Search

Pine Script® language reference manual

Variables

bar_index

Current bar index. Numbering is zero-based, index of the first bar is 0.

TYPE

series int

EXAMPLE

```
//@version=5
indicator("bar_index")
plot(bar_index)
plot(bar_index > 5000 ? close : 0)
```

REMARKS

Note that **bar_index** has replaced **n** variable in version 4.

Note that bar indexing starts from 0 on the first historical bar.

Please note that using this variable/function can cause indicator repainting.

SEE ALSO

last_bar_index barstate.isfirst barstate.islast barstate.isrealtime

barstate.isconfirmed

Returns true if the script is calculating the last (closing) update of the current bar. The next script calculation will be on the new bar data.

TYPE

series bool

REMARKS

Pine Script® code that uses this variable could calculate differently on history and real-time data.

It is NOT recommended to use barstate.isconfirmed in request.security expression. Its value requested from request.security is unpredictable.

Please note that using this variable/function can cause indicator repainting.

SEE ALSO

barstate.isfirst barstate.islast barstate.ishistory barstate.isrealtime barstate.isnew barstate.islastconfirmedhistory

barstate.isfirst

Returns true if current bar is first bar in barset, false otherwise.

TYPE

series bool

REMARKS

Pine Script® code that uses this variable could calculate differently on history and real-time data.

Please note that using this variable/function can cause indicator repainting.

SEE ALSO

barstate.islast barstate.ishistory barstate.isrealtime barstate.isnew barstate.isconfirmed barstate.islastconfirmedhistory

barstate.ishistory

Returns true if current bar is a historical bar, false otherwise.

TYPE

series bool

REMARKS

Pine Script® code that uses this variable could calculate differently on history and real-time data.

Please note that using this variable/function can cause indicator repainting.

SEE ALSO

barstate.isfirst barstate.islast barstate.isrealtime barstate.isnew barstate.isconfirmed barstate.islastconfirmedhistory

barstate.islast

Returns true if current bar is the last bar in barset, false otherwise. This condition is true for all real-time bars in barset.

TYPE

series bool

REMARKS

Pine Script® code that uses this variable could calculate differently on history and real-time data.

Please note that using this variable/function can cause indicator repainting.

SEE ALSO

barstate.isfirst barstate.ishistory barstate.isrealtime barstate.isnew barstate.isconfirmed barstate.islastconfirmedhistory

barstate.islastconfirmedhistory

Returns true if script is executing on the dataset's last bar when market is closed, or script is executing on the bar immediately preceding the real-time bar, if market is open. Returns false otherwise.

TYPE

series bool

REMARKS

Pine Script® code that uses this variable could calculate differently on history and real-time data.

Please note that using this variable/function can cause indicator repainting.

SEE ALSO

barstate.isfirst barstate.islast barstate.ishistory barstate.isrealtime barstate.isnew

barstate.isnew

Returns true if script is currently calculating on new bar, false otherwise. This variable is true when calculating on historical bars or on first update of a newly generated real-time bar.

TYPE

series bool

REMARKS

Pine Script® code that uses this variable could calculate differently on history and real-time data.

Please note that using this variable/function can cause indicator repainting.

SEE ALSO

barstate.isfirst barstate.islast barstate.ishistory barstate.isrealtime barstate.isconfirmed

barstate.islastconfirmedhistory

barstate.isrealtime

Returns true if current bar is a real-time bar, false otherwise.

TYPE

series bool

REMARKS

Pine Script® code that uses this variable could calculate differently on history and real-time data.

Please note that using this variable/function can cause indicator repainting.

SEE ALSO

barstate.isfirst barstate.islast barstate.ishistory barstate.isnew barstate.isconfirmed

barstate.islastconfirmedhistory

box.all

Returns an array filled with all the current boxes drawn by the script.

TYPE

array<box>

EXAMPLE

```
//@version=5
indicator("box.all")
//delete all boxes
box.new(time, open, time + 60 * 60 * 24, close, xloc=xloc.bar_time, border_style=line.styl
a_allBoxes = box.all
if array.size(a_allBoxes) > 0
    for i = 0 to array.size(a_allBoxes) - 1
        box.delete(array.get(a_allBoxes, i))
```

REMARKS

The array is read-only. Index zero of the array is the ID of the oldest object on the chart.

SEE ALSO

box.new line.all label.all table.all

chart.bg_color

Returns the color of the chart's background from the "Chart settings/Appearance/Background" field. When a gradient is selected, the middle point of the gradient is returned.

TYPE

input color

SEE ALSO

chart.fg_color

chart.fg_color

Returns a color providing optimal contrast with chart.bg_color.

TYPE

input color

SEE ALSO

chart.bg_color

chart.is_heikinashi

TYPE

simple bool

RETURNS

Returns true if the chart type is Heikin Ashi, false otherwise.

SEE ALSO

chart.is_renko chart.is_linebreak chart.is_kagi chart.is_pnf chart.is_range

chart.is_kagi

TYPE

simple bool

RETURNS

Returns true if the chart type is Kagi, false otherwise.

SEE ALSO

chart.is_renko chart.is_linebreak chart.is_heikinashi chart.is_pnf chart.is_range

chart.is_linebreak

TYPE

simple bool

RETURNS

Returns true if the chart type is Line break, false otherwise.

SEE ALSO

chart.is_renko chart.is_heikinashi chart.is_kagi chart.is_pnf chart.is_range

chart.is_pnf

TYPE

simple bool

RETURNS

Returns true if the chart type is Point & figure, false otherwise.

SEE ALSO

chart.is_renko chart.is_linebreak chart.is_kagi chart.is_heikinashi chart.is_range

chart.is_range

TYPE

simple bool

RETURNS

Returns true if the chart type is Range, false otherwise.

SEE ALSO

chart.is_renko chart.is_linebreak chart.is_kagi chart.is_pnf chart.is_heikinashi

chart.is_renko

TYPE

simple bool

RETURNS

Returns true if the chart type is Renko, false otherwise.

SEE ALSO

chart.is_heikinashi chart.is_linebreak chart.is_kagi chart.is_pnf chart.is_range

chart.is_standard

TYPE

simple bool

RETURNS

Returns true if the chart type is bars, candles, hollow candles, line, area or baseline, false otherwise.

SEE ALSO

chart.is_renko chart.is_linebreak chart.is_kagi chart.is_pnf chart.is_range chart.is_heikinashi

chart.left_visible_bar_time

The time of the leftmost bar currently visible on the chart.

TYPE

input int

REMARKS

Scripts using this variable will automatically re-execute when its value updates to reflect changes in the chart, which can be caused by users scrolling the chart, or new real-time bars.

Alerts created on a script that includes this variable will only use the value assigned to the variable at the moment of the alert's creation, regardless of whether the value changes afterward, which may lead to repainting.

SEE ALSO

chart.right_visible_bar_time

chart.right_visible_bar_time

The time of the rightmost bar currently visible on the chart.

TYPE

input int

REMARKS

Scripts using this variable will automatically re-execute when its value updates to reflect changes in the chart, which can be caused by users scrolling the chart, or new real-time bars.

Alerts created on a script that includes this variable will only use the value assigned to the variable at the moment of the alert's creation, regardless of whether the value changes afterward, which may lead to repainting.

SEE ALSO

chart.left_visible_bar_time

close

Close price of the current bar when it has closed, or last traded price of a yet incomplete, realtime bar.

TYPE

series float

REMARKS

Previous values may be accessed with square brackets operator [], e.g. close[1], close[2].

SEE ALSO

open high low volume time hl2 hlc3 hlcc4 ohlc4

dayofmonth

Date of current bar time in exchange timezone.

TYPE

series int

REMARKS

Note that this variable returns the day based on the time of the bar's open. For overnight sessions (e.g. EURUSD, where Monday session starts on Sunday, 17:00) this value can be lower by 1 than the day of the trading day.

SEE ALSO

dayofmonth time year month weekofyear dayofweek hour minute second

dayofweek

Day of week for current bar time in exchange timezone.

TYPE

series int

REMARKS

Note that this variable returns the day based on the time of the bar's open. For overnight sessions (e.g. EURUSD, where Monday session starts on Sunday, 17:00) this value can be lower by 1 than the day of the trading day.

You can use dayofweek.sunday, dayofweek.monday, dayofweek.tuesday, dayofweek.wednesday, dayofweek.thursday, dayofweek.friday and dayofweek.saturday variables for comparisons.

SEE ALSO

dayofweek time year month weekofyear dayofmonth hour minute second

dividends.future_amount

Returns the payment amount of the upcoming dividend in the currency of the current instrument, or na if this data isn't available.

TYPE

series float

REMARKS

This value is only fetched once during the script's initial calculation. The variable will return the same value until the script is recalculated, even after the expected Payment date of the next dividend.

dividends.future_ex_date

Returns the Ex-dividend date (Ex-date) of the current instrument's next dividend payment, or na if this data isn't available. Ex-dividend date signifies when investors are no longer entitled to a payout from the most recent dividend. Only those who purchased shares before this day are entitled to the dividend payment.

TYPE

series int

RETURNS

UNIX time, expressed in milliseconds.

REMARKS

This value is only fetched once during the script's initial calculation. The variable will return the same value until the script is recalculated, even after the expected Payment date of the next dividend.

dividends.future_pay_date

Returns the Payment date (Pay date) of the current instrument's next dividend payment, or na if this data isn't available. Payment date signifies the day when eligible investors will receive the dividend payment.

TYPE

series int

RETURNS

UNIX time, expressed in milliseconds.

REMARKS

This value is only fetched once during the script's initial calculation. The variable will return the same value until the script is recalculated, even after the expected Payment date of the next dividend.

earnings.future_eps

Returns the estimated Earnings per Share of the next earnings report in the currency of the instrument, or na if this data isn't available.

TYPE

series float

REMARKS

This value is only fetched once during the script's initial calculation. The variable will return the same value until the script is recalculated, even after the expected time of the next earnings report.

SEE ALSO

`request.earnings`

earnings.future_period_end_time

Checks the data for the next earnings report and returns the UNIX timestamp of the day when the financial period covered by those earnings ends, or na if this data isn't available.

TYPE

series int

RETURNS

UNIX time, expressed in milliseconds.

REMARKS

This value is only fetched once during the script's initial calculation. The variable will return the same value until the script is recalculated, even after the expected time of the next earnings report.

SEE ALSO

`request.earnings`

earnings.future_revenue

Returns the estimated Revenue of the next earnings report in the currency of the instrument, or na if this data isn't available.

TYPE

series float

REMARKS

This value is only fetched once during the script's initial calculation. The variable will return the same value until the script is recalculated, even after the expected time of the next earnings report.

SEE ALSO

`request.earnings`

earnings.future_time

Returns a UNIX timestamp indicating the expected time of the next earnings report, or na if this data isn't available.

TYPE

series int

RETURNS

UNIX time, expressed in milliseconds.

REMARKS

This value is only fetched once during the script's initial calculation. The variable will return the same value until the script is recalculated, even after the expected time of the next earnings report.

SEE ALSO

request.earnings

high

Current high price.

TYPE

series float

REMARKS

Previous values may be accessed with square brackets operator [], e.g. high[1], high[2].

SEE ALSO

open low close volume time hl2 hlc3 hlcc4 ohlc4

hl2

Is a shortcut for (high + low)/2

TYPE

series float

SEE ALSO

open high low close volume time hlc3 hlcc4 ohlc4

hlc3

Is a shortcut for (high + low + close)/3

TYPE

series float

SEE ALSO

open high low close volume time hl2 hlcc4 ohlc4

hlcc4

Is a shortcut for (high + low + close + close)/4

TYPE

series float

SEE ALSO

open high low close volume time hl2 hlc3 ohlc4

hour

Current bar hour in exchange timezone.

TYPE

series int

SEE ALSO

hour time year month weekofyear dayofmonth dayofweek minute second

label.all

Returns an array filled with all the current labels drawn by the script.

TYPE

array<label>

EXAMPLE

```
//@version=5
indicator("label.all")
//delete all labels
label.new(bar_index, close)
a_allLabels = label.all
```

```
    if array.size(a_allLabels) > 0
        for i = 0 to array.size(a_allLabels) - 1
            label.delete(array.get(a_allLabels, i))
```

REMARKS

The array is read-only. Index zero of the array is the ID of the oldest object on the chart.

SEE ALSO

label.new line.all box.all table.all

last_bar_index

Bar index of the last chart bar. Bar indices begin at zero on the first bar.

TYPE

series int

EXAMPLE

```
//@version=5
strategy("Mark Last X Bars For Backtesting", overlay = true, calc_on_every_tick = true)
lastBarsFilterInput = input.int(100, "Bars Count:")
// Here, we store the 'last_bar_index' value that is known from the beginning of the scrip
// The 'last_bar_index' will change when new real-time bars appear, so we declare 'lastbar
var lastbar = last_bar_index
// Check if the current bar_index is 'lastBarsFilterInput' removed from the last bar on th
allowedToTrade = (lastbar - bar_index <= lastBarsFilterInput) or barstate.isrealtime
bgcolor(allowedToTrade ? color.new(color.green, 80) : na)
```

RETURNS

Last historical bar index for closed markets, or the real-time bar index for open markets.

REMARKS

Please note that using this variable can cause indicator repainting.

SEE ALSO

bar_index last_bar_time barstate.ishistory barstate.isrealtime

last_bar_time

Time in UNIX format of the last chart bar. It is the number of milliseconds that have elapsed since 00:00:00 UTC, 1 January 1970.

TYPE

series int

REMARKS

Please note that using this variable/function can cause indicator repainting.

Note that this variable returns the timestamp based on the time of the bar's open.

SEE ALSO

time timenow timestamp last_bar_index

line.all

Returns an array filled with all the current lines drawn by the script.

TYPE

array<line>

EXAMPLE

```
//@version=5
indicator("line.all")
//delete all lines
line.new(bar_index - 10, close, bar_index, close)
a_allLines = line.all
if array.size(a_allLines) > 0
    for i = 0 to array.size(a_allLines) - 1
        line.delete(array.get(a_allLines, i))
```

REMARKS

The array is read-only. Index zero of the array is the ID of the oldest object on the chart.

SEE ALSO

line.new label.all box.all table.all

linefill.all

Returns an array filled with all the current linefill objects drawn by the script.

TYPE

array<linefill>

REMARKS

The array is read-only. Index zero of the array is the ID of the oldest object on the chart.

low

Current low price.

TYPE

series float

REMARKS

Previous values may be accessed with square brackets operator [], e.g. low[1], low[2].

SEE ALSO

open high close volume time hl2 hlc3 hlcc4 ohlc4

minute

Current bar minute in exchange timezone.

TYPE

series int

SEE ALSO

minute time year month weekofyear dayofmonth dayofweek hour second

month

Current bar month in exchange timezone.

TYPE

series int

REMARKS

Note that this variable returns the month based on the time of the bar's open. For overnight sessions (e.g. EURUSD, where Monday session starts on Sunday, 17:00) this value can be lower by 1 than the month of the trading day.

SEE ALSO

month time year weekofyear dayofmonth dayofweek hour minute second

na

A keyword signifying "not available", indicating that a variable has no assigned value.

TYPE

simple na

EXAMPLE

```
//@version=5
indicator("na")
// CORRECT
// Plot no value when on bars zero to nine. Plot `close` on other bars.
plot(bar_index < 10 ? na : close)
// CORRECT ALTERNATIVE
// Initialize `a` to `na`. Reassign `close` to `a` on bars 10 and later.
float a = na
if bar_index >= 10
    a := close
plot(a)

// INCORRECT
// Trying to test the preceding bar's `close` for `na`.
// Will not work correctly on bar zero, when `close[1]` is `na`.
plot(close[1] == na ? close : close[1])
// CORRECT
// Use the `na()` function to test for `na`.
plot(na(close[1]) ? close : close[1])
// CORRECT ALTERNATIVE
// `nz()` tests `close[1]` for `na`. It returns `close[1]` if it is not `na`, and `close`
plot(nz(close[1], close))
```

REMARKS

Do not use this variable with comparison operators to test values for `na`, as it might lead to unexpected behavior. Instead, use the na function. Note that `na` can be used to initialize variables when the initialization statement also specifies the variable's type.

SEE ALSO

na nz fixnan

ohlc4

Is a shortcut for (open + high + low + close)/4

TYPE

series float

SEE ALSO

open high low close volume time hl2 hlc3 hlcc4

open

Current open price.

TYPE

series float

REMARKS

Previous values may be accessed with square brackets operator [], e.g. open[1], open[2].

SEE ALSO

high low close volume time hl2 hlc3 hlcc4 ohlc4

polyline.all

Returns an array containing all current polyline instances drawn by the script.

TYPE

array<polyline>

REMARKS

The array is read-only. Index zero of the array references the ID of the oldest polyline object on the chart.

second

Current bar second in exchange timezone.

TYPE

series int

SEE ALSO

second time year month weekofyear dayofmonth dayofweek hour minute

session.isfirstbar

Returns true if the current bar is the first bar of the day's session, false otherwise. If extended session information is used, only returns `true` on the first bar of the pre-market bars.

TYPE

series bool

EXAMPLE

```
//@version=5
strategy("`session.isfirstbar` Example", overlay = true)
longCondition = year >= 2022
// Place a long order at the `close` of the trading session's first bar.
if session.isfirstbar and longCondition
    strategy.entry("Long", strategy.long)

// Close the long position at the `close` of the trading session's last bar.
if session.islastbar and barstate.isconfirmed
    strategy.close("Long", immediately = true)
```

SEE ALSO

session.isfirstbar_regular session.islastbar session.islastbar_regular

session.isfirstbar_regular

Returns true on the first regular session bar of the day, false otherwise. The result is the same whether extended session information is used or not.

TYPE

series bool

EXAMPLE

```
//@version=5
```

```
strategy("`session.isfirstbar_regular` Example", overlay = true)
longCondition = year >= 2022
// Place a long order at the `close` of the trading session's first bar.
if session.isfirstbar and longCondition
    strategy.entry("Long", strategy.long)
// Close the long position at the `close` of the trading session's last bar.
if session.islastbar_regular and barstate.isconfirmed
    strategy.close("Long", immediately = true)
```

SEE ALSO

session.isfirstbar session.islastbar

session.islastbar

Returns true if the current bar is the last bar of the day's session, false otherwise. If extended session information is used, only returns `true` on the last bar of the post-market bars.

TYPE

series bool

EXAMPLE

```
//@version=5
strategy("`session.islastbar` Example", overlay = true)
longCondition = year >= 2022
// Place a long order at the `close` of the trading session's last bar.
// The position will enter `on the `open` of next session's first bar.
if session.islastbar and longCondition
    strategy.entry("Long", strategy.long)
 // Close 'Long' position at the close of the last bar of the trading session
if session.islastbar and barstate.isconfirmed
    strategy.close("Long", immediately = true)
```

REMARKS

This variable is not guaranteed to return true once in every session because the last bar of the session might not exist if no trades occur during what should be the session's last bar.

This variable is not guaranteed to work as expected on non-standard chart types, e.g., Renko.

SEE ALSO

session.isfirstbar session.islastbar_regular

session.islastbar_regular

Returns true on the last regular session bar of the day, false otherwise. The result is the same whether extended session information is used or not.

TYPE

series bool

EXAMPLE

```
//@version=5
strategy("`session.islastbar_regular` Example", overlay = true)
longCondition = year >= 2022
// Place a long order at the `close` of the trading session's first bar.
if session.isfirstbar and longCondition
    strategy.entry("Long", strategy.long)
// Close the long position at the `close` of the trading session's last bar.
if session.islastbar_regular and barstate.isconfirmed
    strategy.close("Long", immediately = true)
```

REMARKS

This variable is not guaranteed to return true once in every session because the last bar of the session might not exist if no trades occur during what should be the session's last bar.

This variable is not guaranteed to work as expected on non-standard chart types, e.g., Renko.

SEE ALSO

session.isfirstbar session.islastbar session.isfirstbar_regular

session.ismarket

Returns true if the current bar is a part of the regular trading hours (i.e. market hours), false otherwise.

TYPE

series bool

SEE ALSO

session.ispremarket session.ispostmarket

session.ispostmarket

Returns true if the current bar is a part of the post-market, false otherwise. On non-intraday charts always returns `false`.

TYPE

series bool

SEE ALSO

session.ismarket session.ispremarket

session.ispremarket

Returns true if the current bar is a part of the pre-market, false otherwise. On non-intraday charts always returns `false`.

TYPE

series bool

SEE ALSO

session.ismarket session.ispostmarket

strategy.account_currency

Returns the currency used to calculate results, which can be set in the strategy's properties.

TYPE

simple string

SEE ALSO

strategy strategy.convert_to_account strategy.convert_to_symbol

strategy.avg_losing_trade

Returns the average amount of money lost per losing trade. Calculated as the sum of losses divided by the number of losing trades.

TYPE

series float

SEE ALSO

`strategy.avg_losing_trade_percent`

strategy.avg_losing_trade_percent

Returns the average percentage loss per losing trade. Calculated as the sum of loss percentages divided by the number of losing trades.

TYPE

series float

SEE ALSO

`strategy.avg_losing_trade`

strategy.avg_trade

Returns the average amount of money gained or lost per trade. Calculated as the sum of all profits and losses divided by the number of closed trades.

TYPE

series float

SEE ALSO

`strategy.avg_trade_percent`

strategy.avg_trade_percent

Returns the average percentage gain or loss per trade. Calculated as the sum of all profit and loss percentages divided by the number of closed trades.

TYPE

series float

SEE ALSO

`strategy.avg_trade`

strategy.avg_winning_trade

Returns the average amount of money gained per winning trade. Calculated as the sum of profits divided by the number of winning trades.

TYPE

series float

SEE ALSO

strategy.avg_winning_trade_percent

strategy.avg_winning_trade_percent

Returns the average percentage gain per winning trade. Calculated as the sum of profit percentages divided by the number of winning trades.

TYPE

series float

SEE ALSO

strategy.avg_winning_trade

strategy.closedtrades

Number of trades, which were closed for the whole trading interval.

TYPE

series int

SEE ALSO

strategy.position_size strategy.opentrades strategy.wintrades strategy.losstrades strategy.eventrades

strategy.equity

Current equity (strategy.initial_capital + strategy.netprofit + strategy.openprofit).

TYPE

series float

SEE ALSO

strategy.netprofit strategy.openprofit strategy.position_size

strategy.eventrades

Number of breakeven trades for the whole trading interval.

TYPE

series int

SEE ALSO

strategy.position_size strategy.opentrades strategy.closedtrades strategy.wintrades strategy.losstrades

strategy.grossloss

Total currency value of all completed losing trades.

TYPE

series float

SEE ALSO

strategy.netprofit strategy.grossprofit

strategy.grossloss_percent

The total value of all completed losing trades, expressed as a percentage of the initial capital.

TYPE

series float

SEE ALSO

strategy.grossloss

strategy.grossprofit

Total currency value of all completed winning trades.

TYPE

series float

SEE ALSO

strategy.netprofit strategy.grossloss

strategy.grossprofit_percent

The total currency value of all completed winning trades, expressed as a percentage of the initial capital.

TYPE

series float

SEE ALSO

strategy.grossprofit

strategy.initial_capital

The amount of initial capital set in the strategy properties.

TYPE

series float

SEE ALSO

strategy

strategy.losstrades

Number of unprofitable trades for the whole trading interval.

TYPE

series int

SEE ALSO

strategy.position_size strategy.opentrades strategy.closedtrades strategy.wintrades strategy.eventrades

strategy.margin_liquidation_price

When margin is used in a strategy, returns the price point where a simulated margin call will occur and liquidate enough of the position to meet the margin requirements.

TYPE

series float

EXAMPLE

```
//@version=5
strategy("Margin call management", overlay = true, margin_long = 25, margin_short = 25,
  default_qty_type = strategy.percent_of_equity, default_qty_value = 395)

float maFast = ta.sma(close, 14)
float maSlow = ta.sma(close, 28)

if ta.crossover(maFast, maSlow)
    strategy.entry("Long", strategy.long)

if ta.crossunder(maFast, maSlow)
    strategy.entry("Short", strategy.short)

changePercent(v1, v2) =>
    float result = (v1 - v2) * 100 / math.abs(v2)

// exit when we're 10% away from a margin call, to prevent it.
if math.abs(changePercent(close, strategy.margin_liquidation_price)) <= 10
    strategy.close("Long")
    strategy.close("Short")
```

REMARKS

The variable returns na if the strategy does not use margin, i.e., the strategy declaration statement does not specify an argument for the `margin_long` or `margin_short` parameter.

strategy.max_contracts_held_all

Maximum number of contracts/shares/lots/units in one trade for the whole trading interval.

TYPE

series float

SEE ALSO

strategy.position_size strategy.max_contracts_held_long strategy.max_contracts_held_short

strategy.max_contracts_held_long

Maximum number of contracts/shares/lots/units in one long trade for the whole trading interval.

TYPE

series float

SEE ALSO

strategy.position_size strategy.max_contracts_held_all strategy.max_contracts_held_short

strategy.max_contracts_held_short

Maximum number of contracts/shares/lots/units in one short trade for the whole trading interval.

TYPE

series float

SEE ALSO

strategy.position_size strategy.max_contracts_held_all strategy.max_contracts_held_long

strategy.max_drawdown

Maximum equity drawdown value for the whole trading interval.

TYPE

series float

SEE ALSO

strategy.netprofit strategy.equity strategy.max_runup

strategy.max_drawdown_percent

The maximum equity drawdown value for the whole trading interval, expressed as a percentage and calculated by formula: `Lowest Value During Trade / (Entry Price x Quantity) * 100`.

TYPE

series float

SEE ALSO

strategy.max_drawdown

strategy.max_runup

Maximum equity run-up value for the whole trading interval.

TYPE

series float

SEE ALSO

strategy.netprofit strategy.equity strategy.max_drawdown

strategy.max_runup_percent

The maximum equity run-up value for the whole trading interval, expressed as a percentage and calculated by formula: `Highest Value During Trade / (Entry Price x Quantity) * 100`.

TYPE

series float

SEE ALSO

strategy.max_runup

strategy.netprofit

Total currency value of all completed trades.

TYPE

series float

SEE ALSO

strategy.openprofit strategy.position_size strategy.grossprofit strategy.grossloss

strategy.netprofit_percent

The total value of all completed trades, expressed as a percentage of the initial capital.

TYPE

series float

SEE ALSO

strategy.netprofit

strategy.openprofit

Current unrealized profit or loss for all open positions.

TYPE

series float

SEE ALSO

strategy.netprofit strategy.position_size

strategy.openprofit_percent

The current unrealized profit or loss for all open positions, expressed as a percentage and calculated by formula: `openPL / realizedEquity * 100`.

TYPE

series float

SEE ALSO

strategy.openprofit

strategy.opentrades

Number of market position entries, which were not closed and remain opened. If there is no open market position, 0 is returned.

TYPE

series int

SEE ALSO

strategy.position_size

strategy.opentrades.capital_held

Returns the capital amount currently held by open trades.

TYPE

series float

EXAMPLE

```
//@version=5
strategy(
   "strategy.opentrades.capital_held example", overlay=false,  margin_long=50, margin_shor
   default_qty_type = strategy.percent_of_equity, default_qty_value = 100
 )

// Enter a short position on the first bar.
if barstate.isfirst
    strategy.entry("Short", strategy.short)

// Plot the capital held by the short position.
plot(strategy.opentrades.capital_held, "Capital held")
// Highlight the chart background if the position is completely closed by margin calls.
bgcolor(bar_index > 0 and strategy.opentrades.capital_held == 0 ? color.new(color.red, 60)
```

REMARKS

This variable returns na if the strategy does not simulate funding trades with a portion of the hypothetical account, i.e., if the strategy function does not include nonzero `margin_long` or `margin_short` arguments.

strategy.position_avg_price

Average entry price of current market position. If the market position is flat, 'NaN' is returned.

TYPE

series float

SEE ALSO

strategy.position_size

strategy.position_entry_name

Name of the order that initially opened current market position.

TYPE

series string

SEE ALSO

strategy.position_size

strategy.position_size

Direction and size of the current market position. If the value is > 0, the market position is long. If the value is < 0, the market position is short. The absolute value is the number of contracts/shares/lots/units in trade (position size).

TYPE

series float

SEE ALSO

strategy.position_avg_price

strategy.wintrades

Number of profitable trades for the whole trading interval.

TYPE

series int

SEE ALSO

strategy.position_size strategy.opentrades strategy.closedtrades strategy.losstrades

strategy.eventrades

syminfo.basecurrency

Returns a string containing the code representing the symbol's base currency (i.e., the traded currency or coin) if the instrument is a Forex or Crypto pair or a derivative based on such a pair. Otherwise, it returns an empty string. For example, this variable returns "EUR" for "EURJPY", "BTC" for "BTCUSDT", "CAD" for "CME:6C1!", and "" for "NASDAQ:AAPL".

TYPE

simple string

SEE ALSO

syminfo.currency   syminfo.ticker

syminfo.country

Returns the two-letter code of the country where the symbol is traded, in the ISO 3166-1 alpha-2 format, or na if the exchange is not directly tied to a specific country. For example, on "NASDAQ:AAPL" it will return "US", on "LSE:AAPL" it will return "GB", and on "BITSTAMP:BTCUSD it will return na.

TYPE

simple string

syminfo.currency

Returns a string containing the code representing the currency of the symbol's prices. For example, this variable returns "USD" for "NASDAQ:AAPL" and "JPY" for "EURJPY".

TYPE

simple string

SEE ALSO

syminfo.basecurrency   syminfo.ticker   currency.USD   currency.EUR

syminfo.description

Description for the current symbol.

TYPE

simple string

SEE ALSO

symminfo.ticker symminfo.prefix

syminfo.employees

The number of employees the company has.

TYPE

simple int

EXAMPLE

```
//@version=5
indicator("syminfo simple")
//@variable A table containing information about a company's employees, shareholders, and
var result_table = table.new(position = position.top_right, columns = 2, rows = 5, border_
if barstate.islastconfirmedhistory
    // Add header cells
    table.cell(table_id = result_table, column = 0, row = 0, text = "name")
    table.cell(table_id = result_table, column = 1, row = 0, text = "value")
    // Add employee info cells.
    table.cell(table_id = result_table, column = 0, row = 1, text = "employees")
    table.cell(table_id = result_table, column = 1, row = 1, text = str.tostring(syminfo.e
    // Add shareholder cells.
    table.cell(table_id = result_table, column = 0, row = 2, text = "shareholders")
    table.cell(table_id = result_table, column = 1, row = 2, text = str.tostring(syminfo.s
    // Add float shares outstanding cells.
    table.cell(table_id = result_table, column = 0, row = 3, text = "shares_outstanding_fl
    table.cell(table_id = result_table, column = 1, row = 3, text = str.tostring(syminfo.s
    // Add total shares outstanding cells.
    table.cell(table_id = result_table, column = 0, row = 4, text = "shares_outstanding_to
    table.cell(table_id = result_table, column = 1, row = 4, text = str.tostring(syminfo.s
```

SEE ALSO

syminfo.shareholders syminfo.shares_outstanding_float syminfo.shares_outstanding_total

syminfo.expiration_date

A UNIX timestamp representing the start of the last day of the current futures contract. This variable is only compatible with non-continuous futures symbols. On other symbols, it returns na.

TYPE

simple int

syminfo.industry

Returns the industry of the symbol, or na if the symbol has no industry. Example: "Internet Software/Services", "Packaged software", "Integrated Oil", "Motor Vehicles", etc. These are the same values one can see in the chart's "Symbol info" window.

TYPE

simple string

REMARKS

A sector is a broad section of the economy. An industry is a narrower classification. NASDAQ:CAT (Caterpillar, Inc.) for example, belongs to the "Producer Manufacturing" sector and the "Trucks/Construction/Farm Machinery" industry.

syminfo.minmove

Returns a whole number used to calculate the smallest increment between a symbol's price movements (syminfo.mintick). It is the numerator in the syminfo.mintick formula: `syminfo.minmove / syminfo.pricescale = syminfo.mintick` .

TYPE

simple int

SEE ALSO

ticker.new syminfo.ticker timeframe.period timeframe.multiplier syminfo.root

syminfo.mintick

Min tick value for the current symbol.

TYPE

simple float

SEE ALSO

syminfo.pointvalue

syminfo.pointvalue

Point value for the current symbol.

TYPE

simple float

SEE ALSO

syminfo.mintick

syminfo.prefix

Prefix of current symbol name (i.e. for 'CME_EOD:TICKER' prefix is 'CME_EOD').

TYPE

simple string

EXAMPLE

```
//@version=5
indicator("syminfo.prefix")

// If current chart symbol is 'BATS:MSFT' then syminfo.prefix is 'BATS'.
if barstate.islastconfirmedhistory
    label.new(bar_index, high, text=syminfo.prefix)
```

SEE ALSO

syminfo.ticker syminfo.tickerid

syminfo.pricescale

Returns a whole number used to calculate the smallest increment between a symbol's price movements (syminfo.mintick). It is the denominator in the syminfo.mintick formula: `syminfo.minmove / syminfo.pricescale = syminfo.mintick`.

TYPE

simple int

SEE ALSO

ticker.new syminfo.ticker timeframe.period timeframe.multiplier syminfo.root

syminfo.recommendations_buy

The number of analysts who gave the current symbol a "Buy" rating.

TYPE

series int

EXAMPLE

```
//@version=5
indicator("syminfo recommendations", overlay = true)
//@variable A table containing information about analyst recommendations.
var table ratings = table.new(position.top_right, 8, 2, frame_color = #000000)
if barstate.islastconfirmedhistory
    //@variable The time value one year from the date of the last analyst recommendations.
    int YTD = syminfo.target_price_date + timeframe.in_seconds("12M") * 1000
    // Add header cells.
    table.cell(ratings, 0, 0, "Start Date", bgcolor = color.gray, text_color = #000000, te
    table.cell(ratings, 1, 0, "End Date", bgcolor = color.gray, text_color = #000000, text
    table.cell(ratings, 2, 0, "Buy", bgcolor = color.teal, text_color = #000000, text_size
    table.cell(ratings, 3, 0, "Strong Buy", bgcolor = color.lime, text_color = #000000, te
    table.cell(ratings, 4, 0, "Sell", bgcolor = color.maroon, text_color = #000000, text_s
    table.cell(ratings, 5, 0, "Strong Sell", bgcolor = color.red, text_color = #000000, te
    table.cell(ratings, 6, 0, "Hold", bgcolor = color.orange, text_color = #000000, text_s
    table.cell(ratings, 7, 0, "Total", bgcolor = color.silver, text_color = #000000, text_
    // Recommendation strings
    string startDate         = str.format_time(syminfo.recommendations_date, "yyyy-MM-dd")
    string endDate           = str.format_time(YTD, "yyyy-MM-dd")
    string buyRatings        = str.tostring(syminfo.recommendations_buy)
    string strongBuyRatings  = str.tostring(syminfo.recommendations_buy_strong)
    string sellRatings       = str.tostring(syminfo.recommendations_sell)
    string strongSellRatings = str.tostring(syminfo.recommendations_sell_strong)
    string holdRatings       = str.tostring(syminfo.recommendations_hold)
    string totalRatings      = str.tostring(syminfo.recommendations_total)
    // Add value cells
    table.cell(ratings, 0, 1, startDate, bgcolor = color.gray, text_color = #000000, text_
    table.cell(ratings, 1, 1, endDate, bgcolor = color.gray, text_color = #000000, text_si
    table.cell(ratings, 2, 1, buyRatings, bgcolor = color.teal, text_color = #000000, text
    table.cell(ratings, 3, 1, strongBuyRatings, bgcolor = color.lime, text_color = #00000
    table.cell(ratings, 4, 1, sellRatings, bgcolor = color.maroon, text_color = #000000, t
```

SEE ALSO

syminfo.recommendations_buy_strong
syminfo.recommendations_date
syminfo.recommendations_hold
syminfo.recommendations_total
syminfo.recommendations_sell

syminfo.recommendations_sell_strong

syminfo.recommendations_buy_strong

The number of analysts who gave the current symbol a "Strong Buy" rating.

