.ema(source, length)
```

Accepted type keywords: int, float, bool, string, color, any UDT, any enum, array<T>, map<K, V>, matrix<T>.

Tuple return

Return multiple values with a bracketed list:

```
bollinger(src, len, mult) =>
    basis = ta.sma(src, len)
    dev   = mult * ta.stdev(src, len)
    [basis, basis + dev, basis - dev]

[mid, up, lo] = bollinger(close, 20, 2)
```

The caller must destructure or discard the result — a tuple return cannot be assigned to a single variable.

Default arguments

Specify defaults with = after the parameter name:

```
smooth(source, length = 20) =>
    ta.ema(source, length)

m1 = smooth(close)        // length defaults to 20
m2 = smooth(close, 50)
m3 = smooth(close, length = 50)   // named argument
```

Named arguments can be used at any call site, in any order.

Nested functions

A function body may declare helper functions, visible only inside the enclosing function:

```
weightedMean(src, len) =>
    weight(i) => math.exp(-i / 10.0)
    total  = 0.0
    norm   = 0.0
    for i = 0 to len - 1
        total := total + src[i] * weight(i)
        norm  := norm  + weight(i)
    total / norm
```

Overloading

Multiple functions can share a name as long as the arity or the first parameter's type differ:

```
roll(float x) => x * 1.0
roll(int x)   => x * 1
roll(x, len)  => ta.sma(x, len)
```

The compiler picks the best match at each call site; ambiguous calls raise an error.

Recursion

Recursive calls are not supported — a user-defined function cannot call itself. Rewrite with a loop or an explicit stack.

Series semantics

A function that transitively calls a series-mode built-in (like ta.ema) is itself a series-mode function. It must be called on every bar from the top-level body for its internal state to stay in sync. The compiler warns when a series-mode call sits inside a conditional block.

Methods

Declaring a method

Prefix a function definition with method:

```
method double(float this) =>
    this * 2.0

method magnitude(array<float> this) =>
    sumSquares = 0.0
    for v in this
        sumSquares := sumSquares + v * v
    math.sqrt(sumSquares)
```

The first parameter is called this by convention; its type determines the receiver type.

Everything else works like a user-defined function (default arguments, overloads, nested helpers).

Calling a method

Both forms are equivalent:

```
x = 5.0
plot(x.double())              // method call
plot(double(x))               // ordinary function call
```

Methods chain:

```
prices = array.from(1.0, 2.0, 3.0, 4.0)
m = prices.magnitude()        // 5.477…
```

Built-in methods

Several built-in types expose methods out of the box:

Arrays: .push(), .pop(), .get(), .set(), .size(), .sort(), .avg(), .sum(), .stdev(), .first(), .last(), .slice(), .join(), ...

Maps: .put(), .get(), .remove(), .contains(), .size(), .keys(), .values(), ...

Matrices: .get(), .set(), .rows(), .columns(), .add_row(), .add_col(), ...

Strings: .length(), .contains(), .lower(), .upper(), .split(), .replace_all(), ...

UDTs: all fields and user-defined methods.

```
arr = array.from(3.0, 1.0, 2.0)
arr.sort()
log.info(arr.join(","))        // "1,2,3"
```

Methods on UDTs

Methods on user-defined types pair naturally with the type:

```
type Pivot
    int   bar
    float price
    bool  isHigh

method label(Pivot this) =>
    this.isHigh ? "H" : "L"

method distanceTo(Pivot this, float px) =>
    math.abs(this.price - px)

p = Pivot.new(bar_index, high, true)
log.info(p.label() + " at " + str.tostring(p.price))
log.info("gap: " + str.tostring(p.distanceTo(close)))
```

Exporting methods

Inside a library() script, a method may be exported just like a regular function:

```
//@version=1
library("MyMath")

export method square(float this) =>
    this * this
```

Users of the library call it the same way:

```
import 1000000072/MyMath/1 as mm
y = (3.0).square() + mm.square(4.0)
```

Exported methods follow the standard library contract (typed parameters, no input.*() inside the body, no reads of input-derived globals).

Objects

Declaring a type

```
type Pivot
    int   bar
    float price
    bool  isHigh
```

Fields are listed indented, one per line.

Each field has an explicit type keyword (int, float, bool, string, color, another UDT, an array/map/matrix type, or a reference type like line).

A field can declare a default: float price = na.

Field defaults make .new() arguments optional:

```
type ZoneBox
    int   left   = na
    int   right  = na
    float top    = na
    float bottom = na
    color borderColor = color.gray
    color fillColor   = color.new(color.gray, 90)
```

Creating an instance

```
p = Pivot.new(bar_index, high, true)
log.info("pivot " + (p.isHigh ? "H" : "L") + " @ " + str.tostring(p.price))
```

TypeName.new(...) is a constructor; arguments are positional, in field order, or named.

Field-less construction is allowed: p = Pivot.new() leaves every field at its default (na for primitives, "" for strings, white for colors).

Reading and writing fields

```
p.price           // read
p.price := 100.5  // write
```

A field write uses :=. Reading is a bare dot-access.

Nested types

Types can contain other UDTs:

```
type Trendline
    Pivot start
    Pivot end
    color col
```

Field assignment on a nested value chains:

```
t.start.price := high
```

Methods on UDTs

See Methods — a method whose this is a UDT reads like native syntax:

```
method label(Pivot this) =>
    this.isHigh ? "H" : "L"

log.info(Pivot.new(0, 100.0, true).label())   // "H"
```

Reference semantics

UDTs are reference types: assigning or passing one does not copy it. Two names bound to the same instance observe each other's writes:

```
a = Pivot.new(0, 100, true)
b = a
b.price := 200
log.info(str.tostring(a.price))  // 200
```

Create a copy with a fresh .new() call (or write a method copy() on the type).

Arrays of UDTs

A common pattern is an array of UDT instances for history:

```
var array<Pivot> pivots = array.new<Pivot>()

p = ta.pivothigh(5, 5)
if not na(p)
    pivots.push(Pivot.new(bar_index[5], p, true))
```

Walk the array with for ... in and dot-access to field read:

```
for piv in pivots
    label.new(piv.bar, piv.price, piv.isHigh ? "H" : "L")
```

Exported types

Libraries can export types for other scripts to use:

```
//@version=1
library("PivotTypes")

export type Pivot
    int   bar
    float price
    bool  isHigh

export method label(Pivot this) =>
    this.isHigh ? "H" : "L"
```

A library must export a UDT if any exported function/method accepts or returns a reference of that type.

Enums

Declaring an enum

```
enum Trend
    up    = "Bullish"
    down  = "Bearish"
    flat  = "Neutral"
```

Each member has an identifier (used in code) and an optional title (shown to users in dropdowns).

Omitting the title defaults it to the identifier:

```
enum Signal
    buy
    sell
    hold
```

Reading a member

```
t = Trend.up
if t == Trend.up
    log.info("bull")
```

Enum values are compared with ==/!=. They are not strings — you cannot concatenate them with + "foo". Call str.tostring(t) to get the display title, or str.tostring(t, format.id) to get the identifier.

As inputs

Pass an enum as an input to get a dropdown automatically:

```
mode = input.enum(Trend.up, "Trend mode")
```

The dialog renders one row per member, labelled with the title.

Exporting from a library

An enum used in an exported function's signature must itself be exported:

```
//@version=1
library("Signals")

export enum Signal
    buy  = "Buy"
    sell = "Sell"
    hold = "Hold"

export decide(float fast, float slow) =>
    fast > slow ? Signal.buy : fast < slow ? Signal.sell : Signal.hold
```

Iteration

Enum members can be collected into an array via enumname.all:

```
for s in Trend.all
    log.info(str.tostring(s))

enumname.all yields members in declaration order.
```

Arrays

Creation

CALL	RESULT
array.new<T>()	empty
array.new<T>(size)	size elements, all na
array.new<T>(size, default)	size elements, all default
array.from(a, b, c, ...)	typed-inferred from the first argument
array.copy(src)	shallow copy

Element access

```
first = seeded.get(0)            // 1.0
seeded.set(0, 42.0)              // mutate
```

.get(i) / .set(i, v) take a 0-based index.

Out-of-range reads return na (never throw); writes raise a runtime error.

Length

```
n = seeded.size()
isEmpty = n == 0
```

Mutating operations

CALL	MEANING
.push(v)	append
.pop()	remove and return the last element
.unshift(v)	prepend
.shift()	remove and return the first element
.insert(i, v)	insert at index i
.remove(i)	delete at index i
.clear()	drop all elements
.reverse()	in-place reverse
.sort(order?)	in-place sort; order is order.ascending (default) or order.descending
.sort_indices(order?)	return an array of indices in sorted order
.fill(v, from?, to?)	overwrite elements in a range

Non-mutating accessors

Numeric helpers over array<int> / array<float>:

.sum() .avg() .median() .stdev() .variance() .mode()

.max() .min() .range()

.first() .last() .slice(from, to)

.covariance(other)

.includes(v) .indexof(v) .lastindexof(v)

.binary_search(v) .binary_search_leftmost(v) .binary_search_rightmost(v)

String helper:

.join(separator)

Walking an array

```
for v in prices
    sum := sum + v

for [i, v] in prices
    log.info(str.tostring(i) + " → " + str.tostring(v))
```

See Loops.

Size caps

An array may hold up to 100,000 elements. Attempting to grow beyond that raises a runtime error (array size exceeds 100000).

Working-set collections (per-bar scratch space) are rolled back on tick snapshot restoration, same as varip-less var.

Maps

Creation

CALL	RESULT
map.new<K, V>()	empty
map.copy(src)	shallow copy

Lookup and mutation

CALL	MEANING
.put(k, v)	insert or overwrite
.get(k)	the value, or na if the key is absent
.contains(k)	boolean
.remove(k)	delete; returns the previous value or na
.clear()	drop everything
.size()	element count

Walking a map

```
for [key, value] in counts
    log.info(key + "=" + str.tostring(value))
```

Or over the keys and values separately:

```
keys   = counts.keys()      // array<K>
values = counts.values()    // array<V>
```

Order

Iteration order equals insertion order. Overwriting an existing key does not reorder it. Deleting and re-inserting does.

Size caps

A map may hold up to 50,000 entries.

Matrices

Creation

CALL	RESULT
matrix.new<T>(rows, cols)	all na
matrix.new<T>(rows, cols, default)	all default
matrix.copy(src)	shallow copy

Element access

```
v = m.get(row, col)
m.set(row, col, v + 1)
```

Zero-indexed; out-of-range reads return na, writes raise an error.

Shape

CALL	MEANING
.rows()	row count
.columns()	column count
.row(i)	copy of row i as array<T>
.col(j)	copy of column j as array<T>

Reshaping

CALL	MEANING

.add_row(i, values?)	insert row values (or all na) at index i
.add_col(j, values?)	insert column at index j
.remove_row(i) / .remove_col(j)	delete
.swap_rows(a, b) / .swap_cols(a, b)	in-place swap
.reshape(rows, cols)	in-place reshape (row-major)
.reverse()	reverse row order
.transpose()	returns a transposed copy

Numeric helpers

For matrix<float> / matrix<int>:

.sum() .avg() .max() .min() .mode() .median()

.sort(order?, byRow?, byCol?)

.mult(other) // matrix × matrix, or matrix × scalar

.inv() // inverse (square, non-singular)

.pinv() // Moore–Penrose pseudo-inverse

.rank() // rank

.trace() // sum of the main diagonal

.eigenvalues() // array<float>

When to prefer an array of arrays

Rows share the same length? → matrix.

Rows are of different lengths, or you walk one row heavily? → array<array<float>>.

Matrices are the natural shape for small linear-algebra problems. For large data processing use arrays — they are faster for pure walks.

Size caps

A matrix may have at most 1,000 rows and 1,000 columns.

Built-Ins

Built-in series

NAME	MEANING
open	current bar's open price
high	current bar's high
low	current bar's low
close	current bar's close
volume	current bar's volume
hl2	(high + low) / 2
hlc3	(high + low + close) / 3
ohlc4	(open + high + low + close) / 4
hlcc4	(high + low + close + close) / 4
time	bar's open time, ms UTC
time_close	bar's close time, ms UTC
bar_index	0-indexed bar number
last_bar_index	index of the chart's last bar
last_bar_time	time of the chart's last bar
timenow	wall-clock time, ms UTC

Historical access is uniform: close[1], volume[5], bar_index[0] all read bar-ago values.

Series namespaces

The lists below are generated from the compiler's own tables (dcs-compiler 0.3.51) — if a name is not here, the compiler rejects it with Unknown function 'ns.name'.

ta.* — technical analysis (61 functions + 6 tuple functions)

Moving averages and smoothers: sma, ema, rma, wma, vwma, swma, alma, hma, linreg, vwap Oscillators: rsi, cci, cmo, cog, mfi, mom, roc, stoch, tsi, wpr, rci Volatility / dispersion: atr, tr, stdev, dev, variance, bbw, kcw, range Trend: sar Direction: change, rising, falling, crossover, crossunder, cross Reductions: highest, lowest, highestbars, lowestbars, max, min, cum, median, mode, percentrank, percentile_nearest_rank, percentile_linear_interpolation, correlation Pivots: pivothigh, pivotlow Volume: obv, accdist, nvi,

pvi, pvt, wad, wvad, iii Query: barssince, valuewhen

Tuple-returning forms (destructure with [a, b] = …): ta.macd(src, fast, slow, signal) → [macd, signal, hist], ta.bb(src, len, mult) → [middle, upper, lower], ta.kc(src, len, mult, useTrueRange?) → [middle, upper, lower], ta.dmi(diLen, adxSmoothing) → [+DI, -DI, ADX], ta.supertrend(factor, atrLen) → [supertrend, direction], ta.vwap(src, anchor?, stdevMult?) → [vwap, upper, lower].

ta.cum, ta.obv, ta.accdist, ta.nvi, ta.pvi, ta.wad, ta.pvt are cumulative: a script using them is recomputed from the first bar on every tick (see Profiling).

The short-form names (sma(…), rsi(…), crossover(…), …) still compile for the 48 classic functions.

math.* — numeric functions (24) and constants

abs, max, min, sqrt, pow, log, log10, exp, round, round_to_mintick, floor, ceil, sign, avg, sum, sin, cos, tan, asin, acos, atan, todegrees, toradians, random. Constants: math.pi, math.e. (math.phi and math.rphi are not defined — see What is not supported.)

str.* — strings (18)

tostring, tonumber, format, format_time, length, contains, startswith, endswith, pos, match, substring, split, replace, replace_all, repeat, trim, lower, upper.

See Strings.

color.* — colors

17 constants: aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, orange, purple, red, silver, teal, white, yellow (plus any #rrggbb / #rrggbbaa literal). Functions: new(c, transp), rgb(r, g, b, transp?), from_gradient(value, bottom, top, bottomColor, topColor), r(c), g(c), b(c), t(c).

See Colors.

input.* — inputs (13)

int, float, bool, string, color, source, symbol, timeframe, session, time, price, textarea, enum, plus the generic input(defval, title, …).

See Inputs.

array.* — arrays (43 operations + constructors)

Constructors: array.new<type>(size?, initial?), array.from(…), and the typed new_float, new_int, new_bool, new_string, new_color, new_line, new_label, new_box, new_table, new_linefill. Mutation: push, unshift, pop, shift, set, insert, remove, clear, fill, reverse, sort, concat. Access: get, first, last, size, slice, copy, indexof, lastindexof, includes, join, every, some, sort_indices, binary_search, binary_search_leftmost, binary_search_rightmost. Statistics: sum, avg, min, max, abs, median, mode, range, stdev, variance, covariance, standardize, percentrank, percentile_nearest_rank, percentile_linear_interpolation.

map.* — maps (10)

new<K, V>(), put, put_all, get, contains, remove, clear, copy, size, keys, values.

matrix.* — matrices (50)

Shape: new<type>(rows, cols, initial?), rows, columns, elements_count, reshape, submatrix, transpose, reverse, copy, concat. Rows / columns: row, col, add_row, add_col, remove_row, remove_col, swap_rows, swap_columns, fill, first, last, get, set, sort. Algebra: sum, diff, mult, pow, det, inv, pinv, rank, trace, eigenvalues, eigenvectors, kron. Statistics: avg, median, mode, min, max. Predicates: is_square, is_zero, is_binary, is_identity, is_diagonal, is_antidiagonal, is_symmetric, is_antisymmetric, is_triangular, is_stochastic.

request.* — cross-symbol / cross-timeframe data

request.security, request.security_lower_tf, request.dividends, request.splits, request.earnings, request.financial, request.economic (compiles, returns na — no data source yet). request.quandl, request.seed, request.currency_rate and request.footprint are not available.

See Other timeframes and data.

```
strategy.* — order management
```

Orders: strategy.entry, strategy.exit, strategy.close, strategy.close_all, strategy.order, strategy.cancel, strategy.cancel_all. Risk rules: strategy.risk.allow_entry_in, max_position_size, max_drawdown, max_intraday_loss, max_cons_loss_days, max_intraday_filled_orders.

Account values (30): position_size, position_avg_price, position_entry_name, opentrades, closedtrades, wintrades, losstrades, eventrades, equity, netprofit, openprofit, grossprofit, grossloss, initial_capital, max_drawdown, max_runup, avg_trade, avg_winning_trade, avg_losing_trade, avg_trade_percent, avg_winning_trade_percent, avg_losing_trade_percent, netprofit_percent, openprofit_percent, grossprofit_percent, grossloss_percent, max_contracts_held_all, max_contracts_held_long, max_contracts_held_short, account_currency.

Per-trade accessors (18) on strategy.closedtrades.*(index) / strategy.opentrades.*(index): size, profit, profit_percent, commission, entry_id, entry_price, entry_time, entry_bar_index, entry_comment, exit_id, exit_price, exit_time, exit_bar_index, exit_comment, max_runup, max_runup_percent, max_drawdown, max_drawdown_percent.

See Strategies.

Drawing namespaces

line.* — new, delete, copy, get_x1/get_y1/get_x2/get_y2, set_x1/set_y1/set_x2/set_y2, set_xy1/set_xy2, set_xloc, set_color, set_width, set_style, set_extend, get_price, line.all.

box.* — new, delete, copy, get_left/get_right/get_top/get_bottom, set_left/set_right/set_top/set_bottom, set_lefttop/set_rightbottom, set_bgcolor, set_border_color, set_border_width, set_border_style, set_extend, set_text, set_text_color, set_text_size, set_text_halign/set_text_valign, set_text_wrap, set_text_font_family, set_text_formatting, box.all.

label.* — new, delete, copy, get_x/get_y, get_text, set_x/set_y/set_xy, set_xloc/set_yloc, set_text, set_textcolor, set_color, set_style, set_size, set_textalign, set_tooltip, set_text_font_family, set_text_formatting, label.all.

polyline.* — new(points, curved?, closed?, xloc?, line_color?, fill_color?, line_style?, line_width?), delete, polyline.all; points are chart.point objects (chart.point.new, from_index, from_time, now, copy).

linefill.* — new(line1, line2, color), delete, get_line1/get_line2, set_color, linefill.all.

table.* — new(position, columns, rows, ...), delete, clear, cell, merge_cells, set_position, set_bgcolor, set_frame_color/set_frame_width, set_border_color/set_border_width, and per cell cell_set_text, cell_set_bgcolor, cell_set_text_color, cell_set_text_size, cell_set_text_halign/cell_set_text_valign, cell_set_tooltip, cell_set_width/cell_set_height, cell_set_text_font_family, cell_set_text_formatting.

See Lines and boxes, Text and shapes, Tables.

log.*, alertcondition and alert

log.info(msg), log.warning(msg), log.error(msg) — the editor panel's Logs pane (filters, source jump, scroll to bar: see Debugging).

alertcondition(cond, title, message) — declarative, bound to the chart.

alert(message, freq?) — dynamic, called from the body; default alert.freq_once_per_bar.

See Alerts.

Constant namespaces

Namespaces whose members are compile-time constants:

color.* — see Colors

shape.* — shape.triangleup, shape.triangledown, shape.cross, shape.circle, shape.square, shape.diamond, shape.flag, shape.arrowup, shape.arrowdown, shape.labelup, shape.labeldown, shape.xcross

size.* — size.auto, size.tiny, size.small, size.normal, size.large, size.huge

location.* — location.absolute, location.abovebar, location.belowbar, location.top, location.bottom

plot.* — plot.style_line, plot.style_stepline, plot.style_histogram, plot.style_cross, plot.style_area, plot.style_columns, plot.style_circles, plot.style_areabr, plot.style_linebr, plot.style_stepline_diamond, plus plot.linestyle_solid/dashed/dotted

```
hline.* — hline.style_solid, hline.style_dashed, hline.style_dotted
```

line.style_*, extend.left/right/both/none, scale.right/left/none

display.* — display.none, display.pane, display.data_window, display.price_scale, display.status_line, display.all

format.* — format.inherit, format.price, format.volume, format.percent, format.mintick

font.*, position.*, text.align_left/center/right, text.wrap_none/auto

Symbol / chart namespaces

syminfo.* — strings tickerid, ticker, symbol, prefix, root, main_tickerid, description, type, isin, currency, basecurrency, country, timezone, session, volumetype, sector, industry; numbers mintick, minmove, pricescale, pointvalue, mincontract, expiration_date, shareholders, shares_outstanding_total, shares_outstanding_float, recommendations_buy, recommendations_hold, recommendations_sell, recommendations_total, employees (always na). Values and sources: Chart information.

timeframe.* — period, multiplier, main_period, isseconds, isminutes, isintraday, isdaily, isweekly, ismonthly, isdwm; functions in_seconds(tf?), from_seconds(n), change(tf).

barstate.* — isfirst, islast, ishistory, isrealtime, isnew, isconfirmed, islastconfirmedhistory.

chart.* — is_standard, is_heikinashi, is_renko, is_kagi, is_linebreak, is_pnf, is_range, bg_color, fg_color, left_visible_bar_time / right_visible_bar_time (≈ first / last loaded bar — there is no viewport model server-side).

session.* — ismarket (always true), ispremarket / ispostmarket (always false: executions run on regular-session data), isfirstbar, islastbar, isfirstbar_regular, islastbar_regular.

ticker.* — new, modify, standard, heikinashi, inherit, renko, linebreak, kagi, pointfigure (the last four build the ticker string; the non-standard bars themselves come from the chart type the user selected).

session.* — .ismarket, .ispremarket, .ispostmarket, .isfirstbar, .islastbar

barstate.* — .isconfirmed, .islast, .isfirst, .ishistory, .isrealtime, .isnew, .islastconfirmedhistory

chart.* — .left_visible_bar_time, .right_visible_bar_time, .is_heikinashi/isrenko/iskagi/islinebreak/isrange/ispnf/isstandard, .fg_color, .bg_color

dayofweek.*, currency.*

Logging