TYPE

series int

EXAMPLE

```
//@version=5
indicator("syminfo recommendations", overlay = true)
//@variable A table containing information about analyst recommendations.
var table ratings = table.new(position.top_right, 8, 2, frame_color = #000000)
if barstate.islastconfirmedhistory
    //@variable The time value one year from the date of the last analyst recommendations.
    int YTD = syminfo.target_price_date + timeframe.in_seconds("12M") * 1000
    // Add header cells.
    table.cell(ratings, 0, 0, "Start Date", bgcolor = color.gray, text_color = #000000, te
    table.cell(ratings, 1, 0, "End Date", bgcolor = color.gray, text_color = #000000, text
    table.cell(ratings, 2, 0, "Buy", bgcolor = color.teal, text_color = #000000, text_size
    table.cell(ratings, 3, 0, "Strong Buy", bgcolor = color.lime, text_color = #000000, te
    table.cell(ratings, 4, 0, "Sell", bgcolor = color.maroon, text_color = #000000, text_s
    table.cell(ratings, 5, 0, "Strong Sell", bgcolor = color.red, text_color = #000000, te
    table.cell(ratings, 6, 0, "Hold", bgcolor = color.orange, text_color = #000000, text_s
    table.cell(ratings, 7, 0, "Total", bgcolor = color.silver, text_color = #000000, text_
    // Recommendation strings
    string startDate         = str.format_time(syminfo.recommendations_date, "yyyy-MM-dd")
    string endDate           = str.format_time(YTD, "yyyy-MM-dd")
    string buyRatings        = str.tostring(syminfo.recommendations_buy)
    string strongBuyRatings  = str.tostring(syminfo.recommendations_buy_strong)
    string sellRatings       = str.tostring(syminfo.recommendations_sell)
    string strongSellRatings = str.tostring(syminfo.recommendations_sell_strong)
    string holdRatings       = str.tostring(syminfo.recommendations_hold)
    string totalRatings      = str.tostring(syminfo.recommendations_total)
    // Add value cells
    table.cell(ratings, 0, 1, startDate, bgcolor = color.gray, text_color = #000000, text_
    table.cell(ratings, 1, 1, endDate, bgcolor = color.gray, text_color = #000000, text_si
    table.cell(ratings, 2, 1, buyRatings, bgcolor = color.teal, text_color = #000000, text
    table.cell(ratings, 3, 1, strongBuyRatings, bgcolor = color.lime, text_color = #00000
    table.cell(ratings, 4, 1, sellRatings, bgcolor = color.maroon, text_color = #000000, t
```

SEE ALSO

syminfo.recommendations_buy syminfo.recommendations_date syminfo.recommendations_hold

syminfo.recommendations_total syminfo.recommendations_sell

syminfo.recommendations_sell_strong

syminfo.recommendations_date

The starting date of the last set of recommendations for the current symbol.

TYPE

series int

EXAMPLE

```
//@version=5
indicator("syminfo recommendations", overlay = true)
//@variable A table containing information about analyst recommendations.
var table ratings = table.new(position.top_right, 8, 2, frame_color = #000000)
if barstate.islastconfirmedhistory
    //@variable The time value one year from the date of the last analyst recommendations.
    int YTD = syminfo.target_price_date + timeframe.in_seconds("12M") * 1000
    // Add header cells.
    table.cell(ratings, 0, 0, "Start Date", bgcolor = color.gray, text_color = #000000, te
    table.cell(ratings, 1, 0, "End Date", bgcolor = color.gray, text_color = #000000, text
    table.cell(ratings, 2, 0, "Buy", bgcolor = color.teal, text_color = #000000, text_size
    table.cell(ratings, 3, 0, "Strong Buy", bgcolor = color.lime, text_color = #000000, te
    table.cell(ratings, 4, 0, "Sell", bgcolor = color.maroon, text_color = #000000, text_s
    table.cell(ratings, 5, 0, "Strong Sell", bgcolor = color.red, text_color = #000000, te
    table.cell(ratings, 6, 0, "Hold", bgcolor = color.orange, text_color = #000000, text_s
    table.cell(ratings, 7, 0, "Total", bgcolor = color.silver, text_color = #000000, text_
    // Recommendation strings
    string startDate         = str.format_time(syminfo.recommendations_date, "yyyy-MM-dd")
    string endDate           = str.format_time(YTD, "yyyy-MM-dd")
    string buyRatings        = str.tostring(syminfo.recommendations_buy)
    string strongBuyRatings  = str.tostring(syminfo.recommendations_buy_strong)
    string sellRatings       = str.tostring(syminfo.recommendations_sell)
    string strongSellRatings = str.tostring(syminfo.recommendations_sell_strong)
    string holdRatings       = str.tostring(syminfo.recommendations_hold)
    string totalRatings      = str.tostring(syminfo.recommendations_total)
    // Add value cells
    table.cell(ratings, 0, 1, startDate, bgcolor = color.gray, text_color = #000000, text_
    table.cell(ratings, 1, 1, endDate, bgcolor = color.gray, text_color = #000000, text_si
    table.cell(ratings, 2, 1, buyRatings, bgcolor = color.teal, text_color = #000000, text
    table.cell(ratings, 3, 1, strongBuyRatings, bgcolor = color.lime, text_color = #000000
    table.cell(ratings, 4, 1, sellRatings, bgcolor = color.maroon, text_color = #000000, t
```

SEE ALSO

syminfo.recommendations_buy syminfo.recommendations_buy_strong

syminfo.recommendations_hold syminfo.recommendations_total syminfo.recommendations_sell

syminfo.recommendations_sell_strong

syminfo.recommendations_hold

The number of analysts who gave the current symbol a "Hold" rating.

TYPE

series int

EXAMPLE

```
//@version=5
indicator("syminfo recommendations", overlay = true)
//@variable A table containing information about analyst recommendations.
var table ratings = table.new(position.top_right, 8, 2, frame_color = #000000)
if barstate.islastconfirmedhistory
    //@variable The time value one year from the date of the last analyst recommendations.
    int YTD = syminfo.target_price_date + timeframe.in_seconds("12M") * 1000
    // Add header cells.
    table.cell(ratings, 0, 0, "Start Date", bgcolor = color.gray, text_color = #000000, te
    table.cell(ratings, 1, 0, "End Date", bgcolor = color.gray, text_color = #000000, text
    table.cell(ratings, 2, 0, "Buy", bgcolor = color.teal, text_color = #000000, text_size
    table.cell(ratings, 3, 0, "Strong Buy", bgcolor = color.lime, text_color = #000000, te
    table.cell(ratings, 4, 0, "Sell", bgcolor = color.maroon, text_color = #000000, text_s
    table.cell(ratings, 5, 0, "Strong Sell", bgcolor = color.red, text_color = #000000, te
    table.cell(ratings, 6, 0, "Hold", bgcolor = color.orange, text_color = #000000, text_s
    table.cell(ratings, 7, 0, "Total", bgcolor = color.silver, text_color = #000000, text_
    // Recommendation strings
    string startDate         = str.format_time(syminfo.recommendations_date, "yyyy-MM-dd")
    string endDate           = str.format_time(YTD, "yyyy-MM-dd")
    string buyRatings        = str.tostring(syminfo.recommendations_buy)
    string strongBuyRatings  = str.tostring(syminfo.recommendations_buy_strong)
    string sellRatings       = str.tostring(syminfo.recommendations_sell)
    string strongSellRatings = str.tostring(syminfo.recommendations_sell_strong)
    string holdRatings       = str.tostring(syminfo.recommendations_hold)
    string totalRatings      = str.tostring(syminfo.recommendations_total)
    // Add value cells
    table.cell(ratings, 0, 1, startDate, bgcolor = color.gray, text_color = #000000, text_
    table.cell(ratings, 1, 1, endDate, bgcolor = color.gray, text_color = #000000, text_si
    table.cell(ratings, 2, 1, buyRatings, bgcolor = color.teal, text_color = #000000, text
    table.cell(ratings, 3, 1, strongBuyRatings, bgcolor = color.lime, text_color = #000000
    table.cell(ratings, 4, 1, sellRatings, bgcolor = color.maroon, text_color = #000000, t
```

SEE ALSO

syminfo.recommendations_buy syminfo.recommendations_buy_strong

syminfo.recommendations_date syminfo.recommendations_total syminfo.recommendations_sell

syminfo.recommendations_sell_strong

syminfo.recommendations_sell

The number of analysts who gave the current symbol a "Sell" rating.

TYPE

series int

EXAMPLE

```
//@version=5
indicator("syminfo recommendations", overlay = true)
//@variable A table containing information about analyst recommendations.
var table ratings = table.new(position.top_right, 8, 2, frame_color = #000000)
if barstate.islastconfirmedhistory
    //@variable The time value one year from the date of the last analyst recommendations.
    int YTD = syminfo.target_price_date + timeframe.in_seconds("12M") * 1000
    // Add header cells.
    table.cell(ratings, 0, 0, "Start Date", bgcolor = color.gray, text_color = #000000, te
    table.cell(ratings, 1, 0, "End Date", bgcolor = color.gray, text_color = #000000, text
    table.cell(ratings, 2, 0, "Buy", bgcolor = color.teal, text_color = #000000, text_size
    table.cell(ratings, 3, 0, "Strong Buy", bgcolor = color.lime, text_color = #000000, te
    table.cell(ratings, 4, 0, "Sell", bgcolor = color.maroon, text_color = #000000, text_s
    table.cell(ratings, 5, 0, "Strong Sell", bgcolor = color.red, text_color = #000000, te
    table.cell(ratings, 6, 0, "Hold", bgcolor = color.orange, text_color = #000000, text_s
    table.cell(ratings, 7, 0, "Total", bgcolor = color.silver, text_color = #000000, text_
    // Recommendation strings
    string startDate         = str.format_time(syminfo.recommendations_date, "yyyy-MM-dd")
    string endDate           = str.format_time(YTD, "yyyy-MM-dd")
    string buyRatings        = str.tostring(syminfo.recommendations_buy)
    string strongBuyRatings  = str.tostring(syminfo.recommendations_buy_strong)
    string sellRatings       = str.tostring(syminfo.recommendations_sell)
    string strongSellRatings = str.tostring(syminfo.recommendations_sell_strong)
    string holdRatings       = str.tostring(syminfo.recommendations_hold)
    string totalRatings      = str.tostring(syminfo.recommendations_total)
    // Add value cells
    table.cell(ratings, 0, 1, startDate, bgcolor = color.gray, text_color = #000000, text_
    table.cell(ratings, 1, 1, endDate, bgcolor = color.gray, text_color = #000000, text_si
    table.cell(ratings, 2, 1, buyRatings, bgcolor = color.teal, text_color = #000000, text
    table.cell(ratings, 3, 1, strongBuyRatings, bgcolor = color.lime, text_color = #00000
    table.cell(ratings, 4, 1, sellRatings, bgcolor = color.maroon, text_color = #000000, t
```

SEE ALSO

syminfo.recommendations_buy syminfo.recommendations_buy_strong

syminfo.recommendations_date   syminfo.recommendations_hold

syminfo.recommendations_total   syminfo.recommendations_sell_strong

syminfo.recommendations_sell_strong

The number of analysts who gave the current symbol a "Strong Sell" rating.

TYPE

series int

EXAMPLE

```
//@version=5
indicator("syminfo recommendations", overlay = true)
//@variable A table containing information about analyst recommendations.
var table ratings = table.new(position.top_right, 8, 2, frame_color = #000000)
if barstate.islastconfirmedhistory
    //@variable The time value one year from the date of the last analyst recommendations.
    int YTD = syminfo.target_price_date + timeframe.in_seconds("12M") * 1000
    // Add header cells.
    table.cell(ratings, 0, 0, "Start Date", bgcolor = color.gray, text_color = #000000, te
    table.cell(ratings, 1, 0, "End Date", bgcolor = color.gray, text_color = #000000, text
    table.cell(ratings, 2, 0, "Buy", bgcolor = color.teal, text_color = #000000, text_size
    table.cell(ratings, 3, 0, "Strong Buy", bgcolor = color.lime, text_color = #000000, te
    table.cell(ratings, 4, 0, "Sell", bgcolor = color.maroon, text_color = #000000, text_s
    table.cell(ratings, 5, 0, "Strong Sell", bgcolor = color.red, text_color = #000000, te
    table.cell(ratings, 6, 0, "Hold", bgcolor = color.orange, text_color = #000000, text_s
    table.cell(ratings, 7, 0, "Total", bgcolor = color.silver, text_color = #000000, text_
    // Recommendation strings
    string startDate         = str.format_time(syminfo.recommendations_date, "yyyy-MM-dd")
    string endDate           = str.format_time(YTD, "yyyy-MM-dd")
    string buyRatings        = str.tostring(syminfo.recommendations_buy)
    string strongBuyRatings  = str.tostring(syminfo.recommendations_buy_strong)
    string sellRatings       = str.tostring(syminfo.recommendations_sell)
    string strongSellRatings = str.tostring(syminfo.recommendations_sell_strong)
    string holdRatings       = str.tostring(syminfo.recommendations_hold)
    string totalRatings      = str.tostring(syminfo.recommendations_total)
    // Add value cells
    table.cell(ratings, 0, 1, startDate, bgcolor = color.gray, text_color = #000000, text_
    table.cell(ratings, 1, 1, endDate, bgcolor = color.gray, text_color = #000000, text_si
    table.cell(ratings, 2, 1, buyRatings, bgcolor = color.teal, text_color = #000000, text
    table.cell(ratings, 3, 1, strongBuyRatings, bgcolor = color.lime, text_color = #000000
    table.cell(ratings, 4, 1, sellRatings, bgcolor = color.maroon, text_color = #000000, t
```

SEE ALSO

syminfo.recommendations_buy   syminfo.recommendations_buy_strong

syminfo.recommendations_date syminfo.recommendations_hold

syminfo.recommendations_total syminfo.recommendations_sell

syminfo.recommendations_total

The total number of recommendations for the current symbol.

TYPE

series int

EXAMPLE

```
//@version=5
indicator("syminfo recommendations", overlay = true)
//@variable A table containing information about analyst recommendations.
var table ratings = table.new(position.top_right, 8, 2, frame_color = #000000)
if barstate.islastconfirmedhistory
    //@variable The time value one year from the date of the last analyst recommendations.
    int YTD = syminfo.target_price_date + timeframe.in_seconds("12M") * 1000
    // Add header cells.
    table.cell(ratings, 0, 0, "Start Date", bgcolor = color.gray, text_color = #000000, te
    table.cell(ratings, 1, 0, "End Date", bgcolor = color.gray, text_color = #000000, text
    table.cell(ratings, 2, 0, "Buy", bgcolor = color.teal, text_color = #000000, text_size
    table.cell(ratings, 3, 0, "Strong Buy", bgcolor = color.lime, text_color = #000000, te
    table.cell(ratings, 4, 0, "Sell", bgcolor = color.maroon, text_color = #000000, text_s
    table.cell(ratings, 5, 0, "Strong Sell", bgcolor = color.red, text_color = #000000, te
    table.cell(ratings, 6, 0, "Hold", bgcolor = color.orange, text_color = #000000, text_s
    table.cell(ratings, 7, 0, "Total", bgcolor = color.silver, text_color = #000000, text_
    // Recommendation strings
    string startDate         = str.format_time(syminfo.recommendations_date, "yyyy-MM-dd")
    string endDate           = str.format_time(YTD, "yyyy-MM-dd")
    string buyRatings        = str.tostring(syminfo.recommendations_buy)
    string strongBuyRatings  = str.tostring(syminfo.recommendations_buy_strong)
    string sellRatings       = str.tostring(syminfo.recommendations_sell)
    string strongSellRatings = str.tostring(syminfo.recommendations_sell_strong)
    string holdRatings       = str.tostring(syminfo.recommendations_hold)
    string totalRatings      = str.tostring(syminfo.recommendations_total)
    // Add value cells
    table.cell(ratings, 0, 1, startDate, bgcolor = color.gray, text_color = #000000, text_
    table.cell(ratings, 1, 1, endDate, bgcolor = color.gray, text_color = #000000, text_si
    table.cell(ratings, 2, 1, buyRatings, bgcolor = color.teal, text_color = #000000, text
    table.cell(ratings, 3, 1, strongBuyRatings, bgcolor = color.lime, text_color = #00000
    table.cell(ratings, 4, 1, sellRatings, bgcolor = color.maroon, text_color = #000000, t
```

SEE ALSO

syminfo.recommendations_buy syminfo.recommendations_buy_strong

syminfo.recommendations_date syminfo.recommendations_hold syminfo.recommendations_sell

syminfo.recommendations_sell_strong

syminfo.root

Root for derivatives like futures contract. For other symbols returns the same value as syminfo.ticker.

TYPE

simple string

EXAMPLE

```
//@version=5
indicator("syminfo.root")

// If the current chart symbol is continuous futures ('ES1!'), it would display 'ES'.
if barstate.islastconfirmedhistory
    label.new(bar_index, high, syminfo.root)
```

SEE ALSO

syminfo.ticker syminfo.tickerid

syminfo.sector

Returns the sector of the symbol, or na if the symbol has no sector. Example: "Electronic Technology", "Technology services", "Energy Minerals", "Consumer Durables", etc. These are the same values one can see in the chart's "Symbol info" window.

TYPE

simple string

REMARKS

A sector is a broad section of the economy. An industry is a narrower classification. NASDAQ:CAT (Caterpillar, Inc.) for example, belongs to the "Producer Manufacturing" sector and the "Trucks/Construction/Farm Machinery" industry.

symInfo.session

Session type of the chart main series. Possible values are session.regular, session.extended.

TYPE

simple string

SEE ALSO

session.regular   session.extended

syminfo.shareholders

The number of shareholders the company has.

TYPE

simple int

EXAMPLE

```
//@version=5
indicator("syminfo simple")
//@variable A table containing information about a company's employees, shareholders, and
var result_table = table.new(position = position.top_right, columns = 2, rows = 5, border_
if barstate.islastconfirmedhistory
    // Add header cells
    table.cell(table_id = result_table, column = 0, row = 0, text = "name")
    table.cell(table_id = result_table, column = 1, row = 0, text = "value")
    // Add employee info cells.
    table.cell(table_id = result_table, column = 0, row = 1, text = "employees")
    table.cell(table_id = result_table, column = 1, row = 1, text = str.tostring(syminfo.e
    // Add shareholder cells.
    table.cell(table_id = result_table, column = 0, row = 2, text = "shareholders")
    table.cell(table_id = result_table, column = 1, row = 2, text = str.tostring(syminfo.s
    // Add float shares outstanding cells.
    table.cell(table_id = result_table, column = 0, row = 3, text = "shares_outstanding_fl
    table.cell(table_id = result_table, column = 1, row = 3, text = str.tostring(syminfo.s
    // Add total shares outstanding cells.
    table.cell(table_id = result_table, column = 0, row = 4, text = "shares_outstanding_to
    table.cell(table_id = result_table, column = 1, row = 4, text = str.tostring(syminfo.s
```

SEE ALSO

syminfo.employees   syminfo.shares_outstanding_float   syminfo.shares_outstanding_total

syminfo.shares_outstanding_float

The total number of shares outstanding a company has available, excluding any of its restricted shares.

TYPE

simple float

EXAMPLE

```
//@version=5
indicator("syminfo simple")
//@variable A table containing information about a company's employees, shareholders, and
var result_table = table.new(position = position.top_right, columns = 2, rows = 5, border_
if barstate.islastconfirmedhistory
    // Add header cells
    table.cell(table_id = result_table, column = 0, row = 0, text = "name")
    table.cell(table_id = result_table, column = 1, row = 0, text = "value")
    // Add employee info cells.
    table.cell(table_id = result_table, column = 0, row = 1, text = "employees")
    table.cell(table_id = result_table, column = 1, row = 1, text = str.tostring(syminfo.e
    // Add shareholder cells.
    table.cell(table_id = result_table, column = 0, row = 2, text = "shareholders")
    table.cell(table_id = result_table, column = 1, row = 2, text = str.tostring(syminfo.s
    // Add float shares outstanding cells.
    table.cell(table_id = result_table, column = 0, row = 3, text = "shares_outstanding_fl
    table.cell(table_id = result_table, column = 1, row = 3, text = str.tostring(syminfo.s
    // Add total shares outstanding cells.
    table.cell(table_id = result_table, column = 0, row = 4, text = "shares_outstanding_to
    table.cell(table_id = result_table, column = 1, row = 4, text = str.tostring(syminfo.s
```

SEE ALSO

syminfo.employees syminfo.shareholders syminfo.shares_outstanding_total

syminfo.shares_outstanding_total

The total number of shares outstanding a company has available, including restricted shares held by insiders, major shareholders, and employees.

TYPE

simple int

EXAMPLE

```
//@version=5
indicator("syminfo simple")
//@variable A table containing information about a company's employees, shareholders, and
var result_table = table.new(position = position.top_right, columns = 2, rows = 5, border_
if barstate.islastconfirmedhistory
    // Add header cells
    table.cell(table_id = result_table, column = 0, row = 0, text = "name")
    table.cell(table_id = result_table, column = 1, row = 0, text = "value")
    // Add employee info cells.
    table.cell(table_id = result_table, column = 0, row = 1, text = "employees")
    table.cell(table_id = result_table, column = 1, row = 1, text = str.tostring(syminfo.e
    // Add shareholder cells.
    table.cell(table_id = result_table, column = 0, row = 2, text = "shareholders")
    table.cell(table_id = result_table, column = 1, row = 2, text = str.tostring(syminfo.s
    // Add float shares outstanding cells.
    table.cell(table_id = result_table, column = 0, row = 3, text = "shares_outstanding_fl
    table.cell(table_id = result_table, column = 1, row = 3, text = str.tostring(syminfo.s
    // Add total shares outstanding cells.
    table.cell(table_id = result_table, column = 0, row = 4, text = "shares_outstanding_to
    table.cell(table_id = result_table, column = 1, row = 4, text = str.tostring(syminfo.s
```

SEE ALSO

syminfo.employees   syminfo.shareholders   syminfo.shares_outstanding_float

syminfo.target_price_average

The average of the last yearly price targets for the symbol predicted by analysts.

TYPE

series float

EXAMPLE

```
//@version=5
indicator("syminfo target_price")
if barstate.islastconfirmedhistory
    //@variable The time value one year from the date of the last analyst recommendations.
    int YTD = syminfo.target_price_date + timeframe.in_seconds("12M") * 1000
    //@variable A line connecting the current `close` to the highest yearly price estimate
    highLine = line.new(time, close, YTD, syminfo.target_price_high, color = color.green,
    //@variable A line connecting the current `close` to the lowest yearly price estimate.
    lowLine = line.new(time, close, YTD, syminfo.target_price_low, color = color.red, xloc
    //@variable A line connecting the current `close` to the median yearly price estimate.
    medianLine = line.new(time, close, YTD, syminfo.target_price_median, color = color.gra
    //@variable A line connecting the current `close` to the average yearly price estimate
    averageLine = line.new(time, close, YTD, syminfo.target_price_average, color = color.c
    // Fill the space between targets
    linefill.new(lowLine, medianLine, color.new(color.red, 90))
    linefill.new(medianLine, highLine, color.new(color.green, 90))
    // Create a label displaying the total number of analyst estimates.
    string estimatesText = str.format("Number of estimates: {0}", syminfo.target_price_est
    label.new(bar_index, close, estimatesText, textcolor = color.white, size = size.large)
```

SEE ALSO

symininfo.target_price_date symininfo.target_price_estimates syminfo.target_price_high syminfo.target_price_low syminfo.target_price_median

syminfo.target_price_date

The starting date of the last price target prediction for the current symbol.

TYPE

series int

EXAMPLE

```
//@version=5
indicator("syminfo target_price")
if barstate.islastconfirmedhistory
    //@variable The time value one year from the date of the last analyst recommendations.
    int YTD = syminfo.target_price_date + timeframe.in_seconds("12M") * 1000
    //@variable A line connecting the current `close` to the highest yearly price estimate
    highLine = line.new(time, close, YTD, syminfo.target_price_high, color = color.green,
    //@variable A line connecting the current `close` to the lowest yearly price estimate.
    lowLine = line.new(time, close, YTD, syminfo.target_price_low, color = color.red, xloc
    //@variable A line connecting the current `close` to the median yearly price estimate.
    medianLine = line.new(time, close, YTD, syminfo.target_price_median, color = color.gra
    //@variable A line connecting the current `close` to the average yearly price estimate
    averageLine = line.new(time, close, YTD, syminfo.target_price_average, color = color.c
    // Fill the space between targets
    linefill.new(lowLine, medianLine, color.new(color.red, 90))
    linefill.new(medianLine, highLine, color.new(color.green, 90))
    // Create a label displaying the total number of analyst estimates.
    string estimatesText = str.format("Number of estimates: {0}", syminfo.target_price_est
    label.new(bar_index, close, estimatesText, textcolor = color.white, size = size.large)
```

SEE ALSO

syminfo.target_price_average syminfo.target_price_estimates syminfo.target_price_high syminfo.target_price_low syminfo.target_price_median

syminfo.target_price_estimates

The latest total number of price target predictions for the current symbol.

TYPE

series float

EXAMPLE

```
//@version=5
indicator("syminfo target_price")
if barstate.islastconfirmedhistory
    //@variable The time value one year from the date of the last analyst recommendations.
    int YTD = syminfo.target_price_date + timeframe.in_seconds("12M") * 1000
    //@variable A line connecting the current `close` to the highest yearly price estimate
    highLine = line.new(time, close, YTD, syminfo.target_price_high, color = color.green,
    //@variable A line connecting the current `close` to the lowest yearly price estimate.
    lowLine = line.new(time, close, YTD, syminfo.target_price_low, color = color.red, xloc
    //@variable A line connecting the current `close` to the median yearly price estimate.
    medianLine = line.new(time, close, YTD, syminfo.target_price_median, color = color.gra
    //@variable A line connecting the current `close` to the average yearly price estimate
    averageLine = line.new(time, close, YTD, syminfo.target_price_average, color = color.c
    // Fill the space between targets
    linefill.new(lowLine, medianLine, color.new(color.red, 90))
    linefill.new(medianLine, highLine, color.new(color.green, 90))
    // Create a label displaying the total number of analyst estimates.
    string estimatesText = str.format("Number of estimates: {0}", syminfo.target_price_est
    label.new(bar_index, close, estimatesText, textcolor = color.white, size = size.large)
```

SEE ALSO

syminfo.target_price_average syminfo.target_price_date syminfo.target_price_high

syminfo.target_price_low syminfo.target_price_median

syminfo.target_price_high

The last highest yearly price target for the symbol predicted by analysts.

TYPE

series float

EXAMPLE

```
//@version=5
indicator("syminfo target_price")
if barstate.islastconfirmedhistory
    //@variable The time value one year from the date of the last analyst recommendations.
    int YTD = syminfo.target_price_date + timeframe.in_seconds("12M") * 1000
    //@variable A line connecting the current `close` to the highest yearly price estimate
    highLine = line.new(time, close, YTD, syminfo.target_price_high, color = color.green,
    //@variable A line connecting the current `close` to the lowest yearly price estimate.
    lowLine = line.new(time, close, YTD, syminfo.target_price_low, color = color.red, xloc
    //@variable A line connecting the current `close` to the median yearly price estimate.
    medianLine = line.new(time, close, YTD, syminfo.target_price_median, color = color.gra
    //@variable A line connecting the current `close` to the average yearly price estimate
    averageLine = line.new(time, close, YTD, syminfo.target_price_average, color = color.c
    // Fill the space between targets
    linefill.new(lowLine, medianLine, color.new(color.red, 90))
    linefill.new(medianLine, highLine, color.new(color.green, 90))
    // Create a label displaying the total number of analyst estimates.
    string estimatesText = str.format("Number of estimates: {0}", syminfo.target_price_est
    label.new(bar_index, close, estimatesText, textcolor = color.white, size = size.large)
```

SEE ALSO

syminfo.target_price_average syminfo.target_price_date syminfo.target_price_estimates

syminfo.target_price_low syminfo.target_price_median

syminfo.target_price_low

The last lowest yearly price target for the symbol predicted by analysts.

TYPE

series float

EXAMPLE

```
//@version=5
indicator("syminfo target_price")
if barstate.islastconfirmedhistory
    //@variable The time value one year from the date of the last analyst recommendations.
    int YTD = syminfo.target_price_date + timeframe.in_seconds("12M") * 1000
    //@variable A line connecting the current `close` to the highest yearly price estimate
    highLine = line.new(time, close, YTD, syminfo.target_price_high, color = color.green,
    //@variable A line connecting the current `close` to the lowest yearly price estimate.
    lowLine = line.new(time, close, YTD, syminfo.target_price_low, color = color.red, xloc
    //@variable A line connecting the current `close` to the median yearly price estimate.
    medianLine = line.new(time, close, YTD, syminfo.target_price_median, color = color.gra
    //@variable A line connecting the current `close` to the average yearly price estimate
    averageLine = line.new(time, close, YTD, syminfo.target_price_average, color = color.c
    // Fill the space between targets
    linefill.new(lowLine, medianLine, color.new(color.red, 90))
    linefill.new(medianLine, highLine, color.new(color.green, 90))
    // Create a label displaying the total number of analyst estimates.
    string estimatesText = str.format("Number of estimates: {0}", syminfo.target_price_est
    label.new(bar_index, close, estimatesText, textcolor = color.white, size = size.large)
```

SEE ALSO

syminfo.target_price_average syminfo.target_price_date syminfo.target_price_estimates syminfo.target_price_high syminfo.target_price_median

syminfo.target_price_median

The median of the last yearly price targets for the symbol predicted by analysts.

TYPE

series float

EXAMPLE

```
//@version=5
indicator("syminfo target_price")
if barstate.islastconfirmedhistory
    //@variable The time value one year from the date of the last analyst recommendations.
    int YTD = syminfo.target_price_date + timeframe.in_seconds("12M") * 1000
    //@variable A line connecting the current `close` to the highest yearly price estimate
    highLine = line.new(time, close, YTD, syminfo.target_price_high, color = color.green,
    //@variable A line connecting the current `close` to the lowest yearly price estimate.
    lowLine = line.new(time, close, YTD, syminfo.target_price_low, color = color.red, xloc
    //@variable A line connecting the current `close` to the median yearly price estimate.
    medianLine = line.new(time, close, YTD, syminfo.target_price_median, color = color.gra
    //@variable A line connecting the current `close` to the average yearly price estimate
    averageLine = line.new(time, close, YTD, syminfo.target_price_average, color = color.c
    // Fill the space between targets
    linefill.new(lowLine, medianLine, color.new(color.red, 90))
    linefill.new(medianLine, highLine, color.new(color.green, 90))
    // Create a label displaying the total number of analyst estimates.
    string estimatesText = str.format("Number of estimates: {0}", syminfo.target_price_est
    label.new(bar_index, close, estimatesText, textcolor = color.white, size = size.large)
```

SEE ALSO

syminfo.target_price_average syminfo.target_price_date syminfo.target_price_estimates symInfo.target_price_high symInfo.target_price_low

syminfo.ticker

Symbol name without exchange prefix, e.g. 'MSFT'.

TYPE

simple string

SEE ALSO

syminfo.tickerid timeframe.period timeframe.multiplier syminfo.root

syminfo.tickerid

Returns the full form of the ticker ID representing a symbol, for use as an argument in functions with a `ticker` or `symbol` parameter. It always includes the prefix (exchange) and ticker separated by a colon ("NASDAQ:AAPL"), but it can also include other symbol data such as dividend adjustment, chart type, currency conversion, etc.

TYPE

simple string

REMARKS

Because the value of this variable does not always use a simple "prefix:ticker" format, it is a poor candidate for use in boolean comparisons or string manipulation functions. In those contexts, run the variable's result through ticker.standard to purify it. This will remove any extraneous information and return a ticker ID consistently formatted using the "prefix:ticker" structure.

SEE ALSO

ticker.new syminfo.ticker timeframe.period timeframe.multiplier syminfo.root

syminfo.timezone

Timezone of the exchange of the chart main series. Possible values see in timestamp.

TYPE

simple string

SEE ALSO

timestamp

syminfo.type

The type of market the symbol belongs to. The values are "stock", "fund", "dr", "right", "bond", "warrant", "structured", "index", "forex", "futures", "spread", "economic", "fundamental", "crypto", "spot", "swap", "option", "commodity".

TYPE

simple string

SEE ALSO

syminfo.ticker

syminfo.volumetype

Volume type of the current symbol. Possible values are: "base" for base currency, "quote" for quote currency, "tick" for the number of transactions, and "n/a" when there is no volume or its type is not specified.

TYPE

simple string

REMARKS

Only some data feed suppliers provide information qualifying volume. As a result, the variable will return a value on some symbols only, mostly in the crypto sector.

SEE ALSO

syminfo.type

ta.accdist

Accumulation/distribution index.

TYPE

series float

ta.iii

Intraday Intensity Index.

TYPE

series float

EXAMPLE

```
//@version=5
indicator("Intraday Intensity Index")
plot(ta.iii, color=color.yellow)

// the same on pine
f_iii() =>
    (2 * close - high - low) / ((high - low) * volume)

plot(f_iii())
```

ta.nvi

Negative Volume Index.

TYPE

series float

EXAMPLE

```
//@version=5
indicator("Negative Volume Index")

plot(ta.nvi, color=color.yellow)

// the same on pine
f_nvi() =>
    float ta_nvi = 1.0
    float prevNvi = (nz(ta_nvi[1], 0.0) == 0.0)  ? 1.0: ta_nvi[1]
    if nz(close, 0.0) == 0.0 or nz(close[1], 0.0) == 0.0
        ta_nvi := prevNvi
    else
        ta_nvi := (volume < nz(volume[1], 0.0)) ? prevNvi + ((close - close[1]) / close[1
    result = ta_nvi

plot(f_nvi())
```

ta.obv

On Balance Volume.

TYPE

series float

EXAMPLE

```
//@version=5
indicator("On Balance Volume")
plot(ta.obv, color=color.yellow)

// the same on pine
f_obv() =>
    ta.cum(math.sign(ta.change(close)) * volume)

plot(f_obv())
```

ta.pvi

Positive Volume Index.

TYPE

series float

EXAMPLE

```
//@version=5
indicator("Positive Volume Index")

plot(ta.pvi, color=color.yellow)

// the same on pine
f_pvi() =>
    float ta_pvi = 1.0
    float prevPvi = (nz(ta_pvi[1], 0.0) == 0.0)  ? 1.0: ta_pvi[1]
    if nz(close, 0.0) == 0.0 or nz(close[1], 0.0) == 0.0
        ta_pvi := prevPvi
    else
        ta_pvi := (volume > nz(volume[1], 0.0)) ? prevPvi + ((close - close[1]) / close[1
    result = ta_pvi
```

```
plot(f_pvi())
```

ta.pvt

Price-Volume Trend.

TYPE

series float

EXAMPLE

```
//@version=5
indicator("Price-Volume Trend")
plot(ta.pvt, color=color.yellow)

// the same on pine
f_pvt() =>
    ta.cum((ta.change(close) / close[1]) * volume)

plot(f_pvt())
```

ta.tr

True range, equivalent to `ta.tr(handle_na = false)` . It is calculated as `math.max(high - low, math.abs(high - close[1]), math.abs(low - close[1]))` .

TYPE

series float

SEE ALSO

ta.tr ta.atr

ta.vwap

Volume Weighted Average Price. It uses hlc3 as its source series.

TYPE

series float

SEE ALSO

ta.vwap

ta.wad

Williams Accumulation/Distribution.

TYPE

series float

EXAMPLE

```
//@version=5
indicator("Williams Accumulation/Distribution")
plot(ta.wad, color=color.yellow)

// the same on pine
f_wad() =>
    trueHigh = math.max(high, close[1])
    trueLow = math.min(low, close[1])
    mom = ta.change(close)
    gain = (mom > 0) ? close - trueLow : (mom < 0) ? close - trueHigh : 0
    ta.cum(gain)

plot(f_wad())
```

ta.wvad

Williams Variable Accumulation/Distribution.

TYPE

series float

EXAMPLE

```
//@version=5
indicator("Williams Variable Accumulation/Distribution")
plot(ta.wvad, color=color.yellow)

// the same on pine
f_wvad() =>
```

```
    (close - open) / (high - low) * volume

plot(f_wvad())
```

table.all

Returns an array filled with all the current tables drawn by the script.

TYPE

array<table>

EXAMPLE

```
//@version=5
indicator("table.all")
//delete all tables
table.new(position = position.top_right, columns = 2, rows = 1, bgcolor = color.yellow, bc
a_allTables = table.all
if array.size(a_allTables) > 0
    for i = 0 to array.size(a_allTables) - 1
        table.delete(array.get(a_allTables, i))
```

REMARKS

The array is read-only. Index zero of the array is the ID of the oldest object on the chart.

SEE ALSO

table.new line.all label.all box.all

time

Current bar time in UNIX format. It is the number of milliseconds that have elapsed since 00:00:00 UTC, 1 January 1970.

TYPE

series int

REMARKS

Note that this variable returns the timestamp based on the time of the bar's open. Because of that, for overnight sessions (e.g. EURUSD, where Monday session starts on Sunday, 17:00) this variable can return time before the specified date of the trading day. For example, on

EURUSD, `dayofmonth(time)` can be lower by 1 than the date of the trading day, because the bar for the current day actually opens one day prior.

SEE ALSO

time time_close timenow year month weekofyear dayofmonth dayofweek hour minute second

time_close

The time of the current bar's close in UNIX format. It represents the number of milliseconds elapsed since 00:00:00 UTC, 1 January 1970. On tick charts and price-based charts such as Renko, line break, Kagi, point & figure, and range, this variable's series holds an na timestamp for the latest realtime bar (because the future closing time is unpredictable), but valid timestamps for all previous bars.

TYPE

series int

SEE ALSO

time timenow year month weekofyear dayofmonth dayofweek hour minute second

time_tradingday

The beginning time of the trading day the current bar belongs to, in UNIX format (the number of milliseconds that have elapsed since 00:00:00 UTC, 1 January 1970).

TYPE

series int

REMARKS

The variable is useful for overnight sessions, where the current day's session can start on the previous calendar day (e.g., on FXCM:EURUSD the Monday session will start on Sunday, 17:00 in the exchange timezone). Unlike `time`, which would return the timestamp for Sunday at 17:00 for the Monday daily bar, `time_tradingday` will return the timestamp for Monday, 00:00 UTC.