```
log.info(msg), log.warning(msg), log.error(msg).
```

See Debugging.

Full reference

Every built-in ships with a tested signature in the compiler's verification corpus.

Built-In Reference

Global functions (declarations, plotting, time)

FUNCTION	SIGNATURE
alert	alert(message, freq) → void
alertcondition	alertcondition(condition, title?, message?) — Registers an alert trigger. Server-side alert rules fire when condition is true.
barcolor	barcolor(color, offset, editable, show_last, title, display) → void
bgcolor	bgcolor(color, offset, editable, show_last, title, display, force_overlay) → void
bool	bool(x) → const bool
color	color(x) → const color
dayofmonth	dayofmonth(time, timezone) → series int
dayofweek	dayofweek(time, timezone) → series int
fill	fill(plot1, plot2, color?) — Fills the background between two plots or hlines.
fixnan	fixnan(source) → series float
float	float(x) → const float
hline	hline(price, title?, color?, linestyle?) — Horizontal line at a fixed price level.
hour	hour(time, timezone) → series int
indicator	indicator(title, overlay?, ...) — Declares the script as an indicator. overlay=true draws on the price chart; otherwise a separate pane is used.
input	input(defval, title, tooltip, inline, group, display, active) → input bool
int	int(x) → simple int
library	library(title, overlay, dynamic_requests) → void
max_bars_back	max_bars_back(var, num) → void
minute	minute(time, timezone) → series int
month	month(time, timezone) → series int
na	na(x) → simple bool
nz	nz(source, replacement?) → series — Replaces na values with replacement (default 0).
plot	plot(series, title?, color?, linewidth?, style?) — Plots a series as a line (or histogram/columns via style).
plotarrow	plotarrow(series, title, colorup, colordown, offset, minheight, maxheight, editable, show_last, display, format, precision, force_overlay) → void
plotbar	plotbar(open, high, low, close, title, color, editable, show_last, display, format, precision, force_overlay) → void
plotcandle	plotcandle(open, high, low, close, title, color, wickcolor, editable, show_last, bordercolor, display, format, precision, force_overlay) → void
plotchar	plotchar(condition, title?, char?, location?, color?) — Draws a text character on bars where condition is true.
plotshape	plotshape(condition, title?, style?, location?, color?, text?, size?) — Draws a shape on bars where condition is true.
second	second(time, timezone) → series int
strategy	strategy(title, overlay?, ...) — Declares the script as a strategy that can simulate entries/exits. pyramiding caps same-direction entries (default 1); default_qty_type (strategy.fixed / strategy.percent_of_equity / strategy.cash) with default_qty_value sizes orders that omit qty=; slippage (ticks) worsens market/stop fills; commission_type (strategy.commission.percent / .cash_per_contract / .cash_per_order) with commission_value sets the commission model applied in backtest reports and trade accessors.
string	string(x) → const string
time	time(timeframe, session, bars_back, timeframe_bars_back) → series int
time_close	time_close(timeframe, session, bars_back, timeframe_bars_back) → series int
timestamp	timestamp(dateString) → const int
weekofyear	weekofyear(time, timezone) → series int
year	year(time, timezone) → series int

ta.* — technical analysis

FUNCTION	SIGNATURE
ta.alma	ta.alma(series, length, offset, sigma, floor) → series float
ta.atr	ta.atr(length) → series float
ta.barssince	ta.barssince(condition) → series int
ta.bb	ta.bb(series, length, mult) → [series float, series float, series float]
ta.bbw	ta.bbw(series, length, mult) → series float
ta.cci	ta.cci(source, length) → series float
ta.change	ta.change(source, length) → series float
ta.cmo	ta.cmo(series, length) → series float

ta.cog	ta.cog(source, length) → series float
ta.correlation	ta.correlation(source1, source2, length) → series float
ta.cross	ta.cross(source1, source2) → series bool
ta.crossover	ta.crossover(source1, source2) → series bool
ta.crossunder	ta.crossunder(source1, source2) → series bool
ta.cum	ta.cum(source) → series float
ta.dev	ta.dev(source, length) → series float
ta.dmi	ta.dmi(diLength, adxSmoothing) → [series float, series float, series float]
ta.ema	ta.ema(source, length) → series float
ta.falling	ta.falling(source, length) → series bool
ta.highest	ta.highest(source, length) → series float
ta.highestbars	ta.highestbars(source, length) → series int
ta.hma	ta.hma(source, length) → series float
ta.kc	ta.kc(series, length, mult, useTrueRange) → [series float, series float, series float]
ta.kcw	ta.kcw(series, length, mult, useTrueRange) → series float
ta.linreg	ta.linreg(source, length, offset) → series float
ta.lowest	ta.lowest(source, length) → series float
ta.lowestbars	ta.lowestbars(source, length) → series int
ta.macd	ta.macd(source, fastlen, slowlen, siglen) → [series float, series float, series float]
ta.max	ta.max(source) → series float
ta.median	ta.median(source, length) → series float
ta.mfi	ta.mfi(series, length) → series float
ta.min	ta.min(source) → series float
ta.mode	ta.mode(source, length) → series float
ta.mom	ta.mom(source, length) → series float
ta.percentile_linear_interpolation	ta.percentile_linear_interpolation(source, length, percentage) → series float
ta.percentile_nearest_rank	ta.percentile_nearest_rank(source, length, percentage) → series float
ta.percentrank	ta.percentrank(source, length) → series float
ta.pivot_point_levels	ta.pivot_point_levels(type, anchor, developing) → array<float>
ta.pivothigh	ta.pivothigh(leftbars, rightbars) → series float
ta.pivotlow	ta.pivotlow(leftbars, rightbars) → series float
ta.range	ta.range(source, length) → series float
ta.rci	ta.rci(source, length) → series float
ta.rising	ta.rising(source, length) → series bool
ta.rma	ta.rma(source, length) → series float
ta.roc	ta.roc(source, length) → series float
ta.rsi	ta.rsi(source, length) → series float
ta.sar	ta.sar(start, inc, max) → series float
ta.sma	ta.sma(source, length) → series float
ta.stdev	ta.stdev(source, length, biased) → series float
ta.stoch	ta.stoch(source, high, low, length) → series float
ta.supertrend	ta.supertrend(factor, atrPeriod) → [series float, series float]
ta.swma	ta.swma(source) → series float
ta.tr	ta.tr(handle_na) → series float
ta.tsi	ta.tsi(source, short_length, long_length) → series float
ta.valuewhen	ta.valuewhen(condition, source, occurrence) → series float
ta.variance	ta.variance(source, length, biased) → series float
ta.vwap	ta.vwap(source, anchor) → series float
ta.vwma	ta.vwma(source, length) → series float
ta.wma	ta.wma(source, length) → series float
ta.wpr	ta.wpr(length) → series float

math.*

FUNCTION	SIGNATURE
math.abs	math.abs(number) → simple int

math.acos	math.acos(angle) → simple float
math.asin	math.asin(angle) → simple float
math.atan	math.atan(angle) → simple float
math.avg	math.avg(number0, number1, ...) → simple float
math.ceil	math.ceil(number) → simple int
math.cos	math.cos(angle) → simple float
math.exp	math.exp(number) → simple float
math.floor	math.floor(number) → simple int
math.log	math.log(number) → simple float
math.log10	math.log10(number) → simple float
math.max	math.max(number0, number1, ...) → const int
math.min	math.min(number0, number1, ...) → const int
math.pow	math.pow(base, exponent) → simple float
math.random	math.random(min, max, seed) → series float
math.round	math.round(number) → simple int
math.round_to_mintick	math.round_to_mintick(number) → simple float
math.sign	math.sign(number) → simple float
math.sin	math.sin(angle) → simple float
math.sqrt	math.sqrt(number) → simple float
math.sum	math.sum(source, length) → series float
math.tan	math.tan(angle) → simple float
math.todegrees	math.todegrees(radians) → series float
math.toradians	math.toradians(degrees) → series float

str.* — strings

FUNCTION	SIGNATURE
str.contains	str.contains(source, str) → const bool
str.endswith	str.endswith(source, str) → const bool
str.format	str.format(formatString, arg0, arg1, ...) → simple string
str.format_time	str.format_time(time, format, timezone) → series string
str.length	str.length(string) → const int
str.lower	str.lower(source) → const string
str.match	str.match(source, regex) → simple string
str.pos	str.pos(source, str) → const int
str.repeat	str.repeat(source, repeat, separator) → const string
str.replace	str.replace(source, target, replacement, occurrence) → const string
str.replace_all	str.replace_all(source, target, replacement) → simple string
str.split	str.split(string, separator) → array<string>
str.startswith	str.startswith(source, str) → const bool
str.substring	str.substring(source, begin_pos, end_pos) → const string
str.tonumber	str.tonumber(string) → series float
str.tostring	str.tostring(value) → series string
str.trim	str.trim(source) → const string
str.upper	str.upper(source) → const string

color.*

FUNCTION	SIGNATURE
color.b	color.b(color) → series float
color.from_gradient	color.from_gradient(value, bottom_value, top_value, bottom_color, top_color) → series color
color.g	color.g(color) → series float
color.new	color.new(color, transp) → const color
color.r	color.r(color) → series float
color.rgb	color.rgb(red, green, blue, transp) → series color
color.t	color.t(color) → series float

input.* — script inputs

FUNCTION	SIGNATURE
input.bool	input.bool(defval, title, tooltip, inline, group, confirm, display, active) → input bool
input.color	input.color(defval, title, tooltip, inline, group, confirm, display, active) → input color
input.enum	input.enum(defval, title, options, tooltip, inline, group, confirm, display, active) → input enum
input.float	input.float(defval, title, minval, maxval, step, tooltip, inline, group, confirm, display, active) → input float
input.int	input.int(defval, title, minval, maxval, step, tooltip, inline, group, confirm, display, active) → input int
input.price	input.price(defval, title, tooltip, inline, group, confirm, display, active) → input float
input.session	input.session(defval, title, options, tooltip, inline, group, confirm, display, active) → input string
input.source	input.source(defval, title, tooltip, inline, group, display, active, confirm) → series float
input.string	input.string(defval, title, options, tooltip, inline, group, confirm, display, active) → input string
input.symbol	input.symbol(defval, title, tooltip, inline, group, confirm, display, active) → input string
input.text_area	input.text_area(defval, title, tooltip, group, confirm, display, active) → input string
input.time	input.time(defval, title, tooltip, inline, group, confirm, display, active) → input int
input.timeframe	input.timeframe(defval, title, options, tooltip, inline, group, confirm, display, active) → input string

request.* — other symbols, timeframes and data

FUNCTION	SIGNATURE
request.dividends	request.dividends(ticker, field?) → series — Dividend per share, visible from each ex-date onward. field is dividends.gross (default) or dividends.net.
request.earnings	request.earnings(ticker, field?) → series — Quarterly earnings per share, stepped at fiscal-quarter boundaries. field is earnings.actual (default) or earnings.standardized.
request.economic	request.economic(country_code, field, gaps, ignore_invalid_symbol) → series float
request.financial	request.financial(symbol, financial_id, period) → series — Fundamental data (e.g. "TOTAL_REVENUE", "NET_INCOME", "EARNINGS_PER_SHARE") stepped onto chart bars at fiscal-period boundaries. period is "FY", "FQ", or "TTM". Server executions resolve from the securities financial cache.
request.security	request.security(symbol, timeframe, expression) → series — Evaluates expression on another symbol/timeframe. Server executions preload every referenced timeframe from real market data.
request.security_lower_tf	request.security_lower_tf(symbol, timeframe, expression, ignore_invalid_symbol, currency, ignore_invalid_timeframe, calc_bars_count) → array<type>
request.splits	request.splits(ticker, field) → series — Split/bonus ratio components, visible from each ex-date onward. field is splits.numerator or splits.denominator.

ticker.* — synthetic tickers

FUNCTION	SIGNATURE
ticker.heikinashi	ticker.heikinashi(symbol) → simple string
ticker.inherit	ticker.inherit(from_tickerid, symbol) → simple string
ticker.kagi	ticker.kagi(symbol, reversal) → simple string
ticker.linebreak	ticker.linebreak(symbol, number_of_lines) → simple string
ticker.modify	ticker.modify(tickerid, session, adjustment, backadjustment, settlement_as_close) → simple string
ticker.new	ticker.new(prefix, ticker, session, adjustment, backadjustment, settlement_as_close) → simple string
ticker.pointfigure	ticker.pointfigure(symbol, source, style, param, reversal) → simple string
ticker.renko	ticker.renko(symbol, style, param, request_wicks, source) → simple string
ticker.standard	ticker.standard(symbol) → simple string

array.*

FUNCTION	SIGNATURE
array.abs	array.abs(id) → array<float>
array.avg	array.avg(id) → series float
array.binary_search	array.binary_search(id, val) → series int
array.binary_search_leftmost	array.binary_search_leftmost(id, val) → series int
array.binary_search_rightmost	array.binary_search_rightmost(id, val) → series int
array.clear	array.clear(id) → void
array.concat	array.concat(id1, id2) → array<type>
array.copy	array.copy(id) → array<type>
array.covariance	array.covariance(id1, id2, biased) → series float
array.every	array.every(id) → series bool
array.fill	array.fill(id, value, index_from, index_to) → void
array.first	array.first(id) → series <type>

array.from	array.from(arg0, arg1, ...) → array<type>
array.get	array.get(id, index) → series <type>
array.includes	array.includes(id, value) → series bool
array.indexof	array.indexof(id, value) → series int
array.insert	array.insert(id, index, value) → void
array.join	array.join(id, separator) → series string
array.last	array.last(id) → series <type>
array.lastindexof	array.lastindexof(id, value) → series int
array.max	array.max(id, nth) → series float
array.median	array.median(id) → series float
array.min	array.min(id, nth) → series float
array.mode	array.mode(id) → series float
array.new<type>	array.new<type>(size, initial_value) → array<type>
array.new_bool	array.new_bool(size, initial_value) → array<bool>
array.new_box	array.new_box(size, initial_value) → array<box>
array.new_color	array.new_color(size, initial_value) → array<color>
array.new_float	array.new_float(size, initial_value) → array<float>
array.new_int	array.new_int(size, initial_value) → array<int>
array.new_label	array.new_label(size, initial_value) → array<label>
array.new_line	array.new_line(size, initial_value) → array<line>
array.new_linefill	array.new_linefill(size, initial_value) → array<linefill>
array.new_string	array.new_string(size, initial_value) → array<string>
array.new_table	array.new_table(size, initial_value) → array<table>
array.percentile_linear_interpolation	array.percentile_linear_interpolation(id, percentage) → series float
array.percentile_nearest_rank	array.percentile_nearest_rank(id, percentage) → series float
array.percentrank	array.percentrank(id, index) → series float
array.pop	array.pop(id) → series <type>
array.push	array.push(id, value) → void
array.range	array.range(id) → series float
array.remove	array.remove(id, index) → series <type>
array.reverse	array.reverse(id) → void
array.set	array.set(id, index, value) → void
array.shift	array.shift(id) → series <type>
array.size	array.size(id) → series int
array.slice	array.slice(id, index_from, index_to) → array<type>
array.some	array.some(id) → series bool
array.sort	array.sort(id, order) → void
array.sort_indices	array.sort_indices(id, order) → array<int>
array.standardize	array.standardize(id) → array<float>
array.stdev	array.stdev(id, biased) → series float
array.sum	array.sum(id) → series float
array.unshift	array.unshift(id, value) → void
array.variance	array.variance(id, biased) → series float

map.*

FUNCTION	SIGNATURE
map.clear	map.clear(id) → void
map.contains	map.contains(id, key) → series bool
map.copy	map.copy(id) → map<keyType, valueType>
map.get	map.get(id, key) → <value_type>
map.keys	map.keys(id) → array<type>
map.new<type,type>	—
map.put	map.put(id, key, value) → <value_type>
map.put_all	map.put_all(id, id2) → void

map.remove	map.remove(id, key) → <value_type>
map.size	map.size(id) → series int
map.values	map.values(id) → array<type>

matrix.*

FUNCTION	SIGNATURE
matrix.add_col	matrix.add_col(id, column, array_id) → void
matrix.add_row	matrix.add_row(id, row, array_id) → void
matrix.avg	matrix.avg(id) → series float
matrix.col	matrix.col(id, column) → array<type>
matrix.columns	matrix.columns(id) → series int
matrix.concat	matrix.concat(id1, id2) → matrix<type>
matrix.copy	matrix.copy(id) → matrix<type>
matrix.det	matrix.det(id) → series float
matrix.diff	matrix.diff(id1, id2) → matrix<int>
matrix.eigenvalues	matrix.eigenvalues(id) → array<float>
matrix.eigenvectors	matrix.eigenvectors(id) → matrix<float>
matrix.elements_count	matrix.elements_count(id) → series int
matrix.fill	matrix.fill(id, value, from_row, to_row, from_column, to_column) → void
matrix.get	matrix.get(id, row, column) → <matrix_type>
matrix.inv	matrix.inv(id) → matrix<float>
matrix.is_antidiagonal	matrix.is_antidiagonal(id) → series bool
matrix.is_antisymmetric	matrix.is_antisymmetric(id) → series bool
matrix.is_binary	matrix.is_binary(id) → series bool
matrix.is_diagonal	matrix.is_diagonal(id) → series bool
matrix.is_identity	matrix.is_identity(id) → series bool
matrix.is_square	matrix.is_square(id) → series bool
matrix.is_stochastic	matrix.is_stochastic(id) → series bool
matrix.is_symmetric	matrix.is_symmetric(id) → series bool
matrix.is_triangular	matrix.is_triangular(id) → series bool
matrix.is_zero	matrix.is_zero(id) → series bool
matrix.kron	matrix.kron(id1, id2) → matrix<float>
matrix.max	matrix.max(id) → series float
matrix.median	matrix.median(id) → series float
matrix.min	matrix.min(id) → series float
matrix.mode	matrix.mode(id) → series float
matrix.mult	matrix.mult(id1, id2) → matrix<int>
matrix.new<type>	matrix.new<type>(rows, columns, initial_value) → matrix<type>
matrix.pinv	matrix.pinv(id) → matrix<float>
matrix.pow	matrix.pow(id, power) → matrix<float>
matrix.rank	matrix.rank(id) → series int
matrix.remove_col	matrix.remove_col(id, column) → array<type>
matrix.remove_row	matrix.remove_row(id, row) → array<type>
matrix.reshape	matrix.reshape(id, rows, columns) → void
matrix.reverse	matrix.reverse(id) → void
matrix.row	matrix.row(id, row) → array<type>
matrix.rows	matrix.rows(id) → series int
matrix.set	matrix.set(id, row, column, value) → void
matrix.sort	matrix.sort(id, column, order) → void
matrix.submatrix	matrix.submatrix(id, from_row, to_row, from_column, to_column) → matrix<type>
matrix.sum	matrix.sum(id1, id2) → matrix<int>
matrix.swap_columns	matrix.swap_columns(id, column1, column2) → void
matrix.swap_rows	matrix.swap_rows(id, row1, row2) → void
matrix.trace	matrix.trace(id) → series float
matrix.transpose	matrix.transpose(id) → matrix<type>

line.*

FUNCTION	SIGNATURE
line.copy	line.copy(id) → series line
line.delete	line.delete(id) → void
line.get_price	line.get_price(id, x) → series float
line.get_x1	line.get_x1(id) → series int
line.get_x2	line.get_x2(id) → series int
line.get_y1	line.get_y1(id) → series float
line.get_y2	line.get_y2(id) → series float
line.new	line.new(x1, y1, x2, y2, ...) → line — Creates a line segment.
line.set_color	line.set_color(id, color) → void
line.set_extend	line.set_extend(id, extend) → void
line.set_first_point	line.set_first_point(id, point) → void
line.set_second_point	line.set_second_point(id, point) → void
line.set_style	line.set_style(id, style) → void
line.set_width	line.set_width(id, width) → void
line.set_x1	line.set_x1(id, x) → void
line.set_x2	line.set_x2(id, x) → void
line.set_xloc	line.set_xloc(id, x1, x2, xloc) → void
line.set_xy1	line.set_xy1(id, x, y) → void
line.set_xy2	line.set_xy2(id, x, y) → void
line.set_y1	line.set_y1(id, y) → void
line.set_y2	line.set_y2(id, y) → void

label.*

FUNCTION	SIGNATURE
label.copy	label.copy(id) → series label
label.delete	label.delete(id) → void
label.get_text	label.get_text(id) → series string
label.get_x	label.get_x(id) → series int
label.get_y	label.get_y(id) → series float
label.new	label.new(x, y, text?, ...) → label — Creates a text label.
label.set_color	label.set_color(id, color) → void
label.set_point	label.set_point(id, point) → void
label.set_size	label.set_size(id, size) → void
label.set_style	label.set_style(id, style) → void
label.set_text	label.set_text(id, text) → void
label.set_text_font_family	label.set_text_font_family(id, text_font_family) → void
label.set_text_formatting	label.set_text_formatting(id, text_formatting) → void
label.set_textalign	label.set_textalign(id, textalign) → void
label.set_textcolor	label.set_textcolor(id, textcolor) → void
label.set_tooltip	label.set_tooltip(id, tooltip) → void
label.set_x	label.set_x(id, x) → void
label.set_xloc	label.set_xloc(id, x, xloc) → void
label.set_xy	label.set_xy(id, x, y) → void
label.set_y	label.set_y(id, y) → void
label.set_yloc	label.set_yloc(id, yloc) → void

box.*

FUNCTION	SIGNATURE
box.copy	box.copy(id) → series box
box.delete	box.delete(id) → void
box.get_bottom	box.get_bottom(id) → series float
box.get_left	box.get_left(id) → series int
box.get_right	box.get_right(id) → series int

box.get_top	box.get_top(id) → series float
box.new	box.new(left, top, right, bottom, ...) → box — Creates a rectangle in bar-index/price coordinates. Mutate with box.set_* functions.
box.set_bgcolor	box.set_bgcolor(id, color) → void
box.set_border_color	box.set_border_color(id, color) → void
box.set_border_style	box.set_border_style(id, style) → void
box.set_border_width	box.set_border_width(id, width) → void
box.set_bottom	box.set_bottom(id, bottom) → void
box.set_bottom_right_point	box.set_bottom_right_point(id, point) → void
box.set_extend	box.set_extend(id, extend) → void
box.set_left	box.set_left(id, left) → void
box.set_lefttop	box.set_lefttop(id, left, top) → void
box.set_right	box.set_right(id, right) → void
box.set_rightbottom	box.set_rightbottom(id, right, bottom) → void
box.set_text	box.set_text(id, text) → void
box.set_text_color	box.set_text_color(id, text_color) → void
box.set_text_font_family	box.set_text_font_family(id, text_font_family) → void
box.set_text_formatting	box.set_text_formatting(id, text_formatting) → void
box.set_text_halign	box.set_text_halign(id, text_halign) → void
box.set_text_size	box.set_text_size(id, text_size) → void
box.set_text_valign	box.set_text_valign(id, text_valign) → void
box.set_text_wrap	box.set_text_wrap(id, text_wrap) → void
box.set_top	box.set_top(id, top) → void
box.set_top_left_point	box.set_top_left_point(id, point) → void
box.set_xloc	box.set_xloc(id, left, right, xloc) → void

table.*

FUNCTION	SIGNATURE
table.cell	table.cell(table, column, row, text?, ...) — Sets one cell of a table.
table.cell_set_bgcolor	table.cell_set_bgcolor(table_id, column, row, bgcolor) → void
table.cell_set_height	table.cell_set_height(table_id, column, row, height) → void
table.cell_set_text	table.cell_set_text(table_id, column, row, text) → void
table.cell_set_text_color	table.cell_set_text_color(table_id, column, row, text_color) → void
table.cell_set_text_font_family	table.cell_set_text_font_family(table_id, column, row, text_font_family) → void
table.cell_set_text_formatting	table.cell_set_text_formatting(table_id, column, row, text_formatting) → void
table.cell_set_text_halign	table.cell_set_text_halign(table_id, column, row, text_halign) → void
table.cell_set_text_size	table.cell_set_text_size(table_id, column, row, text_size) → void
table.cell_set_text_valign	table.cell_set_text_valign(table_id, column, row, text_valign) → void
table.cell_set_tooltip	table.cell_set_tooltip(table_id, column, row, tooltip) → void
table.cell_set_width	table.cell_set_width(table_id, column, row, width) → void
table.clear	table.clear(table_id, start_column, start_row, end_column, end_row) → void
table.delete	table.delete(table_id) → void
table.merge_cells	table.merge_cells(table_id, start_column, start_row, end_column, end_row) → void
table.new	table.new(position, columns, rows, ...) → table — Creates a chart-corner data table. Fill cells with table.cell().
table.set_bgcolor	table.set_bgcolor(table_id, bgcolor) → void
table.set_border_color	table.set_border_color(table_id, border_color) → void
table.set_border_width	table.set_border_width(table_id, border_width) → void
table.set_frame_color	table.set_frame_color(table_id, frame_color) → void
table.set_frame_width	table.set_frame_width(table_id, frame_width) → void
table.set_position	table.set_position(table_id, position) → void

polyline.*

FUNCTION	SIGNATURE
polyline.delete	polyline.delete(id) → void
polyline.new	polyline.new(points, curved, closed, xloc, line_color, fill_color, line_style, line_width, force_overlay) → series polyline

linefill.*

FUNCTION	SIGNATURE
linefill.delete	linefill.delete(id) → void
linefill.get_line1	linefill.get_line1(id) → series line
linefill.get_line2	linefill.get_line2(id) → series line
linefill.new	linefill.new(line1, line2, color) → series linefill
linefill.set_color	linefill.set_color(id, color) → void

`chart.point.*`

FUNCTION	SIGNATURE
chart.point.copy	chart.point.copy(id) → chart.point
chart.point.from_index	chart.point.from_index(index, price) → chart.point
chart.point.from_time	chart.point.from_time(time, price) → chart.point
chart.point.new	chart.point.new(time, index, price) → chart.point
chart.point.now	chart.point.now(price) → chart.point

`strategy.* — orders, risk, trade accessors`

FUNCTION	SIGNATURE
strategy.cancel	strategy.cancel(id) — Cancels the working entry or exit order with the given id.
strategy.cancel_all	strategy.cancel_all() → void
strategy.close	strategy.close(id, when?) — Closes the open trade with the given id at the bar close.
strategy.close_all	strategy.close_all(when?) — Closes every open trade at the bar close and cancels working exit orders.
strategy.closedtrades.commission	strategy.closedtrades.commission(trade_num) → series float
strategy.closedtrades.entry_bar_index	strategy.closedtrades.entry_bar_index(trade_num) → series int
strategy.closedtrades.entry_comment	strategy.closedtrades.entry_comment(trade_num) → series string
strategy.closedtrades.entry_id	strategy.closedtrades.entry_id(trade_num) → series string
strategy.closedtrades.entry_price	strategy.closedtrades.entry_price(trade_num) → series float
strategy.closedtrades.entry_time	strategy.closedtrades.entry_time(trade_num) → series int
strategy.closedtrades.exit_bar_index	strategy.closedtrades.exit_bar_index(trade_num) → series int
strategy.closedtrades.exit_comment	strategy.closedtrades.exit_comment(trade_num) → series string
strategy.closedtrades.exit_id	strategy.closedtrades.exit_id(trade_num) → series string
strategy.closedtrades.exit_price	strategy.closedtrades.exit_price(trade_num) → series float
strategy.closedtrades.exit_time	strategy.closedtrades.exit_time(trade_num) → series int
strategy.closedtrades.max_drawdown	strategy.closedtrades.max_drawdown(trade_num) → series float
strategy.closedtrades.max_drawdown_percent	strategy.closedtrades.max_drawdown_percent(trade_num) → series float
strategy.closedtrades.max_runup	strategy.closedtrades.max_runup(trade_num) → series float
strategy.closedtrades.max_runup_percent	strategy.closedtrades.max_runup_percent(trade_num) → series float
strategy.closedtrades.profit	strategy.closedtrades.profit(trade_num) → series float
strategy.closedtrades.profit_percent	strategy.closedtrades.profit_percent(trade_num) → series float
strategy.closedtrades.size	strategy.closedtrades.size(trade_num) → series float
strategy.convert_to_account	strategy.convert_to_account(value) → series float
strategy.convert_to_symbol	strategy.convert_to_symbol(value) → series float
strategy.default_entry_qty	strategy.default_entry_qty(fill_price) → series float
strategy.entry	strategy.entry(id, direction, qty?, when?, limit?, stop?, oca_name?, oca_type?) — Opens a position at the bar close, or — with limit=/stop= — places a working entry order filled intrabar on later bars (both = stop-limit). An opposite open position is reversed (closed first); same-direction adds are capped by the declaration's pyramiding. oca_name groups working orders; when one fills, siblings react per their oca_type (strategy.oca.cancel / .reduce / .none). Omitted qty uses the default sizing. Idempotent per id; re-issuing replaces the working order.
strategy.exit	strategy.exit(id, from_entry?, stop?, limit?, profit?, loss?, trail_price?, trail_points?, trail_offset?, qty?, qty_percent?, when?) — Registers a working exit order, evaluated intrabar against subsequent bars' OHLC. stop/limit are prices; profit/loss are tick distances from the entry price; trail_price (level) or trail_points (distance) arms a trail that follows by trail_offset. qty/qty_percent exit part of the trade. Gap opens fill at the open; when both stop and limit are touched in one bar, the stop fills.

strategy.opentrades.commission	strategy.opentrades.commission(trade_num) → series float
strategy.opentrades.entry_bar_index	strategy.opentrades.entry_bar_index(trade_num) → series int
strategy.opentrades.entry_comment	strategy.opentrades.entry_comment(trade_num) → series string
strategy.opentrades.entry_id	strategy.opentrades.entry_id(trade_num) → series string
strategy.opentrades.entry_price	strategy.opentrades.entry_price(trade_num) → series float
strategy.opentrades.entry_time	strategy.opentrades.entry_time(trade_num) → series int
strategy.opentrades.max_drawdown	strategy.opentrades.max_drawdown(trade_num) → series float
strategy.opentrades.max_drawdown_percent	strategy.opentrades.max_drawdown_percent(trade_num) → series float
strategy.opentrades.max_runup	strategy.opentrades.max_runup(trade_num) → series float
strategy.opentrades.max_runup_percent	strategy.opentrades.max_runup_percent(trade_num) → series float
strategy.opentrades.profit	strategy.opentrades.profit(trade_num) → series float
strategy.opentrades.profit_percent	strategy.opentrades.profit_percent(trade_num) → series float
strategy.opentrades.size	strategy.opentrades.size(trade_num) → series float
strategy.order	strategy.order(id, direction, qty?, when?) — Market order that nets against the opposite position FIFO; any remainder opens a new trade under id.
strategy.risk.allow_entry_in	strategy.risk.allow_entry_in(value) → void
strategy.risk.max_cons_loss_days	strategy.risk.max_cons_loss_days(count, alert_message) → void
strategy.risk.max_drawdown	strategy.risk.max_drawdown(value, type, alert_message) → void
strategy.risk.max_intraday_filled_orders	strategy.risk.max_intraday_filled_orders(count, alert_message) → void
strategy.risk.max_intraday_loss	strategy.risk.max_intraday_loss(value, type, alert_message) → void
strategy.risk.max_position_size	strategy.risk.max_position_size(contracts) → void

log.*

FUNCTION	SIGNATURE
log.error	log.error(message) → void
log.info	log.info(message) → void
log.warning	log.warning(message) → void

runtime.*

FUNCTION	SIGNATURE
runtime.error	runtime.error(message) → void

timeframe.*

FUNCTION	SIGNATURE
timeframe.change	timeframe.change(timeframe) → series bool
timeframe.from_seconds	timeframe.from_seconds(seconds) → simple string
timeframe.in_seconds	timeframe.in_seconds(timeframe) → simple int

syminfo.*

FUNCTION	SIGNATURE
syminfo.prefix	syminfo.prefix(symbol) → simple string
syminfo.ticker	syminfo.ticker(symbol) → simple string

Part 3 – Visuals

Visuals Overview

Plots

plot() is the main way to display a numeric series. It appears once at script scope and the host renders a point per bar from the value you supply.

Colors

Colors are 24-bit RGB values, optionally with an 8-bit alpha channel.

Fills

A fill is a tinted region between two plots or two hline levels.

Levels

Horizontal levels mark constant prices (overbought bands, round numbers, average entry) that span the whole chart pane.

Lines and Boxes

Lines, boxes, and polylines are drawing objects: you create them with .new(), hold the returned reference, and later modify or delete it.

Backgrounds

bgcolor() tints the background of each bar. It is the right tool for visualising sessions, filter windows, or regime labels — any colour-only, per-bar state.

Bar Coloring

barcolor() recolours the chart's candles themselves. The host repaints the body and wick in the colour you supply; na leaves the candle in its default colour.

Bar Plotting

Two declarative calls render *synthetic candles* from your own OHLC series:

Text and Shapes

ChartScript exposes three declarative shape plotters (plotshape, plotchar, plotarrow) and one drawing primitive (label.new).

Tables

A table is a floating UI element pinned to one of the chart's nine positions. It is the right tool for always-visible, grid-shaped summaries (strategy report, regime read-out, session...

Plots

Signature

```
plot(series, title, color, linewidth, style, trackprice, histbase, offset,
     join, editable, show_last, display, format, precision, force_overlay,
     linestyle)
```

PARAMETER	TYPE	DEFAULT	MEANING
series	series float / int	required	The value to plot. na on a bar hides the point.
title	const string	""	Appears in the legend and the Style tab.
color	series color	#2962FF	Line / fill colour. Can vary per bar.
linewidth	const int	1	1–4.
style	const	plot.style_line	See Styles.
linestyle	const	plot.linestyle_solid	plot.linestyle_dashed, plot.linestyle_dotted.
trackprice	const bool	false	Draw a dotted horizontal line at the last value across the whole pane.
histbase	const float	0	Baseline for style_histogram / style_columns / style_area.
offset	const int	0	Shift the plot n bars left/right.
join	const bool	true	Join segments across na bars (false → style_linebr-like gaps).
editable	const bool	true	false hides the Style tab row for this plot.

show_last	const int	–	Render only the last N bars.
display	const	display.all	Where the value appears (pane, data window, price scale, status line).
format	const	format.inherit	Label number format.
precision	const int	–	Decimal places.
force_overlay	const bool	false	Pin this plot to the price pane even in an oscillator script.

Styles

STYLE	RENDERED AS
plot.style_line	continuous line (default)
plot.style_linebr	line with gaps across na bars
plot.style_stepline	step line (horizontal between bars)
plot.style_stepline_diamond	step with diamond markers at each step
plot.style_histogram	vertical bars from histbase to value
plot.style_columns	thick histogram (thick volume columns)
plot.style_cross	× marker at each bar
plot.style_circles	◯ marker
plot.style_area	filled area between histbase and value
plot.style_areabr	style_area with gaps across na

Examples