When used on timeframes higher than 1D, `time_tradingday` returns the trading day of the last day inside the bar (e.g. on 1W, it will return the last trading day of the week).

SEE ALSO

time time_close

timeframe.isdaily

Returns true if current resolution is a daily resolution, false otherwise.

TYPE

simple bool

SEE ALSO

timeframe.isdwm timeframe.isintraday timeframe.isminutes timeframe.isseconds timeframe.isticks timeframe.isweekly timeframe.ismonthly

timeframe.isdwm

Returns true if current resolution is a daily or weekly or monthly resolution, false otherwise.

TYPE

simple bool

SEE ALSO

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

timeframe.isintraday

Returns true if current resolution is an intraday (minutes or seconds) resolution, false otherwise.

TYPE

simple bool

SEE ALSO

timeframe.isminutes timeframe.isseconds timeframe.isticks timeframe.isdwm timeframe.isdaily timeframe.isweekly timeframe.ismonthly

timeframe.isminutes

Returns true if current resolution is a minutes resolution, false otherwise.

TYPE

simple bool

SEE ALSO

timeframe.isdwm timeframe.isintraday timeframe.isseconds timeframe.isticks timeframe.isdaily timeframe.isweekly timeframe.ismonthly

timeframe.ismonthly

Returns true if current resolution is a monthly resolution, false otherwise.

TYPE

simple bool

SEE ALSO

timeframe.isdwm timeframe.isintraday timeframe.isminutes timeframe.isseconds timeframe.isticks timeframe.isdaily timeframe.isweekly

timeframe.isseconds

Returns true if current resolution is a seconds resolution, false otherwise.

TYPE

simple bool

SEE ALSO

timeframe.isdwm timeframe.isintraday timeframe.isminutes timeframe.isticks timeframe.isdaily timeframe.isweekly timeframe.ismonthly

timeframe.isticks

Returns true if current resolution is a ticks resolution, false otherwise.

TYPE

simple bool

SEE ALSO

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

timeframe.isweekly

Returns true if current resolution is a weekly resolution, false otherwise.

TYPE

simple bool

SEE ALSO

timeframe.isdwm timeframe.isintraday timeframe.isminutes timeframe.isseconds timeframe.isticks timeframe.isdaily timeframe.ismonthly

timeframe.multiplier

Multiplier of resolution, e.g. '60' - 60, 'D' - 1, '5D' - 5, '12M' - 12.

TYPE

simple int

SEE ALSO

syminfo.ticker syminfo.tickerid timeframe.period

timeframe.period

A string representation of the chart's timeframe. The returned string's format is "[<quantity>][<units>]", where <quantity> and <units> are in some cases absent. <quantity> is the number of units, but it is absent if that number is 1. <unit> is "S" for seconds, "D" for days, "W" for weeks, "M" for months, but it is absent for minutes. No <unit> exists for hours.

The variable will return: "10S" for 10 seconds, "60" for 60 minutes, "D" for one day, "2W" for two weeks, "3M" for one quarter.

Can be used as an argument with any function containing a `timeframe` parameter.

TYPE

simple string

SEE ALSO

syminfo.ticker syminfo.tickerid timeframe.multiplier

timenow

Current time in UNIX format. It is the number of milliseconds that have elapsed since 00:00:00 UTC, 1 January 1970.

TYPE

series int

REMARKS

Please note that using this variable/function can cause indicator repainting.

SEE ALSO

timestamp time time_close year month weekofyear dayofmonth dayofweek hour minute second

volume

Current bar volume.

TYPE

series float

REMARKS

Previous values may be accessed with square brackets operator [], e.g. volume[1], volume[2].

SEE ALSO

open high low close time hl2 hlc3 hlcc4 ohlc4

weekofyear

Week number of current bar time in exchange timezone.

TYPE

series int

REMARKS

Note that this variable returns the week based on the time of the bar's open. For overnight sessions (e.g. EURUSD, where Monday session starts on Sunday, 17:00) this value can be lower by 1 than the week of the trading day.

SEE ALSO

weekofyear time year month dayofmonth dayofweek hour minute second

year

Current bar year in exchange timezone.

TYPE

series int

REMARKS

Note that this variable returns the year based on the time of the bar's open. For overnight sessions (e.g. EURUSD, where Monday session starts on Sunday, 17:00) this value can be lower by 1 than the year of the trading day.

SEE ALSO

year time month weekofyear dayofmonth dayofweek hour minute second

Constants

adjustment.dividends

Constant for dividends adjustment type (dividends adjustment is applied).

TYPE

const string

SEE ALSO

adjustment.none adjustment.splits ticker.new

adjustment.none

Constant for none adjustment type (no adjustment is applied).

TYPE

const string

SEE ALSO

adjustment.splits adjustment.dividends ticker.new

adjustment.splits

Constant for splits adjustment type (splits adjustment is applied).

TYPE

const string

SEE ALSO

adjustment.none adjustment.dividends ticker.new

alert.freq_all

A named constant for use with the `freq` parameter of the alert() function.

All function calls trigger the alert.

TYPE

const string

SEE ALSO

alert

alert.freq_once_per_bar

A named constant for use with the `freq` parameter of the alert() function.

The first function call during the bar triggers the alert.

TYPE

const string

SEE ALSO

alert

alert.freq_once_per_bar_close

A named constant for use with the `freq` parameter of the alert() function.

The function call triggers the alert only when it occurs during the last script iteration of the real-time bar, when it closes.

TYPE

const string

SEE ALSO

alert

backadjustment.inherit

A constant to specify the value of the `backadjustment` parameter in ticker.new and ticker.modify functions.

TYPE

const backadjustment

SEE ALSO

ticker.new ticker.modify backadjustment.on backadjustment.off

backadjustment.off

A constant to specify the value of the `backadjustment` parameter in ticker.new and ticker.modify functions.

TYPE

const backadjustment

SEE ALSO

ticker.new ticker.modify backadjustment.on backadjustment.inherit

backadjustment.on

A constant to specify the value of the `backadjustment` parameter in ticker.new and ticker.modify functions.

TYPE

const backadjustment

SEE ALSO

ticker.new ticker.modify backadjustment.inherit backadjustment.off

barmerge.gaps_off

Merge strategy for requested data. Data is merged continuously without gaps, all the gaps are filled with the previous nearest existing value.

TYPE

const barmerge_gaps

SEE ALSO

request.security barmerge.gaps_on

barmerge.gaps_on

Merge strategy for requested data. Data is merged with possible gaps (na values).

TYPE

const barmerge_gaps

SEE ALSO

request.security barmerge.gaps_off

barmerge.lookahead_off

Merge strategy for the requested data position. Requested barset is merged with current barset in the order of sorting bars by their close time. This merge strategy disables effect of getting data from "future" on calculation on history.

TYPE

const barmerge_lookahead

SEE ALSO

request.security barmerge.lookahead_on

barmerge.lookahead_on

Merge strategy for the requested data position. Requested barset is merged with current barset in the order of sorting bars by their opening time. This merge strategy can lead to undesirable effect of getting data from "future" on calculation on history. This is unacceptable in backtesting strategies, but can be useful in indicators.

TYPE

const barmerge_lookahead

SEE ALSO

request.security barmerge.lookahead_off

color.aqua

Is a named constant for #00BCD4 color.

TYPE

const color

SEE ALSO

color.black color.silver color.gray color.white color.maroon color.red color.purple color.fuchsia color.green color.lime color.olive color.yellow color.navy color.blue color.teal color.orange

color.black

Is a named constant for #363A45 color.

TYPE

const color

SEE ALSO

color.silver color.gray color.white color.maroon color.red color.purple color.fuchsia color.green color.lime color.olive color.yellow color.navy color.blue color.teal color.aqua color.orange

color.blue

Is a named constant for #2962ff color.

TYPE

const color

SEE ALSO

color.black color.silver color.gray color.white color.maroon color.red color.purple color.fuchsia color.green color.lime color.olive color.yellow color.navy color.teal color.aqua color.orange

color.fuchsia

Is a named constant for #E040FB color.

TYPE

const color

SEE ALSO

color.black color.silver color.gray color.white color.maroon color.red color.purple color.green color.lime color.olive color.yellow color.navy color.blue color.teal color.aqua color.orange

color.gray

Is a named constant for #787B86 color.

TYPE

const color

SEE ALSO

color.black color.silver color.white color.maroon color.red color.purple color.fuchsia color.green color.lime color.olive color.yellow color.navy color.blue color.teal color.aqua color.orange

color.green

Is a named constant for #4CAF50 color.

TYPE

const color

SEE ALSO

color.black color.silver color.gray color.white color.maroon color.red color.purple color.fuchsia color.lime color.olive color.yellow color.navy color.blue color.teal color.aqua color.orange

color.lime

Is a named constant for #00E676 color.

TYPE

const color

SEE ALSO

color.black color.silver color.gray color.white color.maroon color.red color.purple color.fuchsia color.green color.olive color.yellow color.navy color.blue color.teal color.aqua color.orange

color.maroon

Is a named constant for #880E4F color.

TYPE

const color

SEE ALSO

color.black color.silver color.gray color.white color.red color.purple color.fuchsia color.green color.lime color.olive color.yellow color.navy color.blue color.teal color.aqua color.orange

color.navy

Is a named constant for #311B92 color.

TYPE

const color

SEE ALSO

color.black color.silver color.gray color.white color.maroon color.red color.purple color.fuchsia color.green color.lime color.olive color.yellow color.blue color.teal color.aqua color.orange

color.olive

Is a named constant for #808000 color.

TYPE

const color

SEE ALSO

color.black color.silver color.gray color.white color.maroon color.red color.purple color.fuchsia color.green color.lime color.yellow color.navy color.blue color.teal color.aqua color.orange

color.orange

Is a named constant for #FF9800 color.

TYPE

const color

SEE ALSO

color.black color.silver color.gray color.white color.maroon color.red color.purple color.fuchsia color.green color.lime color.olive color.yellow color.navy color.blue color.teal color.aqua

color.purple

Is a named constant for #9C27B0 color.

TYPE

const color

SEE ALSO

color.black color.silver color.gray color.white color.maroon color.red color.fuchsia color.green color.lime color.olive color.yellow color.navy color.blue color.teal color.aqua color.orange

color.red

Is a named constant for #FF5252 color.

TYPE

const color

SEE ALSO

color.black color.silver color.gray color.white color.maroon color.purple color.fuchsia color.green color.lime color.olive color.yellow color.navy color.blue color.teal color.aqua color.orange

color.silver

Is a named constant for #B2B5BE color.

TYPE

const color

SEE ALSO

color.black color.gray color.white color.maroon color.red color.purple color.fuchsia color.green color.lime color.olive color.yellow color.navy color.blue color.teal color.aqua color.orange

color.teal

Is a named constant for #00897B color.

TYPE

const color

SEE ALSO

color.black color.silver color.gray color.white color.maroon color.red color.purple color.fuchsia color.green color.lime color.olive color.yellow color.navy color.blue color.aqua color.orange

color.white

Is a named constant for #FFFFFF color.

TYPE

const color

SEE ALSO

color.black color.silver color.gray color.maroon color.red color.purple color.fuchsia color.green color.lime color.olive color.yellow color.navy color.blue color.teal color.aqua color.orange

color.yellow

Is a named constant for #FFEB3B color.

TYPE

const color

SEE ALSO

color.black color.silver color.gray color.white color.maroon color.red color.purple color.fuchsia color.green color.lime color.olive color.navy color.blue color.teal color.aqua color.orange

currency.AUD

Australian dollar.

TYPE

const string

SEE ALSO

strategy

currency.BTC

Bitcoin.

TYPE

const string

SEE ALSO

strategy

currency.CAD

Canadian dollar.

TYPE

const string

SEE ALSO

strategy

currency.CHF

Swiss franc.

TYPE

const string

SEE ALSO

strategy

currency.ETH

Ethereum.

TYPE

const string

SEE ALSO

strategy

currency.EUR

Euro.

TYPE

const string

SEE ALSO

strategy

currency.GBP

Pound sterling.

TYPE

const string

SEE ALSO

strategy

currency.HKD

Hong Kong dollar.

TYPE

const string

SEE ALSO

strategy

currency.INR

Indian rupee.

TYPE

const string

SEE ALSO

strategy

currency.JPY

Japanese yen.

TYPE

const string

SEE ALSO

strategy

currency.KRW

South Korean won.

TYPE

const string

SEE ALSO

strategy

currency.MYR

Malaysian ringgit.

TYPE

const string

SEE ALSO

strategy

currency.NOK

Norwegian krone.

TYPE

const string

SEE ALSO

strategy

currency.NONE

Unspecified currency.

TYPE

const string

SEE ALSO

strategy

currency.NZD

New Zealand dollar.

TYPE

const string

SEE ALSO

strategy

currency.RUB

Russian ruble.

TYPE

const string

SEE ALSO

strategy

currency.SEK

Swedish krona.

TYPE

const string

SEE ALSO

strategy

currency.SGD

Singapore dollar.

TYPE

const string

SEE ALSO

strategy

currency.TRY

Turkish lira.

TYPE

const string

SEE ALSO

strategy

currency.USD

United States dollar.

TYPE

const string

SEE ALSO

strategy

currency.USDT

Tether.

TYPE

const string

SEE ALSO

strategy

currency.ZAR

South African rand.

TYPE

const string

SEE ALSO

strategy

dayofweek.friday

Is a named constant for return value of dayofweek function and value of dayofweek variable.

TYPE

const int

SEE ALSO

dayofweek.sunday dayofweek.monday dayofweek.tuesday dayofweek.wednesday dayofweek.thursday dayofweek.saturday

dayofweek.monday

Is a named constant for return value of dayofweek function and value of dayofweek variable.

TYPE

const int

SEE ALSO

dayofweek.sunday dayofweek.tuesday dayofweek.wednesday dayofweek.thursday dayofweek.friday dayofweek.saturday

dayofweek.saturday

Is a named constant for return value of dayofweek function and value of dayofweek variable.

TYPE

const int

SEE ALSO

dayofweek.sunday dayofweek.monday dayofweek.tuesday dayofweek.wednesday dayofweek.thursday dayofweek.friday

dayofweek.sunday

Is a named constant for return value of dayofweek function and value of dayofweek variable.

TYPE

const int

SEE ALSO

dayofweek.monday dayofweek.tuesday dayofweek.wednesday dayofweek.thursday dayofweek.friday dayofweek.saturday

dayofweek.thursday

Is a named constant for return value of dayofweek function and value of dayofweek variable.

TYPE

const int

SEE ALSO

dayofweek.sunday dayofweek.monday dayofweek.tuesday dayofweek.wednesday dayofweek.friday dayofweek.saturday

dayofweek.tuesday

Is a named constant for return value of dayofweek function and value of dayofweek variable.

TYPE

const int

SEE ALSO

dayofweek.sunday dayofweek.monday dayofweek.wednesday dayofweek.thursday dayofweek.friday dayofweek.saturday

dayofweek.wednesday

Is a named constant for return value of dayofweek function and value of dayofweek variable.

TYPE

const int

SEE ALSO

dayofweek.sunday dayofweek.monday dayofweek.tuesday dayofweek.thursday dayofweek.friday dayofweek.saturday

display.all

A named constant for use with the `display` parameter of `plot*()` and `input*()` functions. Displays plotted or input values in all possible locations.

TYPE

const plot_simple_display

SEE ALSO

plot plotshape plotchar plotarrow plotbar plotcandle

display.data_window

A named constant for use with the `display` parameter of `plot*()` and `input*()` functions. Displays plotted or input values in the Data Window, a menu accessible from the chart's right sidebar.

TYPE

const plot_display

SEE ALSO

plot plotshape plotchar plotarrow plotbar plotcandle

display.none

A named constant for use with the `display` parameter of `plot*()` and `input*()` functions. `plot*()` functions using this will not display their plotted values anywhere. However, alert template messages and fill functions can still use the values, and they will appear in exported chart data. `input*()` functions using this constant will only display their values within the script's settings.

TYPE

const plot_simple_display

SEE ALSO

plot plotshape plotchar plotarrow plotbar plotcandle

display.pane

A named constant for use with the `display` parameter of `plot*()` functions. Displays plotted values in the chart pane used by the script.

TYPE

const plot_display

SEE ALSO

plot plotshape plotchar plotarrow plotbar plotcandle

display.price_scale

A named constant for use with the `display` parameter of `plot*()` functions. Displays the plot’s label and value on the price scale if the chart's settings allow it.

TYPE

const plot_display

SEE ALSO

plot plotshape plotchar plotarrow plotbar plotcandle

display.status_line

A named constant for use with the `display` parameter of `plot*()` and `input*()` functions. Displays plotted or input values in the status line next to the script's name on the chart if the chart's settings allow it.

TYPE

const plot_display

SEE ALSO

plot plotshape plotchar plotarrow plotbar plotcandle

dividends.gross

A named constant for the request.dividends function. Is used to request the dividends return on a stock before deductions.

TYPE

const string

SEE ALSO

`request.dividends`

dividends.net

A named constant for the request.dividends function. Is used to request the dividends return on a stock after deductions.

TYPE

const string

SEE ALSO

`request.dividends`

earnings.actual

A named constant for the request.earnings function. Is used to request the earnings value as it was reported.

TYPE

const string

SEE ALSO

`request.earnings`

earnings.estimate

A named constant for the request.earnings function. Is used to request the estimated earnings value.

TYPE

const string

SEE ALSO

request.earnings

earnings.standardized

A named constant for the request.earnings function. Is used to request the standardized earnings value.

TYPE

const string

SEE ALSO

request.earnings

extend.both

A named constant for line.new and line.set_extend functions.

TYPE

const string

SEE ALSO

line.new line.set_extend extend.none extend.left extend.right

extend.left

A named constant for line.new and line.set_extend functions.

TYPE

const string

SEE ALSO

line.new line.set_extend extend.none extend.right extend.both

extend.none

A named constant for line.new and line.set_extend functions.

TYPE

const string

SEE ALSO

line.new line.set_extend extend.left extend.right extend.both

extend.right

A named constant for line.new and line.set_extend functions.

TYPE

const string

SEE ALSO

line.new line.set_extend extend.none extend.left extend.both

false

Literal representing a bool value, and result of a comparison operation.

REMARKS

See the User Manual for comparison operators and logical operators.

SEE ALSO

bool

font.family_default

Default text font for box.new, box.set_text_font_family, label.new, label.set_text_font_family, table.cell and table.cell_set_text_font_family functions.

TYPE

const string

SEE ALSO

box.new box.set_text_font_family label.new label.set_text_font_family table.cell table.cell_set_text_font_family font.family_monospace

font.family_monospace

Monospace text font for box.new, box.set_text_font_family, label.new, label.set_text_font_family, table.cell and table.cell_set_text_font_family functions.

TYPE

const string

SEE ALSO

box.new box.set_text_font_family label.new label.set_text_font_family table.cell

table.cell_set_text_font_family font.family_default

format.inherit

Is a named constant for selecting the formatting of the script output values from the parent series in the indicator function.

TYPE

const string

SEE ALSO

indicator format.price format.volume format.percent

format.mintick

Is a named constant to use with the str.tostring function. Passing a number to str.tostring with this argument rounds the number to the nearest value that can be divided by syminfo.mintick, without the remainder, with ties rounding up, and returns the string version of said value with trailing zeros.

TYPE

const string

SEE ALSO

indicator format.inherit format.price format.volume

format.percent

Is a named constant for selecting the formatting of the script output values as a percentage in the indicator function. It adds a percent sign after values.

TYPE

const string

REMARKS

The default precision is 2, regardless of the precision of the chart itself. This can be changed with the 'precision' argument of the indicator function.

SEE ALSO

indicator format.inherit format.price format.volume

format.price

Is a named constant for selecting the formatting of the script output values as prices in the indicator function.

TYPE

const string

REMARKS

If format is format.price, default precision value is set. You can use the precision argument of indicator function to change the precision value.

SEE ALSO

indicator format.inherit format.volume format.percent

format.volume

Is a named constant for selecting the formatting of the script output values as volume in the indicator function, e.g. '5183' will be formatted as '5.183K'.

The decimal precision rules defined by this variable take precedence over other precision settings. When an indicator, strategy, or `plot*()` call uses this `format` option, the function's `precision` parameter will not affect the result.

TYPE

const string

SEE ALSO

indicator format.inherit format.price format.percent

hline.style_dashed

Is a named constant for dashed linestyle of hline function.

TYPE

const hline_style

SEE ALSO

hline.style_solid hline.style_dotted

hline.style_dotted

Is a named constant for dotted linestyle of hline function.

TYPE

const hline_style

SEE ALSO

hline.style_solid hline.style_dashed

hline.style_solid

Is a named constant for solid linestyle of hline function.

TYPE

const hline_style

SEE ALSO

hline.style_dotted hline.style_dashed

label.style_arrowdown

Label style for label.new and label.set_style functions.

TYPE

const string

SEE ALSO

label.new label.set_style label.set_textalign label.style_none label.style_xcross label.style_cross label.style_triangleup label.style_triangledown label.style_flag label.style_circle label.style_arrowup label.style_label_up label.style_label_down label.style_label_left label.style_label_right label.style_label_lower_left label.style_label_lower_right label.style_label_upper_left label.style_label_upper_right label.style_label_center label.style_square label.style_diamond

label.style_arrowup

Label style for label.new and label.set_style functions.

TYPE

const string

SEE ALSO

label.new label.set_style label.set_textalign label.style_none label.style_xcross label.style_cross label.style_triangleup label.style_triangledown label.style_flag label.style_circle label.style_arrowdown label.style_label_up label.style_label_down label.style_label_left label.style_label_right label.style_label_lower_left label.style_label_lower_right label.style_label_upper_left label.style_label_upper_right label.style_label_center label.style_square label.style_diamond

label.style_circle

Label style for label.new and label.set_style functions.

TYPE

const string

SEE ALSO

label.new label.set_style label.set_textalign label.style_none label.style_xcross
label.style_cross label.style_triangleup label.style_triangledown label.style_flag
label.style_arrowup label.style_arrowdown label.style_label_up label.style_label_down
label.style_label_left label.style_label_right label.style_label_lower_left
label.style_label_lower_right label.style_label_upper_left label.style_label_upper_right
label.style_label_center label.style_square label.style_diamond

label.style_cross

Label style for label.new and label.set_style functions.

TYPE

const string

SEE ALSO

label.new label.set_style label.set_textalign label.style_none label.style_xcross
label.style_triangleup label.style_triangledown label.style_flag label.style_circle
label.style_arrowup label.style_arrowdown label.style_label_up label.style_label_down
label.style_label_left label.style_label_right label.style_label_lower_left
label.style_label_lower_right label.style_label_upper_left label.style_label_upper_right
label.style_label_center label.style_square label.style_diamond

label.style_diamond

Label style for label.new and label.set_style functions.

TYPE

const string

SEE ALSO

label.new label.set_style label.set_textalign label.style_none label.style_xcross
label.style_cross label.style_triangleup label.style_triangledown label.style_flag

label.style_circle label.style_arrowup label.style_arrowdown label.style_label_up

label.style_label_down label.style_label_left label.style_label_right

label.style_label_lower_left label.style_label_lower_right label.style_label_upper_left

label.style_label_upper_right label.style_label_center label.style_square

label.style_flag

Label style for label.new and label.set_style functions.

TYPE

const string

SEE ALSO

label.new label.set_style label.set_textalign label.style_none label.style_xcross

label.style_cross label.style_triangleup label.style_triangledown label.style_circle

label.style_arrowup label.style_arrowdown label.style_label_up label.style_label_down

label.style_label_left label.style_label_right label.style_label_lower_left

label.style_label_lower_right label.style_label_upper_left label.style_label_upper_right

label.style_label_center label.style_square label.style_diamond

label.style_label_center

Label style for label.new and label.set_style functions.

TYPE

const string

SEE ALSO

label.new label.set_style label.set_textalign label.style_none label.style_xcross

label.style_cross label.style_triangleup label.style_triangledown label.style_flag

label.style_circle label.style_arrowup label.style_arrowdown label.style_label_up

label.style_label_down label.style_label_left label.style_label_right label.style_label_lower_left label.style_label_lower_right label.style_label_upper_left

label.style_label_upper_right label.style_square label.style_diamond

label.style_label_down

Label style for label.new and label.set_style functions.

TYPE

const string

SEE ALSO

label.new label.set_style label.set_textalign label.style_none label.style_xcross

label.style_cross label.style_triangleup label.style_triangledown label.style_flag

label.style_circle label.style_arrowup label.style_arrowdown label.style_label_up

label.style_label_left label.style_label_right label.style_label_lower_left

label.style_label_lower_right label.style_label_upper_left label.style_label_upper_right

label.style_label_center label.style_square label.style_diamond

label.style_label_left

Label style for label.new and label.set_style functions.

TYPE

const string

SEE ALSO

label.new label.set_style label.set_textalign label.style_none label.style_xcross

label.style_cross label.style_triangleup label.style_triangledown label.style_flag

label.style_circle label.style_arrowup label.style_arrowdown label.style_label_up

label.style_label_down label.style_label_right label.style_label_lower_left

label.style_label_lower_right label.style_label_upper_left label.style_label_upper_right

label.style_label_center label.style_square label.style_diamond

label.style_label_lower_left

Label style for label.new and label.set_style functions.

TYPE

const string

SEE ALSO

label.new label.set_style label.set_textalign label.style_none label.style_xcross label.style_cross label.style_triangleup label.style_triangledown label.style_flag label.style_circle label.style_arrowup label.style_arrowdown label.style_label_up label.style_label_down label.style_label_left label.style_label_right label.style_label_lower_right label.style_label_upper_left label.style_label_upper_right label.style_label_center label.style_square label.style_diamond

label.style_label_lower_right

Label style for label.new and label.set_style functions.

TYPE

const string

SEE ALSO

label.new label.set_style label.set_textalign label.style_none label.style_xcross label.style_cross label.style_triangleup label.style_triangledown label.style_flag label.style_circle label.style_arrowup label.style_arrowdown label.style_label_up label.style_label_down label.style_label_left label.style_label_right label.style_label_lower_left label.style_label_upper_left label.style_label_upper_right label.style_label_center label.style_square label.style_diamond

label.style_label_right

Label style for label.new and label.set_style functions.

TYPE

const string

SEE ALSO

label.new label.set_style label.set_textalign label.style_none label.style_xcross label.style_cross label.style_triangleup label.style_triangledown label.style_flag label.style_circle label.style_arrowup label.style_arrowdown label.style_label_up label.style_label_down label.style_label_left label.style_label_lower_left label.style_label_lower_right label.style_label_upper_left label.style_label_upper_right label.style_label_center label.style_square label.style_diamond

label.style_label_up

Label style for label.new and label.set_style functions.

TYPE

const string

SEE ALSO

label.new label.set_style label.set_textalign label.style_none label.style_xcross label.style_cross label.style_triangleup label.style_triangledown label.style_flag label.style_circle label.style_arrowup label.style_arrowdown label.style_label_down label.style_label_left label.style_label_right label.style_label_lower_left label.style_label_lower_right label.style_label_upper_left label.style_label_upper_right label.style_label_center label.style_square label.style_diamond

label.style_label_upper_left

Label style for label.new and label.set_style functions.

TYPE

const string

SEE ALSO

label.new label.set_style label.set_textalign label.style_none label.style_xcross label.style_cross label.style_triangleup label.style_triangledown label.style_flag label.style_circle label.style_arrowup label.style_arrowdown label.style_label_up label.style_label_down label.style_label_left label.style_label_right label.style_label_lower_left label.style_label_lower_right label.style_label_upper_right label.style_label_center label.style_square label.style_diamond

label.style_label_upper_right

Label style for label.new and label.set_style functions.

TYPE

const string

SEE ALSO

label.new label.set_style label.set_textalign label.style_none label.style_xcross label.style_cross label.style_triangleup label.style_triangledown label.style_flag label.style_circle label.style_arrowup label.style_arrowdown label.style_label_up label.style_label_down label.style_label_left label.style_label_right label.style_label_lower_left label.style_label_lower_right label.style_label_upper_left label.style_label_center label.style_square label.style_diamond

label.style_none

Label style for label.new and label.set_style functions.

TYPE

const string

SEE ALSO

label.new label.set_style label.set_textalign label.style_xcross label.style_cross label.style_triangleup label.style_triangledown label.style_flag label.style_circle label.style_arrowup label.style_arrowdown label.style_label_up label.style_label_down label.style_label_left label.style_label_right label.style_label_center label.style_square label.style_diamond

label.style_square

Label style for label.new and label.set_style functions.

TYPE

const string

SEE ALSO

label.new label.set_style label.set_textalign label.style_none label.style_xcross label.style_cross label.style_triangleup label.style_triangledown label.style_flag label.style_circle label.style_arrowup label.style_arrowdown label.style_label_up label.style_label_down label.style_label_left label.style_label_right label.style_label_lower_left label.style_label_lower_right label.style_label_upper_left label.style_label_upper_right label.style_label_center label.style_diamond

label.style_text_outline

Label style for label.new and label.set_style functions.

TYPE

const string

SEE ALSO

label.new label.set_style label.set_textalign label.style_none label.style_xcross label.style_cross label.style_triangleup label.style_triangledown label.style_flag label.style_circle label.style_arrowup label.style_arrowdown label.style_label_up label.style_label_down label.style_label_left label.style_label_right

label.style_label_lower_left label.style_label_lower_right label.style_label_upper_left

label.style_label_upper_right label.style_label_center label.style_square

label.style_diamond

label.style_triangledown

Label style for label.new and label.set_style functions.

TYPE

const string

SEE ALSO

label.new label.set_style label.set_textalign label.style_none label.style_xcross

label.style_cross label.style_triangleup label.style_flag label.style_circle

label.style_arrowup label.style_arrowdown label.style_label_up label.style_label_down

label.style_label_left label.style_label_right label.style_label_lower_left

label.style_label_lower_right label.style_label_upper_left label.style_label_upper_right

label.style_label_center label.style_square label.style_diamond

label.style_triangleup

Label style for label.new and label.set_style functions.

TYPE

const string

SEE ALSO

label.new label.set_style label.set_textalign label.style_none label.style_xcross

label.style_cross label.style_triangledown label.style_flag label.style_circle

label.style_arrowup label.style_arrowdown label.style_label_up label.style_label_down

label.style_label_left label.style_label_right label.style_label_lower_left label.style_label_lower_right label.style_label_upper_left label.style_label_upper_right

label.style_label_center label.style_square label.style_diamond

label.style_xcross

Label style for label.new and label.set_style functions.

TYPE

const string

SEE ALSO

label.new label.set_style label.set_textalign label.style_none label.style_cross

label.style_triangleup label.style_triangledown label.style_flag label.style_circle

label.style_arrowup label.style_arrowdown label.style_label_up label.style_label_down

label.style_label_left label.style_label_right label.style_label_center label.style_square

label.style_diamond

line.style_arrow_both

Line style for line.new and line.set_style functions. Solid line with arrows on both points.

TYPE

const string

SEE ALSO

line.new line.set_style line.style_solid line.style_dotted line.style_dashed

line.style_arrow_left line.style_arrow_right

line.style_arrow_left

Line style for line.new and line.set_style functions. Solid line with arrow on the first point.

TYPE

const string

SEE ALSO

line.new line.set_style line.style_solid line.style_dotted line.style_dashed

line.style_arrow_right line.style_arrow_both

line.style_arrow_right

Line style for line.new and line.set_style functions. Solid line with arrow on the second point.

TYPE

const string

SEE ALSO

line.new line.set_style line.style_solid line.style_dotted line.style_dashed

line.style_arrow_left line.style_arrow_both

line.style_dashed

Line style for line.new and line.set_style functions.

TYPE

const string

SEE ALSO

line.new line.set_style line.style_solid line.style_dotted line.style_arrow_left

line.style_arrow_right line.style_arrow_both

line.style_dotted

Line style for line.new and line.set_style functions.

TYPE

const string

SEE ALSO

line.new line.set_style line.style_solid line.style_dashed line.style_arrow_left line.style_arrow_right line.style_arrow_both

line.style_solid

Line style for line.new and line.set_style functions.

TYPE

const string

SEE ALSO

line.new line.set_style line.style_dotted line.style_dashed line.style_arrow_left line.style_arrow_right line.style_arrow_both

location.abovebar

Location value for plotshape, plotchar functions. Shape is plotted above main series bars.

TYPE

const string

SEE ALSO

plotshape plotchar location.belowbar location.top location.bottom location.absolute

location.absolute

Location value for plotshape, plotchar functions. Shape is plotted on chart using indicator value as a price coordinate.

TYPE

const string

SEE ALSO

plotshape plotchar location.abovebar location.belowbar location.top location.bottom

location.belowbar

Location value for plotshape, plotchar functions. Shape is plotted below main series bars.

TYPE

const string

SEE ALSO

plotshape plotchar location.abovebar location.top location.bottom location.absolute

location.bottom

Location value for plotshape, plotchar functions. Shape is plotted near the bottom chart border.

TYPE

const string

SEE ALSO

plotshape plotchar location.abovebar location.belowbar location.top location.absolute

location.top

Location value for plotshape, plotchar functions. Shape is plotted near the top chart border.

TYPE

const string

SEE ALSO

plotshape plotchar location.abovebar location.belowbar location.bottom location.absolute

math.e

Is a named constant for Euler's number. It is equal to 2.7182818284590452.

TYPE

const float

SEE ALSO

math.phi math.pi math.rphi

math.phi

Is a named constant for the golden ratio. It is equal to 1.6180339887498948.

TYPE

const float

SEE ALSO

math.e math.pi math.rphi

math.pi

Is a named constant for Archimedes' constant. It is equal to 3.1415926535897932.

TYPE

const float

SEE ALSO

math.e math.phi math.rphi

math.rphi

Is a named constant for the golden ratio conjugate. It is equal to 0.6180339887498948.

TYPE

const float

SEE ALSO

math.e math.pi math.phi

order.ascending

Determines the sort order of the array from the smallest to the largest value.

TYPE

const sort_order

SEE ALSO

array.new_float array.sort

order.descending

Determines the sort order of the array from the largest to the smallest value.

TYPE

const sort_order

SEE ALSO

array.new_float array.sort

plot.style_area

A named constant for the 'Area' style, to be used as an argument for the `style` parameter in the plot function.

TYPE

const plot_style

SEE ALSO

plot plot.style_steplinebr plot.style_line plot.style_linebr plot.style_stepline plot.style_stepline_diamond plot.style_histogram plot.style_areabr plot.style_cross plot.style_columns plot.style_circles

plot.style_areabr

A named constant for the 'Area With Breaks' style, to be used as an argument for the `style` parameter in the plot function. Similar to plot.style_area, except the gaps in the data are not filled.

TYPE

const plot_style

SEE ALSO

plot plot.style_steplinebr plot.style_line plot.style_linebr plot.style_stepline plot.style_stepline_diamond plot.style_histogram plot.style_cross plot.style_area plot.style_columns plot.style_circles

plot.style_circles

A named constant for the 'Circles' style, to be used as an argument for the `style` parameter in the plot function.

TYPE

const plot_style

SEE ALSO

plot plot.style_steplinebr plot.style_line plot.style_linebr plot.style_stepline plot.style_stepline_diamond plot.style_histogram plot.style_cross plot.style_area plot.style_areabr plot.style_columns

plot.style_columns

A named constant for the 'Columns' style, to be used as an argument for the `style` parameter in the plot function.

TYPE

const plot_style

SEE ALSO

plot plot.style_steplinebr plot.style_line plot.style_linebr plot.style_stepline plot.style_stepline_diamond plot.style_histogram plot.style_cross plot.style_area plot.style_areabr plot.style_circles

plot.style_cross

A named constant for the 'Cross' style, to be used as an argument for the `style` parameter in the plot function.

TYPE

const plot_style

SEE ALSO

plot plot.style_steplinebr plot.style_line plot.style_linebr plot.style_stepline plot.style_stepline_diamond plot.style_histogram plot.style_area plot.style_areabr plot.style_columns plot.style_circles

plot.style_histogram

A named constant for the 'Histogram' style, to be used as an argument for the `style` parameter in the plot function.

TYPE

const plot_style

SEE ALSO

plot plot.style_steplinebr plot.style_line plot.style_linebr plot.style_stepline plot.style_stepline_diamond plot.style_cross plot.style_area plot.style_areabr plot.style_columns plot.style_circles

plot.style_line

A named constant for the 'Line' style, to be used as an argument for the `style` parameter in the plot function.

TYPE

const plot_style

SEE ALSO

plot plot.style_steplinebr plot.style_linebr plot.style_stepline plot.style_stepline_diamond plot.style_histogram plot.style_cross plot.style_area plot.style_areabr plot.style_columns plot.style_circles

plot.style_linebr

A named constant for the 'Line With Breaks' style, to be used as an argument for the `style` parameter in the plot function. Similar to plot.style_line, except the gaps in the data are not filled.

TYPE

const plot_style

SEE ALSO

plot plot.style_steplinebr plot.style_line plot.style_stepline plot.style_stepline_diamond plot.style_histogram plot.style_cross plot.style_area plot.style_areabr plot.style_columns plot.style_circles

plot.style_stepline

A named constant for the 'Step Line' style, to be used as an argument for the `style` parameter in the plot function.