```
//@version=1
indicator("Plot gallery", overlay=false)
```

plot(close - open, "Body (histogram)", style=plot.style_histogram, color=close > open ? color.lime : color.red)

```
plot(math.avg(open, close),        "Body midpoint",    color=color.aqua, linewidth=2)
```

plot(math.min(open, close), "Body low (step)", color=color.navy, style=plot.style_stepline, linewidth=3)

plot(math.max(open, close), "Body high (dot)", color=color.gray, style=plot.style_circles, linewidth=3)

Multi-colour MA:

```
ma     = ta.alma(hl2, 40, 0.85, 6)
var color almaColor = color.silver
almaColor := ma > ma[2] ? color.lime : ma < ma[2] ? color.purple : almaColor
plot(ma, "Two-colour ALMA", color=almaColor, linewidth=2)
```

Dashed guide:

```
plot(2.0,  "Upper guide", color=color.new(color.orange, 25), linestyle=plot.linestyle_dashed)
plot(-2.0, "Lower guide", color=color.new(color.orange, 25), linestyle=plot.linestyle_dashed)
```

show_last for a trailing window

Render only the last N bars — handy for anchored volume profiles or last-value level overlays:

```
plot(ta.ema(close, 20), "EMA 20", color=color.aqua, trackprice=true, show_last=1)
```

display

Control which chart surfaces show the plot:

```
plot(close,  display=display.pane)                          // pane only (default)
plot(volume, display=display.data_window + display.pane)    // pane + the Data Window read-out
plot(delta,  display=display.status_line + display.pane)    // pane + the top-of-chart status line
```

plot(rsi, display=display.all - display.price_scale) // everywhere except the right scale

Combine modes with +; subtract with -.

offset for lead/lag

```
plot(ta.sma(close, 20), "SMA 20 forward 10 bars", offset=10)
```

A positive offset shifts the plot into the future, a negative one into the past.

Dynamic colours

The color argument is a series value — it can change bar to bar:

```
diff = close - ta.sma(close, 20)
plot(diff, "Deviation", color=diff >= 0 ? color.lime : color.red)
```

For many-color palettes see Colors.

Colors

Literals

color opaqueRed = #FF5252

color halfRed = #80FF5252 // 50% opaque red (alpha first)

Named constants

The following names are available under color.*. The hex values are stable across runs:

NAME	HEX
red	#FF5252
green	#4CAF50
blue	#2196F3
orange	#FF9800
yellow	#FFEB3B
teal	#00897B
aqua	#00BCD4
lime	#00E676
purple	#9C27B0
maroon	#880E4F
navy	#311B92
gray	#787B86
silver	#B2B5BE
white	#FFFFFF
black	#363A45

Bare short names (aqua, silver, red, ...) also resolve.

```
color.new(color, transparency)
```

Produce a translucent version of a colour:

```
tint = color.new(color.aqua, 85)   // 85% transparent
```

Transparency is a 0–100 integer where 0 is opaque and 100 is fully invisible.

```
color.rgb(r, g, b, transparency = 0)
```

Build a colour from numeric channel values:

```
custom = color.rgb(255, 180, 40, 0)
```

Each channel is 0–255; transparency 0–100.

Gradients

```
score = ta.rsi(close, 14)
tint  = color.from_gradient(score, 30, 70, color.red, color.lime)
plot(close, color=tint)
```

color.from_gradient(x, low, high, cLow, cHigh) returns cLow when x <= low, cHigh when x >= high, and a linearly interpolated colour between.

When your anchors must bracket, prefer math.max(low, math.min(x, high)) to clamp before calling.

Theme-aware colours

Two theme-driven sentinels are available:

chart.fg_color — the chart's foreground text colour (adapts to light/dark theme at render time).

```
chart.bg_color — the chart's background colour.
hline(0, "Zero", color=color.new(chart.fg_color, 75), linestyle=hline.style_solid)
```

Both can be used as a const-colour argument; the host substitutes the real theme colour just before rendering.

Channel accessors

CALL	RETURNS
color.r(c)	red channel 0–255
color.g(c)	green
color.b(c)	blue
color.t(c)	transparency 0–100

Palettes

A plot can declare a palette whose entries are coloured independently from the Style tab:

```
// Four-state regime
regime = switch
    close > ta.ema(close, 50)  and close > ta.ema(close, 200) => 3
    close > ta.ema(close, 200)                                => 2
    close < ta.ema(close, 50)  and close < ta.ema(close, 200) => 0
    => 1

palette = switch regime
    0 => color.red
    1 => color.orange
    2 => color.aqua
    3 => color.lime
    => color.gray

plot(close, "Regime close", color=palette, linewidth=2)
```

When all branches of a ternary / switch produce color.<name> values, the dialog shows a swatch per branch using the variable's name as the swatch title.

Fills

```
fill(plot1, plot2, color, title?)
upper = ta.sma(close, 20) + ta.stdev(close, 20) * 2
lower = ta.sma(close, 20) - ta.stdev(close, 20) * 2
pU = plot(upper, "Upper",  color=color.aqua)
pL = plot(lower, "Lower",  color=color.aqua)
fill(pU, pL, color=color.new(color.aqua, 85), title="Band")
```

plot1 / plot2 are the handles returned by plot() or hline().

color can be a series — the fill tint recalculates per bar.

Title appears in the Style tab so the user can switch the fill on/off.

Named arguments

```
fill(plot1 = pU, plot2 = pL, color = color.new(color.aqua, 85))
fill(hline1 = h1, hline2 = h2, color = color.new(color.orange, 90))
```

Dynamic fills

Both the plot values and the colour can vary per bar:

```
pUp = plot(close, color=close > open ? color.lime : na)
pDn = plot(open,  color=close > open ? color.red  : na)
fill(pUp, pDn, color=close > open ? color.new(color.lime, 85) : color.new(color.red, 85))
```

Gradient fills

Supply top_color/bottom_color/top_value/bottom_value for a vertical gradient between the two plots:

```
fill(pU, pL,
     top_value    = upper, top_color    = color.new(color.lime, 70),
     bottom_value = lower, bottom_color = color.new(color.red,  70))
```

top_value / bottom_value are the series anchors (usually the plot values themselves).

The gradient is drawn linearly between those anchors each bar.

fillgaps for na bars

Pass fillgaps = true to bridge over na bars instead of stopping the fill:

```
fill(pU, pL, color=color.new(color.aqua, 85), fillgaps=true)
```

linefill.new() for arbitrary line pairs

When the two lines are line.new() drawings rather than plots, use linefill:

```
l1 = line.new(bar_index - 10, high, bar_index, high, color=color.lime)
l2 = line.new(bar_index - 10, low,  bar_index, low,  color=color.red)
linefill.new(l1, l2, color=color.new(color.yellow, 90))
```

The returned reference can be deleted or recoloured later with linefill.set_color() / linefill.delete().

Levels

```
hline(price, title?, color?, linestyle?, linewidth?, editable?, display?)
hline(70, "Overbought", color=color.red,  linestyle=hline.style_dashed)
hline(30, "Oversold",   color=color.lime, linestyle=hline.style_dashed)
hline(50, "Midline",    color=color.gray, linestyle=hline.style_dotted)
```

PARAMETER	TYPE	DEFAULT	MEANING
price	const float	required	The y-coordinate. hline(50) draws a horizontal line at 50.
title	const string	""	Dialog label.
color	const color	#2962FF	Line colour.
linestyle	const	hline.style_solid	.style_dashed, .style_dotted.
linewidth	const int	1	1–4.
editable	const bool	true	false hides the Style tab row.
display	const	display.all	Combine display.* modes same as plot.

Returning the handle

hline() returns a reference you can pass to fill():

```
upper = hline(70, "Upper", color=color.red)
lower = hline(30, "Lower", color=color.lime)
fill(upper, lower, color=color.new(color.gray, 92))
```

trackprice vs hline

hline(50) — a fixed y-coordinate for the whole chart.

plot(ta.sma(close, 20), trackprice=true) — a dotted horizontal line at the plot's last value, moving every bar.

Use hline for constant levels, trackprice for last-value trackers.

When to prefer a drawing

line.new() is the right choice when the level moves (support/resistance built from pivots), when it should extend only part of the chart, or when it should be deletable:

```
var line lastSup = na
p = ta.pivotlow(5, 5)
if not na(p)
    line.delete(lastSup)
    lastSup := line.new(bar_index[5], p, bar_index + 50, p,
                        color=color.lime, extend=extend.right)
```

Lines and Boxes

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

x1/x2 default to bar_index coordinates; set xloc = xloc.bar_time to use epoch-ms times instead.

extend is one of extend.none (default), extend.left, extend.right, extend.both.

style is line.style_solid (default), line.style_dashed, line.style_dotted, line.style_arrow_left, line.style_arrow_right,

line.style_arrow_both.

```
if ta.crossover(close, ta.sma(close, 20))
    line.new(bar_index, close, bar_index + 20, close + ta.atr(14) * 2,
             color=color.lime, style=line.style_dashed, extend=extend.right)
```

Updating a line

line.set_xy1(l, bar_index, close)

line.set_xy2(l, bar_index + 50, close)

```
line.set_color(l, color.red)
line.set_width(l, 2)
line.set_style(l, line.style_dotted)
line.set_extend(l, extend.right)
```

Deleting

```
line.delete(l)
```

A deleted reference becomes na.

```
box.new()
box.new(left, top, right, bottom, border_color?, border_width?, border_style?,
        bgcolor?, text?, text_color?, text_size?, text_halign?, text_valign?,
        xloc?, extend?)

if volume > ta.sma(volume, 20) * 2
    box.new(bar_index - 1, high + 2, bar_index + 1, low - 2,
            border_color=color.yellow, bgcolor=color.new(color.yellow, 85),
            text="Volume spike", text_color=color.yellow, text_size=size.tiny,
            text_halign=text.align_center, text_valign=text.align_top)
```

Updates mirror lines: box.set_lefttop, box.set_rightbottom, box.set_bgcolor, box.set_border_color, box.set_border_width, box.set_border_style, box.set_text, box.set_text_color, box.set_extend, box.delete.

```
polyline.new()
```

A polyline is a connected sequence of segments:

```
points = array.new<chart.point>()
for i = 0 to 20
    points.push(chart.point.new(bar_index[i], high[i]))
polyline.new(points, color=color.lime, line_width=2, line_style=line.style_solid, curved=false)
```

curved = true fits a Bezier curve through the points instead of straight segments.

closed = true connects the last point back to the first.

Caps on polylines are configured via max_polylines_count on the declaration (default and maximum 100).

```
linefill.new()
```

Fills a region between two line.new() drawings:

```
up = line.new(bar_index - 20, high[20], bar_index, high, color=color.lime)
dn = line.new(bar_index - 20, low[20],  bar_index, low,  color=color.red)
linefill.new(up, dn, color=color.new(color.yellow, 92))
```

Z-order among drawings

Drawings sit on top of plots in order of declaration: the last new() call renders on top. Reordering is done by deleting and re-creating.

Reference lifetimes

A drawing persists until you delete it or the chart is destroyed.

The script's declaration caps (max_lines_count, max_boxes_count, …) govern how many can coexist; exceeding the cap deletes the oldest.

Store references in var or array<line>/array<box> to survive across bars.

Backgrounds

```
bgcolor(color, offset?, editable?, title?, display?, force_overlay?)
bgcolor(close > open ? color.new(color.lime, 92) : color.new(color.red, 92))
```

color is a series value; return na on bars where no tint should apply.

offset shifts the tint n bars left/right (defaults to 0).

force_overlay = true paints the tint on the price pane even when the script is an oscillator.

Common patterns

Highlight a session

```
isPremarket = session.ispremarket
bgcolor(isPremarket ? color.new(color.orange, 92) : na, title="Premarket")
```

Highlight a time-range filter

```
useTimeFilter = input.bool(true, "Use time filter")
startTime     = input.time(timestamp("2020-01-01 00:00"), "Start")
endTime       = input.time(timestamp("2099-01-01 00:00"), "End")

inRange = time >= startTime and time <= endTime
bgcolor(useTimeFilter and not inRange ? color.new(color.red, 85) : na, title="Outside range")
```

Regime colouring

```
regime = switch
    close > ta.ema(close, 50) and close > ta.ema(close, 200) => "bull"
    close < ta.ema(close, 50) and close < ta.ema(close, 200) => "bear"
    => "flat"

bgcolor(switch regime
    "bull" => color.new(color.lime, 92)
    "bear" => color.new(color.red,  92)
    => color.new(color.gray, 95))
```

Z-order

bgcolor() renders behind the price series and the plots.

Multiple bgcolor() calls alpha-blend in declaration order (last call on top of earlier ones).

When to prefer box.new

bgcolor paints every selected bar edge-to-edge in the pane and takes constant time.

box.new is better when you need a region with a border, text label, or that extends outside the chart's current bar range (zones from historical pivots).

Bar Coloring

```
barcolor(color, offset?, editable?, title?, display?)
isBullMarubozu = close == high and open == low
isBearMarubozu = close == low  and open == high
barcolor(isBullMarubozu ? color.lime : isBearMarubozu ? color.red : na,
         title="Marubozu")
```

Returns from this call stack per bar — each bar paints independently.

offset shifts the painted bar n positions left/right.

When to use it

Highlighting bars that match a pattern (engulfings, outside bars, high-volume candles).

Showing a long/short regime by colouring every bar that qualifies.

Painting a "quality" overlay over a transparent base series.

Use bgcolor() instead when you want a tint behind the whole bar area rather than a colour on the candle itself.

Translucency

A translucent barcolor() blends with the pane background, not with the original candle colour — the override repaints the body from scratch over the pane background. If you need the original candle's hue to show through, use bgcolor() instead.

Multi-condition example

```
avgBody = ta.sma(math.abs(close - open), 20)

upBig   = close > open and (close - open) > 2 * avgBody
dnBig   = close < open and (open - close) > 2 * avgBody
inside  = high <= high[1] and low >= low[1]

barcolor(upBig   ? color.lime      :
         dnBig   ? color.red       :
         inside  ? color.gray      : na,
         title="Pattern bar")
```

Bar Plotting

```
plotcandle()
plotcandle(open, high, low, close, title?, color?, wickcolor?, bordercolor?,
           editable?, show_last?, display?, force_overlay?)
```

Heikin-Ashi example:

```
haClose = (open + high + low + close) / 4
var float haOpen = na
haOpen := na(haOpen[1]) ? (open + close) / 2 : (haOpen[1] + haClose[1]) / 2
haHigh  = math.max(high, math.max(haOpen, haClose))
haLow   = math.min(low,  math.min(haOpen, haClose))

bull = haClose >= haOpen
plotcandle(haOpen, haHigh, haLow, haClose, "Heikin-Ashi",
           color      = bull ? color.lime : color.red,
           wickcolor  = bull ? color.lime : color.red,
           bordercolor= bull ? color.lime : color.red)
```

color paints the candle body (and the fill, if hollow).

wickcolor and bordercolor paint the wick and the body outline. If omitted they default to color.

```
plotbar()
plotbar(open, high, low, close, title?, color?, editable?, show_last?, display?)
```

OHLC bars share the single color argument for the vertical line and the open/close ticks:

```
plotbar(open, high, low, close, "Bars",
        color = close > open ? color.lime : color.red)
```

Volume example — multi-colour columns

Pair plotcandle with color for shaded body plus wick:

```
plotcandle(math.min(open, close), high, low, math.max(open, close),
           "Wick-less body",
           color = close > open ? color.lime : color.red,
           wickcolor = color.gray, bordercolor = color.gray)
```

Z-order

Synthetic candles always render on the price pane. In an overlay = false script they draw in their own pane along with the oscillator; force_overlay = true moves them to the price pane.

Text and Shapes

```
plotshape()
```

Renders a shape on each bar whose value is non-na.

```
plotshape(cond, title, style, location, color, text?, textcolor?, size?,
          offset?, show_last?, display?, editable?, force_overlay?)
```

PARAMETER	TYPE	DEFAULT	MEANING
cond	series bool/int	required	Truthy → emit a shape at location.

style	const	shape.triangleup	See Shape styles.
location	const	location.abovebar	.abovebar, .belowbar, .top, .bottom, .absolute.
color	series color	#2962FF	
text	const string	""	Caption rendered next to the shape.
textcolor	series color	plot colour	
size	const	size.auto	.tiny, .small, .normal, .large, .huge.

```
cross = ta.crossover(close, ta.sma(close, 20))
plotshape(cross, "Cross up", shape.triangleup, location.belowbar,
          color=color.lime, size=size.tiny)
```

Shape styles

shape.triangleup, shape.triangledown, shape.cross, shape.xcross, shape.circle, shape.square, shape.diamond, shape.flag, shape.arrowup, shape.arrowdown, shape.labelup, shape.labeldown.

location.absolute

Place the shape at the exact price in location.absolute's value:

```
level = ta.pivothigh(5, 5)
plotshape(not na(level) ? level : na, style=shape.circle, location=location.absolute,
          color=color.red, size=size.tiny)

plotchar()
```

Renders a single character glyph. Faster to render than a shape; the glyph can be anything in the Unicode BMP:

```
plotchar(ta.crossover(rsi, 70),  "RSI OB", "▲", location.abovebar, color=color.red,  size=size.tiny)
plotchar(ta.crossunder(rsi, 30), "RSI OS", "▼", location.belowbar, color=color.lime, size=size.tiny)

plotarrow()
```

Up/down arrows sized by magnitude:

```
delta = close - open
plotarrow(delta, "Body size", colorup=color.lime, colordown=color.red,
          minheight=5, maxheight=50)
```

Positive values render an up arrow, negative values a down arrow, na nothing.

Arrow pixel height scales linearly between minheight and maxheight from the smallest-magnitude bar to the largest.

```
label.new()
```

A drawing object — supports multiline text, tooltip, custom style, and update/delete:

```
label.new(x, y, text?, xloc?, yloc?, color?, style?, textcolor?, size?,
          textalign?, tooltip?, font_family?)

pivot = ta.pivothigh(5, 5)
if not na(pivot)
    label.new(bar_index[5], pivot, str.tostring(pivot, "#.##"),
              style=label.style_label_down, color=color.new(color.red, 80),
              textcolor=color.white, size=size.small)
```

Label styles

label.style_label_up, label.style_label_down, label.style_label_left, label.style_label_right, label.style_label_center, label.style_label_lower_left, label.style_label_lower_right, label.style_label_upper_left, label.style_label_upper_right, label.style_arrowup, label.style_arrowdown, label.style_square, label.style_circle, label.style_diamond, label.style_cross, label.style_xcross, label.style_none (text only, no shape).

yloc

yloc.price (default) — place at the supplied y value.

yloc.abovebar — anchor against this bar's high (y is ignored).

yloc.belowbar — anchor against this bar's low.

Updating a label

```
label.set_text(l, "updated")
label.set_xy(l, bar_index, close)
label.set_color(l, color.aqua)
```

label.set_tooltip(l, "detailed hover text here")

```
label.delete(l)
```

Multiline and alignment

Newlines in text are preserved. textalign controls horizontal alignment inside the label box: text.align_left, text.align_center (default), text.align_right.

Tables

Creating a table

```
table.new(position, columns, rows, bgcolor?, frame_color?, frame_width?,
          border_color?, border_width?, force_overlay?)
```

position is a position.* constant: .top_left, .top_center, .top_right, .middle_left, .middle_center, .middle_right, .bottom_left, .bottom_center, .bottom_right.

columns × rows fixes the grid at creation; it cannot change later.

Both are capped at 20: a request for more rows or columns is clamped to 20, and table.cell calls outside the grid are ignored. For a longer list, lay the data out as several key/value column pairs (4 or 6 columns) instead of one tall table.

force_overlay = true renders the table on the price pane even inside an oscillator script.

```
var table t = table.new(position.top_right, 2, 4,
                        bgcolor=color.new(color.black, 20),
                        border_color=color.gray, border_width=1)
```

Table creation is usually guarded by var so the same table is reused across bars.

Writing cells

```
table.cell(table_id, column, row, text,
           text_color?, bgcolor?, text_size?, text_halign?, text_valign?,
           tooltip?, text_font_family?, text_formatting?)

table.cell(t, 0, 0, "Price",   text_color=color.gray,  text_halign=text.align_left)
table.cell(t, 1, 0, str.tostring(close, "#.##"),
                            text_color=color.white, text_halign=text.align_right)
```

Updating a single cell field without rewriting the whole row:

```
table.cell_set_text(t, 1, 0, str.tostring(close, "#.##"))
table.cell_set_bgcolor(t, 1, 0, close > open ? color.new(color.lime, 85) : color.new(color.red, 85))
table.cell_set_text_color(t, 1, 0, color.white)
```

Alignment and sizing

Horizontal: text.align_left, text.align_center, text.align_right.

Vertical: text.align_top, text.align_center, text.align_bottom.

Size: size.auto (default), size.tiny, size.small, size.normal, size.large, size.huge.

Text wrapping is controlled per-cell:

table.cell(t, 0, 0, "A long block of commentary that wraps nicely",

```
           text_wrap=text.wrap_auto)
```

Styling

bgcolor on table.new() tints the whole table.

bgcolor on table.cell() tints the cell only.

text_formatting accepts text.format_bold, text.format_italic, text.format_bold | text.format_italic.

Spans

Merge adjacent cells into one wider or taller cell:

```
table.merge_cells(t, 0, 0, 1, 0)   // merge columns 0..1 of row 0 (header)
```

Only rectangular merges are supported. Writing to a merged cell writes to the top-left of the merged region.

Hiding and deleting

```
table.clear(t, 0, 0, 1, 3)   // clear a sub-rectangle
```

table.delete(t) // remove the whole table

Setting a cell's text to "" renders a blank cell but keeps the slot allocated.

Idiomatic pattern — once per bar on the last bar

Tables are usually populated only on the last bar to avoid rewriting values that will immediately be overwritten:

```
var table t = table.new(position.top_right, 2, 3, bgcolor=color.new(color.black, 10))

if barstate.islast
    table.cell(t, 0, 0, "Close",    text_color=color.gray,  text_halign=text.align_left)
    table.cell(t, 1, 0, str.tostring(close, "#.##"),
                                text_color=color.white, text_halign=text.align_right)
    table.cell(t, 0, 1, "RSI 14",
                                text_color=color.gray,  text_halign=text.align_left)
    table.cell(t, 1, 1, str.tostring(ta.rsi(close, 14), "0.0"),
                                text_color=color.white, text_halign=text.align_right)
```

Part 4 – Writing Scripts

Writing Scripts Overview

Debugging

Four channels help you understand what a running script is doing: the log.* family and the Logs pane (filters, source jump, scroll to bar), visible plots, the Problems panel, and Profiler mode.

Limitations

Every hard limit the platform enforces — source size, history depth, compute budgets, drawing caps, collection sizes, backtest and alert limits — with the current values.

Profiling and Optimization

Profiler mode shows per-line runtime share and execution counts on the live execution; the patterns that follow keep a per-bar script inside its compute budget.

Publishing

A script becomes usable to someone else — including yourself on a different chart — only after it is published. Publishing an indicator or strategy stores a version on the server and…

Style Guide

Conventions that make a ChartScript source readable, diffable, and reviewable.

What Is Not Supported

The built-ins the platform does not provide — variables, functions and constants that fail to compile or always return na — and what to use instead.

Debugging

```
log.info / log.warning / log.error
log.info("entering, close=" + str.tostring(close))
log.info("rsi={0,number,#.##} atr={1}", ta.rsi(close, 14), ta.atr(14))   // str.format-style placeholders

if ta.crossover(close, ta.sma(close, 20))
    log.warning("Cross up on " + str.tostring(bar_index))

if close < 0
    log.error("Negative close — bad data?")
```

One entry is written each time the call executes: once per historical bar, and on every tick of the live bar.

Each entry carries the level, the bar time (the bar's open time; the system time is shown on hover), the script (id@version), the message and the source line of the log.* call.

Live bar: the entries of the open bar are replaced on every tick (the pane always shows the latest tick's lines for that bar); once the bar closes they are final. They are not rolled back the way plots are.

Caps: at most 500 lines per bar (a warning log limit reached — further entries dropped is appended once) and 10,000 per execution; the Logs pane keeps the latest 10,000. Logs survive until the script restarts (re-publish, inputs change, symbol or timeframe change).

log.* is not allowed inside a request.security expression (it runs in another context).

The Logs pane

Open it with the Logs button in the editor header. It merges the logs of every execution you currently have on charts and offers:

CONTROL	WHAT IT DOES
Info / Warning / Error toggles	hide a level (the calls still run and still count toward the caps)
Search box	plain-text filter on the message; Aa = match case, ab = whole word, .* = regular expression (invalid patterns are flagged and match nothing); matches are highlighted
From date-time	hide entries whose bar time is before the chosen moment
counter N / total	how many lines the filters let through
source (hover a line)	opens the editor at the log.* call that produced it
scroll to bar (hover a line)	moves the chart to that bar, loading older history when needed

Tips: a log.* call inside a loop multiplies per bar and eats the 500-line budget quickly — collect loop values into an array or a string and log once after the loop; gate noisy calls behind an input.bool or a time range (input.time) so the useful history reaches further back.

Plot-as-print

The oldest trick: plot a numeric value you can read off the chart.