TYPE

const plot_style

SEE ALSO

plot plot.style_line plot.style_steplinebr plot.style_stepline_diamond plot.style_linebr plot.style_histogram plot.style_cross plot.style_area plot.style_areabr plot.style_columns plot.style_circles

plot.style_stepline_diamond

A named constant for the 'Step Line With Diamonds' style, to be used as an argument for the `style` parameter in the plot function. Similar to plot.style_stepline, except the data changes are also marked with the Diamond shapes.

TYPE

const plot_style

SEE ALSO

plot plot.style_steplinebr plot.style_line plot.style_linebr plot.style_histogram plot.style_cross plot.style_area plot.style_areabr plot.style_columns plot.style_circles

plot.style_steplinebr

A named constant for the 'Step line with Breaks' style, to be used as an argument for the `style` parameter in the plot function.

TYPE

const plot_style

SEE ALSO

plot plot.style_circles plot.style_line plot.style_linebr plot.style_stepline plot.style_stepline_diamond plot.style_histogram plot.style_cross plot.style_area plot.style_areabr plot.style_columns

position.bottom_center

Table position is used in table.new, table.cell functions.

Binds the table to the bottom edge in the center.

TYPE

const string

SEE ALSO

table.new table.cell table.set_position position.top_left position.top_center position.top_right position.middle_left position.middle_center position.middle_right position.bottom_left

position.bottom_left

Table position is used in table.new, table.cell functions.

Binds the table to the bottom left of the screen.

TYPE

const string

SEE ALSO

table.new table.cell table.set_position position.top_left position.top_center position.top_right position.middle_left position.middle_center position.middle_right

position.bottom_center

position.bottom_right

Table position is used in table.new, table.cell functions.

Binds the table to the bottom right of the screen.

TYPE

const string

SEE ALSO

table.new table.cell table.set_position position.top_left position.top_center

position.top_right position.middle_left position.middle_center position.middle_right

position.bottom_left position.bottom_center

position.middle_center

Table position is used in table.new, table.cell functions.

Binds the table to the center of the screen.

TYPE

const string

SEE ALSO

table.new table.cell table.set_position position.top_left position.top_center

position.top_right position.middle_left position.middle_right position.bottom_left

position.bottom_center

position.middle_left

Table position is used in table.new, table.cell functions.

Binds the table to the left side of the screen.

TYPE

const string

SEE ALSO

table.new table.cell table.set_position position.top_left position.top_center position.top_right position.middle_center position.middle_right position.bottom_left position.bottom_center

position.middle_right

Table position is used in table.new, table.cell functions.

Binds the table to the right side of the screen.

TYPE

const string

SEE ALSO

table.new table.cell table.set_position position.top_left position.top_center position.top_right position.middle_left position.middle_center position.bottom_left position.bottom_center

position.top_center

Table position is used in table.new, table.cell functions.

Binds the table to the top edge in the center.

TYPE

const string

SEE ALSO

table.new table.cell table.set_position position.top_left position.top_right position.middle_left position.middle_center position.middle_right position.bottom_left position.bottom_center

position.top_left

Table position is used in table.new, table.cell functions.

Binds the table to the upper-left edge.

TYPE

const string

SEE ALSO

table.new table.cell table.set_position position.top_center position.top_right position.middle_left position.middle_center position.middle_right position.bottom_left position.bottom_center

position.top_right

Table position is used in table.new, table.cell functions.

Binds the table to the upper-right edge.

TYPE

const string

SEE ALSO

table.new table.cell table.set_position position.top_left position.top_center position.middle_left position.middle_center position.middle_right position.bottom_left position.bottom_center

scale.left

Scale value for indicator function. Indicator is added to the left price scale.

TYPE

const scale_type

SEE ALSO

indicator

scale.none

Scale value for indicator function. Indicator is added in 'No Scale' mode. Can be used only with 'overlay=true'.

TYPE

const scale_type

SEE ALSO

indicator

scale.right

Scale value for indicator function. Indicator is added to the right price scale.

TYPE

const scale_type

SEE ALSO

indicator

session.extended

Constant for extended session type (with extended hours data).

TYPE

const string

SEE ALSO

session.regular syminfo.session

session.regular

Constant for regular session type (no extended hours data).

TYPE

const string

SEE ALSO

session.extended syminfo.session

settlement_as_close.inherit

A constant to specify the value of the `settlement_as_close` parameter in ticker.new and ticker.modify functions.

TYPE

const settlement

SEE ALSO

ticker.new ticker.modify settlement_as_close.on settlement_as_close.off

settlement_as_close.off

A constant to specify the value of the `settlement_as_close` parameter in ticker.new and ticker.modify functions.

TYPE

const settlement

SEE ALSO

ticker.new ticker.modify settlement_as_close.on settlement_as_close.inherit

settlement_as_close.on

A constant to specify the value of the `settlement_as_close` parameter in ticker.new and ticker.modify functions.

TYPE

const settlement

SEE ALSO

ticker.new ticker.modify settlement_as_close.inherit settlement_as_close.off

shape.arrowdown

Shape style for plotshape function.

TYPE

const string

SEE ALSO

plotshape

shape.arrowup

Shape style for plotshape function.

TYPE

const string

SEE ALSO

plotshape

shape.circle

Shape style for plotshape function.

TYPE

const string

SEE ALSO

plotshape

shape.cross

Shape style for plotshape function.

TYPE

const string

SEE ALSO

plotshape

shape.diamond

Shape style for plotshape function.

TYPE

const string

SEE ALSO

plotshape

shape.flag

Shape style for plotshape function.

TYPE

const string

SEE ALSO

plotshape

shape.labeldown

Shape style for plotshape function.

TYPE

const string

SEE ALSO

plotshape

shape.labelup

Shape style for plotshape function.

TYPE

const string

SEE ALSO

plotshape

shape.square

Shape style for plotshape function.

TYPE

const string

SEE ALSO

plotshape

shape.triangledown

Shape style for plotshape function.

TYPE

const string

SEE ALSO

plotshape

shape.triangleup

Shape style for plotshape function.

TYPE

const string

SEE ALSO

plotshape

shape.xcross

Shape style for plotshape function.

TYPE

const string

SEE ALSO

plotshape

size.auto

Size value for plotshape, plotchar functions. The size of the shape automatically adapts to the size of the bars.

TYPE

const string

SEE ALSO

plotshape plotchar label.set_size size.tiny size.small size.normal size.large size.huge

size.huge

Size value for plotshape, plotchar functions. The size of the shape constantly huge.

TYPE

const string

SEE ALSO

plotshape plotchar label.set_size size.auto size.tiny size.small size.normal size.large

size.large

Size value for plotshape, plotchar functions. The size of the shape constantly large.

TYPE

const string

SEE ALSO

plotshape plotchar label.set_size size.auto size.tiny size.small size.normal size.huge

size.normal

Size value for plotshape, plotchar functions. The size of the shape constantly normal.

TYPE

const string

SEE ALSO

plotshape plotchar label.set_size size.auto size.tiny size.small size.large size.huge

size.small

Size value for plotshape, plotchar functions. The size of the shape constantly small.

TYPE

const string

SEE ALSO

plotshape plotchar label.set_size size.auto size.tiny size.normal size.large size.huge

size.tiny

Size value for plotshape, plotchar functions. The size of the shape constantly tiny.

TYPE

const string

SEE ALSO

plotshape plotchar label.set_size size.auto size.small size.normal size.large size.huge

splits.denominator

A named constant for the request.splits function. Is used to request the denominator (the number below the line in a fraction) of a splits.

TYPE

const string

SEE ALSO

request.splits

splits.numerator

A named constant for the request.splits function. Is used to request the numerator (the number above the line in a fraction) of a splits.

TYPE

const string

SEE ALSO

request.splits

strategy.cash

This is one of the arguments that can be supplied to the `default_qty_type` parameter in the strategy declaration statement. It is only relevant when no value is used for the 'qty' parameter in strategy.entry or strategy.order function calls. It specifies that an amount of cash in the `strategy.account_currency` will be used to enter trades.

TYPE

const string

EXAMPLE

```
//@version=5
strategy("strategy.cash", overlay = true, default_qty_value = 50, default_qty_type = strat

if bar_index == 0
    // As 'qty' is not defined, the previously defined values for the `default_qty_type` a
    // `qty` is calculated as (default_qty_value)/(close price). If current price is $5, t
    strategy.entry("EN", strategy.long)
if bar_index == 2
    strategy.close("EN")
```

SEE ALSO

strategy

strategy.commission.cash_per_contract

Commission type for an order. Money displayed in the account currency per contract.

TYPE

const string

SEE ALSO

strategy

strategy.commission.cash_per_order

Commission type for an order. Money displayed in the account currency per order.

TYPE

const string

SEE ALSO

strategy

strategy.commission.percent

Commission type for an order. A percentage of the cash volume of order.

TYPE

const string

SEE ALSO

strategy

strategy.direction.all

It allows strategy to open both long and short positions.

TYPE

const string

SEE ALSO

strategy.risk.allow_entry_in

strategy.direction.long

It allows strategy to open only long positions.

TYPE

const string

SEE ALSO

strategy.risk.allow_entry_in

strategy.direction.short

It allows strategy to open only short positions.

TYPE

const string

SEE ALSO

strategy.risk.allow_entry_in

strategy.fixed

This is one of the arguments that can be supplied to the `default_qty_type` parameter in the strategy declaration statement. It is only relevant when no value is used for the ‘qty’ parameter in strategy.entry or strategy.order function calls. It specifies that a number of contracts/shares/lots will be used to enter trades.

TYPE

const string

EXAMPLE

```
//@version=5
strategy("strategy.fixed", overlay = true, default_qty_value = 50, default_qty_type = stra

if bar_index == 0
    // As ‘qty’ is not defined, the previously defined values for the `default_qty_type` a
    // qty = 50
    strategy.entry("EN", strategy.long)
if bar_index == 2
    strategy.close("EN")
```

SEE ALSO

strategy

strategy.long

A named constant for use with the `direction` parameter of the strategy.entry and strategy.order commands. It specifies that the command creates a buy order.

TYPE

const strategy_direction

SEE ALSO

strategy.entry strategy.exit strategy.order

strategy.oca.cancel

A named constant for use with the `oca_type` parameter of the strategy.entry and strategy.order commands. It specifies that the strategy cancels the unfilled order when another order with the same `oca_name` and `oca_type` executes.

TYPE

const string

REMARKS

Strategies cannot cancel or reduce pending orders from an OCA group if they execute on the same tick. For example, if the market price triggers two stop orders from strategy.order calls with the same `oca_*` arguments, the strategy cannot fully or partially cancel either one.

SEE ALSO

strategy.entry strategy.exit strategy.order

strategy.oca.none

A named constant for use with the `oca_type` parameter of the strategy.entry and strategy.order commands. It specifies that the order executes independently of all other orders, including those with the same `oca_name`.

TYPE

const string

SEE ALSO

strategy.entry strategy.exit strategy.order

strategy.oca.reduce

A named constant for use with the `oca_type` parameter of the strategy.entry and strategy.order commands. It specifies that when another order with the same `oca_name` and `oca_type` executes, the strategy reduces the unfilled order by that order's size. If the unfilled order's size reaches 0 after reduction, it is the same as canceling the order entirely.

TYPE

const string

REMARKS

Strategies cannot cancel or reduce pending orders from an OCA group if they execute on the same tick. For example, if the market price triggers two stop orders from strategy.order calls with the same `oca_*` arguments, the strategy cannot fully or partially cancel either one.

Orders from strategy.exit automatically use this OCA type, and they belong to the same OCA group by default.

SEE ALSO

strategy.entry strategy.exit strategy.order

strategy.percent_of_equity

This is one of the arguments that can be supplied to the `default_qty_type` parameter in the strategy declaration statement. It is only relevant when no value is used for the ‘qty’ parameter in strategy.entry or strategy.order function calls. It specifies that a percentage (0-100) of equity will be used to enter trades.

TYPE

const string

EXAMPLE

```
//@version=5
strategy("strategy.percent_of_equity", overlay = false, default_qty_value = 100, default_q

// As 'qty' is not defined, the previously defined values for the `default_qty_type` and `
if bar_index == 0
    strategy.entry("EN", strategy.long)
if bar_index == 2
    strategy.close("EN")
plot(strategy.equity)

 // The 'qty' parameter is set to 10. Entering position with fixed size of 10 contracts ar
if bar_index == 4
    strategy.entry("EN", strategy.long, qty = 10)
if bar_index == 6
    strategy.close("EN")
```

SEE ALSO

strategy

strategy.short

A named constant for use with the `direction` parameter of the strategy.entry and strategy.order commands. It specifies that the command creates a sell order.

TYPE

const strategy_direction

SEE ALSO

strategy.entry strategy.exit strategy.order

text.align_bottom

Vertical text alignment for box.new, box.set_text_valign, table.cell and table.cell_set_text_valign functions.

TYPE

const string

SEE ALSO

table.cell table.cell_set_text_valign text.align_center text.align_left text.align_right

text.align_center

Text alignment for box.new, box.set_text_halign, box.set_text_valign, label.new and label.set_textalign functions.

TYPE

const string

SEE ALSO

label.new label.set_style text.align_left text.align_right

text.align_left

Horizontal text alignment for box.new, box.set_text_halign, label.new and label.set_textalign functions.

TYPE

const string

SEE ALSO

label.new label.set_style text.align_center text.align_right

text.align_right

Horizontal text alignment for box.new, box.set_text_halign, label.new and label.set_textalign functions.

TYPE

const string

SEE ALSO

label.new label.set_style text.align_center text.align_left

text.align_top

Vertical text alignment for box.new, box.set_text_valign, table.cell and table.cell_set_text_valign functions.

TYPE

const string

SEE ALSO

table.cell table.cell_set_text_valign text.align_center text.align_left text.align_right

text.wrap_auto

Automatic wrapping mode for box.new and box.set_text_wrap functions.

TYPE

const string

SEE ALSO

box.new box.set_text box.set_text_wrap

text.wrap_none

Disabled wrapping mode for box.new and box.set_text_wrap functions.

TYPE

const string

SEE ALSO

box.new box.set_text box.set_text_wrap

true

Literal representing one of the values a bool variable can hold, or an expression can evaluate to when it uses comparison or logical operators.

REMARKS

See the User Manual for comparison operators and logical operators.

SEE ALSO

bool

xloc.bar_index

A constant that specifies how functions that create and modify Pine drawings interpret x-coordinates. If `xloc = xloc.bar_index`, the drawing object treats each x-coordinate as a `bar_index` value.

TYPE

const string

SEE ALSO

xloc.bar_time line.new label.new box.new polyline.new line.set_xloc label.set_xloc

xloc.bar_time

A constant that specifies how functions that create and modify Pine drawings interpret x-coordinates. If `xloc = xloc.bar_time`, the drawing object treats each x-coordinate as a UNIX timestamp, expressed in milliseconds.

TYPE

const string

SEE ALSO

xloc.bar_index line.new label.new box.new polyline.new line.set_xloc label.set_xloc xloc.bar_index

yloc.abovebar

A named constant that specifies the algorithm of interpretation of y-value in function label.new.

TYPE

const string

SEE ALSO

label.new label.set_yloc yloc.price yloc.belowbar

yloc.belowbar

A named constant that specifies the algorithm of interpretation of y-value in function label.new.

TYPE

const string

SEE ALSO

label.new label.set_yloc yloc.price yloc.abovebar

yloc.price

A named constant that specifies the algorithm of interpretation of y-value in function label.new.

TYPE

const string

SEE ALSO

label.new label.set_yloc yloc.abovebar yloc.belowbar

Functions

alert()

Creates an alert event when called during the real-time bar, which will trigger a script alert based on "alert function events" if one was previously created for the indicator or strategy through the "Create Alert" dialog box.

SYNTAX

```
alert(message, freq) → void
```

ARGUMENTS

message (series string) Message sent when the alert triggers. Required argument.

freq (input string) The triggering frequency. Possible values are: alert.freq_all (all function calls trigger the alert), alert.freq_once_per_bar (the first function call during the bar triggers the alert), alert.freq_once_per_bar_close (the function call triggers the alert only when it occurs during the last script iteration of the real-time bar, when it closes). The default is alert.freq_once_per_bar.

EXAMPLE

```
//@version=5
indicator("`alert()` example", "", true)
ma = ta.sma(close, 14)
xUp = ta.crossover(close, ma)
if xUp
    // Trigger the alert the first time a cross occurs during the real-time bar.
    alert("Price (" + str.tostring(close) + ") crossed over MA (" + str.tostring(ma) +  ")
plot(ma)
plotchar(xUp, "xUp", "▲", location.top, size = size.tiny)
```

REMARKS

The Help Center explains how to create such alerts.

Contrary to alertcondition, alert calls do NOT count as an additional plot.

Function calls can be located in both global and local scopes.

Function calls do not display anything on the chart.

The 'freq' argument only affects the triggering frequency of the function call where it is used.

SEE ALSO

alertcondition

alertcondition()

Creates alert condition, that is available in Create Alert dialog. Please note, that alertcondition does NOT create an alert, it just gives you more options in Create Alert dialog. Also, alertcondition effect is invisible on chart.

SYNTAX

```
alertcondition(condition, title, message) → void
```

ARGUMENTS

condition (series bool) Series of boolean values that is used for alert. True values mean alert fire, false - no alert. Required argument.

title (const string) Title of the alert condition. Optional argument.

message (const string) Message to display when alert fires. Optional argument.

EXAMPLE

```
//@version=5
indicator("alertcondition", overlay=true)
alertcondition(close >= open, title='Alert on Green Bar', message='Green Bar!')
```

REMARKS

Please note that an alertcondition call generates an additional plot. All such calls are taken into account when we calculate the number of the output series per script.

SEE ALSO

alert

array.abs() 2 overloads

Returns an array containing the absolute value of each element in the original array.

SYNTAX & OVERLOADS

```
array.abs(id) → array<float>
```

```
array.abs(id) → array<int>
```

ARGUMENTS

id (array<int/float>) An array object.

SEE ALSO

array.new_float array.insert array.slice array.reverse order.ascending order.descending

array.avg() 2 overloads

The function returns the mean of an array's elements.

SYNTAX & OVERLOADS

```
array.avg(id) → series float
```

```
array.avg(id) → series int
```

ARGUMENTS

id (array<int/float>) An array object.

EXAMPLE

```
//@version=5
indicator("array.avg example")
a = array.new_float(0)
for i = 0 to 9
    array.push(a, close[i])
plot(array.avg(a))
```

RETURNS

Mean of array's elements.

SEE ALSO

array.new_float array.max array.min array.stdev

array.binary_search()

The function returns the index of the value, or -1 if the value is not found. The array to search must be sorted in ascending order.

SYNTAX

```
array.binary_search(id, val) → series int
```

ARGUMENTS

id (array<int/float>) An array object.

val (series int/float) The value to search for in the array.

EXAMPLE

```
//@version=5
indicator("array.binary_search")
a = array.from(5, -2, 0, 9, 1)
array.sort(a) // [-2, 0, 1, 5, 9]
position = array.binary_search(a, 0) // 1
plot(position)
```

REMARKS

A binary search works on arrays pre-sorted in ascending order. It begins by comparing an element in the middle of the array with the target value. If the element matches the target value, its position in the array is returned. If the element's value is greater than the target value, the search continues in the lower half of the array. If the element's value is less than the target value, the search continues in the upper half of the array. By doing this recursively, the algorithm progressively eliminates smaller and smaller portions of the array in which the target value cannot lie.

SEE ALSO

array.new_float array.insert array.slice array.reverse order.ascending order.descending

array.binary_search_leftmost()

The function returns the index of the value if it is found. When the value is not found, the function returns the index of the next smallest element to the left of where the value would lie if it was in the array. The array to search must be sorted in ascending order.

SYNTAX

```
array.binary_search_leftmost(id, val) → series int
```

ARGUMENTS

id (array<int/float>) An array object.

val (series int/float) The value to search for in the array.

EXAMPLE

```
//@version=5
indicator("array.binary_search_leftmost")
a = array.from(5, -2, 0, 9, 1)
array.sort(a) // [-2, 0, 1, 5, 9]
position = array.binary_search_leftmost(a, 3) // 2
plot(position)
```

EXAMPLE

```
//@version=5
indicator("array.binary_search_leftmost, repetitive elements")
a = array.from(4, 5, 5, 5)
// Returns the index of the first instance.
position = array.binary_search_leftmost(a, 5)
```

```
plot(position) // Plots 1
```

REMARKS

A binary search works on arrays pre-sorted in ascending order. It begins by comparing an element in the middle of the array with the target value. If the element matches the target value, its position in the array is returned. If the element's value is greater than the target value, the search continues in the lower half of the array. If the element's value is less than the target value, the search continues in the upper half of the array. By doing this recursively, the algorithm progressively eliminates smaller and smaller portions of the array in which the target value cannot lie.

SEE ALSO

array.new_float array.insert array.slice array.reverse order.ascending order.descending

array.binary_search_rightmost()

The function returns the index of the value if it is found. When the value is not found, the function returns the index of the element to the right of where the value would lie if it was in the array. The array must be sorted in ascending order.

SYNTAX

```
array.binary_search_rightmost(id, val) → series int
```

ARGUMENTS

id (array<int/float>) An array object.

val (series int/float) The value to search for in the array.

EXAMPLE

```
//@version=5
indicator("array.binary_search_rightmost")
a = array.from(5, -2, 0, 9, 1)
array.sort(a) // [-2, 0, 1, 5, 9]
position = array.binary_search_rightmost(a, 3) // 3
plot(position)
```

EXAMPLE

```
//@version=5
indicator("array.binary_search_rightmost, repetitive elements")
a = array.from(4, 5, 5, 5)
// Returns the index of the last instance.
position = array.binary_search_rightmost(a, 5)
plot(position) // Plots 3
```

REMARKS

A binary search works on sorted arrays in ascending order. It begins by comparing an element in the middle of the array with the target value. If the element matches the target value, its position in the array is returned. If the element's value is greater than the target value, the search continues in the lower half of the array. If the element's value is less than the target value, the search continues in the upper half of the array. By doing this recursively, the algorithm progressively eliminates smaller and smaller portions of the array in which the target value cannot lie.

SEE ALSO

array.new_float array.insert array.slice array.reverse order.ascending order.descending

array.clear()

The function removes all elements from an array.

SYNTAX

```
array.clear(id) → void
```

ARGUMENTS

id (any array type) An array object.

EXAMPLE

```
//@version=5
indicator("array.clear example")
a = array.new_float(5,high)
array.clear(a)
array.push(a, close)
plot(array.get(a,0))
plot(array.size(a))
```

SEE ALSO

array.new_float array.insert array.push array.remove array.pop

array.concat()

The function is used to merge two arrays. It pushes all elements from the second array to the first array, and returns the first array.

SYNTAX

```
array.concat(id1, id2) → array<type>
```

ARGUMENTS

id1 (any array type) The first array object.

id2 (any array type) The second array object.

EXAMPLE

```
//@version=5
indicator("array.concat example")
a = array.new_float(0,0)
b = array.new_float(0,0)
for i = 0 to 4
    array.push(a, high[i])
    array.push(b, low[i])
c = array.concat(a,b)
plot(array.size(a))
plot(array.size(b))
plot(array.size(c))
```

RETURNS

The first array with merged elements from the second array.

SEE ALSO

array.new_float array.insert array.slice

array.copy()

The function creates a copy of an existing array.

SYNTAX

```
array.copy(id) → array<type>
```

ARGUMENTS

id (any array type) An array object.

EXAMPLE

```
//@version=5
indicator("array.copy example")
length = 5
a = array.new_float(length, close)
b = array.copy(a)
a := array.new_float(length, open)
plot(array.sum(a) / length)
plot(array.sum(b) / length)
```

RETURNS

A copy of an array.

SEE ALSO

array.new_float array.get array.slice array.sort

array.covariance()

The function returns the covariance of two arrays.

SYNTAX

```
array.covariance(id1, id2, biased) → series float
```

ARGUMENTS

id1 (array<int/float>) An array object.

id2 (array<int/float>) An array object.

biased (series bool) Determines which estimate should be used. Optional. The default is true.

EXAMPLE

```
//@version=5
indicator("array.covariance example")
a = array.new_float(0)
b = array.new_float(0)
```

```
for i = 0 to 9
    array.push(a, close[i])
    array.push(b, open[i])
plot(array.covariance(a, b))
```

RETURNS

The covariance of two arrays.

REMARKS

If `biased` is true, function will calculate using a biased estimate of the entire population, if false - unbiased estimate of a sample.

SEE ALSO

array.new_float array.max array.stdev array.avg array.variance

array.every()

Returns true if all elements of the `id` array are true, false otherwise.

SYNTAX

```
array.every(id) → series bool
```

ARGUMENTS

id (array<bool>) An array object.

REMARKS

This function also works with arrays of int and float types, in which case zero values are considered false, and all others true.

SEE ALSO

array.some array.get

array.fill()

The function sets elements of an array to a single value. If no index is specified, all elements are set. If only a start index (default 0) is supplied, the elements starting at that index are set. If both index parameters are used, the elements from the starting index up to but not including the end index (default na) are set.

SYNTAX

```
array.fill(id, value, index_from, index_to) → void
```

ARGUMENTS

id (any array type) An array object.

value (series <type of the array's elements>) Value to fill the array with.

index_from (series int) Start index, default is 0.

index_to (series int) End index, default is na. Must be one greater than the index of the last element to set.

EXAMPLE

```
//@version=5
indicator("array.fill example")
a = array.new_float(10)
array.fill(a, close)
plot(array.sum(a))
```

SEE ALSO

array.new_float array.set array.slice

array.first()

Returns the array's first element. Throws a runtime error if the array is empty.

SYNTAX

```
array.first(id) → series <type>
```

ARGUMENTS

id (any array type) An array object.

EXAMPLE

```
//@version=5
indicator("array.first example")
arr = array.new_int(3, 10)
plot(array.first(arr))
```

SEE ALSO

array.last array.get

array.from() 12 overloads

The function takes a variable number of arguments with one of the types: int, float, bool, string, label, line, color, box, table, linefill, and returns an array of the corresponding type.

SYNTAX & OVERLOADS

```
array.from(arg0, arg1, ...) → array<type>
```

```
array.from(arg0, arg1, ...) → array<series enum>
```

```
array.from(arg0, arg1, ...) → array<label>
```

```
array.from(arg0, arg1, ...) → array<line>
```

```
array.from(arg0, arg1, ...) → array<box>
```

```
array.from(arg0, arg1, ...) → array<table>
```

```
array.from(arg0, arg1, ...) → array<linefill>
```

```
array.from(arg0, arg1, ...) → array<string>
```

```
array.from(arg0, arg1, ...) → array<color>
```

```
array.from(arg0, arg1, ...) → array<int>
```

```
array.from(arg0, arg1, ...) → array<float>
```

```
array.from(arg0, arg1, ...) → array<bool>
```

ARGUMENTS

arg0, arg1, ... (<arg..._type>) Array arguments.

EXAMPLE

```
//@version=5
indicator("array.from_example", overlay = false)
arr = array.from("Hello", "World!") // arr (array<string>) will contain 2 elements: {Hell
plot(close)
```

RETURNS

The array element's value.

REMARKS

This function can accept up to 4,000 'int', 'float', 'bool', or 'color' arguments. For all other types, including user-defined types, the limit is 999.

array.get()

The function returns the value of the element at the specified index.

SYNTAX

```
array.get(id, index) → series <type>
```

ARGUMENTS

id (any array type) An array object.

index (series int) The index of the element whose value is to be returned.

EXAMPLE

```
//@version=5
indicator("array.get example")
a = array.new_float(0)
for i = 0 to 9
    array.push(a, close[i] - open[i])
plot(array.get(a, 9))
```

RETURNS

The array element's value.

SEE ALSO

array.new_float array.set array.slice array.sort

array.includes()

The function returns true if the value was found in an array, false otherwise.

SYNTAX

```
array.includes(id, value) → series bool
```

ARGUMENTS

id (any array type) An array object.

value (series <type of the array's elements>) The value to search in the array.

EXAMPLE

```
//@version=5
indicator("array.includes example")
a = array.new_float(5,high)
p = close
if array.includes(a, high)
    p := open
plot(p)
```

RETURNS

True if the value was found in the array, false otherwise.

SEE ALSO

array.new_float array.indexof array.shift array.remove array.insert

array.indexof()

The function returns the index of the first occurrence of the value, or -1 if the value is not found.

SYNTAX

```
array.indexof(id, value) → series int
```

ARGUMENTS

id (any array type) An array object.

value (series <type of the array's elements>) The value to search in the array.

EXAMPLE

```
//@version=5
indicator("array.indexof example")
a = array.new_float(5,high)
index = array.indexof(a, high)
plot(index)
```

RETURNS

The index of an element.

SEE ALSO

array.lastindexof   array.get   array.lastindexof   array.remove   array.insert

array.insert()

The function changes the contents of an array by adding new elements in place.

SYNTAX

```
array.insert(id, index, value) → void
```

ARGUMENTS

id (any array type) An array object.

index (series int) The index at which to insert the value.

value (series <type of the array's elements>) The value to add to the array.

EXAMPLE

```
//@version=5
indicator("array.insert example")
a = array.new_float(5, close)
array.insert(a, 0, open)
plot(array.get(a, 5))
```

SEE ALSO

array.new_float   array.set   array.push   array.remove   array.pop   array.unshift

array.join()

The function creates and returns a new string by concatenating all the elements of an array, separated by the specified separator string.

SYNTAX

```
array.join(id, separator) → series string
```

ARGUMENTS

id (array<int/float/string>) An array object.

separator (series string) The string used to separate each array element.

EXAMPLE

```
//@version=5
indicator("array.join example")
a = array.new_float(5, 5)
label.new(bar_index, close, array.join(a, ","))
```

SEE ALSO

array.new_float   array.set   array.insert   array.remove   array.pop   array.unshift

array.last()

Returns the array's last element. Throws a runtime error if the array is empty.

SYNTAX

```
array.last(id) → series <type>
```

ARGUMENTS

id (any array type) An array object.

EXAMPLE

```
//@version=5
indicator("array.last example")
arr = array.new_int(3, 10)
plot(array.last(arr))
```

SEE ALSO

array.first array.get

array.lastindexof()

The function returns the index of the last occurrence of the value, or -1 if the value is not found.

SYNTAX

```
array.lastindexof(id, value) → series int
```

ARGUMENTS

id (any array type) An array object.

value (series <type of the array's elements>) The value to search in the array.

EXAMPLE

```
//@version=5
indicator("array.lastindexof example")
a = array.new_float(5,high)
index = array.lastindexof(a, high)
plot(index)
```

RETURNS

The index of an element.

SEE ALSO

array.new_float array.set array.push array.remove array.insert

array.max() 4 overloads

The function returns the greatest value, or the nth greatest value in a given array.

SYNTAX & OVERLOADS

```
array.max(id) → series float
```

```
array.max(id) → series int
```

```
array.max(id, nth) → series float
```

```
array.max(id, nth) → series int
```

ARGUMENTS

id (array<int/float>) An array object.

EXAMPLE

```
//@version=5
indicator("array.max")
a = array.from(5, -2, 0, 9, 1)
thirdHighest = array.max(a, 2) // 1
plot(thirdHighest)
```

RETURNS

The greatest or the nth greatest value in the array.

SEE ALSO

array.new_float array.min array.sum

array.median() 2 overloads

The function returns the median of an array's elements.

SYNTAX & OVERLOADS

```
array.median(id) → series float
```

```
array.median(id) → series int
```

ARGUMENTS

id (array<int/float>) An array object.

EXAMPLE

```
//@version=5
indicator("array.median example")
a = array.new_float(0)
for i = 0 to 9
    array.push(a, close[i])
plot(array.median(a))
```

RETURNS

The median of the array's elements.

SEE ALSO

array.median array.avg array.variance array.min

array.min() 4 overloads

The function returns the smallest value, or the nth smallest value in a given array.

SYNTAX & OVERLOADS

```
array.min(id) → series float
```

```
array.min(id) → series int
```

```
array.min(id, nth) → series float
```

```
array.min(id, nth) → series int
```

ARGUMENTS

id (array<int/float>) An array object.

EXAMPLE

```
//@version=5
indicator("array.min")
a = array.from(5, -2, 0, 9, 1)
secondLowest = array.min(a, 1) // 0
```

```
plot(secondLowest)
```

RETURNS

The smallest or the nth smallest value in the array.

SEE ALSO

array.new_float array.max array.sum

array.mode() 2 overloads

The function returns the mode of an array's elements. If there are several values with the same frequency, it returns the smallest value.

SYNTAX & OVERLOADS

```
array.mode(id) → series float
```

```
array.mode(id) → series int
```

ARGUMENTS

id (array<int/float>) An array object.

EXAMPLE

```
//@version=5
indicator("array.mode example")
a = array.new_float(0)
for i = 0 to 9
    array.push(a, close[i])
plot(array.mode(a))
```

RETURNS

The most frequently occurring value from the `id` array. If none exists, returns the smallest value instead.

SEE ALSO

array.new_float ta.mode matrix.mode array.avg array.variance array.min

array.new_bool()

The function creates a new array object of bool type elements.

SYNTAX

```
array.new_bool(size, initial_value) → array<bool>
```

ARGUMENTS

size (series int) Initial size of an array. Optional. The default is 0.

initial_value (series bool) Initial value of all array elements. Optional. The default is 'na'.

EXAMPLE

```
//@version=5
indicator("array.new_bool example")
length = 5
a = array.new_bool(length, close > open)
plot(array.get(a, 0) ? close : open)
```

RETURNS

The ID of an array object which may be used in other array.*() functions.

REMARKS

An array index starts from 0.

SEE ALSO

array.new_float array.get array.slice array.sort

array.new_box()

The function creates a new array object of box type elements.

SYNTAX

```
array.new_box(size, initial_value) → array<box>
```

ARGUMENTS

size (series int) Initial size of an array. Optional. The default is 0.

initial_value (series box) Initial value of all array elements. Optional. The default is 'na'.

EXAMPLE

```
//@version=5
indicator("array.new_box example")
boxes = array.new_box()
array.push(boxes, box.new(time, close, time+2, low, xloc=xloc.bar_time))
plot(1)
```

RETURNS

The ID of an array object which may be used in other array.*() functions.

REMARKS

An array index starts from 0.

SEE ALSO

array.new_float array.get array.slice

array.new_color()

The function creates a new array object of color type elements.

SYNTAX

```
array.new_color(size, initial_value) → array<color>
```

ARGUMENTS

size (series int) Initial size of an array. Optional. The default is 0.

initial_value (series color) Initial value of all array elements. Optional. The default is 'na'.

EXAMPLE

```
//@version=5
indicator("array.new_color example")
length = 5
a = array.new_color(length, color.red)
plot(close, color = array.get(a, 0))
```

RETURNS

The ID of an array object which may be used in other array.*() functions.

REMARKS

An array index starts from 0.

SEE ALSO

array.new_float   array.get   array.slice   array.sort

array.new_float()

The function creates a new array object of float type elements.

SYNTAX

```
array.new_float(size, initial_value) → array<float>
```

ARGUMENTS

size (series int) Initial size of an array. Optional. The default is 0.

initial_value (series int/float) Initial value of all array elements. Optional. The default is 'na'.

EXAMPLE

```
//@version=5
indicator("array.new_float example")
length = 5
a = array.new_float(length, close)
plot(array.sum(a) / length)
```

RETURNS

The ID of an array object which may be used in other array.*() functions.

REMARKS

An array index starts from 0.

SEE ALSO

array.new_color   array.new_bool   array.get   array.slice   array.sort

array.new_int()

The function creates a new array object of int type elements.

SYNTAX

```
array.new_int(size, initial_value) → array<int>
```

ARGUMENTS

size (series int) Initial size of an array. Optional. The default is 0.

initial_value (series int) Initial value of all array elements. Optional. The default is 'na'.

EXAMPLE

```
//@version=5
indicator("array.new_int example")
length = 5
a = array.new_int(length, int(close))
plot(array.sum(a) / length)
```

RETURNS

The ID of an array object which may be used in other array.*() functions.

REMARKS

An array index starts from 0.

SEE ALSO

array.new_float array.get array.slice array.sort

array.new_label()

The function creates a new array object of label type elements.

SYNTAX

```
array.new_label(size, initial_value) → array<label>
```

ARGUMENTS

size (series int) Initial size of an array. Optional. The default is 0.

initial_value (series label) Initial value of all array elements. Optional. The default is 'na'.

EXAMPLE

```
//@version=5
indicator("array.new_label example", overlay = true, max_labels_count = 500)

//@variable The number of labels to show on the chart.
int labelCount = input.int(50, "Labels to show", 1, 500)

//@variable An array of `label` objects.
var array<label> labelArray = array.new_label()

//@variable A `chart.point` for the new label.
labelPoint = chart.point.from_index(bar_index, close)
//@variable The text in the new label.
string labelText = na
//@variable The color of the new label.
color labelColor = na
//@variable The style of the new label.
string labelStyle = na

// Set the label attributes for rising bars.
if close > open
    labelText  := "Rising"
    labelColor := color.green
    labelStyle := label.style_label_down
// Set the label attributes for falling bars.
else if close < open
    labelText  := "Falling"
    labelColor := color.red
    labelStyle := label.style_label_up

// Add a new label to the `labelArray` when the chart bar closed at a new value.
if close != open
    labelArray.push(label.new(labelPoint, labelText, color = labelColor, style = labelStyl
// Remove the first element and delete its label when the size of the `labelArray` exceeds
if labelArray.size() > labelCount
    label.delete(labelArray.shift())
```

RETURNS

The ID of an array object which may be used in other array.*() functions.