```
raw = (close - open) / open * 100
plot(raw, "body %", display=display.data_window)
```

display=display.data_window keeps the value out of the pane but puts it in the Data Window read-out so you can hover individual bars.

Table-based readout

A tiny table in a corner is often the fastest way to see the running state of several variables:

```
var table t = table.new(position.bottom_right, 2, 4, bgcolor=color.new(color.black, 20))
if barstate.islast
    table.cell(t, 0, 0, "RSI",     text_color=color.gray,  text_halign=text.align_left)
    table.cell(t, 1, 0, str.tostring(ta.rsi(close, 14), "0.00"),
                                text_color=color.white, text_halign=text.align_right)
    table.cell(t, 0, 1, "ATR",     text_color=color.gray,  text_halign=text.align_left)
    table.cell(t, 1, 1, str.tostring(ta.atr(14), "0.00"),
                                text_color=color.white, text_halign=text.align_right)
```

Breakpoints

A log.* call before and after a suspected hot section narrows a hang:

```
log.info("before heavy loop")
for i = 0 to 1_000_000 by 100
    …
log.info("after")
```

The last message that fires localises the problem.

Problems panel

The editor's Problems list shows compile diagnostics as you type. Click an entry to jump to the offending line. See Errors overview.

Compile vs runtime

A compile error prevents publish — you'll see it in the editor before the script runs.

A runtime error (runtime.error, a budget overrun, a request without data) stops the execution and shows on the indicator's legend and in the Problems strip; warnings injected by the platform (no data for a symbol, log limit reached) land in Logs and the script keeps running.

Where does the time go?

When the question is how slow rather than what value, switch on Profiler mode (Script menu): every significant line gets its share of runtime and its execution count right beside the code. See Profiling and optimization.

Limitations

Source and compilation

LIMIT	VALUE
Source size	600 KiB
Imported libraries per script (transitive)	24
Diagnostics per compile	no cap — every error and warning is reported
//@version= annotation	any integer accepted; the compiler's behaviour does not change by version

Execution

LIMIT	VALUE
Chart history available to a script	5,000 bars (close[5000] and beyond is na; longer length arguments raise history lookback exceeded)
Full compute (whole history, on start / bar close)	10 s hard timeout

Live tick compute	2 s hard timeout; ticks slower than 150 ms are paced to every 5 s / 30 s / 60 s instead of killed
Loop iterations	100,000 per loop (for / while); the loop exits, the script continues
request.security targets	512 unique (symbol, timeframe) pairs per script
Executions per user	5 custom + community indicators on a chart at a time (built-in indicators are unlimited)

Visuals (per script, oldest object dropped silently when exceeded)

KIND	DEFAULT	MAX_*_COUNT RANGE
Lines	500	1–500
Labels	500	1–500
Boxes	500	1–500
Polylines	100	1–100
Linefills	500	—
Tables	8	— (20 × 20 cells each)

Output size: one live update (delta) is capped at 256 KB and a snapshot at 4 MB. A script whose drawings exceed that is stopped with a LIMIT error — reduce drawing counts or max_*_count.

Collections

COLLECTION	MAX SIZE
array	100,000 elements
map	
matrix	1,000 rows × 1,000 columns

Strategies and backtests

LIMIT	VALUE
Backtest bars	100,000 (deep history is decimated to 6,000 curve points for the report; metrics use every bar)
Bar-magnifier intrabars	200,000 per run
Backtest wall time	60 s, +1 s per 2,500 bars
Concurrent backtests per executor	4 (further requests wait or fail fast)

Alerts

LIMIT	VALUE
Runaway guard	a rule that fires 15 times inside 3 minutes is stopped
only_once rules	stop after their first fire
Webhook targets	public http(s) hosts only; private / loopback / link-local addresses are rejected
Rules per user, message length	no fixed cap today

Accepted but ignored

These are parsed but have no effect:

```
currency= on strategy() — reports are in the symbol's native currency
```

max_bars_back / max_bars_back() — the runtime always keeps the full loaded history

```
timeframe_gaps= on indicator(), explicit_plot_zorder= (plots already draw in declaration order)
request.economic() — compiles with a warning and returns na (no data source yet)
```

Everything else on strategy() — calc_on_every_tick, calc_on_order_fills, use_bar_magnifier, fill_orders_on_standard_ohlc, margin_long / margin_short, slippage, commission_* — is enforced by the backtester; see Strategies.

Profiling and Optimization

Profiler mode

Before optimising, measure. Profiler mode shows, beside every significant line of your script, how much of the script's total runtime that line is responsible for and how many times it executed — measured on the real execution running on your chart.

Turn it on: publish the script (Add to Chart) so it is running, then open the Script menu (the script name in the editor toolbar) → Profiler mode. Turn it off the same way, or by closing the editor.

What you see:

a bar in the gutter and 12.3% · 4,512× after the code: the line's share of the script's total compute time (history plus every live tick

so far) and its execution count;

🔥 on the three heaviest lines;

on hover: the time spent, the executions, the average per execution and — for an if / for / while / switch — the block time (the whole structure) next to the line time (block minus the statements inside it);

a status strip above the Problems list explaining why nothing shows yet (not published, not on a chart, still starting) or that the buffer no longer matches the published version.

How to read it:

Counts reflect every context a line runs in. A statement inside a for executes once per iteration per bar; a user function's body lines add up across all its call sites, as expected.

Code inside a request.security expression runs in the requested context and is not instrumented; its cost lands on the line holding the request.security call.

Conditions of an else if / switch chain are part of the enclosing structure's line time (they cannot be split per branch).

Declarations, imports, input.* lines and literal assignments are not profiled.

Percentages are of the script body's own time; profiling adds overhead, so numbers are estimates — compare lines against each other, not against a stopwatch.

Under the hood a profiled run is a second, instrumented execution of your script (the chart keeps using the normal one), available only to the script's author; the editor's line numbers must match the published version, so re-publish after edits before reading the numbers.

The patterns below are the usual fixes once the profiler has pointed at a line.

Compute once, display selectively

A series-mode function (ta.sma, ta.rsi, ...) should sit at the top level. If you only want to display it on some bars, build the value every bar and gate the plot:

```
// 😩 recomputes `ta.sma` state when the condition is true, loses it when false
plot(cond ? ta.sma(close, 20) : na)

// ✓ compute always, display selectively
ma = ta.sma(close, 20)
plot(cond ? ma : na)
```

Avoid request.security for the same thing twice

Every call pays the fetch cost. Fetch once, use many times:

```
// 😩 fetches twice
dClose = request.security(syminfo.tickerid, "D", close)
dOpen  = request.security(syminfo.tickerid, "D", open)

// ✓ fetches once, destructures
[dOpen, dClose] = request.security(syminfo.tickerid, "D", [open, close])
```

Keep drawings bounded

```
// 😩 one line per bar — hits the 500 cap, oldest lines silently disappear
line.new(bar_index, high, bar_index + 10, high)

// ✓ delete the previous reference before creating a new one
var line last = na
if ta.change(ta.highest(high, 20)) != 0
    line.delete(last)
    last := line.new(bar_index, high, bar_index + 10, high)
```

Prefer plots over labels

Labels and lines are rendered individually and interact with the picker. A single plot() renders a continuous series in one pass. For high-density per-bar annotations use plotchar/plotshape instead of label.new:

```
plotchar(ta.crossover(close, ta.sma(close, 20)), "cross", "▲", location.belowbar, color=color.lime)
```

Index arrays, don't walk them

```
// 😩 O(N) per lookup
for v in sortedPivots
    if v > level
        found := v
        break

// ✓ O(log N) with a sorted array
idx = sortedPivots.binary_search_leftmost(level)
```

Pre-aggregate inside loops

```
// 😩 recomputes `ta.sma(close, 20)` on each iteration
for i = 0 to length - 1
    diff = close[i] - ta.sma(close, 20)[i]

// ✓ compute once
sma20 = ta.sma(close, 20)
for i = 0 to length - 1
    diff = close[i] - sma20[i]
```

Minimise str.format on every bar

Formatted strings are expensive. For a value that updates once per bar, build the string only on the last bar:

```
var table t = table.new(position.top_right, 1, 1, bgcolor=color.new(color.black, 20))
if barstate.islast
    table.cell(t, 0, 0, str.format("close={0,.2f}", close))
```

Publishing

From the editor

Write the script.

Click ▷ Add to chart (indicator / strategy) or ▷ Publish library.

If this is the first save, pick a name and visibility.

The compiler runs server-side; a successful compile creates version v1.

Each subsequent save bumps the version — v1 → v2 → v3.

Versions are immutable. You cannot rewrite v1; the next save becomes v2. Consumers pin a specific version (import you/MyMa/1 as m), so an update never silently reaches them.

Visibility

Private (default) — only you can see, add, or import it.

Unlisted — anyone with the link / id can add or import it; it does not appear in listings.

Public — listed in the Indicators hub for all users.

Visibility is per script (all versions) and can be changed any time from the editor's visibility dialog.

For libraries, the publish flow surfaces the canonical import path and copies it to your clipboard:

```
import <ownerId>/<Name>/<version> as <Alias>
```

Naming rules

SCRIPT KIND	NAME RULE
Indicator / strategy	Anything up to 100 characters.
Library	[A-Za-z0-9_]+ only — library names ride import paths.

The editor sanitises a library name that breaks this rule before saving (and says so in the Logs panel); the server rejects an unsafe name outright.

Protected scripts

A protected script publishes the compiled artifact but hides the source — users can add it to a chart and configure its inputs, but they cannot read or fork the code (Protected scripts cannot be forked). A protected library can still be imported by anyone the visibility allows: the compiler resolves it server-side and the source never reaches the importer.

Marketplace listing

Public scripts appear in the Indicators hub, filtered by kind (indicator / strategy / library). Marketplace listings additionally carry the creator's catalogue data: title, description, preview image, and the plan (free, or paid with monthly / yearly prices). Adding a paid script to a chart requires an active plan — without one the add is refused with SUBSCRIPTION_REQUIRED and a link to the listing.

If the plan expires while the script is on a chart, the stream is ended within ten minutes (exactly at the plan end when the end time is known): the indicator shows SUBSCRIPTION_EXPIRED: your subscription to this indicator has ended — renew it to keep using it, stops updating, and any alert rules on it are deactivated. After renewing, re-add the script to the chart and re-enable the alerts.

A user can have at most 5 custom + community scripts on a chart at a time; built-in indicators are not counted.

Alerts and distribution

A publicly-published strategy with alert_message= on orders can be armed by any user as a durable trigger = order_fill alert — fills reach their configured webhook / push without the script owner needing to run anything. See Alerts.

Updating a published script

Saving a new version does not touch charts that already run an older one. "Add to chart" from the hub always adds the latest version; a chart that has the script open is told about the new contract and can refresh to it. Deleting a script hides it from listings and stops new adds; executions already running finish their session.

Credit and licensing

Include a top-of-script header comment with your name and licence.

Published code may be reused personally, but public re-publication needs the author's explicit permission — crediting the original author in both the code and the publication description is expected.

Checklist before publishing

Script compiles with no warnings beyond the deliberately-ignored ones.

All inputs have a title= and a tooltip= where behaviour isn't obvious.

All plots have a title= and sensible color=.

A header comment describing what the script does and how to read it.

Default inputs produce something recognisable on an un-configured chart.

For strategies: a sensible initial_capital and commission model — the Strategy Tester report is only useful if these are honest.

Style Guide

File shape

```
//@version=1
// <one-line description>
// <author / licence>

indicator("Title", overlay=true)

// ─── Constants ──────────────────────────────────────────────────
```

color BULL = color.lime

color BEAR = color.red

```
// ─── Inputs ─────────────────────────────────────────────────────
length  = input.int(14, "Length", minval=1)
source  = input.source(close, "Source")

// ─── Helpers ────────────────────────────────────────────────────
smooth(src, len) =>
    ta.ema(src, len)

// ─── Calculations ───────────────────────────────────────────────
sm = smooth(source, length)

// ─── Visuals ────────────────────────────────────────────────────
plot(sm, "Smoothed", color=color.aqua, linewidth=2)

// ─── Alerts ─────────────────────────────────────────────────────
```

alertcondition(ta.crossover(close, sm), "Cross up", "{{ticker}} crossed above its smoothed series")

The six-section layout (Constants / Inputs / Helpers / Calculations / Visuals / Alerts) scales — add Risk / Strategy sections after Visuals for strategies, Imports before Constants for library-consuming scripts.

Naming

lowerCamelCase for variables and parameters: kernelLength, isTrendingUp, smoothedRsi.

UPPER_SNAKE_CASE for script-level constants: MAX_ZONES, BULL_COLOR.

PascalCase for UDT and enum type names: type Pivot, enum Trend.

Prefix variables with their unit when they could be confused: len vs lenBars vs lenPct vs lenPrice.

Suffix input ids with their intent: showMa (bool), maLen (int), maColor (color).

Formatting

4-space indentation.

One blank line between logical sections; two blank lines are unnecessary.

Line length under ~110 characters is a target, not a rule.

Align = within a block of consecutive declarations if it helps readability — but don't fight the editor to maintain alignment.