REMARKS

An array index starts from 0.

SEE ALSO

array.new_float array.get array.slice

array.new_line()

The function creates a new array object of line type elements.

SYNTAX

```
array.new_line(size, initial_value) → array<line>
```

ARGUMENTS

size (series int) Initial size of an array. Optional. The default is 0.

initial_value (series line) Initial value of all array elements. Optional. The default is 'na'.

EXAMPLE

```
//@version=5
indicator("array.new_line example")
// draw last 15 lines
var a = array.new_line()
array.push(a, line.new(bar_index - 1, close[1], bar_index, close))
if array.size(a) > 15
    ln = array.shift(a)
    line.delete(ln)
```

RETURNS

The ID of an array object which may be used in other array.*() functions.

REMARKS

An array index starts from 0.

SEE ALSO

array.new_float array.get array.slice

array.new_linefill()

The function creates a new array object of linefill type elements.

SYNTAX

```
array.new_linefill(size, initial_value) → array<linefill>
```

ARGUMENTS

size (series int) Initial size of an array.

initial_value (series linefill) Initial value of all array elements.

RETURNS

The ID of an array object which may be used in other array.*() functions.

REMARKS

An array index starts from 0.

array.new_string()

The function creates a new array object of string type elements.

SYNTAX

```
array.new_string(size, initial_value) → array<string>
```

ARGUMENTS

size (series int) Initial size of an array. Optional. The default is 0.

initial_value (series string) Initial value of all array elements. Optional. The default is 'na'.

EXAMPLE

```
//@version=5
indicator("array.new_string example")
length = 5
a = array.new_string(length, "text")
label.new(bar_index, close, array.get(a, 0))
```

RETURNS

The ID of an array object which may be used in other array.*() functions.

REMARKS

An array index starts from 0.

SEE ALSO

array.new_float array.get array.slice

array.new_table()

The function creates a new array object of table type elements.

SYNTAX

```
array.new_table(size, initial_value) → array<table>
```

ARGUMENTS

size (series int) Initial size of an array. Optional. The default is 0.

initial_value (series table) Initial value of all array elements. Optional. The default is 'na'.

EXAMPLE

```
//@version=5
indicator("table array")
tables = array.new_table()
array.push(tables, table.new(position = position.top_left, rows = 1, columns = 2, bgcolor
plot(1)
```

RETURNS

The ID of an array object which may be used in other array.*() functions.

REMARKS

An array index starts from 0.

SEE ALSO

array.new_float array.get array.slice

array.new<type>()

The function creates a new array object of <type> elements.

SYNTAX

```
array.new<type>(size, initial_value) → array<type>
```

ARGUMENTS

size (series int) Initial size of an array. Optional. The default is 0.

initial_value (<array_type>) Initial value of all array elements. Optional. The default is 'na'.

EXAMPLE

```
//@version=5
```

```
indicator("array.new<string> example")
a = array.new<string>(1, "Hello, World!")
label.new(bar_index, close, array.get(a, 0))
```

EXAMPLE

```
//@version=5
indicator("array.new<color> example")
a = array.new<color>()
array.push(a, color.red)
array.push(a, color.green)
plot(close, color = array.get(a, close > open ? 1 : 0))
```

EXAMPLE

```
//@version=5
indicator("array.new<float> example")
length = 5
var a = array.new<float>(length, close)
if array.size(a) == length
    array.remove(a, 0)
    array.push(a, close)
plot(array.sum(a) / length, "SMA")
```

EXAMPLE

```
//@version=5
indicator("array.new<line> example")
// draw last 15 lines
var a = array.new<line>()
array.push(a, line.new(bar_index - 1, close[1], bar_index, close))
if array.size(a) > 15
    ln = array.shift(a)
    line.delete(ln)
```

RETURNS

The ID of an array object which may be used in other array.*() functions.

REMARKS

An array index starts from 0.

If you want to initialize an array and specify all its elements at the same time, then use the function array.from.

SEE ALSO

array.from array.push array.get array.size array.remove array.shift array.sum

array.percentile_linear_interpolation() 2 overloads

Returns the value for which the specified percentage of array values (percentile) are less than or equal to it, using linear interpolation.

SYNTAX & OVERLOADS

```
array.percentile_linear_interpolation(id, percentage) → series float
```

```
array.percentile_linear_interpolation(id, percentage) → series int
```

ARGUMENTS

id (array<int/float>) An array object.

percentage (series int/float) The percentage of values that must be equal or less than the returned value.

REMARKS

In statistics, the percentile is the percent of ranking items that appear at or below a certain score. This measurement shows the percentage of scores within a standard frequency distribution that is lower than the percentile rank you're measuring. Linear interpolation estimates the value between two ranks.

SEE ALSO

array.new_float array.insert array.slice array.reverse order.ascending order.descending

array.percentile_nearest_rank() 2 overloads

Returns the value for which the specified percentage of array values (percentile) are less than or equal to it, using the nearest-rank method.

SYNTAX & OVERLOADS

```
array.percentile_nearest_rank(id, percentage) → series float
```

```
array.percentile_nearest_rank(id, percentage) → series int
```

ARGUMENTS

id (array<int/float>) An array object.

percentage (series int/float) The percentage of values that must be equal or less than the returned value.

REMARKS

In statistics, the percentile is the percent of ranking items that appear at or below a certain score. This measurement shows the percentage of scores within a standard frequency distribution that is lower than the percentile rank you're measuring.

SEE ALSO

array.new_float array.insert array.slice array.reverse order.ascending order.descending

array.percentrank() 2 overloads

Returns the percentile rank of the element at the specified `index`.

SYNTAX & OVERLOADS

```
array.percentrank(id, index) → series float
```

```
array.percentrank(id, index) → series int
```

ARGUMENTS

id (array<int/float>) An array object.

index (series int) The index of the element for which the percentile rank should be calculated.

REMARKS

Percentile rank is the percentage of how many elements in the array are less than or equal to the reference value.

SEE ALSO

array.new_float array.insert array.slice array.reverse order.ascending order.descending

array.pop()

The function removes the last element from an array and returns its value.

SYNTAX

```
array.pop(id) → series <type>
```

ARGUMENTS

id (any array type) An array object.

EXAMPLE

```
//@version=5
indicator("array.pop example")
a = array.new_float(5,high)
removedEl = array.pop(a)
plot(array.size(a))
plot(removedEl)
```

RETURNS

The value of the removed element.

SEE ALSO

array.new_float array.set array.push array.remove array.insert array.shift

array.push()

The function appends a value to an array.

SYNTAX

```
array.push(id, value) → void
```

ARGUMENTS

id (any array type) An array object.

value (series <type of the array's elements>) The value of the element added to the end of the array.

EXAMPLE

```
//@version=5
indicator("array.push example")
a = array.new_float(5, 0)
```

```
array.push(a, open)
plot(array.get(a, 5))
```

SEE ALSO

array.new_float array.set array.insert array.remove array.pop array.unshift

array.range() 2 overloads

The function returns the difference between the min and max values from a given array.

SYNTAX & OVERLOADS

```
array.range(id) → series float
```

```
array.range(id) → series int
```

ARGUMENTS

id (array<int/float>) An array object.

EXAMPLE

```
//@version=5
indicator("array.range example")
a = array.new_float(0)
for i = 0 to 9
    array.push(a, close[i])
plot(array.range(a))
```

RETURNS

The difference between the min and max values in the array.

SEE ALSO

array.new_float array.min array.max array.sum

array.remove()

The function changes the contents of an array by removing the element with the specified index.

SYNTAX

```
array.remove(id, index) → series <type>
```

ARGUMENTS

id (any array type) An array object.

index (series int) The index of the element to remove.

EXAMPLE

```
//@version=5
indicator("array.remove example")
a = array.new_float(5,high)
removedEl = array.remove(a, 0)
plot(array.size(a))
plot(removedEl)
```

RETURNS

The value of the removed element.

SEE ALSO

array.new_float array.set array.push array.insert array.pop array.shift

array.reverse()

The function reverses an array. The first array element becomes the last, and the last array element becomes the first.

SYNTAX

```
array.reverse(id) → void
```

ARGUMENTS

id (any array type) An array object.

EXAMPLE

```
//@version=5
indicator("array.reverse example")
a = array.new_float(0)
for i = 0 to 9
```

```
    array.push(a, close[i])
plot(array.get(a, 0))
array.reverse(a)
plot(array.get(a, 0))
```

SEE ALSO

array.new_float array.sort array.push array.set array.avg

array.set()

The function sets the value of the element at the specified index.

SYNTAX

```
array.set(id, index, value) → void
```

ARGUMENTS

id (any array type) An array object.

index (series int) The index of the element to be modified.

value (series <type of the array's elements>) The new value to be set.

EXAMPLE

```
//@version=5
indicator("array.set example")
a = array.new_float(10)
for i = 0 to 9
    array.set(a, i, close[i])
plot(array.sum(a) / 10)
```

SEE ALSO

array.new_float array.get array.slice

array.shift()

The function removes an array's first element and returns its value.

SYNTAX

```
array.shift(id) → series <type>
```

ARGUMENTS

id (any array type) An array object.

EXAMPLE

```
//@version=5
indicator("array.shift example")
a = array.new_float(5,high)
removedEl = array.shift(a)
plot(array.size(a))
plot(removedEl)
```

RETURNS

The value of the removed element.

SEE ALSO

array.unshift array.set array.push array.remove array.includes

array.size()

The function returns the number of elements in an array.

SYNTAX

```
array.size(id) → series int
```

ARGUMENTS

id (any array type) An array object.

EXAMPLE

```
//@version=5
indicator("array.size example")
a = array.new_float(0)
for i = 0 to 9
    array.push(a, close[i])
// note that changes in slice also modify original array
slice = array.slice(a, 0, 5)
array.push(slice, open)
// size was changed in slice and in original array
plot(array.size(a))
plot(array.size(slice))
```

RETURNS

The number of elements in the array.

SEE ALSO

array.new_float array.sum array.slice array.sort

array.slice()

The function creates a slice from an existing array. If an object from the slice changes, the changes are applied to both the new and the original arrays.

SYNTAX

```
array.slice(id, index_from, index_to) → array<type>
```

ARGUMENTS

id (any array type) An array object.

index_from (series int) Zero-based index at which to begin extraction.

index_to (series int) Zero-based index before which to end extraction. The function extracts up to but not including the element with this index.

EXAMPLE

```
//@version=5
indicator("array.slice example")
a = array.new_float(0)
for i = 0 to 9
    array.push(a, close[i])
// take elements from 0 to 4
// *note that changes in slice also modify original array
slice = array.slice(a, 0, 5)
plot(array.sum(a) / 10)
plot(array.sum(slice) / 5)
```

RETURNS

A shallow copy of an array's slice.

SEE ALSO

array.new_float array.get array.slice array.sort

array.some()

Returns true if at least one element of the `id` array is true, false otherwise.

SYNTAX

```
array.some(id) → series bool
```

ARGUMENTS

id (array<bool>) An array object.

REMARKS

This function also works with arrays of int and float types, in which case zero values are considered false, and all others true.

SEE ALSO

array.every array.get

array.sort()

The function sorts the elements of an array.

SYNTAX

```
array.sort(id, order) → void
```

ARGUMENTS

id (array<int/float/string>) An array object.

order (series sort_order) The sort order: order.ascending (default) or order.descending.

EXAMPLE

```
//@version=5
indicator("array.sort example")
a = array.new_float(0,0)
for i = 0 to 5
    array.push(a, high[i])
array.sort(a, order.descending)
if barstate.islast
    label.new(bar_index, close, str.tostring(a))
```

SEE ALSO

array.new_float array.insert array.slice array.reverse order.ascending order.descending

array.sort_indices()

Returns an array of indices which, when used to index the original array, will access its elements in their sorted order. It does not modify the original array.

SYNTAX

```
array.sort_indices(id, order) → array<int>
```

ARGUMENTS

id (array<int/float/string>) An array object.

order (series sort_order) The sort order: order.ascending or order.descending. Optional. The default is order.ascending.

EXAMPLE

```
//@version=5
indicator("array.sort_indices")
a = array.from(5, -2, 0, 9, 1)
sortedIndices = array.sort_indices(a) // [1, 2, 4, 0, 3]
indexOfSmallestValue = array.get(sortedIndices, 0) // 1
smallestValue = array.get(a, indexOfSmallestValue) // -2
plot(smallestValue)
```

SEE ALSO

array.new_float array.insert array.slice array.reverse order.ascending order.descending

array.standardize() 2 overloads

The function returns the array of standardized elements.

SYNTAX & OVERLOADS

```
array.standardize(id) → array<float>
array.standardize(id) → array<int>
```

ARGUMENTS

id (array<int/float>) An array object.

EXAMPLE

```
//@version=5
indicator("array.standardize example")
a = array.new_float(0)
for i = 0 to 9
    array.push(a, close[i])
b = array.standardize(a)
plot(array.min(b))
plot(array.max(b))
```

RETURNS

The array of standardized elements.

SEE ALSO

array.max array.min array.mode array.avg array.variance array.stdev

array.stdev() 2 overloads

The function returns the standard deviation of an array's elements.

SYNTAX & OVERLOADS

```
array.stdev(id, biased) → series float
```

```
array.stdev(id, biased) → series int
```

ARGUMENTS

id (array<int/float>) An array object.

biased (series bool) Determines which estimate should be used. Optional. The default is true.

EXAMPLE

```
//@version=5
```

```
indicator("array.stdev example")
a = array.new_float(0)
for i = 0 to 9
    array.push(a, close[i])
plot(array.stdev(a))
```

RETURNS

The standard deviation of the array's elements.

REMARKS

If `biased` is true, function will calculate using a biased estimate of the entire population, if false - unbiased estimate of a sample.

SEE ALSO

array.new_float array.max array.min array.avg

array.sum() 2 overloads

The function returns the sum of an array's elements.

SYNTAX & OVERLOADS

```
array.sum(id) → series float
```

```
array.sum(id) → series int
```

ARGUMENTS

id (array<int/float>) An array object.

EXAMPLE

```
//@version=5
indicator("array.sum example")
a = array.new_float(0)
for i = 0 to 9
    array.push(a, close[i])
plot(array.sum(a))
```

RETURNS

The sum of the array's elements.

SEE ALSO

array.new_float array.max array.min

array.unshift()

The function inserts the value at the beginning of the array.

SYNTAX

```
array.unshift(id, value) → void
```

ARGUMENTS

id (any array type) An array object.

value (series <type of the array's elements>) The value to add to the start of the array.

EXAMPLE

```
//@version=5
indicator("array.unshift example")
a = array.new_float(5, 0)
array.unshift(a, open)
plot(array.get(a, 0))
```

SEE ALSO

array.shift array.set array.insert array.remove array.indexof

array.variance() 2 overloads

The function returns the variance of an array's elements.

SYNTAX & OVERLOADS

```
array.variance(id, biased) → series float
```

```
array.variance(id, biased) → series int
```

ARGUMENTS

id (array<int/float>) An array object.

biased (series bool) Determines which estimate should be used. Optional. The default is true.

EXAMPLE

```
//@version=5
indicator("array.variance example")
a = array.new_float(0)
for i = 0 to 9
    array.push(a, close[i])
plot(array.variance(a))
```

RETURNS

The variance of the array's elements.

REMARKS

If `biased` is true, function will calculate using a biased estimate of the entire population, if false - unbiased estimate of a sample.

SEE ALSO

array.new_float array.stdev array.min array.avg array.covariance

barcolor()

Set color of bars.

SYNTAX

```
barcolor(color, offset, editable, show_last, title, display) → void
```

ARGUMENTS

color (series color) Color of bars. You can use constants like 'red' or '#ff001a' as well as complex expressions like 'close >= open ? color.green : color.red'. Required argument.

offset (series int) Shifts the color series to the left or to the right on the given number of bars. Default is 0.

editable (const bool) If true then barcolor style will be editable in Format dialog. Default is true.

show_last (input int) If set, defines the number of bars (from the last bar back to the past) to fill on chart.

title (const string) Title of the barcolor. Optional argument.

display (input plot_simple_display) Controls where the barcolor is displayed. Possible values are: display.none, display.all. Default is display.all.

EXAMPLE

```
//@version=5
indicator("barcolor example", overlay=true)
barcolor(close < open ? color.black : color.white)
```

SEE ALSO

bgcolor plot fill

bgcolor()

Fill background of bars with specified color.

SYNTAX

```
bgcolor(color, offset, editable, show_last, title, display, force_overlay) → void
```

ARGUMENTS

color (series color) Color of the filled background. You can use constants like 'red' or '#ff001a' as well as complex expressions like 'close >= open ? color.green : color.red'. Required argument.

offset (series int) Shifts the color series to the left or to the right on the given number of bars. Default is 0.

editable (const bool) If true then bgcolor style will be editable in Format dialog. Default is true.

show_last (input int) If set, defines the number of bars (from the last bar back to the past) to fill on chart.

title (const string) Title of the bgcolor. Optional argument.

display (input plot_simple_display) Controls where the bgcolor is displayed. Possible values are: display.none, display.all. Default is display.all.

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

EXAMPLE

```
//@version=5
indicator("bgcolor example", overlay=true)
bgcolor(close < open ? color.new(color.red,70) : color.new(color.green, 70))
```

SEE ALSO

barcolor plot fill

bool() 4 overloads

Casts na to bool

SYNTAX & OVERLOADS

```
bool(x) → const bool
```

```
bool(x) → input bool
```

```
bool(x) → simple bool
```

```
bool(x) → series bool
```

ARGUMENTS

x (simple int/float/bool) The value to convert to the specified type, usually na.

RETURNS

The value of the argument after casting to bool.

SEE ALSO

float int color string line label

box()

Casts na to box.

SYNTAX

```
box(x) → series box
```

ARGUMENTS

x (series box) The value to convert to the specified type, usually na.

RETURNS

The value of the argument after casting to box.

SEE ALSO

float int bool color string line label

box.copy()

Clones the box object.

SYNTAX

```
box.copy(id) → series box
```

ARGUMENTS

id (series box) Box object.

EXAMPLE

```
//@version=5
indicator('Last 50 bars price ranges', overlay = true)
LOOKBACK = 50
highest = ta.highest(LOOKBACK)
lowest = ta.lowest(LOOKBACK)
if barstate.islastconfirmedhistory
    var BoxLast = box.new(bar_index[LOOKBACK], highest, bar_index, lowest, bgcolor = color
    var BoxPrev = box.copy(BoxLast)
    box.set_lefttop(BoxPrev, bar_index[LOOKBACK * 2], highest[50])
    box.set_rightbottom(BoxPrev, bar_index[LOOKBACK], lowest[50])
    box.set_bgcolor(BoxPrev, color.new(color.red, 80))
```

SEE ALSO

box.new box.delete

box.delete()

Deletes the specified box object. If it has already been deleted, does nothing.

SYNTAX

```
box.delete(id) → void
```

ARGUMENTS

id (series box) A box object to delete.

SEE ALSO

box.new

box.get_bottom()

Returns the price value of the bottom border of the box.

SYNTAX

```
box.get_bottom(id) → series float
```

ARGUMENTS

id (series box) A box object.

RETURNS

The price value.

SEE ALSO

box.new box.set_bottom

box.get_left()

Returns the bar index or the UNIX time (depending on the last value used for 'xloc') of the left border of the box.

SYNTAX

```
box.get_left(id) → series int
```

ARGUMENTS

id (series box) A box object.

RETURNS

A bar index or a UNIX timestamp (in milliseconds).

SEE ALSO

box.new box.set_left

box.get_right()

Returns the bar index or the UNIX time (depending on the last value used for 'xloc') of the right border of the box.

SYNTAX

```
box.get_right(id) → series int
```

ARGUMENTS

id (series box) A box object.

RETURNS

A bar index or a UNIX timestamp (in milliseconds).

SEE ALSO

box.new box.set_right

box.get_top()

Returns the price value of the top border of the box.

SYNTAX

```
box.get_top(id) → series float
```

ARGUMENTS

id (series box) A box object.

RETURNS

The price value.

SEE ALSO

box.new box.set_top

box.new() 2 overloads

Creates a new box object.

SYNTAX & OVERLOADS

```
box.new(top_left, bottom_right, border_color, border_width, border_style, extend, xloc,
bgcolor, text, text_size, text_color, text_halign, text_valign, text_wrap,
text_font_family, force_overlay) → series box
```

```
box.new(left, top, right, bottom, border_color, border_width, border_style, extend, xloc,
bgcolor, text, text_size, text_color, text_halign, text_valign, text_wrap,
text_font_family, force_overlay) → series box
```

ARGUMENTS

top_left (chart.point) A chart.point object that specifies the top-left corner location of the box.

bottom_right (chart.point) A chart.point object that specifies the bottom-right corner location of the box.

border_color (series color) Color of the four borders. Optional. The default is color.blue.

border_width (series int) Width of the four borders, in pixels. Optional. The default is 1 pixel.

border_style (series string) Style of the four borders. Possible values: line.style_solid, line.style_dotted, line.style_dashed. Optional. The default value is line.style_solid.

extend (series string) When extend.none is used, the horizontal borders start at the left border and end at the right border. With extend.left or extend.right, the horizontal borders are extended indefinitely to the left or right of the box, respectively. With extend.both, the horizontal borders are extended on both sides. Optional. The default value is extend.none.

xloc (series string) Determines whether the arguments to 'left' and 'right' are a bar index or a time value. If xloc = xloc.bar_index, the arguments must be a bar index. If xloc = xloc.bar_time, the arguments must be a UNIX time. Possible values: xloc.bar_index and xloc.bar_time. Optional. The default is xloc.bar_index.

bgcolor (series color) Background color of the box. Optional. The default is color.blue.

text (series string) The text to be displayed inside the box. Optional. The default is empty string.

text_size (series string) The size of the text. An optional parameter, the default value is size.auto. Possible values: size.auto, size.tiny, size.small, size.normal, size.large, size.huge.

text_color (series color) The color of the text. Optional. The default is color.black.

text_halign (series string) The horizontal alignment of the box's text. Optional. The default value is text.align_center. Possible values: text.align_left, text.align_center, text.align_right.

text_valign (series string) The vertical alignment of the box's text. Optional. The default value is text.align_center. Possible values: text.align_top, text.align_center, text.align_bottom.

text_wrap (series string) Defines whether the text is presented in a single line, extending past the width of the box if necessary, or wrapped so every line is no wider than the box itself (and clipped by the bottom border of the box if the height of the resulting wrapped text is higher than the height of the box). Optional. The default value is text.wrap_none. Possible values: text.wrap_none, text.wrap_auto.

text_font_family (series string) The font family of the text. Optional. The default value is font.family_default. Possible values: font.family_default, font.family_monospace.

force_overlay (const bool) If true, the drawing will display on the main chart pane, even when the script occupies a separate pane. Optional. The default is false.

EXAMPLE

```
//@version=5
indicator("box.new")
var b = box.new(time, open, time + 60 * 60 * 24, close, xloc=xloc.bar_time, border_style=l
box.set_lefttop(b, time, 100)
box.set_rightbottom(b, time + 60 * 60 * 24, 500)
box.set_bgcolor(b, color.green)
```

RETURNS

The ID of a box object which may be used in box.set_*() and box.get_*() functions.

SEE ALSO

box.delete box.get_left box.get_top box.get_right box.get_bottom

box.set_top_left_point box.set_left box.set_top box.set_bottom_right_point

box.set_right box.set_bottom box.set_border_color box.set_bgcolor

box.set_border_width box.set_border_style box.set_extend

box.set_bgcolor()

Sets the background color of the box.

SYNTAX

```
box.set_bgcolor(id, color) → void
```

ARGUMENTS

id (series box) A box object.

color (series color) New background color.

SEE ALSO

box.new

box.set_border_color()

Sets the border color of the box.

SYNTAX

```
box.set_border_color(id, color) → void
```

ARGUMENTS

id (series box) A box object.

color (series color) New border color.

SEE ALSO

box.new

box.set_border_style()

Sets the border style of the box.

SYNTAX

```
box.set_border_style(id, style) → void
```

ARGUMENTS

id (series box) A box object.

style (series string) New border style.

SEE ALSO

box.new   line.style_solid

box.set_border_width()

Sets the border width of the box.

SYNTAX

```
box.set_border_width(id, width) → void
```

ARGUMENTS

id (series box) A box object.

width (series int) Width of the four borders, in pixels.

SEE ALSO

box.new

box.set_bottom()

Sets the bottom coordinate of the box.

SYNTAX

```
box.set_bottom(id, bottom) → void
```

ARGUMENTS

id (series box) A box object.

bottom (series int/float) Price value of the bottom border.

SEE ALSO

box.new   box.get_bottom

box.set_bottom_right_point()

Sets the bottom-right corner location of the `id` box to `point`.

SYNTAX

```
box.set_bottom_right_point(id, point) → void
```

ARGUMENTS

id (series box) A box object.

point (chart.point) A chart.point object.

box.set_extend()

Sets extending type of the border of this box object. When extend.none is used, the horizontal borders start at the left border and end at the right border. With extend.left or extend.right, the horizontal borders are extended indefinitely to the left or right of the box, respectively. With extend.both, the horizontal borders are extended on both sides.

SYNTAX

```
box.set_extend(id, extend) → void
```

ARGUMENTS

id (series box) A box object.

extend (series string) New extending type.

SEE ALSO

box.new extend.none

box.set_left()

Sets the left coordinate of the box.

SYNTAX

```
box.set_left(id, left) → void
```

ARGUMENTS

id (series box) A box object.

left (series int) Bar index or bar time of the left border. Note that objects positioned using xloc.bar_index cannot be drawn further than 500 bars into the future.

SEE ALSO

box.new box.get_left

box.set_lefttop()

Sets the left and top coordinates of the box.

SYNTAX

```
box.set_lefttop(id, left, top) → void
```

ARGUMENTS

id (series box) A box object.

left (series int) Bar index or bar time of the left border.

top (series int/float) Price value of the top border.

SEE ALSO

box.new box.get_left box.get_top

box.set_right()

Sets the right coordinate of the box.

SYNTAX

```
box.set_right(id, right) → void
```

ARGUMENTS

id (series box) A box object.

right (series int) Bar index or bar time of the right border. Note that objects positioned using xloc.bar_index cannot be drawn further than 500 bars into the future.

SEE ALSO

box.new box.get_right

box.set_rightbottom()

Sets the right and bottom coordinates of the box.

SYNTAX

```
box.set_rightbottom(id, right, bottom) → void
```

ARGUMENTS

id (series box) A box object.

right (series int) Bar index or bar time of the right border.

bottom (series int/float) Price value of the bottom border.

SEE ALSO

box.new box.get_right box.get_bottom

box.set_text()

The function sets the text in the box.

SYNTAX

```
box.set_text(id, text) → void
```

ARGUMENTS

id (series box) A box object.

text (series string) The text to be displayed inside the box.

SEE ALSO

box.set_text_color box.set_text_size box.set_text_valign box.set_text_halign

box.set_text_color()

The function sets the color of the text inside the box.

SYNTAX

```
box.set_text_color(id, text_color) → void
```

ARGUMENTS

id (series box) A box object.

text_color (series color) The color of the text.

SEE ALSO

box.set_text box.set_text_size box.set_text_valign box.set_text_halign

box.set_text_font_family()

The function sets the font family of the text inside the box.

SYNTAX

```
box.set_text_font_family(id, text_font_family) → void
```

ARGUMENTS

id (series box) A box object.

text_font_family (series string) The font family of the text. Possible values: font.family_default, font.family_monospace.

EXAMPLE

```
//@version=5
indicator("Example of setting the box font")
if barstate.islastconfirmedhistory
    b = box.new(bar_index, open-ta.tr, bar_index-50, open-ta.tr*5, text="monospace")
    box.set_text_font_family(b, font.family_monospace)
```

SEE ALSO

box.new font.family_default font.family_monospace

box.set_text_halign()

The function sets the horizontal alignment of the box's text.

SYNTAX

```
box.set_text_halign(id, text_halign) → void
```

ARGUMENTS

id (series box) A box object.

text_halign (series string) The horizontal alignment of a box's text. Possible values: text.align_left, text.align_center, text.align_right.

SEE ALSO

box.set_text box.set_text_size box.set_text_valign box.set_text_color

box.set_text_size()

The function sets the size of the box's text.

SYNTAX

```
box.set_text_size(id, text_size) → void
```

ARGUMENTS

id (series box) A box object.

text_size (series string) The size of the text. Possible values: size.auto, size.tiny, size.small, size.normal, size.large, size.huge.

SEE ALSO

box.set_text box.set_text_color box.set_text_valign box.set_text_halign

box.set_text_valign()

The function sets the vertical alignment of a box's text.

SYNTAX

```
box.set_text_valign(id, text_valign) → void
```

ARGUMENTS

id (series box) A box object.

text_valign (series string) The vertical alignment of the box's text. Possible values: text.align_top, text.align_center, text.align_bottom.

SEE ALSO

box.set_text box.set_text_size box.set_text_color box.set_text_halign

box.set_text_wrap()

The function sets the mode of wrapping of the text inside the box.

SYNTAX

```
box.set_text_wrap(id, text_wrap) → void
```

ARGUMENTS

id (series box) A box object.

text_wrap (series string) The mode of the wrapping. Possible values: text.wrap_auto, text.wrap_none.

SEE ALSO

box.set_text box.set_text_size box.set_text_valign box.set_text_halign

box.set_text_color

box.set_top()

Sets the top coordinate of the box.

SYNTAX

```
box.set_top(id, top) → void
```

ARGUMENTS

id (series box) A box object.

top (series int/float) Price value of the top border.

SEE ALSO

box.new box.get_top

box.set_top_left_point()

Sets the top-left corner location of the `id` box to `point`.

SYNTAX

```
box.set_top_left_point(id, point) → void
```

ARGUMENTS

id (series box) A box object.

point (chart.point) A chart.point object.

chart.point.copy()

Creates a copy of a chart.point object with the specified `id`.

SYNTAX

```
chart.point.copy(id) → chart.point
```

ARGUMENTS

id (chart.point) A chart.point object.

chart.point.from_index()

Returns a chart.point object with `index` as its x-coordinate and `price` as its y-coordinate.

SYNTAX

```
chart.point.from_index(index, price) → chart.point
```

ARGUMENTS

index (series int) The x-coordinate of the point, expressed as a bar index value.

price (series int/float) The y-coordinate of the point.

REMARKS

The `time` field values of chart.point instances returned from this function will be na, meaning drawing objects with `xloc` values set to `xloc.bar_time` will not work with them.

chart.point.from_time()

Returns a chart.point object with `time` as its x-coordinate and `price` as its y-coordinate.

SYNTAX

```
chart.point.from_time(time, price) → chart.point
```

ARGUMENTS

time (series int) The x-coordinate of the point, expressed as a UNIX time value, in milliseconds.

price (series int/float) The y-coordinate of the point.

REMARKS

The `index` field values of chart.point instances returned from this function will be na, meaning drawing objects with `xloc` values set to `xloc.bar_index` will not work with them.

chart.point.new()

Creates a new chart.point object with the specified `time`, `index`, and `price`.

SYNTAX

```
chart.point.new(time, index, price) → chart.point
```

ARGUMENTS

time (series int) The x-coordinate of the point, expressed as a UNIX time value, in milliseconds.

index (series int) The x-coordinate of the point, expressed as a bar index value.

price (series int/float) The y-coordinate of the point.

REMARKS

Whether a drawing object uses a point's `time` or `index` field as an x-coordinate depends on the `xloc` type used in the function call that returned the drawing.

It's important to note that this function does not verify that the `time` and `index` values refer to the same bar.

SEE ALSO

polyline.new

chart.point.now()

Returns a chart.point object with `price` as the y-coordinate

SYNTAX

```
chart.point.now(price) → chart.point
```

ARGUMENTS

price (series int/float) The y-coordinate of the point. Optional. The default is close.

REMARKS

The chart.point instance returned from this function records values for its `index` and `time` fields on the bar it executed on, making it suitable for use with drawing objects of any `xloc` type.

color() 4 overloads

Casts na to color

SYNTAX & OVERLOADS

```
color(x) → const color
```

```
color(x) → input color
```

```
color(x) → simple color
```

```
color(x) → series color
```

ARGUMENTS

x (const color) The value to convert to the specified type, usually na.

RETURNS

The value of the argument after casting to color.

SEE ALSO

float int bool string line label

color.b() 4 overloads

Retrieves the value of the color's blue component.

SYNTAX & OVERLOADS

```
color.b(color) → const float
```

```
color.b(color) → input float
```

```
color.b(color) → simple float
```

```
color.b(color) → series float
```

ARGUMENTS

color (const color) Color.

EXAMPLE

```
//@version=5
indicator("color.b", overlay=true)
plot(color.b(color.blue))
```

RETURNS

The value (0 to 255) of the color's blue component.

color.from_gradient()

Based on the relative position of value in the bottom_value to top_value range, the function returns a color from the gradient defined by bottom_color to top_color.

SYNTAX

```
color.from_gradient(value, bottom_value, top_value, bottom_color, top_color) → series
color
```

ARGUMENTS

value (series int/float) Value to calculate the position-dependent color.

bottom_value (series int/float) Bottom position value corresponding to bottom_color.

top_value (series int/float) Top position value corresponding to top_color.

bottom_color (series color) Bottom position color.

top_color (series color) Top position color.

EXAMPLE

```
//@version=5
indicator("color.from_gradient", overlay=true)
color1 = color.from_gradient(close, low, high, color.yellow, color.lime)
color2 = color.from_gradient(ta.rsi(close, 7), 0, 100, color.rgb(255, 0, 0), color.rgb(0,
plot(close, color=color1)
plot(ta.rsi(close,7), color=color2)
```

RETURNS

A color calculated from the linear gradient between bottom_color to top_color.

REMARKS

Using this function will have an impact on the colors displayed in the script's "Settings/Style" tab. See the User Manual for more information.

color.g() 4 overloads

Retrieves the value of the color's green component.

SYNTAX & OVERLOADS

```
color.g(color) → const float
```

```
color.g(color) → input float
```

```
color.g(color) → simple float
```

```
color.g(color) → series float
```

ARGUMENTS

color (const color) Color.

EXAMPLE

```
//@version=5
indicator("color.g", overlay=true)
plot(color.g(color.green))
```

RETURNS

The value (0 to 255) of the color's green component.

color.new() 4 overloads

Function color applies the specified transparency to the given color.

SYNTAX & OVERLOADS

```
color.new(color, transp) → const color
```

```
color.new(color, transp) → input color
```

```
color.new(color, transp) → simple color
```

```
color.new(color, transp) → series color
```

ARGUMENTS

color (const color) Color to apply transparency to.

transp (const int/float) Possible values are from 0 (not transparent) to 100 (invisible).

EXAMPLE

```
//@version=5
indicator("color.new", overlay=true)
plot(close, color=color.new(color.red, 50))
```

RETURNS

Color with specified transparency.

REMARKS

Using arguments that are not constants (e.g., 'simple', 'input' or 'series') will have an impact on the colors displayed in the script's "Settings/Style" tab. See the User Manual for more information.

color.r() 4 overloads

Retrieves the value of the color's red component.

SYNTAX & OVERLOADS

```
color.r(color) → const float
```

```
color.r(color) → input float
```

```
color.r(color) → simple float
```

```
color.r(color) → series float
```

ARGUMENTS

color (const color) Color.

EXAMPLE

```
//@version=5
indicator("color.r", overlay=true)
plot(color.r(color.red))
```

RETURNS

The value (0 to 255) of the color's red component.

color.rgb() 4 overloads

Creates a new color with transparency using the RGB color model.

SYNTAX & OVERLOADS

```
color.rgb(red, green, blue, transp) → const color
```

```
color.rgb(red, green, blue, transp) → input color
```

```
color.rgb(red, green, blue, transp) → simple color
color.rgb(red, green, blue, transp) → series color
```

ARGUMENTS

red (const int/float) Red color component. Possible values are from 0 to 255.

green (const int/float) Green color component. Possible values are from 0 to 255.

blue (const int/float) Blue color component. Possible values are from 0 to 255.

transp (const int/float) Optional. Color transparency. Possible values are from 0 (opaque) to 100 (invisible). Default value is 0.

EXAMPLE

```
//@version=5
indicator("color.rgb", overlay=true)
plot(close, color=color.rgb(255, 0, 0, 50))
```

RETURNS

Color with specified transparency.

REMARKS

Using arguments that are not constants (e.g., 'simple', 'input' or 'series') will have an impact on the colors displayed in the script's "Settings/Style" tab. See the User Manual for more information.

color.t() 4 overloads

Retrieves the color's transparency.

SYNTAX & OVERLOADS

```
color.t(color) → const float
```

```
color.t(color) → input float
```

```
color.t(color) → simple float
```

```
color.t(color) → series float
```

ARGUMENTS

color (const color) Color.

EXAMPLE

```
//@version=5
indicator("color.t", overlay=true)
plot(color.t(color.new(color.red, 50)))
```

RETURNS

The value (0-100) of the color's transparency.

dayofmonth() 2 overloads

SYNTAX & OVERLOADS

```
dayofmonth(time) → series int
```

```
dayofmonth(time, timezone) → series int
```

ARGUMENTS

time (series int) UNIX time in milliseconds.

RETURNS

Day of month (in exchange timezone) for provided UNIX time.

REMARKS

UNIX time is the number of milliseconds that have elapsed since 00:00:00 UTC, 1 January 1970.

Note that this function returns the day based on the time of the bar's open. For overnight sessions (e.g. EURUSD, where Monday session starts on Sunday, 17:00 UTC-4) this value can be lower by 1 than the day of the trading day.

SEE ALSO

dayofmonth time year month dayofweek hour minute second

dayofweek() 2 overloads

SYNTAX & OVERLOADS

```
dayofweek(time) → series int
```

```
dayofweek(time, timezone) → series int
```

ARGUMENTS

time (series int) UNIX time in milliseconds.

RETURNS

Day of week (in exchange timezone) for provided UNIX time.

REMARKS

Note that this function returns the day based on the time of the bar's open. For overnight sessions (e.g. EURUSD, where Monday session starts on Sunday, 17:00) this value can be lower by 1 than the day of the trading day.

UNIX time is the number of milliseconds that have elapsed since 00:00:00 UTC, 1 January 1970.

SEE ALSO

dayofweek time year month dayofmonth hour minute second

fill() 3 overloads

Fills background between two plots or hlines with a given color.

SYNTAX & OVERLOADS

```
fill(hline1, hline2, color, title, editable, fillgaps, display) → void
```

```
fill(plot1, plot2, color, title, editable, show_last, fillgaps, display) → void
```

```
fill(plot1, plot2, top_value, bottom_value, top_color, bottom_color, title, display,
fillgaps, editable) → void
```

ARGUMENTS

hline1 (hline) The first hline object. Required argument.

hline2 (hline) The second hline object. Required argument.

color (series color) Color of the background fill. You can use constants like 'color=color.red' or 'color=#ff001a' as well as complex expressions like 'color = close >= open ? color.green : color.red'. Optional argument.

title (const string) Title of the created fill object. Optional argument.

editable (const bool) If true then fill style will be editable in Format dialog. Default is true.

fillgaps (const bool) Controls continuing fills on gaps, i.e., when one of the plot() calls returns an na value. When true, the last fill will continue on gaps. The default is false.

display (input plot_simple_display) Controls where the fill is displayed. Possible values are: display.none, display.all. Default is display.all.

Fill between two horizontal lines

EXAMPLE

```
//@version=5
indicator("Fill between hlines", overlay = false)
h1 = hline(20)
h2 = hline(10)
fill(h1, h2, color = color.new(color.blue, 90))
```

Fill between two plots

EXAMPLE

```
//@version=5
indicator("Fill between plots", overlay = true)
p1 = plot(open)
p2 = plot(close)
fill(p1, p2, color = color.new(color.green, 90))
```

Gradient fill between two horizontal lines

EXAMPLE

```
//@version=5
indicator("Gradient Fill between hlines", overlay = false)
topVal = input.int(100)
botVal = input.int(0)
topCol = input.color(color.red)
botCol = input.color(color.blue)
topLine = hline(100, color = topCol, linestyle = hline.style_solid)
botLine = hline(0,   color = botCol, linestyle = hline.style_solid)
```

```
fill(topLine, botLine, topVal, botVal, topCol, botCol)
```

SEE ALSO

plot barcolor bgcolor hline color.new

fixnan() 4 overloads

For a given series replaces NaN values with previous nearest non-NaN value.

SYNTAX & OVERLOADS

```
fixnan(source) → series color
```

```
fixnan(source) → series int
```

```
fixnan(source) → series float
```

```
fixnan(source) → series bool
```

ARGUMENTS

source (series color) Source used for the calculation.

RETURNS

Series without na gaps.

SEE ALSO

na na nz

float() 4 overloads

Casts na to float

SYNTAX & OVERLOADS

```
float(x) → const float
```

```
float(x) → input float
float(x) → simple float
```

```
float(x) → series float
```

ARGUMENTS

x (const int/float) The value to convert to the specified type, usually na.

RETURNS

The value of the argument after casting to float.

SEE ALSO

int bool color string line label

hline()

Renders a horizontal line at a given fixed price level.

SYNTAX

```
hline(price, title, color, linestyle, linewidth, editable, display) → hline
```

ARGUMENTS

price (input int/float) Price value at which the object will be rendered. Required argument.

title (const string) Title of the object.

color (input color) Color of the rendered line. Must be a constant value (not an expression). Optional argument.

linestyle (input hline_style) Style of the rendered line. Possible values are: hline.style_solid, hline.style_dotted, hline.style_dashed. Optional argument.

linewidth (input int) Width of the rendered line. Default value is 1.

editable (const bool) If true then hline style will be editable in Format dialog. Default is true.

display (input plot_simple_display) Controls where the hline is displayed. Possible values are: display.none, display.all. Default is display.all.

EXAMPLE

```
//@version=5
indicator("input.hline", overlay=true)
hline(3.14, title='Pi', color=color.blue, linestyle=hline.style_dotted, linewidth=2)

// You may fill the background between any two hlines with a fill() function:
h1 = hline(20)
h2 = hline(10)
fill(h1, h2, color=color.new(color.green, 90))
```

RETURNS

An hline object, that can be used in fill

SEE ALSO

fill

hour() 2 overloads

SYNTAX & OVERLOADS

```
hour(time) → series int
```

```
hour(time, timezone) → series int
```

ARGUMENTS

time (series int) UNIX time in milliseconds.

RETURNS

Hour (in exchange timezone) for provided UNIX time.

REMARKS

UNIX time is the number of milliseconds that have elapsed since 00:00:00 UTC, 1 January 1970.

SEE ALSO

hour time year month dayofmonth dayofweek minute second

indicator()

This declaration statement designates the script as an indicator and sets a number of indicator-related properties.

SYNTAX

```
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,
calc_bars_count, max_polylines_count, dynamic_requests, behind_chart) → void
```

ARGUMENTS

title (const string) The title of the script. It is displayed on the chart when no `shorttitle` argument is used, and becomes the publication's default title when publishing the script.

shorttitle (const string) The script's display name on charts. If specified, it will replace the `title` argument in most chart-related windows. Optional. The default is the argument used for `title`.

overlay (const bool) If true, the indicator will be displayed over the chart. If false, it will be added in a separate pane. Optional. The default is false.

format (const string) Specifies the formatting of the script's displayed values. Possible values: format.inherit, format.price, format.volume, format.percent. Optional. The default is format.inherit.

precision (const int) Specifies the number of digits after the floating point of the script's displayed values. Must be a non-negative integer no greater than 16. If `format` is set to format.inherit and `precision` is specified, the format will instead be set to format.price. When the function's `format` parameter uses format.volume, the `precision` parameter will not affect the result, as the decimal precision rules defined by format.volume supersede other precision settings. Optional. The default is inherited from the precision of the chart's symbol.

scale (const scale_type) The price scale used. Possible values: scale.right, scale.left, scale.none. The scale.none value can only be applied in combination with `overlay = true`. Optional. By default, the script uses the same scale as the chart.

max_bars_back (const int) The length of the historical buffer the script keeps for every variable and function, which determines how many past values can be referenced using the `[]` history-referencing operator. The required buffer size is automatically detected by the Pine Script® runtime. Using this parameter is only necessary when a runtime error occurs because automatic detection fails. More information on the underlying mechanics of the historical buffer can be found in our Help Center. Optional. The default is 0.

timeframe (const string) Adds multi-timeframe functionality to simple scripts. When specified, a "Timeframe" field will be included in the "Calculation" section of the script's "Settings/Inputs" tab. The field's default value will be the argument supplied, whose format must conform to timeframe string specifications. To specify the chart's timeframe, use an empty string or the timeframe.period variable. The parameter cannot be used with scripts using Pine Script® drawings. Optional. The default is timeframe.period.

timeframe_gaps (const bool) Specifies how the indicator's values are displayed on chart bars when the `timeframe` is higher than the chart's. If true, a value only appears on a chart bar when the higher `timeframe` value becomes available, otherwise na is returned (thus a "gap" occurs). With false, what would otherwise be gaps are filled with the latest known value returned, avoiding na values. When specified, a "Wait for timeframe closes" checkbox will be included in the "Calculation" section of the script's "Settings/Inputs" tab. Optional. The default is true.

explicit_plot_zorder (const bool) Specifies the order in which the script's plots, fills, and hlines are rendered. If true, plots are drawn in the order in which they appear in the script's code, each newer plot being drawn above the previous ones. This only applies to `plot*()` functions, fill, and hline. Optional. The default is false.

max_lines_count (const int) The number of last line drawings displayed. Possible values: 1-500. The count is approximate; more drawings than the specified count may be displayed. Optional. The default is 50.

max_labels_count (const int) The number of last label drawings displayed. Possible values: 1-500. The count is approximate; more drawings than the specified count may be displayed. Optional. The default is 50.

max_boxes_count (const int) The number of last box drawings displayed. Possible values: 1-500. The count is approximate; more drawings than the specified count may be displayed. Optional. The default is 50.

calc_bars_count (const int) Limits the initial calculation of a script to the last number of bars specified. When specified, a "Calculated bars" field will be included in the "Calculation" section of the script's "Settings/Inputs" tab. Optional. The default is 0, in which case the script executes on all available bars.

max_polylines_count (const int) The number of last polyline drawings displayed. Possible values: 1-100. The count is approximate; more drawings than the specified count may be displayed. Optional. The default is 50.

dynamic_requests (const bool) Specifies whether the script can dynamically call functions from the `request.*()` namespace. Dynamic `request.*()` calls are allowed within the local scopes of conditional structures (e.g., if), loops (e.g., for), and exported functions. Additionally, such calls allow "series" arguments for many of their parameters. Optional. The default is false. See the User Manual's Dynamic requests section for more information.

behind_chart (const bool) Controls whether the script's plots and drawings in the main chart pane appear behind the chart display (if true), or in front of it (if false). Optional. The default is true.

EXAMPLE

```
//@version=5
indicator("My script", shorttitle="Script")
plot(close)
```

REMARKS

Every indicator script must have one indicator call.

SEE ALSO

strategy library

input() 6 overloads

Adds an input to the Inputs tab of your script's Settings, which allows you to provide configuration options to script users. This function automatically detects the type of the argument used for 'defval' and uses the corresponding input widget.

SYNTAX & OVERLOADS

```
input(defval, title, tooltip, inline, group, display) → input color
```

```
input(defval, title, tooltip, inline, group, display) → input string
```

```
input(defval, title, tooltip, inline, group, display) → input int
```

```
input(defval, title, tooltip, inline, group, display) → input float
```

```
input(defval, title, inline, group, tooltip, display) → series float
```

```
input(defval, title, tooltip, inline, group, display) → input bool
```

ARGUMENTS

defval (const int/float/bool/string/color or source-type built-ins) Determines the default value of the input variable proposed in the script's "Settings/Inputs" tab, from where script users can change it. Source-type built-ins are built-in series float variables that specify the source of the calculation: `close`, `hlc3`, etc.

title (const string) Title of the input. If not specified, the variable name is used as the input's title. If the title is specified, but it is empty, the name will be an empty string.

tooltip (const string) The string that will be shown to the user when hovering over the tooltip icon.

inline (const string) Combines all the input calls using the same argument in one line. The string used as an argument is not displayed. It is only used to identify inputs belonging to the same line.

group (const string) Creates a header above all inputs using the same group argument string. The string is also used as the header's text.

display (const plot_display) Controls where the script will display the input's information, aside from within the script's settings. This option allows one to remove a specific input from the script's status line or the Data Window to ensure only the most necessary inputs are displayed there. Possible values: display.none, display.data_window, display.status_line, display.all. Optional. The default depends on the type of the value passed to `defval`: display.none for bool and color values, display.all for everything else.

EXAMPLE

```
//@version=5
indicator("input", overlay=true)
i_switch = input(true, "On/Off")
plot(i_switch ? open : na)

i_len = input(7, "Length")
i_src = input(close, "Source")
plot(ta.sma(i_src, i_len))

i_border = input(142.50, "Price Border")
hline(i_border)
bgcolor(close > i_border ? color.green : color.red)

i_col = input(color.red, "Plot Color")
plot(close, color=i_col)

i_text = input("Hello!", "Message")
l = label.new(bar_index, high, text=i_text)
label.delete(l[1])
```

RETURNS

Value of input variable.

REMARKS

Result of input function always should be assigned to a variable, see examples above.

SEE ALSO

input.bool input.color input.int input.float input.string input.symbol input.timeframe input.text_area input.session input.source input.time

input.bool()

Adds an input to the Inputs tab of your script's Settings, which allows you to provide configuration options to script users. This function adds a checkmark to the script's inputs.

SYNTAX

```
input.bool(defval, title, tooltip, inline, group, confirm, display) → input bool
```

ARGUMENTS

defval (const bool) Determines the default value of the input variable proposed in the script's "Settings/Inputs" tab, from where the user can change it.

title (const string) Title of the input. If not specified, the variable name is used as the input's title. If the title is specified, but it is empty, the name will be an empty string.

tooltip (const string) The string that will be shown to the user when hovering over the tooltip icon.

inline (const string) Combines all the input calls using the same argument in one line. The string used as an argument is not displayed. It is only used to identify inputs belonging to the same line.

group (const string) Creates a header above all inputs using the same group argument string. The string is also used as the header's text.

confirm (const bool) If true, then user will be asked to confirm input value before indicator is added to chart. Default value is false.

display (const plot_display) Controls where the script will display the input's information, aside from within the script's settings. This option allows one to remove a specific input from the script's status line or the Data Window to ensure only the most necessary inputs are displayed there. Possible values: display.none, display.data_window, display.status_line, display.all. Optional. The default is display.none.

EXAMPLE

```
//@version=5
indicator("input.bool", overlay=true)
i_switch = input.bool(true, "On/Off")
plot(i_switch ? open : na)
```

RETURNS

Value of input variable.

REMARKS

Result of input.bool function always should be assigned to a variable, see examples above.

SEE ALSO

input.int input.float input.string input.text_area input.symbol input.timeframe input.session input.source input.color input.time input

input.color()

Adds an input to the Inputs tab of your script's Settings, which allows you to provide configuration options to script users. This function adds a color picker that allows the user to select a color and transparency, either from a palette or a hex value.

SYNTAX

```
input.color(defval, title, tooltip, inline, group, confirm, display) → input color
```

ARGUMENTS

defval (const color) Determines the default value of the input variable proposed in the script's "Settings/Inputs" tab, from where the user can change it.

title (const string) Title of the input. If not specified, the variable name is used as the input's title. If the title is specified, but it is empty, the name will be an empty string.

tooltip (const string) The string that will be shown to the user when hovering over the tooltip icon.

inline (const string) Combines all the input calls using the same argument in one line. The string used as an argument is not displayed. It is only used to identify inputs belonging to the same line.

group (const string) Creates a header above all inputs using the same group argument string. The string is also used as the header's text.

confirm (const bool) If true, then user will be asked to confirm input value before indicator is added to chart. Default value is false.

display (const plot_display) Controls where the script will display the input's information, aside from within the script's settings. This option allows one to remove a specific input from the script's status line or the Data Window to ensure only the most necessary inputs are displayed there. Possible values: display.none, display.data_window, display.status_line, display.all. Optional. The default is display.none.

EXAMPLE

```
//@version=5
indicator("input.color", overlay=true)
i_col = input.color(color.red, "Plot Color")
plot(close, color=i_col)
```

RETURNS

Value of input variable.

REMARKS

Result of input.color function always should be assigned to a variable, see examples above.

SEE ALSO

input.bool input.int input.float input.string input.text_area input.symbol input.timeframe input.session input.source input.time input

input.enum()

Adds an input to the Inputs tab of your script's Settings, which allows you to provide configuration options to script users. This function adds a dropdown with options based on the enum fields passed to its `defval` and `options` parameters.

The text for each option in the resulting dropdown corresponds to the titles of the included fields. If a field's title is not specified in the enum declaration, its title is the string representation of its name.

SYNTAX

```
input.enum(defval, title, options, tooltip, inline, group, confirm, display) → input enum
```

ARGUMENTS

defval (const enum) Determines the default value of the input, which users can change in the script's "Settings/Inputs" tab. When the `options` parameter has a specified tuple of enum fields, the tuple must include the `defval`.

title (const string) Title of the input. If not specified, the variable name is used as the input's title. If the title is specified, but it is empty, the name will be an empty string.

options (tuple of enum fields: [enumName.field1, enumName.field2, ...]) A list of options to choose from. Optional. By default, the titles of all of the enum's fields are available in the dropdown. Passing a tuple as the `options` argument limits the list to only the included fields.

tooltip (const string) The string that will be shown to the user when hovering over the tooltip icon.

inline (const string) Combines all the input calls using the same argument in one line. The string used as an argument is not displayed. It is only used to identify inputs belonging to the same line.

group (const string) Creates a header above all inputs using the same group argument string. The string is also used as the header's text.

confirm (const bool) If `true`, then user will be asked to confirm input value before indicator is added to chart. Default value is `false`.

display (const plot_display) Controls where the script will display the input's information, aside from within the script's settings. This option allows one to remove a specific input from the script's status line or the Data Window to ensure only the most necessary inputs are displayed there. Possible values: display.none, display.data_window, display.status_line, display.all. Optional. The default is display.all.

EXAMPLE

```
//@version=5
indicator("Session highlight", overlay = true)

//@enum        Contains fields with popular timezones as titles.
//@field exch  Has an empty string as the title to represent the chart timezone.
enum tz
    utc  = "UTC"
    exch = ""
    ny   = "America/New_York"
    chi  = "America/Chicago"
    lon  = "Europe/London"
    tok  = "Asia/Tokyo"

//@variable The session string.
selectedSession = input.session("1200-1500", "Session")
//@variable The selected timezone. The input's dropdown contains the fields in the `tz` er
selectedTimezone = input.enum(tz.utc, "Session Timezone")

//@variable Is `true` if the current bar's time is in the specified session.
bool inSession = false
if not na(time("", selectedSession, str.tostring(selectedTimezone)))
    inSession := true

// Highlight the background when `inSession` is `true`.
bgcolor(inSession ? color.new(color.green, 90) : na, title = "Active session highlight")
```

RETURNS

Value of input variable.

REMARKS

All fields included in the `defval` and `options` arguments must belong to the same enum.

SEE ALSO

input.text_area input.bool input.int input.float input.symbol input.timeframe

input.session input.source input.color input.time input

input.float() 2 overloads

Adds an input to the Inputs tab of your script's Settings, which allows you to provide configuration options to script users. This function adds a field for a float input to the script's inputs.

SYNTAX & OVERLOADS

```
input.float(defval, title, options, tooltip, inline, group, confirm, display) → input
float
```

```
input.float(defval, title, minval, maxval, step, tooltip, inline, group, confirm, display)
→ input float
```

ARGUMENTS

defval (const int/float) Determines the default value of the input variable proposed in the script's "Settings/Inputs" tab, from where script users can change it. When a list of values is used with the `options` parameter, the value must be one of them.

title (const string) Title of the input. If not specified, the variable name is used as the input's title. If the title is specified, but it is empty, the name will be an empty string.

options (tuple of const int/float values: [val1, val2, ...]) A list of options to choose from a dropdown menu, separated by commas and enclosed in square brackets: [val1, val2, ...]. When using this parameter, the `minval`, `maxval` and `step` parameters cannot be used.

tooltip (const string) The string that will be shown to the user when hovering over the tooltip icon.

inline (const string) Combines all the input calls using the same argument in one line. The string used as an argument is not displayed. It is only used to identify inputs belonging to the same line.

group (const string) Creates a header above all inputs using the same group argument string. The string is also used as the header's text.

confirm (const bool) If true, then user will be asked to confirm input value before indicator is added to chart. Default value is false.

display (const plot_display) Controls where the script will display the input's information, aside from within the script's settings. This option allows one to remove a specific input from the script's status line or the Data Window to ensure only the most necessary inputs are displayed there. Possible values: display.none, display.data_window, display.status_line, display.all. Optional. The default is display.all.

EXAMPLE

```
//@version=5
indicator("input.float", overlay=true)
i_angle1 = input.float(0.5, "Sin Angle", minval=-3.14, maxval=3.14, step=0.02)
plot(math.sin(i_angle1) > 0 ? close : open, "sin", color=color.green)

i_angle2 = input.float(0, "Cos Angle", options=[-3.14, -1.57, 0, 1.57, 3.14])
plot(math.cos(i_angle2) > 0 ? close : open, "cos", color=color.red)
```

RETURNS

Value of input variable.

REMARKS

Result of input.float function always should be assigned to a variable, see examples above.

SEE ALSO

input.bool input.int input.string input.text_area input.symbol input.timeframe input.session input.source input.color input.time input

input.int() 2 overloads

Adds an input to the Inputs tab of your script's Settings, which allows you to provide configuration options to script users. This function adds a field for an integer input to the script's inputs.

SYNTAX & OVERLOADS

```
input.int(defval, title, options, tooltip, inline, group, confirm, display) → input int
input.int(defval, title, minval, maxval, step, tooltip, inline, group, confirm, display) →
input int
```

ARGUMENTS

defval (const int) Determines the default value of the input variable proposed in the script's "Settings/Inputs" tab, from where script users can change it. When a list of values is used with the `options` parameter, the value must be one of them.

title (const string) Title of the input. If not specified, the variable name is used as the input's title. If the title is specified, but it is empty, the name will be an empty string.

options (tuple of const int values: [val1, val2, ...]) A list of options to choose from a dropdown menu, separated by commas and enclosed in square brackets: [val1, val2, ...]. When using this parameter, the `minval`, `maxval` and `step` parameters cannot be used.

tooltip (const string) The string that will be shown to the user when hovering over the tooltip icon.

inline (const string) Combines all the input calls using the same argument in one line. The string used as an argument is not displayed. It is only used to identify inputs belonging to the same line.

group (const string) Creates a header above all inputs using the same group argument string. The string is also used as the header's text.

confirm (const bool) If true, then user will be asked to confirm input value before indicator is added to chart. Default value is false.

display (const plot_display) Controls where the script will display the input's information, aside from within the script's settings. This option allows one to remove a specific input from the script's status line or the Data Window to ensure only the most necessary inputs are displayed there. Possible values: display.none, display.data_window, display.status_line, display.all. Optional. The default is display.all.

EXAMPLE

```
//@version=5
indicator("input.int", overlay=true)
i_len1 = input.int(10, "Length 1", minval=5, maxval=21, step=1)
plot(ta.sma(close, i_len1))

i_len2 = input.int(10, "Length 2", options=[5, 10, 21])
plot(ta.sma(close, i_len2))
```

RETURNS

Value of input variable.

REMARKS

Result of input.int function always should be assigned to a variable, see examples above.

SEE ALSO

input.bool input.float input.string input.text_area input.symbol input.timeframe input.session input.source input.color input.time input

input.price()

Adds a price input to the script's "Settings/Inputs" tab. Using `confirm = true` activates the interactive input mode where a price is selected by clicking on the chart.

SYNTAX

```
input.price(defval, title, tooltip, inline, group, confirm, display) → input float
```

ARGUMENTS

defval (const int/float) Determines the default value of the input variable proposed in the script's "Settings/Inputs" tab, from where the user can change it.

title (const string) Title of the input. If not specified, the variable name is used as the input's title. If the title is specified, but it is empty, the name will be an empty string.

tooltip (const string) The string that will be shown to the user when hovering over the tooltip icon.

inline (const string) Combines all the input calls using the same argument in one line. The string used as an argument is not displayed. It is only used to identify inputs belonging to the same line.

group (const string) Creates a header above all inputs using the same group argument string. The string is also used as the header's text.

confirm (const bool) If true, the interactive input mode is enabled and the selection is done by clicking on the chart when the indicator is added to the chart, or by selecting the indicator and moving the selection after that. Optional. The default is false.

display (const plot_display) Controls where the script will display the input's information, aside from within the script's settings. This option allows one to remove a specific input from the script's status line or the Data Window to ensure only the most necessary inputs are displayed there. Possible values: display.none, display.data_window, display.status_line, display.all. Optional. The default is display.all.

EXAMPLE

```
//@version=5
indicator("input.price", overlay=true)
price1 = input.price(title="Date", defval=42)
plot(price1)

price2 = input.price(54, title="Date")
plot(price2)
```

RETURNS

Value of input variable.

REMARKS

When using interactive mode, a time input can be combined with a price input if both function calls use the same argument for their `inline` parameter.

SEE ALSO

input.bool input.int input.float input.string input.text_area input.symbol input.resolution input.session input.source input.color input

input.session()

Adds an input to the Inputs tab of your script's Settings, which allows you to provide configuration options to script users. This function adds two dropdowns that allow the user to specify the beginning and the end of a session using the session selector and returns the result as a string.

SYNTAX

```
input.session(defval, title, options, tooltip, inline, group, confirm, display) → input
string
```

ARGUMENTS

defval (const string) Determines the default value of the input variable proposed in the script's "Settings/Inputs" tab, from where the user can change it. When a list of values is used with the `options` parameter, the value must be one of them.

title (const string) Title of the input. If not specified, the variable name is used as the input's title. If the title is specified, but it is empty, the name will be an empty string.

options (tuple of const string values: [val1, val2, ...]) A list of options to choose from.

tooltip (const string) The string that will be shown to the user when hovering over the tooltip icon.

inline (const string) Combines all the input calls using the same argument in one line. The string used as an argument is not displayed. It is only used to identify inputs belonging to the same line.

group (const string) Creates a header above all inputs using the same group argument string. The string is also used as the header's text.

confirm (const bool) If true, then user will be asked to confirm input value before indicator is added to chart. Default value is false.

display (const plot_display) Controls where the script will display the input's information, aside from within the script's settings. This option allows one to remove a specific input from the script's status line or the Data Window to ensure only the most necessary inputs are displayed there. Possible values: display.none, display.data_window, display.status_line, display.all. Optional. The default is display.all.

EXAMPLE

```
//@version=5
indicator("input.session", overlay=true)
i_sess = input.session("1300-1700", "Session", options=["0930-1600", "1300-1700", "1700-21
t = time(timeframe.period, i_sess)
bgcolor(time == t ? color.green : na)
```

RETURNS

Value of input variable.

REMARKS

Result of input.session function always should be assigned to a variable, see examples above.

SEE ALSO

input.bool input.int input.float input.string input.text_area input.symbol

input.timeframe input.source input.color input.time input

input.source()

Adds an input to the Inputs tab of your script's Settings, which allows you to provide configuration options to script users. This function adds a dropdown that allows the user to select a source for the calculation, e.g. close, hl2, etc. The user can also select an output from another indicator on their chart as the source.

SYNTAX

```
input.source(defval, title, tooltip, inline, group, display, confirm) → series float
```

ARGUMENTS

defval (open/high/low/close/hl2/hlc3/ohlc4/hlcc4) Determines the default value of the input variable proposed in the script's "Settings/Inputs" tab, from where the user can change it.

title (const string) Title of the input. If not specified, the variable name is used as the input's title. If the title is specified, but it is empty, the name will be an empty string.

tooltip (const string) The string that will be shown to the user when hovering over the tooltip icon.

inline (const string) Combines all the input calls using the same argument in one line. The string used as an argument is not displayed. It is only used to identify inputs belonging to the same line.

group (const string) Creates a header above all inputs using the same group argument string. The string is also used as the header's text.

display (const plot_display) Controls where the script will display the input's information, aside from within the script's settings. This option allows one to remove a specific input from the script's status line or the Data Window to ensure only the most necessary inputs are displayed there. Possible values: display.none, display.data_window, display.status_line, display.all. Optional. The default is display.all.

confirm (const bool) If true, then user will be asked to confirm input value before indicator is added to chart. Default value is false.

EXAMPLE

```
//@version=5
indicator("input.source", overlay=true)
i_src = input.source(close, "Source")
plot(i_src)
```

RETURNS

Value of input variable.

REMARKS

Result of input.source function always should be assigned to a variable, see examples above.

SEE ALSO

input.bool input.int input.float input.string input.text_area input.symbol input.timeframe input.session input.color input.time input

input.string()

Adds an input to the Inputs tab of your script's Settings, which allows you to provide configuration options to script users. This function adds a field for a string input to the script's inputs.

SYNTAX

```
input.string(defval, title, options, tooltip, inline, group, confirm, display) → input
string
```

ARGUMENTS

defval (const string) Determines the default value of the input variable proposed in the script's "Settings/Inputs" tab, from where the user can change it. When a list of values is used with the `options` parameter, the value must be one of them.

title (const string) Title of the input. If not specified, the variable name is used as the input's title. If the title is specified, but it is empty, the name will be an empty string.

options (tuple of const string values: [val1, val2, ...]) A list of options to choose from.

tooltip (const string) The string that will be shown to the user when hovering over the tooltip icon.

inline (const string) Combines all the input calls using the same argument in one line. The string used as an argument is not displayed. It is only used to identify inputs belonging to the same line.

group (const string) Creates a header above all inputs using the same group argument string. The string is also used as the header's text.

confirm (const bool) If true, then user will be asked to confirm input value before indicator is added to chart. Default value is false.

display (const plot_display) Controls where the script will display the input's information, aside from within the script's settings. This option allows one to remove a specific input from the script's status line or the Data Window to ensure only the most necessary inputs are displayed there. Possible values: display.none, display.data_window, display.status_line, display.all. Optional. The default is display.all.

EXAMPLE

```
//@version=5
indicator("input.string", overlay=true)
i_text = input.string("Hello!", "Message")
l = label.new(bar_index, high, i_text)
label.delete(l[1])
```

RETURNS

Value of input variable.

REMARKS

Result of input.string function always should be assigned to a variable, see examples above.

SEE ALSO

input.text_area input.bool input.int input.float input.symbol input.timeframe input.session input.source input.color input.time input

input.symbol()

Adds an input to the Inputs tab of your script's Settings, which allows you to provide configuration options to script users. This function adds a field that allows the user to select a specific symbol using the symbol search and returns that symbol, paired with its exchange prefix, as a string.

SYNTAX

```
input.symbol(defval, title, tooltip, inline, group, confirm, display) → input string
```

ARGUMENTS

defval (const string) Determines the default value of the input variable proposed in the script's "Settings/Inputs" tab, from where the user can change it.

title (const string) Title of the input. If not specified, the variable name is used as the input's title. If the title is specified, but it is empty, the name will be an empty string.

tooltip (const string) The string that will be shown to the user when hovering over the tooltip icon.

inline (const string) Combines all the input calls using the same argument in one line. The string used as an argument is not displayed. It is only used to identify inputs belonging to the same line.

group (const string) Creates a header above all inputs using the same group argument string. The string is also used as the header's text.

confirm (const bool) If true, then user will be asked to confirm input value before indicator is added to chart. Default value is false.

display (const plot_display) Controls where the script will display the input's information, aside from within the script's settings. This option allows one to remove a specific input from the script's status line or the Data Window to ensure only the most necessary inputs are displayed there. Possible values: display.none, display.data_window, display.status_line, display.all. Optional. The default is display.all.

EXAMPLE

```
//@version=5
indicator("input.symbol", overlay=true)
i_sym = input.symbol("DELL", "Symbol")
s = request.security(i_sym, 'D', close)
plot(s)
```

RETURNS

Value of input variable.

REMARKS

Result of input.symbol function always should be assigned to a variable, see examples above.

SEE ALSO

input.bool input.int input.float input.string input.text_area input.timeframe

input.session input.source input.color input.time input

input.text_area()

Adds an input to the Inputs tab of your script's Settings, which allows you to provide configuration options to script users. This function adds a field for a multiline text input.

SYNTAX

```
input.text_area(defval, title, tooltip, group, confirm, display) → input string
```

ARGUMENTS

defval (const string) Determines the default value of the input variable proposed in the script's "Settings/Inputs" tab, from where the user can change it.

title (const string) Title of the input. If not specified, the variable name is used as the input's title. If the title is specified, but it is empty, the name will be an empty string.

tooltip (const string) The string that will be shown to the user when hovering over the tooltip icon.

group (const string) Creates a header above all inputs using the same group argument string. The string is also used as the header's text.

confirm (const bool) If true, then user will be asked to confirm input value before indicator is added to chart. Default value is false.

display (const plot_display) Controls where the script will display the input's information, aside from within the script's settings. This option allows one to remove a specific input from the script's status line or the Data Window to ensure only the most necessary inputs are displayed there. Possible values: display.none, display.data_window, display.status_line, display.all. Optional. The default is display.none.

EXAMPLE

```
//@version=5
indicator("input.text_area")
i_text = input.text_area(defval = "Hello \nWorld!", title = "Message")
plot(close)
```

RETURNS

Value of input variable.

REMARKS

Result of input.text_area function always should be assigned to a variable, see examples above.

SEE ALSO

input.string input.bool input.int input.float input.symbol input.timeframe input.session input.source input.color input.time input

input.time()

Adds a time input to the script's "Settings/Inputs" tab. This function adds two input widgets on the same line: one for the date and one for the time. The function returns a date/time value in UNIX format. Using `confirm = true` activates the interactive input mode where a point in time is selected by clicking on the chart.

SYNTAX

```
input.time(defval, title, tooltip, inline, group, confirm, display) → input int
```

ARGUMENTS

defval (const int) Determines the default value of the input variable proposed in the script's "Settings/Inputs" tab, from where the user can change it. The value can be a timestamp function, but only if it uses a date argument in const string format.

title (const string) Title of the input. If not specified, the variable name is used as the input's title. If the title is specified, but it is empty, the name will be an empty string.

tooltip (const string) The string that will be shown to the user when hovering over the tooltip icon.

inline (const string) Combines all the input calls using the same argument in one line. The string used as an argument is not displayed. It is only used to identify inputs belonging to the same line.

group (const string) Creates a header above all inputs using the same group argument string. The string is also used as the header's text.

confirm (const bool) If true, the interactive input mode is enabled and the selection is done by clicking on the chart when the indicator is added to the chart, or by selecting the indicator and moving the selection after that. Optional. The default is false.

display (const plot_display) Controls where the script will display the input's information, aside from within the script's settings. This option allows one to remove a specific input from the script's status line or the Data Window to ensure only the most necessary inputs are displayed there. Possible values: display.none, display.data_window, display.status_line, display.all. Optional. The default is display.none.

EXAMPLE

```
//@version=5
indicator("input.time", overlay=true)
i_date = input.time(timestamp("20 Jul 2021 00:00 +0300"), "Date")
l = label.new(i_date, high, "Date", xloc=xloc.bar_time)
label.delete(l[1])
```

RETURNS

Value of input variable.

REMARKS

When using interactive mode, a price input can be combined with a time input if both function calls use the same argument for their `inline` parameter.

SEE ALSO

input.bool input.int input.float input.string input.text_area input.symbol input.timeframe input.session input.source input.color input

input.timeframe()

Adds an input to the Inputs tab of your script's Settings, which allows you to provide configuration options to script users. This function adds a dropdown that allows the user to select a specific timeframe via the timeframe selector and returns it as a string. The selector includes the custom timeframes a user may have added using the chart's Timeframe dropdown.

SYNTAX

```
input.timeframe(defval, title, options, tooltip, inline, group, confirm, display) → input
string
```

ARGUMENTS

defval (const string) Determines the default value of the input variable proposed in the script's "Settings/Inputs" tab, from where the user can change it. When a list of values is used with the `options` parameter, the value must be one of them.

title (const string) Title of the input. If not specified, the variable name is used as the input's title. If the title is specified, but it is empty, the name will be an empty string.

options (tuple of const string values: [val1, val2, ...]) A list of options to choose from.

tooltip (const string) The string that will be shown to the user when hovering over the tooltip icon.

inline (const string) Combines all the input calls using the same argument in one line. The string used as an argument is not displayed. It is only used to identify inputs belonging to the same line.

group (const string) Creates a header above all inputs using the same group argument string. The string is also used as the header's text.

confirm (const bool) If true, then user will be asked to confirm input value before indicator is added to chart. Default value is false.

display (const plot_display) Controls where the script will display the input's information, aside from within the script's settings. This option allows one to remove a specific input from the script's status line or the Data Window to ensure only the most necessary inputs are displayed there. Possible values: display.none, display.data_window, display.status_line, display.all. Optional. The default is display.all.

EXAMPLE

```
//@version=5
indicator("input.timeframe", overlay=true)
i_res = input.timeframe('D', "Resolution", options=['D', 'W', 'M'])
s = request.security("AAPL", i_res, close)
plot(s)
```

RETURNS

Value of input variable.

REMARKS

Result of input.timeframe function always should be assigned to a variable, see examples above.

SEE ALSO

input.bool input.int input.float input.string input.text_area input.symbol

input.session input.source input.color input.time input

int() 4 overloads

Casts na or truncates float value to int

SYNTAX & OVERLOADS

```
int(x) → const int
```

```
int(x) → input int
```

```
int(x) → simple int
```

```
int(x) → series int
```

ARGUMENTS

x (const int/float) The value to convert to the specified type, usually na.