```
length  = input.int   (14,     "Length", minval=1)
source  = input.source(close,  "Source")
mult    = input.float (2.0,    "Multiplier", step=0.1)
```

Comments

Describe why, not what.

A comment that restates the code (ma = ta.sma(close, 20) // compute the SMA) is noise.

A comment that explains the choice (// We use SMA 20 — not EMA — because the user sees it as "flat".) is signal.

Function design

A pure function (depends only on parameters + current-bar built-ins) is almost always a win over a function that reads top-level state.

Return a tuple instead of mutating globals: [mid, up, lo] = bollinger(src, len, mult).

Keep bodies to one screen's worth; split a long calculation into named helpers.

Inputs

Every input has a title=.

Related inputs share a group=; booleans + their colour swatches share an inline=.

Use tooltip= for anything whose unit or effect isn't obvious.

Use active= to grey out dependent controls instead of silently ignoring them.

Pin numeric input ranges with minval=/maxval=.

Plots

Every plot has a title=.

Pick a palette and stick to it — a script that uses 7 different random greens is harder to read than one that uses color.lime + color.green consistently.

Use display.data_window when the value is useful but noisy on the pane itself.

trackprice=true is for last-value trackers; hline is for constant levels. Don't mix.

Strategies

Decide once whether orders fill at bar close (process_orders_on_close=true) or next bar's open (default). State the choice in a header comment.

initial_capital, default_qty_type, default_qty_value, commission_type, commission_value, slippage always set explicitly — defaults are easy to miss.

Guard entries with barstate.isconfirmed unless you deliberately want tick-level signalling.

Libraries

One concept per library (Zones, Pivots, Signal, …) rather than a kitchen sink.

Export every type a function signature references — the compiler enforces it for UDTs, but it reads cleaner too.

Keep bodies pure and typed.

Example of applying the style

```
//@version=1
// Supertrend with crossover alerts.
// Signals confirm on bar close; stops are placed `atrMult` × ATR from the mid price.

indicator("Supertrend", overlay=true, max_lines_count=100)

// ─── Inputs ──────────────────────────────────────────────────────
grpCore   = "Core"
grpStyle  = "Style"

atrLen    = input.int   (10,  "ATR length",     minval=1, group=grpCore)
atrMult   = input.float (3.0, "ATR multiplier", minval=0.1, step=0.1, group=grpCore)

colUp     = input.color (color.lime, "Bullish", group=grpStyle, inline="col")
colDn     = input.color (color.red,  "Bearish", group=grpStyle, inline="col")

// ─── Calculations ────────────────────────────────────────────────
[trend, dir] = ta.supertrend(atrMult, atrLen)

// ─── Visuals ─────────────────────────────────────────────────────
plot(trend, "Supertrend", color = dir > 0 ? colUp : colDn, linewidth = 2)

// ─── Alerts ──────────────────────────────────────────────────────
alertcondition(ta.change(dir) ==  2, "Flip up",   "{{ticker}} Supertrend flipped bullish")
alertcondition(ta.change(dir) == -2, "Flip down", "{{ticker}} Supertrend flipped bearish")
```

What Is Not Supported

Functions

FUNCTION	STATUS	WHY / ALTERNATIVE
request.footprint() and the footprint.* / volume_row.* accessors	compile error Unknown function	no tick-level order-flow feed is exposed to scripts
request.seed()	compile error	no user-seeded data sets
request.currency_rate()	compile error	no FX conversion feed; prices stay in the symbol's native currency (currency= on request.security is accepted and ignored)
request.quandl()	compile error	retired upstream as well
request.economic()	compiles, returns na with a warning	no macro-economic series source yet
request.dividends(..., dividends.payout_ratio)	accepted, always na	payout ratio is not in the source data (gross / net work)
request.earnings(..., earnings.estimate)	accepted, always na	analyst estimates are not in the source data (actual / standardized work)
method called directly on a built-in series (close.myMethod())	compile error Unknown function 'close.myMethod'	methods work on variables, user-defined objects, arrays, maps and matrices — assign the series to a variable first (c = close, then c.myMethod()) or call the function form

Variables

These names fail to compile (Unknown value / Undeclared identifier).

Global

NAME	TYPE	WHY
ask	series float	Real-time best ask quote; the market-data feed delivered to scripts carries OHLCV only, no quote fields
bid	series float	Real-time best bid quote; the market-data feed delivered to scripts carries OHLCV only, no quote fields

dividends

NAME	TYPE	WHY
dividends.future_amount	series float	Upcoming dividend data is not sourced
dividends.future_ex_date	series int	Upcoming dividend data is not sourced
dividends.future_pay_date	series int	Upcoming dividend data is not sourced

earnings

NAME	TYPE	WHY
earnings.future_eps	series float	Upcoming earnings estimates are not sourced

earnings.future_period_end_time	series int	Upcoming earnings calendar data is not sourced
earnings.future_revenue	series float	Upcoming earnings estimates are not sourced
earnings.future_time	series int	Upcoming earnings calendar data is not sourced

strategy

NAME	TYPE	WHY
strategy.closedtrades.first_index	series int	Closed-trade index window (trade list truncation) is not modelled
strategy.margin_liquidation_price	series float	Margin / liquidation model is not implemented in the strategy engine
strategy.max_drawdown_percent	series float	Strategy engine tracks absolute max drawdown only; the percent variant is not computed
strategy.max_runup_percent	series float	Strategy engine tracks absolute max run-up only; the percent variant is not computed
strategy.opentrades.capital_held	series float	Margin / capital-held model is not implemented in the strategy engine

syminfo

NAME	TYPE	WHY
syminfo.current_contract	simple string	Continuous-futures front contract id; continuous futures tickers are not modelled
syminfo.recommendations_buy_strong	series int	ScanX recommendations give buy/hold/sell/total only; no strong-buy split
syminfo.recommendations_date	series int	Recommendation as-of date is not provided by ScanX
syminfo.recommendations_sell_strong	series int	ScanX recommendations give buy/hold/sell/total only; no strong-sell split
syminfo.target_price_average	series float	Analyst target-price data is not part of the ScanX scrip details we load
syminfo.target_price_date	series int	Analyst target-price data is not part of the ScanX scrip details we load
syminfo.target_price_estimates	series float	Analyst target-price data is not part of the ScanX scrip details we load
syminfo.target_price_high	series float	Analyst target-price data is not part of the ScanX scrip details we load
syminfo.target_price_low	series float	Analyst target-price data is not part of the ScanX scrip details we load
syminfo.target_price_median	series float	Analyst target-price data is not part of the ScanX scrip details we load

Constants

NAME	TYPE	WHY
math.phi	const float	Golden-ratio constant not defined in the compiler
math.rphi	const float	Golden-ratio conjugate constant not defined in the compiler
plot.style_steplinebr	const plot_style	Step line with breaks is not a plot style the chart renderer supports

Platform behaviour and limits

Platform limits — see Limitations: 5,000 bars of history (max_bars_back is a no-op), 500 lines / labels / boxes and 100 polylines, 8 tables of at most 20 × 20 cells, 100,000 iterations per loop, 2 s per live tick, 10 s per full compute, 512 security pairs, 24 imports.

Time zone: hour, dayofweek, timestamp(), str.format_time() and friends default to IST; pass a timezone to change it. See Time.

request.security alignment: the confirmed higher-timeframe value appears on the last chart bar inside the HTF bar; per-bar dynamic timeframes are rejected. See Other timeframes and data.

Strategies: there is no order-trimming limit (so strategy.closedtrades.first_index is not provided); backtests are capped at 100,000 bars instead.

Non-standard charts: bars are transformed server-side; a Renko reversal amount above 1 and source = high/low are not honoured. See Non-standard charts.

Logs and profiling exist with our own UI (filters, source jump, scroll to bar; Profiler mode) — see Debugging and Profiling and optimization.

Part 5 – Errors

Errors Overview

Undeclared Identifier

The compiler saw a name it does not recognise.

Expected End of Line

Something followed a complete statement before the newline — usually a stray token the parser cannot interpret.

Expression Statement Has No Effect

An expression at script scope that doesn't do anything — a bare identifier, a bare arithmetic expression, a function call whose return is thrown away.

Drawing-Object Caps

The script tried to create more lines / labels / boxes / polylines than its declaration allows.

Compute Budget Exceeded

A computation ran past its wall-clock budget (2 s per live tick, 10 s for a full recompute). What the budgets are, how pacing avoids the error, and the usual causes.

Undeclared Identifier

Common causes

1. Typo

```
rsi = ta.rsi(close, 14)
plot(xRsi)       // ⚠ typo
```

Rename or alias the reference.

2. Variable used before declaration

```
plot(rsi)                    // ⚠ not declared yet
rsi = ta.rsi(close, 14)
```

Move the declaration above its first use. ChartScript is single-pass — forward references to top-level identifiers are not allowed.

3. Namespaced function missing the prefix

```
m = sma(close, 20)   // short-form name — still works
m = sma(close, 20)   // but a UDT or variable named `sma` shadows it
```

If a local variable happens to share the name of a bare-alias built-in, the compiler will try to use the local. Rename the local or call the built-in through its namespace (ta.sma).

4. Reaching into a library whose import is missing

```
plot(sig.smooth(close, 5))   // ⚠ 'sig' is undeclared
// forgot the `import` line
```

Add:

```
import 1000000072/Signal/1 as sig
```

5. Enum member mistyped

```
enum Trend
    up
    down
    flat

plot(0, color = Trend.flst)      // ⚠ 'flst' is not a Trend member
```

Related

Expected end of line

Identifiers

Libraries

Expected End of Line

Common causes

1. An unexpected number in an import path

Older scripts wrote numeric owner ids as /1000000072/ and the parser treated the segment as a number literal. Modern ChartScript accepts numeric owners through the normal import grammar, so this message now almost always means a different mistake — a stray token on a line that should end:

```
plot(close) 42        // ⚠ Expected 'end of line', got '42'
```

2. Missing comma inside a call

```
plot(close "SMA 20"   color=color.aqua)
               ^ ⚠ Expected 'end of line', got '"SMA 20"'
```

Add the commas:

```
plot(close, "SMA 20", color=color.aqua)
```

3. Line-continuation operator broken by whitespace

The compiler's line-continuation rule requires the operator and a whitespace to be on the next line:

```
total = volume +
```

volume[1] // ⚠ the second line starts with +, which continues; OK

```
total = volume
      +volume[1]          // ⚠ no space after `+` — treated as a new statement
```

Add a space after the operator or move it to the end of the previous line.

4. Assigning the result of a block to a mis-indented variable

value =

```
if cond                   // ⚠ `if` must be indented
    1
```

else

```
    0
```

Indent the block:

value =

```
    if cond
        1
    else
        0
```

Related

Script structure

Undeclared identifier

Expression Has No Effect

Expression Statement Has No Effect

Common causes

1. Forgotten plot

```
ma = ta.sma(close, 20)
```

ma // ⚠ likely meant plot(ma)

Add the visualisation:

```
plot(ma, "MA 20", color=color.aqua)
```

2. Trailing identifier used as a block return

Some scripts use a bare identifier at the end of a => block to mean "return this":

```
smooth(src, len) =>
    _ret = ta.ema(src, len)
    _ret
```

That's fine inside a function body — the trailing expression is the function's return. The warning fires only when the trailing expression sits at script scope, where there is nothing to return to.

3. Debug remnant

```
math.log(rsi)     // ⚠ leftover from testing
```

Remove or wrap in a log.info:

```
log.info("rsi=" + str.tostring(rsi) + ", log=" + str.tostring(math.log(rsi)))
```

Suppressing the warning

Assign the value to a _ convention name:

```
_debug = math.log(rsi)
```

The compiler does not treat _ names specially, but it does not warn on a plain assignment.

Related

Debugging

Style guide

Drawing-Object Caps

Common cause: a drawing created every bar

```
line.new(bar_index - 20, high[20], bar_index, high, color=color.lime, extend=extend.right)
```

Over 500 bars this hits the cap and the oldest line keeps disappearing.

Fix 1 — set the cap deliberately

```
indicator("…", max_lines_count=50)
```

Keeping only the last 50 lines makes the pruning intentional and keeps the snapshot small (every live drawing is sent to every viewer).

Fix 2 — keep only the active drawing

Hold a reference and delete the previous one when a new one is created:

```
var line current = na
if ta.change(ta.highest(high, 20)) != 0
    line.delete(current)
    current := line.new(bar_index - 20, high[20], bar_index, high,
                        color=color.lime, extend=extend.right)
```

Fix 3 — a ring buffer

```
var array<line> ring = array.new<line>()

if newPivot
    if ring.size() >= 20
        line.delete(ring.shift())
    ring.push(line.new(bar_index, high, bar_index + 10, high, color=color.lime))
```

The ring buffer gives you deterministic control over which drawings survive.

Related

Lines and boxes

Declaration statements

Compute Budget Exceeded

What the budgets are

COMPUTATION	BUDGET	WHEN IT RUNS
Full compute (computeFull)	10 s	On start, on every bar close for cumulative scripts, and after a history reload — the whole loaded history (up to 5,000 bars) in one pass.
Live tick (computeTick)	2 s	Every price update on the forming bar.
Loop guard	100,000 iterations per loop	Compiled into every for / while: the loop exits when its counter passes the guard, the script continues.
Backtest	60 s (+1 s per 2,500 bars)	A Strategy Tester run; stopped and reported as failed beyond it.

A slow script is not killed on the first slow tick. Live ticks are paced: a tick that takes longer than 150 ms moves the script to a slower tier that recomputes at most every 5 s, then 30 s, then 60 s (always on the latest price; bar closes are never skipped). The legend shows a ⏱ chip with "Updating every 30s" while this is happening. Only a single computation that runs past the hard budget above produces this error.

Common causes

1. while with no shrinking guard

```
i = 0
while condition
```

i := i + 1 // but `condition` doesn't shrink → spins until the loop guard trips

Add an explicit upper bound:

```
i = 0
while condition and i < 10_000
    i := i + 1
```

2. Nested loop over bar_index

```
for i = 0 to bar_index          // ⚠ grows with history: 5,000 × 100 iterations per bar
    for j = 0 to 100
        …
```

The outer bound keeps growing. Replace with a fixed lookback or a dedicated array of recent values.

3. Recomputing history on every tick

A script whose result depends on the whole history (ta.cum, ta.obv, running totals in var arrays, request.security on many symbols) is recomputed from the first bar on every live tick. Prefer the streaming ta.* functions, keep per-bar state in var variables, and read Profiling and optimization.

4. request.security with a target that varies per bar

Each unique (symbol, timeframe) pair is fetched once per computation, up to 512 pairs. A symbol string built from per-bar data creates a new pair every bar:

```
sym = str.tostring(bar_index)       // ⚠ unique per bar
x = request.security(sym, "D", close)
```