RETURNS

The value of the argument after casting to int.

SEE ALSO

float bool color string line label

label()

Casts na to label

SYNTAX

```
label(x) → series label
```

ARGUMENTS

x (series label) The value to convert to the specified type, usually na.

RETURNS

The value of the argument after casting to label.

SEE ALSO

float int bool color string line

label.copy()

Clones the label object.

SYNTAX

```
label.copy(id) → series label
```

ARGUMENTS

id (series label) Label object.

EXAMPLE

```
//@version=5
indicator('Last 100 bars highest/lowest', overlay = true)
```

```
LOOKBACK = 100
highest = ta.highest(LOOKBACK)
highestBars = ta.highestbars(LOOKBACK)
lowest = ta.lowest(LOOKBACK)
lowestBars = ta.lowestbars(LOOKBACK)
if barstate.islastconfirmedhistory
    var labelHigh = label.new(bar_index + highestBars, highest, str.tostring(highest), col
    var labelLow = label.copy(labelHigh)
    label.set_xy(labelLow, bar_index + lowestBars, lowest)
    label.set_text(labelLow, str.tostring(lowest))
    label.set_color(labelLow, color.red)
    label.set_style(labelLow, label.style_label_up)
```

RETURNS

New label ID object which may be passed to label.setXXX and label.getXXX functions.

SEE ALSO

label.new label.delete

label.delete()

Deletes the specified label object. If it has already been deleted, does nothing.

SYNTAX

```
label.delete(id) → void
```

ARGUMENTS

id (series label) Label object to delete.

SEE ALSO

label.new

label.get_text()

Returns the text of this label object.

SYNTAX

```
label.get_text(id) → series string
```

ARGUMENTS

id (series label) Label object.

EXAMPLE

```
//@version=5
indicator("label.get_text")
my_label = label.new(time, open, text="Open bar text", xloc=xloc.bar_time)
a = label.get_text(my_label)
label.new(time, close, text = a + " new", xloc=xloc.bar_time)
```

RETURNS

String object containing the text of this label.

SEE ALSO

label.new

label.get_x()

Returns UNIX time or bar index (depending on the last xloc value set) of this label's position.

SYNTAX

```
label.get_x(id) → series int
```

ARGUMENTS

id (series label) Label object.

EXAMPLE

```
//@version=5
indicator("label.get_x")
my_label = label.new(time, open, text="Open bar text", xloc=xloc.bar_time)
a = label.get_x(my_label)
plot(time - label.get_x(my_label)) //draws zero plot
```

RETURNS

UNIX timestamp (in milliseconds) or bar index.

SEE ALSO

label.new

label.get_y()

Returns price of this label's position.

SYNTAX

```
label.get_y(id) → series float
```

ARGUMENTS

id (series label) Label object.

RETURNS

Floating point value representing price.

SEE ALSO

label.new

label.new() 2 overloads

Creates new label object.

SYNTAX & OVERLOADS

```
label.new(point, text, xloc, yloc, color, style, textcolor, size, textalign, tooltip,
text_font_family, force_overlay) → series label
```

```
label.new(x, y, text, xloc, yloc, color, style, textcolor, size, textalign, tooltip,
text_font_family, force_overlay) → series label
```

ARGUMENTS

point (chart.point) A chart.point object that specifies the label's location.

text (series string) Label text. Default is empty string.

xloc (series string) See description of **x** argument. Possible values: xloc.bar_index and xloc.bar_time. Default is xloc.bar_index.

yloc (series string) Possible values are yloc.price, yloc.abovebar, yloc.belowbar. If yloc=yloc.price, **y** argument specifies the price of the label position. If yloc=yloc.abovebar, label is located above bar. If yloc=yloc.belowbar, label is located below bar. Default is yloc.price.

color (series color) Color of the label border and arrow

style (series string) Label style. Possible values: label.style_none, label.style_xcross, label.style_cross, label.style_triangleup, label.style_triangledown, label.style_flag, label.style_circle, label.style_arrowup, label.style_arrowdown, label.style_label_up, label.style_label_down, label.style_label_left, label.style_label_right, label.style_label_lower_left, label.style_label_lower_right, label.style_label_upper_left, label.style_label_upper_right, label.style_label_center, label.style_square, label.style_diamond, label.style_text_outline. Default is label.style_label_down.

textcolor (series color) Text color.

size (series string) Label size. Possible values: size.auto, size.tiny, size.small, size.normal, size.large, size.huge. Default value is size.normal.

textalign (series string) Label text alignment. Possible values: text.align_left, text.align_center, text.align_right. Default value is text.align_center.

tooltip (series string) Hover to see tooltip label.

text_font_family (series string) The font family of the text. Optional. The default value is font.family_default. Possible values: font.family_default, font.family_monospace.

force_overlay (const bool) If true, the drawing will display on the main chart pane, even when the script occupies a separate pane. Optional. The default is false.

EXAMPLE

```
//@version=5
indicator("label.new")
var label1 = label.new(bar_index, low, text="Hello, world!", style=label.style_circle)
label.set_x(label1, 0)
label.set_xloc(label1, time, xloc.bar_time)
label.set_color(label1, color.red)
label.set_size(label1, size.large)
```

RETURNS

Label ID object which may be passed to label.setXXX and label.getXXX functions.

SEE ALSO

label.delete label.set_x label.set_y label.set_xy label.set_xloc label.set_yloc label.set_color label.set_textcolor label.set_style label.set_size label.set_textalign label.set_tooltip

label.set_color()

Sets label border and arrow color.

SYNTAX

```
label.set_color(id, color) → void
```

ARGUMENTS

id (series label) Label object.

color (series color) New label border and arrow color.

SEE ALSO

label.new

label.set_point()

Sets the location of the `id` label to `point`.

SYNTAX

```
label.set_point(id, point) → void
```

ARGUMENTS

id (series label) A label object.

point (chart.point) A chart.point object.

label.set_size()

Sets arrow and text size of the specified label object.

SYNTAX

```
label.set_size(id, size) → void
```

ARGUMENTS

id (series label) Label object.

size (series string) Possible values: size.auto, size.tiny, size.small, size.normal, size.large, size.huge. Default value is size.auto.

SEE ALSO

size.auto size.tiny size.small size.normal size.large size.huge label.new

label.set_style()

Sets label style.

SYNTAX

```
label.set_style(id, style) → void
```

ARGUMENTS

id (series label) Label object.

style (series string) New label style. Possible values: label.style_none, label.style_xcross, label.style_cross, label.style_triangleup, label.style_triangledown, label.style_flag, label.style_circle, label.style_arrowup, label.style_arrowdown, label.style_label_up, label.style_label_down, label.style_label_left, label.style_label_right, label.style_label_lower_left, label.style_label_lower_right, label.style_label_upper_left, label.style_label_upper_right, label.style_label_center, label.style_square, label.style_diamond, label.style_text_outline.

SEE ALSO

label.new

label.set_text()

Sets label text

SYNTAX

```
label.set_text(id, text) → void
```

ARGUMENTS

id (series label) Label object.

text (series string) New label text.

SEE ALSO

label.new

label.set_text_font_family()

The function sets the font family of the text inside the label.

SYNTAX

```
label.set_text_font_family(id, text_font_family) → void
```

ARGUMENTS

id (series label) A label object.

text_font_family (series string) The font family of the text. Possible values: font.family_default, font.family_monospace.

EXAMPLE

```
//@version=5
indicator("Example of setting the label font")
if barstate.islastconfirmedhistory
    l = label.new(bar_index, 0, "monospace", yloc=yloc.abovebar)
    label.set_text_font_family(l, font.family_monospace)
```

SEE ALSO

label.new font.family_default font.family_monospace

label.set_textalign()

Sets the alignment for the label text.

SYNTAX

```
label.set_textalign(id, textalign) → void
```

ARGUMENTS

id (series label) Label object.

textalign (series string) Label text alignment. Possible values: text.align_left, text.align_center, text.align_right.

SEE ALSO

text.align_left text.align_center text.align_right label.new

label.set_textcolor()

Sets color of the label text.

SYNTAX

```
label.set_textcolor(id, textcolor) → void
```

ARGUMENTS

id (series label) Label object.

textcolor (series color) New text color.

SEE ALSO

label.new

label.set_tooltip()

Sets the tooltip text.

SYNTAX

```
label.set_tooltip(id, tooltip) → void
```

ARGUMENTS

id (series label) Label object.

tooltip (series string) Tooltip text.

SEE ALSO

label.new

label.set_x()

Sets bar index or bar time (depending on the xloc) of the label position.

SYNTAX

```
label.set_x(id, x) → void
```

ARGUMENTS

id (series label) Label object.

x (series int) New bar index or bar time of the label position. Note that objects positioned using xloc.bar_index cannot be drawn further than 500 bars into the future.

SEE ALSO

label.new

label.set_xloc()

Sets x-location and new bar index/time value.

SYNTAX

```
label.set_xloc(id, x, xloc) → void
```

ARGUMENTS

id (series label) Label object.

x (series int) New bar index or bar time of the label position.

xloc (series string) New x-location value.

SEE ALSO

xloc.bar_index xloc.bar_time label.new

label.set_xy()

Sets bar index/time and price of the label position.

SYNTAX

```
label.set_xy(id, x, y) → void
```

ARGUMENTS

id (series label) Label object.

x (series int) New bar index or bar time of the label position. Note that objects positioned using xloc.bar_index cannot be drawn further than 500 bars into the future.

y (series int/float) New price of the label position.

SEE ALSO

label.new

label.set_y()

Sets price of the label position

SYNTAX

```
label.set_y(id, y) → void
```

ARGUMENTS

id (series label) Label object.

y (series int/float) New price of the label position.

SEE ALSO

label.new

label.set_yloc()

Sets new y-location calculation algorithm.

SYNTAX

```
label.set_yloc(id, yloc) → void
```

ARGUMENTS

id (series label) Label object.

yloc (series string) New y-location value.

SEE ALSO

yloc.price yloc.abovebar yloc.belowbar label.new

library()

Declaration statement identifying a script as a library.

SYNTAX

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

ARGUMENTS

title (const string) The title of the library and its identifier. It cannot contain spaces, special characters or begin with a digit. It is used as the publication's default title, and to uniquely identify the library in the import statement, when another script uses it. It is also used as the script's name on the chart.

overlay (const bool) If true, the library will be added over the chart. If false, it will be added in a separate pane. Optional. The default is false.

dynamic_requests (const bool) Specifies whether the script can dynamically call functions from the `request.*()` namespace. Dynamic `request.*()` calls are allowed within the local scopes of conditional structures (e.g., if), loops (e.g., for), and exported functions. Additionally, such calls allow "series" arguments for many of their parameters. Optional. The default is false. See the User Manual's Dynamic requests section for more information.

EXAMPLE

```
//@version=5
// @description Math library
library("num_methods", overlay = true)
// Calculate "sinh()" from the float parameter `x`
export sinh(float x) =>
    (math.exp(x) - math.exp(-x)) / 2.0
plot(sinh(0))
```

SEE ALSO

indicator strategy

line()

Casts na to line

SYNTAX

```
line(x) → series line
```

ARGUMENTS

x (series line) The value to convert to the specified type, usually na.

RETURNS

The value of the argument after casting to line.

SEE ALSO

float int bool color string label

line.copy()

Clones the line object.

SYNTAX

```
line.copy(id) → series line
```

ARGUMENTS

id (series line) Line object.

EXAMPLE

```
//@version=5
indicator('Last 100 bars price range', overlay = true)
LOOKBACK = 100
highest = ta.highest(LOOKBACK)
lowest = ta.lowest(LOOKBACK)
if barstate.islastconfirmedhistory
    var lineTop = line.new(bar_index[LOOKBACK], highest, bar_index, highest, color = color
    var lineBottom = line.copy(lineTop)
    line.set_y1(lineBottom, lowest)
    line.set_y2(lineBottom, lowest)
    line.set_color(lineBottom, color.red)
```

RETURNS

New line ID object which may be passed to line.setXXX and line.getXXX functions.

SEE ALSO

line.new line.delete

line.delete()

Deletes the specified line object. If it has already been deleted, does nothing.

SYNTAX

```
line.delete(id) → void
```

ARGUMENTS

id (series line) Line object to delete.

SEE ALSO

line.new

line.get_price()

Returns the price level of a line at a given bar index.

SYNTAX

```
line.get_price(id, x) → series float
```

ARGUMENTS

id (series line) Line object.

x (series int) Bar index for which price is required.

EXAMPLE

```
//@version=5
indicator("GetPrice", overlay=true)
var line l = na
if bar_index == 10
    l := line.new(0, high[5], bar_index, high)
plot(line.get_price(l, bar_index), color=color.green)
```

RETURNS

Price value of line 'id' at bar index 'x'.

REMARKS

The line is considered to have been created using 'extend=extend.both'.

This function can only be called for lines created using 'xloc.bar_index'. If you try to call it for a line created with 'xloc.bar_time', it will generate an error.

SEE ALSO

line.new

line.get_x1()

Returns UNIX time or bar index (depending on the last xloc value set) of the first point of the line.

SYNTAX

```
line.get_x1(id) → series int
```

ARGUMENTS

id (series line) Line object.

EXAMPLE

```
//@version=5
indicator("line.get_x1")
my_line = line.new(time, open, time + 60 * 60 * 24, close, xloc=xloc.bar_time)
a = line.get_x1(my_line)
plot(time - line.get_x1(my_line)) //draws zero plot
```

RETURNS

UNIX timestamp (in milliseconds) or bar index.

SEE ALSO

line.new

line.get_x2()

Returns UNIX time or bar index (depending on the last xloc value set) of the second point of the line.

SYNTAX

```
line.get_x2(id) → series int
```

ARGUMENTS

id (series line) Line object.

RETURNS

UNIX timestamp (in milliseconds) or bar index.

SEE ALSO

line.new

line.get_y1()

Returns price of the first point of the line.

SYNTAX

```
line.get_y1(id) → series float
```

ARGUMENTS

id (series line) Line object.

RETURNS

Price value.

SEE ALSO

line.new

line.get_y2()

Returns price of the second point of the line.

SYNTAX

```
line.get_y2(id) → series float
```

ARGUMENTS

id (series line) Line object.

RETURNS

Price value.

SEE ALSO

line.new

line.new() 2 overloads

Creates new line object.

SYNTAX & OVERLOADS

```
line.new(first_point, second_point, xloc, extend, color, style, width, force_overlay) →
series line
```

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

ARGUMENTS

first_point (chart.point) A chart.point object that specifies the line's starting coordinate.

second_point (chart.point) A chart.point object that specifies the line's ending coordinate.

xloc (series string) See description of **x1** argument. Possible values: xloc.bar_index and xloc.bar_time. Default is xloc.bar_index.

extend (series string) If extend=extend.none, draws segment starting at point (x1, y1) and ending at point (x2, y2). If extend is equal to extend.right or extend.left, draws a ray starting at point (x1, y1) or (x2, y2), respectively. If extend=extend.both, draws a straight line that goes through these points. Default value is extend.none.

color (series color) Line color.

style (series string) Line style. Possible values: line.style_solid, line.style_dotted, line.style_dashed, line.style_arrow_left, line.style_arrow_right, line.style_arrow_both.

width (series int) Line width in pixels.

force_overlay (const bool) If true, the drawing will display on the main chart pane, even when the script occupies a separate pane. Optional. The default is false.

EXAMPLE

```
//@version=5
indicator("line.new")
var line1 = line.new(0, low, bar_index, high, extend=extend.right)
var line2 = line.new(time, open, time + 60 * 60 * 24, close, xloc=xloc.bar_time, style=lir
line.set_x2(line1, 0)
line.set_xloc(line1, time, time + 60 * 60 * 24, xloc.bar_time)
line.set_color(line2, color.green)
line.set_width(line2, 5)
```

RETURNS

Line ID object which may be passed to line.setXXX and line.getXXX functions.

SEE ALSO

line.delete line.set_x1 line.set_y1 line.set_xy1 line.set_x2 line.set_y2 line.set_xy2 line.set_xloc line.set_color line.set_extend line.set_style line.set_width

line.set_color()

Sets the line color

SYNTAX

```
line.set_color(id, color) → void
```

ARGUMENTS

id (series line) Line object.

color (series color) New line color

SEE ALSO

line.new

line.set_extend()

Sets extending type of this line object. If extend=extend.none, draws segment starting at point (x1, y1) and ending at point (x2, y2). If extend is equal to extend.right or extend.left, draws a ray starting at point (x1, y1) or (x2, y2), respectively. If extend=extend.both, draws a straight line that goes through these points.

SYNTAX

```
line.set_extend(id, extend) → void
```

ARGUMENTS

id (series line) Line object.

extend (series string) New extending type.

SEE ALSO

extend.none extend.right extend.left extend.both line.new

line.set_first_point()

Sets the first point of the `id` line to `point`.

SYNTAX

```
line.set_first_point(id, point) → void
```

ARGUMENTS

id (series line) A line object.

point (chart.point) A chart.point object.

line.set_second_point()

Sets the second point of the `id` line to `point`.

SYNTAX

```
line.set_second_point(id, point) → void
```

ARGUMENTS

id (series line) A line object.

point (chart.point) A chart.point object.

line.set_style()

Sets the line style

SYNTAX

```
line.set_style(id, style) → void
```

ARGUMENTS

id (series line) Line object.

style (series string) New line style.

SEE ALSO

line.style_solid line.style_dotted line.style_dashed line.style_arrow_left

line.style_arrow_right line.style_arrow_both line.new

line.set_width()

Sets the line width.

SYNTAX

```
line.set_width(id, width) → void
```

ARGUMENTS

id (series line) Line object.

width (series int) New line width in pixels.

SEE ALSO

line.new

line.set_x1()

Sets bar index or bar time (depending on the xloc) of the first point.

SYNTAX

```
line.set_x1(id, x) → void
```

ARGUMENTS

id (series line) Line object.

x (series int) Bar index or bar time. Note that objects positioned using xloc.bar_index cannot be drawn further than 500 bars into the future.

SEE ALSO

line.new

line.set_x2()

Sets bar index or bar time (depending on the xloc) of the second point.

SYNTAX

```
line.set_x2(id, x) → void
```

ARGUMENTS

id (series line) Line object.

x (series int) Bar index or bar time. Note that objects positioned using xloc.bar_index cannot be drawn further than 500 bars into the future.

SEE ALSO

line.new

line.set_xloc()

Sets x-location and new bar index/time values.

SYNTAX

```
line.set_xloc(id, x1, x2, xloc) → void
```

ARGUMENTS

id (series line) Line object.

x1 (series int) Bar index or bar time of the first point.

x2 (series int) Bar index or bar time of the second point.

xloc (series string) New x-location value.

SEE ALSO

xloc.bar_index xloc.bar_time line.new

line.set_xy1()

Sets bar index/time and price of the first point.

SYNTAX

```
line.set_xy1(id, x, y) → void
```

ARGUMENTS

id (series line) Line object.

x (series int) Bar index or bar time. Note that objects positioned using xloc.bar_index cannot be drawn further than 500 bars into the future.

y (series int/float) Price.

SEE ALSO

`line.new`

line.set_xy2()

Sets bar index/time and price of the second point

SYNTAX

```
line.set_xy2(id, x, y) → void
```

ARGUMENTS

id (series line) Line object.

x (series int) Bar index or bar time.

y (series int/float) Price.

SEE ALSO

`line.new`

line.set_y1()

Sets price of the first point

SYNTAX

```
line.set_y1(id, y) → void
```

ARGUMENTS

id (series line) Line object.

y (series int/float) Price.

SEE ALSO

`line.new`

line.set_y2()

Sets price of the second point.

SYNTAX

```
line.set_y2(id, y) → void
```

ARGUMENTS

id (series line) Line object.

y (series int/float) Price.

SEE ALSO

line.new

linefill()

Casts na to linefill.

SYNTAX

```
linefill(x) → series linefill
```

ARGUMENTS

x (series linefill) The value to convert to the specified type, usually na.

RETURNS

The value of the argument after casting to linefill.

SEE ALSO

float int bool color string line label

linefill.delete()

Deletes the specified linefill object. If it has already been deleted, does nothing.

SYNTAX

```
linefill.delete(id) → void
```

ARGUMENTS

id (series linefill) A linefill object.

linefill.get_line1()

Returns the ID of the first line used in the `id` linefill.

SYNTAX

```
linefill.get_line1(id) → series line
```

ARGUMENTS

id (series linefill) A linefill object.

linefill.get_line2()

Returns the ID of the second line used in the `id` linefill.

SYNTAX

```
linefill.get_line2(id) → series line
```

ARGUMENTS

id (series linefill) A linefill object.

linefill.new()

Creates a new linefill object and displays it on the chart, filling the space between `line1` and `line2` with the color specified in `color`.

SYNTAX

```
linefill.new(line1, line2, color) → series linefill
```

ARGUMENTS

line1 (series line) First line object.

line2 (series line) Second line object.

color (series color) The color used to fill the space between the lines.

RETURNS

The ID of a linefill object that can be passed to other linefill.*() functions.

REMARKS

If any line of the two is deleted, the linefill object is also deleted. If the lines are moved (e.g. via line.set_xy functions), the linefill object is also moved.

If both lines are extended in the same direction relative to the lines themselves (e.g. both have extend.right as the value of their `extend=` parameter), the space between line extensions will also be filled.

linefill.set_color()

The function sets the color of the linefill object passed to it.

SYNTAX

```
linefill.set_color(id, color) → void
```

ARGUMENTS

id (series linefill) A linefill object.

color (series color) The color of the linefill object.

log.error() 2 overloads

Converts the formatting string and value(s) into a formatted string, and sends the result to the "Pine logs" menu tagged with the "error" debug level.

The formatting string can contain literal text and one placeholder in curly braces {} for each value to be formatted. Each placeholder consists of the index of the required argument (beginning at 0) that will replace it, and an optional format specifier. The index represents the position of that argument in the function's argument list.

SYNTAX & OVERLOADS

```
log.error(message) → void
```

```
log.error(formatString, arg0, arg1, ...) → void
```

ARGUMENTS

message (series string) Log message.

EXAMPLE

```
//@version=5
strategy("My strategy", overlay = true, margin_long = 100, margin_short = 100, process_orc
bracketTickSizeInput = input.int(1000, "Stoploss/Take-Profit distance (in ticks)")

longCondition = ta.crossover(ta.sma(close, 14), ta.sma(close, 28))
if (longCondition)
    limitLevel = close * 1.01
    log.info("Long limit order has been placed at {0}", limitLevel)
    strategy.order("My Long Entry Id", strategy.long, limit = limitLevel)

    log.info("Exit orders have been placed: Take-profit at {0}, Stop-loss at {1}", close)
    strategy.exit("Exit", "My Long Entry Id", profit = bracketTickSizeInput, loss = bracke

if strategy.opentrades > 10
    log.warning("{0} positions opened in the same direction in a row. Try adjusting `brack

last10Perc = strategy.initial_capital / 10 > strategy.equity
if (last10Perc and not last10Perc[1])
    log.error("The strategy has lost 90% of the initial capital!")
```

RETURNS

The formatted string.

REMARKS

Any curly braces within an unquoted pattern must be balanced. For example, "ab {0} de" and "ab '}' de" are valid patterns, but "ab {0'}' de", "ab } de" and "''{''" are not.

The function can apply additional formatting to some values inside of the `{}`. The list of additional formatting options can be found in the EXAMPLE section of the str.format article.

The string used as the `formatString` argument can contain single quote characters ('). However, one must pair all single quotes in that string to avoid unexpected formatting results.

The "Pine logs..." button is accessible from the "More" dropdown in the Pine Editor and from the "More" dropdown in the status line of any script that uses `log.*()` functions.

log.info() 2 overloads

Converts the formatting string and value(s) into a formatted string, and sends the result to the "Pine logs" menu tagged with the "info" debug level.

The formatting string can contain literal text and one placeholder in curly braces {} for each value to be formatted. Each placeholder consists of the index of the required argument (beginning at 0) that will replace it, and an optional format specifier. The index represents the position of that argument in the function's argument list.

SYNTAX & OVERLOADS

```
log.info(message) → void
```

```
log.info(formatString, arg0, arg1, ...) → void
```

ARGUMENTS

message (series string) Log message.

EXAMPLE

```
//@version=5
strategy("My strategy", overlay = true, margin_long = 100, margin_short = 100, process_orc
bracketTickSizeInput = input.int(1000, "Stoploss/Take-Profit distance (in ticks)")

longCondition = ta.crossover(ta.sma(close, 14), ta.sma(close, 28))
if (longCondition)
    limitLevel = close * 1.01
    log.info("Long limit order has been placed at {0}", limitLevel)
    strategy.order("My Long Entry Id", strategy.long, limit = limitLevel)

    log.info("Exit orders have been placed: Take-profit at {0}, Stop-loss at {1}", close)
    strategy.exit("Exit", "My Long Entry Id", profit = bracketTickSizeInput, loss = bracke

if strategy.opentrades > 10
    log.warning("{0} positions opened in the same direction in a row. Try adjusting `brack

last10Perc = strategy.initial_capital / 10 > strategy.equity
if (last10Perc and not last10Perc[1])
    log.error("The strategy has lost 90% of the initial capital!")
```

RETURNS

The formatted string.

REMARKS

Any curly braces within an unquoted pattern must be balanced. For example, "ab {0} de" and "ab '}' de" are valid patterns, but "ab {0'}' de", "ab } de" and "''{''" are not.

The function can apply additional formatting to some values inside of the `{}`. The list of additional formatting options can be found in the EXAMPLE section of the str.format article.

The string used as the `formatString` argument can contain single quote characters ('). However, one must pair all single quotes in that string to avoid unexpected formatting results.

The "Pine logs..." button is accessible from the "More" dropdown in the Pine Editor and from the "More" dropdown in the status line of any script that uses `log.*()` functions.

log.warning() 2 overloads

Converts the formatting string and value(s) into a formatted string, and sends the result to the "Pine logs" menu tagged with the "warning" debug level.

The formatting string can contain literal text and one placeholder in curly braces {} for each value to be formatted. Each placeholder consists of the index of the required argument (beginning at 0) that will replace it, and an optional format specifier. The index represents the position of that argument in the function's argument list.

SYNTAX & OVERLOADS

```
log.warning(message) → void
```

```
log.warning(formatString, arg0, arg1, ...) → void
```

ARGUMENTS

message (series string) Log message.

EXAMPLE

```
//@version=5
strategy("My strategy", overlay = true, margin_long = 100, margin_short = 100, process_orc
bracketTickSizeInput = input.int(1000, "Stoploss/Take-Profit distance (in ticks)")

longCondition = ta.crossover(ta.sma(close, 14), ta.sma(close, 28))
if (longCondition)
    limitLevel = close * 1.01
    log.info("Long limit order has been placed at {0}", limitLevel)
    strategy.order("My Long Entry Id", strategy.long, limit = limitLevel)

    log.info("Exit orders have been placed: Take-profit at {0}, Stop-loss at {1}", close)
    strategy.exit("Exit", "My Long Entry Id", profit = bracketTickSizeInput, loss = bracke

if strategy.opentrades > 10
    log.warning("{0} positions opened in the same direction in a row. Try adjusting `brack

last10Perc = strategy.initial_capital / 10 > strategy.equity
if (last10Perc and not last10Perc[1])
```

```
    log.error("The strategy has lost 90% of the initial capital!")
```

RETURNS

The formatted string.

REMARKS

Any curly braces within an unquoted pattern must be balanced. For example, "ab {0} de" and "ab '}' de" are valid patterns, but "ab {0'}' de", "ab } de" and "''{''" are not.

The function can apply additional formatting to some values inside of the `{}`. The list of additional formatting options can be found in the EXAMPLE section of the str.format article.

The string used as the `formatString` argument can contain single quote characters ('). However, one must pair all single quotes in that string to avoid unexpected formatting results.

The "Pine logs..." button is accessible from the "More" dropdown in the Pine Editor and from the "More" dropdown in the status line of any script that uses `log.*()` functions.

map.clear()

Clears the map, removing all key-value pairs from it.

SYNTAX

```
map.clear(id) → void
```

ARGUMENTS

id (any map type) A map object.

EXAMPLE

```
//@version=5
indicator("map.clear example")
oddMap = map.new<int, bool>()
oddMap.put(1, true)
oddMap.put(2, false)
oddMap.put(3, true)
map.clear(oddMap)
plot(oddMap.size())
```

SEE ALSO

map.new<type,type> map.put_all map.keys map.values map.remove

map.contains()

Returns true if the `key` was found in the `id` map, false otherwise.

SYNTAX

```
map.contains(id, key) → series bool
```

ARGUMENTS

id (any map type) A map object.

key (series <type of the map's elements>) The key to search in the map.

EXAMPLE

```
//@version=5
indicator("map.includes example")
a = map.new<string, float>()
a.put("open", open)
p = close
if map.contains(a, "open")
    p := a.get("open")
plot(p)
```

SEE ALSO

map.new<type,type> map.put map.keys map.values map.size

map.copy()

Creates a copy of an existing map.

SYNTAX

```
map.copy(id) → map<keyType, valueType>
```

ARGUMENTS

id (any map type) A map object to copy.

EXAMPLE

```
//@version=5
indicator("map.copy example")
a = map.new<string, int>()
a.put("example", 1)
b = map.copy(a)
a := map.new<string, int>()
a.put("example", 2)
plot(a.get("example"))
plot(b.get("example"))
```

RETURNS

A copy of the `id` map.

SEE ALSO

map.new<type,type> map.put map.keys map.values map.get map.size

map.get()

Returns the value associated with the specified `key` in the `id` map.

SYNTAX

```
map.get(id, key) → <value_type>
```

ARGUMENTS

id (any map type) A map object.

key (series <type of the map's elements>) The key of the value to retrieve.

EXAMPLE

```
//@version=5
indicator("map.get example")
a = map.new<int, int>()
size = 10
for i = 0 to size
    a.put(i, size-i)
plot(map.get(a, 1))
```

SEE ALSO

map.new<type,type> map.put map.keys map.values map.contains

map.keys()

Returns an array of all the keys in the `id` map. The resulting array is a copy and any changes to it are not reflected in the original map.

SYNTAX

```
map.keys(id) → array<type>
```

ARGUMENTS

id (any map type) A map object.

EXAMPLE

```
//@version=5
indicator("map.keys example")
a = map.new<string, float>()
a.put("open", open)
a.put("high", high)
a.put("low", low)
a.put("close", close)
keys = map.keys(a)
ohlc = 0.0
for key in keys
    ohlc += a.get(key)
plot(ohlc/4)
```

REMARKS

Maps maintain insertion order. The elements within the array returned by this function will also be in the insertion order.

SEE ALSO

map.new<type,type> map.put map.get map.values map.size

map.new<type,type>()

Creates a new map object: a collection that consists of key-value pairs, where all keys are of the `keyType`, and all values are of the `valueType`.

`keyType` can be a primitive type or enum. For example: int, float, bool, string, color.

`valueType` can be of any type except `array<>`, `matrix<>`, and `map<>`. User-defined types are allowed, even if they have `array<>`, `matrix<>`, or `map<>` as one of their fields.

SYNTAX

```
map.new<keyType, valueType>() → map<keyType, valueType>
```

EXAMPLE

```
//@version=5
indicator("map.new<string, int> example")
a = map.new<string, int>()
a.put("example", 1)
label.new(bar_index, close, str.tostring(a.get("example")))
```

RETURNS

The ID of a map object which may be used in other map.*() functions.

REMARKS

Each key is unique and can only appear once. When adding a new value with a key that the map already contains, that value replaces the old value associated with the key.

Maps maintain insertion order. Note that the order does not change when inserting a pair with a `key` that's already in the map. The new pair replaces the existing pair with the `key` in such cases.

SEE ALSO

map.put map.keys map.values map.get array.new<type>

map.put()

Puts a new key-value pair into the `id` map.

SYNTAX

```
map.put(id, key, value) → <value_type>
```

ARGUMENTS

id (any map type) A map object.

key (series <type of the map's elements>) The key to put into the map.

value (series <type of the map's elements>) The key value to put into the map.

EXAMPLE

```
//@version=5
indicator("map.put example")
a = map.new<string, float>()
map.put(a, "first", 10)
map.put(a, "second", 15)
prevFirst = map.put(a, "first", 20)
currFirst = a.get("first")
plot(prevFirst)
plot(currFirst)
```

RETURNS

The previous value associated with `key` if the key was already present in the map, or na if the key is new.

REMARKS

Maps maintain insertion order. Note that the order does not change when inserting a pair with a `key` that's already in the map. The new pair replaces the existing pair with the `key` in such cases.

SEE ALSO

map.new<type,type> map.put_all map.keys map.values map.remove

map.put_all()

Puts all key-value pairs from the `id2` map into the `id` map.

SYNTAX

```
map.put_all(id, id2) → void
```

ARGUMENTS

id (any map type) A map object to append to.

id2 (any map type) A map object to be appended.

EXAMPLE

```
//@version=5
indicator("map.put_all example")
a = map.new<string, float>()
b = map.new<string, float>()
```

```
    a.put("first", 10)
    a.put("second", 15)
    b.put("third", 20)
    map.put_all(a, b)
    plot(a.get("third"))
```

SEE ALSO

map.new<type,type> map.put map.keys map.values map.remove

map.remove()

Removes a key-value pair from the `id` map.

SYNTAX

```
map.remove(id, key) → <value_type>
```

ARGUMENTS

id (any map type) A map object.

key (series <type of the map's elements>) The key of the pair to remove from the map.

EXAMPLE

```
//@version=5
indicator("map.remove example")
a = map.new<string, color>()
a.put("firstColor", color.green)
oldColorValue = map.remove(a, "firstColor")
plot(close, color = oldColorValue)
```

RETURNS

The previous value associated with `key` if the key was present in the map, or na if there was no such key.

SEE ALSO

map.new<type,type> map.put map.keys map.values map.clear

map.size()

Returns the number of key-value pairs in the `id` map.

SYNTAX

```
map.size(id) → series int
```

ARGUMENTS

id (any map type) A map object.

EXAMPLE

```
//@version=5
indicator("map.size example")
a = map.new<int, int>()
size = 10
for i = 0 to size
    a.put(i, size-i)
plot(map.size(a))
```

SEE ALSO

map.new<type,type> map.put map.keys map.values map.get

map.values()

Returns an array of all the values in the `id` map. The resulting array is a copy and any changes to it are not reflected in the original map.

SYNTAX

```
map.values(id) → array<type>
```

ARGUMENTS

id (any map type) A map object.

EXAMPLE

```
//@version=5
indicator("map.values example")
a = map.new<string, float>()
a.put("open", open)
a.put("high", high)
a.put("low", low)
a.put("close", close)
```

```
values = map.values(a)
ohlc = 0.0
for value in values
    ohlc += value
plot(ohlc/4)
```

REMARKS

Maps maintain insertion order. The elements within the array returned by this function will also be in the insertion order.

SEE ALSO

map.new<type,type>   map.put   map.get   map.keys   map.size

math.abs() 8 overloads

Absolute value of `number` is `number` if `number` >= 0, or - `number` otherwise.

SYNTAX & OVERLOADS

```
math.abs(number) → const int
```

```
math.abs(number) → input int
```

```
math.abs(number) → const float
```

```
math.abs(number) → simple int
```

```
math.abs(number) → input float
```

```
math.abs(number) → series int
```

```
math.abs(number) → simple float
```

```
math.abs(number) → series float
```

ARGUMENTS

number (const int) The number to use in the calculation.

RETURNS

The absolute value of `number`.

math.acos() 4 overloads

The acos function returns the arccosine (in radians) of number such that cos(acos(y)) = y for y in range [-1, 1].

SYNTAX & OVERLOADS

```
math.acos(angle) → const float
```

```
math.acos(angle) → input float
```

```
math.acos(angle) → simple float
```

```
math.acos(angle) → series float
```

ARGUMENTS

angle (const int/float) The value, in radians, to use in the calculation.

RETURNS

The arc cosine of a value; the returned angle is in the range [0, Pi], or na if y is outside of range [-1, 1].

math.asin() 4 overloads

The asin function returns the arcsine (in radians) of number such that sin(asin(y)) = y for y in range [-1, 1].

SYNTAX & OVERLOADS

```
math.asin(angle) → const float
```

```
math.asin(angle) → input float
```

```
math.asin(angle) → simple float
math.asin(angle) → series float
```

ARGUMENTS

angle (const int/float) The value, in radians, to use in the calculation.

RETURNS

The arcsine of a value; the returned angle is in the range [-Pi/2, Pi/2], or na if y is outside of range [-1, 1].

math.atan() 4 overloads

The atan function returns the arctangent (in radians) of number such that tan(atan(y)) = y for any y.

SYNTAX & OVERLOADS

```
math.atan(angle) → const float
```

```
math.atan(angle) → input float
```

```
math.atan(angle) → simple float
```

```
math.atan(angle) → series float
```

ARGUMENTS

angle (const int/float) The value, in radians, to use in the calculation.

RETURNS

The arc tangent of a value; the returned angle is in the range [-Pi/2, Pi/2].

math.avg() 2 overloads

Calculates average of all given series (elementwise).

SYNTAX & OVERLOADS

```
math.avg(number0, number1, ...) → simple float
math.avg(number0, number1, ...) → series float
```

ARGUMENTS

number0, number1, ... (simple int/float) A sequence of numbers to use in the calculation.

RETURNS

Average.

SEE ALSO

math.sum ta.cum ta.sma

math.ceil() 4 overloads

The ceil function returns the smallest (closest to negative infinity) integer that is greater than or equal to the argument.

SYNTAX & OVERLOADS

```
math.ceil(number) → const int
```

```
math.ceil(number) → input int
```

```
math.ceil(number) → simple int
```

```
math.ceil(number) → series int
```

ARGUMENTS

number (const int/float) The number to use in the calculation.

RETURNS

The smallest integer greater than or equal to the given number.

SEE ALSO

math.floor math.round

math.cos() 4 overloads

The cos function returns the trigonometric cosine of an angle.

SYNTAX & OVERLOADS

```
math.cos(angle) → const float
```

```
math.cos(angle) → input float
```

```
math.cos(angle) → simple float
```

```
math.cos(angle) → series float
```

ARGUMENTS

angle (const int/float) Angle, in radians.

RETURNS

The trigonometric cosine of an angle.

math.exp() 4 overloads

The exp function of `number` is e raised to the power of `number`, where e is Euler's number.

SYNTAX & OVERLOADS

```
math.exp(number) → const float
```

```
math.exp(number) → input float
```

```
math.exp(number) → simple float
```

```
math.exp(number) → series float
```

ARGUMENTS

number (const int/float) The number to use in the calculation.

RETURNS

A value representing e raised to the power of `number`.

SEE ALSO

math.pow

math.floor() 4 overloads

SYNTAX & OVERLOADS

```
math.floor(number) → const int
```

```
math.floor(number) → input int
```

```
math.floor(number) → simple int
```

```
math.floor(number) → series int
```

ARGUMENTS

number (const int/float) The number to use in the calculation.

RETURNS

The largest integer less than or equal to the given number.

SEE ALSO

math.ceil math.round

math.log() 4 overloads

Natural logarithm of any `number` > 0 is the unique y such that e^y = `number`.

SYNTAX & OVERLOADS

```
math.log(number) → const float
```

```
math.log(number) → input float
```

```
math.log(number) → simple float
math.log(number) → series float
```

ARGUMENTS

number (const int/float) The number to use in the calculation.

RETURNS

The natural logarithm of `number`.

SEE ALSO

math.log10

math.log10() 4 overloads

The common (or base 10) logarithm of `number` is the power to which 10 must be raised to obtain the `number`. 10^y = `number`.

SYNTAX & OVERLOADS

```
math.log10(number) → const float
```

```
math.log10(number) → input float
```

```
math.log10(number) → simple float
```

```
math.log10(number) → series float
```

ARGUMENTS

number (const int/float) The number to use in the calculation.

RETURNS

The base 10 logarithm of `number`.

SEE ALSO

math.log

math.max() 8 overloads

Returns the greatest of multiple values.

SYNTAX & OVERLOADS

```
math.max(number0, number1, ...) → const int
```

```
math.max(number0, number1, ...) → const float
```

```
math.max(number0, number1, ...) → input int
```

```
math.max(number0, number1, ...) → simple int
```

```
math.max(number0, number1, ...) → input float
```

```
math.max(number0, number1, ...) → series int
```

```
math.max(number0, number1, ...) → simple float
```

```
math.max(number0, number1, ...) → series float
```

ARGUMENTS

number0, number1, ... (const int) A sequence of numbers to use in the calculation.

EXAMPLE

```
//@version=5
indicator("math.max", overlay=true)
plot(math.max(close, open))
plot(math.max(close, math.max(open, 42)))
```

RETURNS

The greatest of multiple given values.

SEE ALSO

math.min

math.min() 8 overloads

Returns the smallest of multiple values.

SYNTAX & OVERLOADS

```
math.min(number0, number1, ...) → const int
```

```
math.min(number0, number1, ...) → const float
```

```
math.min(number0, number1, ...) → input int
```

```
math.min(number0, number1, ...) → simple int
```

```
math.min(number0, number1, ...) → input float
```

```
math.min(number0, number1, ...) → series int
```

```
math.min(number0, number1, ...) → simple float
```

```
math.min(number0, number1, ...) → series float
```

ARGUMENTS

number0, number1, ... (const int) A sequence of numbers to use in the calculation.

EXAMPLE

```
//@version=5
indicator("math.min", overlay=true)
plot(math.min(close, open))
plot(math.min(close, math.min(open, 42)))
```

RETURNS

The smallest of multiple given values.

SEE ALSO

math.max

math.pow() 4 overloads

Mathematical power function.

SYNTAX & OVERLOADS

```
math.pow(base, exponent) → const float
```

```
math.pow(base, exponent) → input float
```

```
math.pow(base, exponent) → simple float
```

```
math.pow(base, exponent) → series float
```

ARGUMENTS

base (const int/float) Specify the base to use.

exponent (const int/float) Specifies the exponent.

EXAMPLE

```
//@version=5
indicator("math.pow", overlay=true)
plot(math.pow(close, 2))
```

RETURNS

`base` raised to the power of `exponent`. If `base` is a series, it is calculated elementwise.

SEE ALSO

math.sqrt math.exp

math.random()

Returns a pseudo-random value. The function will generate a different sequence of values for each script execution. Using the same value for the optional seed argument will produce a repeatable sequence.

SYNTAX

```
math.random(min, max, seed) → series float
```

ARGUMENTS

min (series int/float) The lower bound of the range of random values. The value is not included in the range. The default is 0.

max (series int/float) The upper bound of the range of random values. The value is not included in the range. The default is 1.

seed (series int) Optional argument. When the same seed is used, allows successive calls to the function to produce a repeatable set of values.

RETURNS

A random value.

math.round() 8 overloads

Returns the value of `number` rounded to the nearest integer, with ties rounding up. If the `precision` parameter is used, returns a float value rounded to that amount of decimal places.

SYNTAX & OVERLOADS

```
math.round(number) → const int
```

```
math.round(number) → input int
```

```
math.round(number) → simple int
```

```
math.round(number) → series int
```

```
math.round(number, precision) → const float
```

```
math.round(number, precision) → input float
```

```
math.round(number, precision) → simple float
```

```
math.round(number, precision) → series float
```

ARGUMENTS

number (const int/float) The value to be rounded.

RETURNS

The value of `number` rounded to the nearest integer, or according to precision.

REMARKS

Note that for 'na' values function returns 'na'.

SEE ALSO

math.ceil math.floor

math.round_to_mintick() 2 overloads

Returns the value rounded to the symbol's mintick, i.e. the nearest value that can be divided by syminfo.mintick, without the remainder, with ties rounding up.

SYNTAX & OVERLOADS

```
math.round_to_mintick(number) → simple float
```

```
math.round_to_mintick(number) → series float
```

ARGUMENTS

number (simple int/float) The value to be rounded.

RETURNS

The `number` rounded to tick precision.

REMARKS

Note that for 'na' values function returns 'na'.

SEE ALSO

math.ceil math.floor

math.sign() 4 overloads

Sign (signum) of `number` is zero if `number` is zero, 1.0 if `number` is greater than zero, -1.0 if `number` is less than zero.

SYNTAX & OVERLOADS

```
math.sign(number) → const float
```

```
math.sign(number) → input float
```

```
math.sign(number) → simple float
```

```
math.sign(number) → series float
```

ARGUMENTS

number (const int/float) The number to use in the calculation.

RETURNS

The sign of the argument.

math.sin() 4 overloads

The sin function returns the trigonometric sine of an angle.

SYNTAX & OVERLOADS

```
math.sin(angle) → const float
```

```
math.sin(angle) → input float
```

```
math.sin(angle) → simple float
```

```
math.sin(angle) → series float
```

ARGUMENTS

angle (const int/float) Angle, in radians.

RETURNS

The trigonometric sine of an angle.

math.sqrt() 4 overloads

Square root of any `number` >= 0 is the unique y >= 0 such that y^2 = `number`.

SYNTAX & OVERLOADS

```
math.sqrt(number) → const float
```

```
math.sqrt(number) → input float
```

```
math.sqrt(number) → simple float
```

```
math.sqrt(number) → series float
```

ARGUMENTS

number (const int/float) The number to use in the calculation.

RETURNS

The square root of `number`.

SEE ALSO

math.pow

math.sum()

The sum function returns the sliding sum of last y values of x.

SYNTAX

```
math.sum(source, length) → series float
```

ARGUMENTS

source (series int/float) Series of values to process.

length (series int) Number of bars (length).

RETURNS

Sum of `source` for `length` bars back.

REMARKS

`na` values in the `source` series are ignored; the function calculates on the `length` quantity of non- `na` values.

SEE ALSO

`ta.cum` `for`

math.tan() 4 overloads

The tan function returns the trigonometric tangent of an angle.

SYNTAX & OVERLOADS

```
math.tan(angle) → const float
```

```
math.tan(angle) → input float
```

```
math.tan(angle) → simple float
```

```
math.tan(angle) → series float
```

ARGUMENTS

angle (const int/float) Angle, in radians.

RETURNS

The trigonometric tangent of an angle.

math.todegrees()

Returns an approximately equivalent angle in degrees from an angle measured in radians.

SYNTAX

```
math.todegrees(radians) → series float
```

ARGUMENTS

radians (series int/float) Angle in radians.

RETURNS

The angle value in degrees.

math.toradians()

Returns an approximately equivalent angle in radians from an angle measured in degrees.

SYNTAX

```
math.toradians(degrees) → series float
```

ARGUMENTS

degrees (series int/float) Angle in degrees.

RETURNS

The angle value in radians.

matrix.add_col() 2 overloads

The function adds a column at the `column` index of the `id` matrix. The column can consist of `na` values, or an array can be used to provide values.

SYNTAX & OVERLOADS

```
matrix.add_col(id, column) → void
```

```
matrix.add_col(id, column, array_id) → void
```

ARGUMENTS

id (any matrix type) A matrix object.

column (series int) The index of the column after which the new column will be inserted. Optional. The default value is matrix.columns.

Adding a column to the matrix

EXAMPLE

```
//@version=5
indicator("`matrix.add_col()` Example 1")

// Create a 2x3 "int" matrix containing values `0`.
m = matrix.new<int>(2, 3, 0)

// Add a column  with `na` values to the matrix.
```

```
    matrix.add_col(m)

    // Display matrix elements.
    if barstate.islastconfirmedhistory
        var t = table.new(position.top_right, 2, 2, color.green)
        table.cell(t, 0, 0, "Matrix elements:")
        table.cell(t, 0, 1, str.tostring(m))
```

Adding an array as a column to the matrix

EXAMPLE

```
//@version=5
indicator("`matrix.add_col()` Example 2")

if barstate.islastconfirmedhistory
    // Create an empty matrix object.
    var m = matrix.new<int>()

    // Create an array with values `1` and `3`.
    var a = array.from(1, 3)

    // Add the `a` array as the first column of the empty matrix.
    matrix.add_col(m, 0, a)

    // Display matrix elements.
    var t = table.new(position.top_right, 2, 2, color.green)
    table.cell(t, 0, 0, "Matrix elements:")
    table.cell(t, 0, 1, str.tostring(m))
```

REMARKS

Rather than add columns to an empty matrix, it is far more efficient to declare a matrix with explicit dimensions and fill it with values. Adding a column is also much slower than adding a row with the matrix.add_row function.

SEE ALSO

matrix.new<type> matrix.get matrix.set matrix.columns matrix.rows matrix.add_row

matrix.add_row() 2 overloads

The function adds a row at the `row` index of the `id` matrix. The row can consist of `na` values, or an array can be used to provide values.

SYNTAX & OVERLOADS

```
matrix.add_row(id, row) → void
```

```
matrix.add_row(id, row, array_id) → void
```

ARGUMENTS

id (any matrix type) A matrix object.

row (series int) The index of the row after which the new row will be inserted. Optional. The default value is matrix.rows.

Adding a row to the matrix

EXAMPLE

```
//@version=5
indicator("`matrix.add_row()` Example 1")

// Create a 2x3 "int" matrix containing values `0`.
m = matrix.new<int>(2, 3, 0)

// Add a row with `na` values to the matrix.
matrix.add_row(m)

// Display matrix elements.
if barstate.islastconfirmedhistory
    var t = table.new(position.top_right, 2, 2, color.green)
    table.cell(t, 0, 0, "Matrix elements:")
    table.cell(t, 0, 1, str.tostring(m))
```

Adding an array as a row to the matrix

EXAMPLE

```
//@version=5
indicator("`matrix.add_row()` Example 2")

if barstate.islastconfirmedhistory
    // Create an empty matrix object.
    var m = matrix.new<int>()

    // Create an array with values `1` and `2`.
    var a = array.from(1, 2)

    // Add the `a` array as the first row of the empty matrix.
    matrix.add_row(m, 0, a)

    // Display matrix elements.
    var t = table.new(position.top_right, 2, 2, color.green)
    table.cell(t, 0, 0, "Matrix elements:")
    table.cell(t, 0, 1, str.tostring(m))
```

REMARKS

Indexing of rows and columns starts at zero. Rather than add rows to an empty matrix, it is far more efficient to declare a matrix with explicit dimensions and fill it with values.

SEE ALSO

matrix.new<type> matrix.get matrix.set matrix.columns matrix.rows matrix.add_col

matrix.avg() 2 overloads

The function calculates the average of all elements in the matrix.

SYNTAX & OVERLOADS

```
matrix.avg(id) → series float
```

```
matrix.avg(id) → series int
```

ARGUMENTS

id (matrix<int/float>) A matrix object.

EXAMPLE

```
//@version=5
indicator("`matrix.avg()` Example")

// Create a 2x2 matrix.
var m = matrix.new<int>(2, 2, na)
// Fill the matrix with values.
matrix.set(m, 0, 0, 1)
matrix.set(m, 0, 1, 2)
matrix.set(m, 1, 0, 3)
matrix.set(m, 1, 1, 4)

// Get the average value of the matrix.
var x = matrix.avg(m)

plot(x, 'Matrix average value')
```

RETURNS

The average value from the `id` matrix.

SEE ALSO

matrix.new<type> matrix.get matrix.set matrix.columns matrix.rows

matrix.col()

The function creates a one-dimensional array from the elements of a matrix column.

SYNTAX

```
matrix.col(id, column) → array<type>
```

ARGUMENTS

id (any matrix type) A matrix object.

column (series int) Index of the required column.

EXAMPLE

```
//@version=5
indicator("`matrix.col()` Example", "", true)

// Create a 2x3 "float" matrix from `hlc3` values.
m = matrix.new<float>(2, 3, hlc3)

// Return an array with the values of the first column of matrix `m`.
a = matrix.col(m, 0)

// Plot the first value from the array `a`.
plot(array.get(a, 0))
```

RETURNS

An array ID containing the `column` values of the `id` matrix.

REMARKS

Indexing of rows starts at 0.

SEE ALSO

matrix.new<type> matrix.get array.get matrix.col matrix.columns

matrix.columns()

The function returns the number of columns in the matrix.

SYNTAX

```
matrix.columns(id) → series int
```

ARGUMENTS

id (any matrix type) A matrix object.

EXAMPLE

```
//@version=5
indicator("`matrix.columns()` Example")

// Create a 2x6 matrix with values `0`.
var m = matrix.new<int>(2, 6, 0)

// Get the quantity of columns in matrix `m`.
var x = matrix.columns(m)

// Display using a label.
if barstate.islastconfirmedhistory
    label.new(bar_index, high, "Columns: " + str.tostring(x) + "\n" + str.tostring(m))
```

RETURNS

The number of columns in the matrix `id`.

SEE ALSO

matrix.new<type> matrix.get matrix.set matrix.col matrix.row matrix.rows

matrix.concat()

The function appends the `m2` matrix to the `m1` matrix.

SYNTAX

```
matrix.concat(id1, id2) → matrix<type>
```

ARGUMENTS

id1 (any matrix type) Matrix object to concatenate into.

id2 (any matrix type) Matrix object whose elements will be appended to `id1`.

EXAMPLE

```
//@version=5
indicator("`matrix.concat()` Example")

// Create a 2x4 "int" matrix containing values `0`.
m1 = matrix.new<int>(2, 4, 0)
// Create a 2x4 "int" matrix containing values `1`.
m2 = matrix.new<int>(2, 4, 1)

// Append matrix `m2` to `m1`.
matrix.concat(m1, m2)

// Display matrix elements.
if barstate.islastconfirmedhistory
    var t = table.new(position.top_right, 2, 2, color.green)
    table.cell(t, 0, 0, "Matrix Elements:")
    table.cell(t, 0, 1, str.tostring(m1))
```

RETURNS

Returns the `id1` matrix concatenated with the `id2` matrix.

REMARKS

The number of columns in both matrices must be identical.

SEE ALSO

matrix.new<type> matrix.get matrix.set matrix.columns matrix.rows

matrix.copy()

The function creates a new matrix which is a copy of the original.

SYNTAX

```
matrix.copy(id) → matrix<type>
```

ARGUMENTS

id (any matrix type) A matrix object to copy.

EXAMPLE

```
//@version=5
indicator("`matrix.copy()` Example")

// For efficiency, execute this code only once.
if barstate.islastconfirmedhistory
    // Create a 2x3 "float" matrix with `1` values.
    var m1 = matrix.new<float>(2, 3, 1)

    // Copy the matrix to a new one.
    // Note that unlike what `matrix.copy()` does,
    // the simple assignment operation `m2 = m1`
    // would NOT create a new copy of the `m1` matrix.
    // It would merely create a copy of its ID referencing the same matrix.
    var m2 = matrix.copy(m1)

    // Display using a table.
    var t = table.new(position.top_right, 5, 2, color.green)
    table.cell(t, 0, 0, "Original Matrix:")
    table.cell(t, 0, 1, str.tostring(m1))
    table.cell(t, 1, 0, "Matrix Copy:")
    table.cell(t, 1, 1, str.tostring(m2))
```

RETURNS

A new matrix object of the copied `id` matrix.

SEE ALSO

matrix.new<type>   matrix.get   matrix.set   matrix.columns   matrix.rows

matrix.det() 2 overloads

The function returns the determinant of a square matrix.

SYNTAX & OVERLOADS

```
matrix.det(id) → series float
```

```
matrix.det(id) → series int
```

ARGUMENTS

id (matrix<int/float>) A matrix object.

EXAMPLE

```
//@version=5
indicator("`matrix.det` Example")
```

```
// Create a 2x2 matrix.
var m = matrix.new<float>(2, 2, na)
// Fill the matrix with values.
matrix.set(m, 0, 0,  3)
matrix.set(m, 0, 1,  7)
matrix.set(m, 1, 0,  1)
matrix.set(m, 1, 1, -4)

// Get the determinant of the matrix.
var x = matrix.det(m)

plot(x, 'Matrix determinant')
```

RETURNS

The determinant value of the `id` matrix.

REMARKS

Function calculation based on the LU decomposition algorithm.

SEE ALSO

matrix.new<type>   matrix.set   matrix.is_square

matrix.diff() 2 overloads

The function returns a new matrix resulting from the subtraction between matrices `id1` and `id2`, or of matrix `id1` and an `id2` scalar (a numerical value).

SYNTAX & OVERLOADS

```
matrix.diff(id1, id2) → matrix<int>
```

```
matrix.diff(id1, id2) → matrix<float>
```

ARGUMENTS

id1 (matrix<int>) Matrix to subtract from.

id2 (series int/float/matrix<int>) Matrix object or a scalar value to be subtracted.

Difference between two matrices

EXAMPLE

```
//@version=5
```

```
indicator("`matrix.diff()` Example 1")

// For efficiency, execute this code only once.
if barstate.islastconfirmedhistory
    // Create a 2x3 matrix containing values `5`.
    var m1 = matrix.new<float>(2, 3, 5)
    // Create a 2x3 matrix containing values `4`.
    var m2 = matrix.new<float>(2, 3, 4)
    // Create a new matrix containing the difference between matrices `m1` and `m2`.
    var m3 = matrix.diff(m1, m2)

    // Display using a table.
    var t = table.new(position.top_right, 1, 2, color.green)
    table.cell(t, 0, 0, "Difference between two matrices:")
    table.cell(t, 0, 1, str.tostring(m3))
```

Difference between a matrix and a scalar value

EXAMPLE

```
//@version=5
indicator("`matrix.diff()` Example 2")

// For efficiency, execute this code only once.
if barstate.islastconfirmedhistory
    // Create a 2x3 matrix with values `4`.
    var m1 = matrix.new<float>(2, 3, 4)

    // Create a new matrix containing the difference between the `m1` matrix and the "int'
    var m2 = matrix.diff(m1, 1)

    // Display using a table.
    var t = table.new(position.top_right, 1, 2, color.green)
    table.cell(t,  0, 0, "Difference between a matrix and a scalar:")
    table.cell(t,  0, 1, str.tostring(m2))
```

RETURNS

A new matrix object containing the difference between `id2` and `id1`.

SEE ALSO

matrix.new<type> matrix.get matrix.set matrix.columns matrix.rows

matrix.eigenvalues() 2 overloads

The function returns an array containing the eigenvalues of a square matrix.

SYNTAX & OVERLOADS

```
matrix.eigenvalues(id) → array<float>
```

```
matrix.eigenvalues(id) → array<int>
```

ARGUMENTS

id (matrix<int/float>) A matrix object.

EXAMPLE

```
//@version=5
indicator("`matrix.eigenvalues()` Example")

// For efficiency, execute this code only once.
if barstate.islastconfirmedhistory
    // Create a 2x2 matrix.
    var m1 = matrix.new<int>(2, 2, na)
    // Fill the matrix with values.
    matrix.set(m1, 0, 0, 2)
    matrix.set(m1, 0, 1, 4)
    matrix.set(m1, 1, 0, 6)
    matrix.set(m1, 1, 1, 8)

    // Get the eigenvalues of the matrix.
    tr = matrix.eigenvalues(m1)

    // Display matrix elements.
    var t = table.new(position.top_right, 2, 2, color.green)
    table.cell(t, 0, 0, "Matrix elements:")
    table.cell(t, 0, 1, str.tostring(m1))
    table.cell(t, 1, 0, "Array of Eigenvalues:")
    table.cell(t, 1, 1, str.tostring(tr))
```

RETURNS

An array containing the eigenvalues of the `id` matrix.

REMARKS

The function is calculated using "The Implicit QL Algorithm".

SEE ALSO

matrix.new<type> matrix.set matrix.eigenvectors

matrix.eigenvectors() 2 overloads

Returns a matrix of eigenvectors, in which each column is an eigenvector of the `id` matrix.

SYNTAX & OVERLOADS

```
matrix.eigenvectors(id) → matrix<float>
```

```
matrix.eigenvectors(id) → matrix<int>
```

ARGUMENTS

id (matrix<int/float>) A matrix object.

EXAMPLE

```
//@version=5
indicator("`matrix.eigenvectors()` Example")

// For efficiency, execute this code only once.
if barstate.islastconfirmedhistory
    // Create a 2x2 matrix
    var m1 = matrix.new<int>(2, 2, 1)
    // Fill the matrix with values.
    matrix.set(m1, 0, 0, 2)
    matrix.set(m1, 0, 1, 4)
    matrix.set(m1, 1, 0, 6)
    matrix.set(m1, 1, 1, 8)

    // Get the eigenvectors of the matrix.
    m2 = matrix.eigenvectors(m1)

    // Display matrix elements.
    var t = table.new(position.top_right, 2, 2, color.green)
    table.cell(t, 0, 0, "Matrix Elements:")
    table.cell(t, 0, 1, str.tostring(m1))
    table.cell(t, 1, 0, "Matrix Eigenvectors:")
    table.cell(t, 1, 1, str.tostring(m2))
```

RETURNS

A new matrix containing the eigenvectors of the `id` matrix.

REMARKS

The function is calculated using "The Implicit QL Algorithm".

SEE ALSO

matrix.new<type>   matrix.get   matrix.set   matrix.eigenvalues

matrix.elements_count()

The function returns the total number of all matrix elements.

SYNTAX

```
matrix.elements_count(id) → series int
```

ARGUMENTS

id (any matrix type) A matrix object.

SEE ALSO

matrix.new<type> matrix.columns matrix.rows

matrix.fill()

The function fills a rectangular area of the `id` matrix defined by the indices `from_column` to `to_column` (not including it) and `from_row` to `to_row` (not including it) with the `value`.

SYNTAX

```
matrix.fill(id, value, from_row, to_row, from_column, to_column) → void
```

ARGUMENTS

id (any matrix type) A matrix object.

value (series <type of the matrix's elements>) The value to fill with.

from_row (series int) Row index from which the fill will begin (inclusive). Optional. The default value is 0.

to_row (series int) Row index where the fill will end (not inclusive). Optional. The default value is matrix.rows.

from_column (series int) Column index from which the fill will begin (inclusive). Optional. The default value is 0.

to_column (series int) Column index where the fill will end (non inclusive). Optional. The default value is matrix.columns.

EXAMPLE

```
//@version=5
indicator("`matrix.fill()` Example")

// Create a 4x5 "int" matrix containing values `0`.
m = matrix.new<float>(4, 5, 0)

// Fill the intersection of rows 1 to 2 and columns 2 to 3 of the matrix with `hl2` values
matrix.fill(m, hl2, 0, 2, 1, 3)

// Display using a label.
if barstate.islastconfirmedhistory
    label.new(bar_index, high, str.tostring(m))
```

SEE ALSO

matrix.new<type>   matrix.get   matrix.set   matrix.columns   matrix.rows

matrix.get()

The function returns the element with the specified index of the matrix.

SYNTAX

```
matrix.get(id, row, column) → <matrix_type>
```

ARGUMENTS

id (any matrix type) A matrix object.

row (series int) Index of the required row.

column (series int) Index of the required column.

EXAMPLE

```
//@version=5
indicator("`matrix.get()` Example", "", true)

// Create a 2x3 "float" matrix from the `hl2` values.
m = matrix.new<float>(2, 3, hl2)

// Return the value of the element at index [0, 0] of matrix `m`.
x = matrix.get(m, 0, 0)

plot(x)
```

RETURNS

The value of the element at the `row` and `column` index of the `id` matrix.

REMARKS

Indexing of the rows and columns starts at zero.

SEE ALSO

matrix.new<type> matrix.set matrix.columns matrix.rows

matrix.inv() 2 overloads

The function returns the inverse of a square matrix.

SYNTAX & OVERLOADS

```
matrix.inv(id) → matrix<float>
```

```
matrix.inv(id) → matrix<int>
```

ARGUMENTS

id (matrix<int/float>) A matrix object.

EXAMPLE

```
//@version=5
indicator("`matrix.inv()` Example")

// For efficiency, execute this code only once.
if barstate.islastconfirmedhistory
    // Create a 2x2 matrix.
    var m1 = matrix.new<int>(2, 2, na)
    // Fill the matrix with values.
    matrix.set(m1, 0, 0, 1)
    matrix.set(m1, 0, 1, 2)
    matrix.set(m1, 1, 0, 3)
    matrix.set(m1, 1, 1, 4)

    // Inverse of the matrix.
    var m2 = matrix.inv(m1)

    // Display matrix elements.
    var t = table.new(position.top_right, 2, 2, color.green)
    table.cell(t, 0, 0, "Original Matrix:")
    table.cell(t, 0, 1, str.tostring(m1))
    table.cell(t, 1, 0, "Inverse matrix:")
    table.cell(t, 1, 1, str.tostring(m2))
```

RETURNS

A new matrix, which is the inverse of the `id` matrix.

REMARKS

The function is calculated using the LU decomposition algorithm.

SEE ALSO

matrix.new<type> matrix.set matrix.pinv matrix.copy str.tostring

matrix.is_antidiagonal()

The function determines if the matrix is anti-diagonal (all elements outside the secondary diagonal are zero).

SYNTAX

```
matrix.is_antidiagonal(id) → series bool
```

ARGUMENTS

id (matrix<int/float>) Matrix object to test.

RETURNS

Returns true if the `id` matrix is anti-diagonal, false otherwise.

REMARKS

Returns false with non-square matrices.

SEE ALSO

matrix.new<type> matrix.set matrix.is_square matrix.is_identity matrix.is_diagonal

matrix.is_antisymmetric()

The function determines if a matrix is antisymmetric (its transpose equals its negative).

SYNTAX

```
matrix.is_antisymmetric(id) → series bool
```

ARGUMENTS

id (matrix<int/float>) Matrix object to test.

RETURNS

Returns true, if the `id` matrix is antisymmetric, false otherwise.

REMARKS

Returns false with non-square matrices.

SEE ALSO

matrix.new<type> matrix.get matrix.set matrix.is_square

matrix.is_binary()

The function determines if the matrix is binary (when all elements of the matrix are 0 or 1).

SYNTAX

```
matrix.is_binary(id) → series bool
```

ARGUMENTS

id (matrix<int/float>) Matrix object to test.

RETURNS

Returns true if the `id` matrix is binary, false otherwise.

SEE ALSO

matrix.new<type> matrix.get matrix.set

matrix.is_diagonal()

The function determines if the matrix is diagonal (all elements outside the main diagonal are zero).

SYNTAX

```
matrix.is_diagonal(id) → series bool
```

ARGUMENTS

id (matrix<int/float>) Matrix object to test.

RETURNS

Returns true if the `id` matrix is diagonal, false otherwise.

REMARKS

Returns false with non-square matrices.

SEE ALSO

matrix.new<type> matrix.set matrix.is_square matrix.is_identity matrix.is_antidiagonal

matrix.is_identity()

The function determines if a matrix is an identity matrix (elements with ones on the main diagonal and zeros elsewhere).

SYNTAX

```
matrix.is_identity(id) → series bool
```

ARGUMENTS

id (matrix<int/float>) Matrix object to test.

RETURNS

Returns true if `id` is an identity matrix, false otherwise.

REMARKS

Returns false with non-square matrices.

SEE ALSO

matrix.new<type> matrix.is_square matrix.is_diagonal

matrix.is_square()

The function determines if the matrix is square (it has the same number of rows and columns).

SYNTAX

```
matrix.is_square(id) → series bool
```

ARGUMENTS

id (any matrix type) Matrix object to test.

RETURNS

Returns true if the `id` matrix is square, false otherwise.

SEE ALSO

`matrix.new<type>` `matrix.get` `matrix.set` `matrix.columns` `matrix.rows`

matrix.is_stochastic()

The function determines if the matrix is stochastic.

SYNTAX

```
matrix.is_stochastic(id) → series bool
```

ARGUMENTS

id (matrix<int/float>) Matrix object to test.

RETURNS

Returns true if the `id` matrix is stochastic, false otherwise.

SEE ALSO

`matrix.new<type>` `matrix.set`

matrix.is_symmetric()

The function determines if a square matrix is symmetric (elements are symmetric with respect to the main diagonal).

SYNTAX

```
matrix.is_symmetric(id) → series bool
```

ARGUMENTS

id (matrix<int/float>) Matrix object to test.

RETURNS

Returns true if the `id` matrix is symmetric, false otherwise.

REMARKS

Returns false with non-square matrices.

SEE ALSO

`matrix.new<type>` `matrix.get` `matrix.set` `matrix.is_square`

matrix.is_triangular()

The function determines if the matrix is triangular (if all elements above or below the main diagonal are zero).

SYNTAX

```
matrix.is_triangular(id) → series bool
```

ARGUMENTS

id (matrix<int/float>) Matrix object to test.

RETURNS

Returns true if the `id` matrix is triangular, false otherwise.

REMARKS

Returns false with non-square matrices.

SEE ALSO

`matrix.new<type>` `matrix.set` `matrix.is_square`

matrix.is_zero()

The function determines if all elements of the matrix are zero.

SYNTAX

```
matrix.is_zero(id) → series bool
```

ARGUMENTS

id (matrix<int/float>) Matrix object to check.

RETURNS

Returns true if all elements of the `id` matrix are zero, false otherwise.

SEE ALSO

matrix.new<type> matrix.get matrix.set

matrix.kron() 2 overloads

The function returns the Kronecker product for the `id1` and `id2` matrices.

SYNTAX & OVERLOADS

```
matrix.kron(id1, id2) → matrix<float>
```

```
matrix.kron(id1, id2) → matrix<int>
```

ARGUMENTS

id1 (matrix<int/float>) First matrix object.

id2 (matrix<int/float>) Second matrix object.

EXAMPLE

```
//@version=5
indicator("`matrix.kron()` Example")

// Display using a table.
if barstate.islastconfirmedhistory
    // Create two matrices with default values `1` and `2`.
    var m1 = matrix.new<float>(2, 2, 1)
    var m2 = matrix.new<float>(2, 2, 2)

    // Calculate the Kronecker product of the matrices.
    var m3 = matrix.kron(m1, m2)

    // Display matrix elements.
    var t = table.new(position.top_right, 5, 2, color.green)
    table.cell(t, 0, 0, "Matrix 1:")
    table.cell(t, 0, 1, str.tostring(m1))
    table.cell(t, 1, 1, "⊗")
    table.cell(t, 2, 0, "Matrix 2:")
    table.cell(t, 2, 1, str.tostring(m2))
    table.cell(t, 3, 1, "=")
    table.cell(t, 4, 0, "Kronecker product:")
    table.cell(t, 4, 1, str.tostring(m3))
```

RETURNS

A new matrix containing the Kronecker product of `id1` and `id2`.

SEE ALSO

matrix.new<type> matrix.mult str.tostring table.new

matrix.max() 2 overloads

The function returns the largest value from the matrix elements.

SYNTAX & OVERLOADS

```
matrix.max(id) → series float
```

```
matrix.max(id) → series int
```

ARGUMENTS

id (matrix<int/float>) A matrix object.

EXAMPLE

```
//@version=5
indicator("`matrix.max()` Example")

// Create a 2x2 matrix.
var m = matrix.new<int>(2, 2, na)
// Fill the matrix with values.
matrix.set(m, 0, 0, 1)
matrix.set(m, 0, 1, 2)
matrix.set(m, 1, 0, 3)
matrix.set(m, 1, 1, 4)

// Get the maximum value in the matrix.
var x = matrix.max(m)

plot(x, 'Matrix maximum value')
```

RETURNS

The maximum value from the `id` matrix.

SEE ALSO

matrix.new<type> matrix.min matrix.avg matrix.sort

matrix.median() 2 overloads

The function calculates the median ("the middle" value) of matrix elements.

SYNTAX & OVERLOADS

```
matrix.median(id) → series float
```

```
matrix.median(id) → series int
```

ARGUMENTS

id (matrix<int/float>) A matrix object.

EXAMPLE

```
//@version=5
indicator("`matrix.median()` Example")

// Create a 2x2 matrix.
m = matrix.new<int>(2, 2, na)
// Fill the matrix with values.
matrix.set(m, 0, 0, 1)
matrix.set(m, 0, 1, 2)
matrix.set(m, 1, 0, 3)
matrix.set(m, 1, 1, 4)

// Get the median of the matrix.
x = matrix.median(m)

plot(x, 'Median of the matrix')
```

REMARKS

Note that na elements of the matrix are not considered when calculating the median.

SEE ALSO

matrix.new<type> matrix.mode matrix.sort matrix.avg

matrix.min() 2 overloads

The function returns the smallest value from the matrix elements.

SYNTAX & OVERLOADS

```
matrix.min(id) → series float
```

```
matrix.min(id) → series int
```

ARGUMENTS

id (matrix<int/float>) A matrix object.

EXAMPLE

```
//@version=5
indicator("`matrix.min()` Example")

// Create a 2x2 matrix.
var m = matrix.new<int>(2, 2, na)
// Fill the matrix with values.
matrix.set(m, 0, 0, 1)
matrix.set(m, 0, 1, 2)
matrix.set(m, 1, 0, 3)
matrix.set(m, 1, 1, 4)

// Get the minimum value from the matrix.
var x = matrix.min(m)

plot(x, 'Matrix minimum value')
```

RETURNS

The smallest value from the `id` matrix.

SEE ALSO

matrix.new<type> matrix.max matrix.avg matrix.sort

matrix.mode() 2 overloads

The function calculates the mode of the matrix, which is the most frequently occurring value from the matrix elements. When there are multiple values occurring equally frequently, the function returns the smallest of those values.

SYNTAX & OVERLOADS

```
matrix.mode(id) → series float
matrix.mode(id) → series int
```

ARGUMENTS

id (matrix<int/float>) A matrix object.

EXAMPLE

```
//@version=5
indicator("`matrix.mode()` Example")

// Create a 2x2 matrix.
var m = matrix.new<int>(2, 2, na)
// Fill the matrix with values.
matrix.set(m, 0, 0, 0)
matrix.set(m, 0, 1, 0)
matrix.set(m, 1, 0, 1)
matrix.set(m, 1, 1, 1)

// Get the mode of the matrix.
var x = matrix.mode(m)

plot(x, 'Mode of the matrix')
```

RETURNS

The most frequently occurring value from the `id` matrix. If none exists, returns the smallest value instead.

REMARKS

Note that na elements of the matrix are not considered when calculating the mode.

SEE ALSO

matrix.new<type> matrix.set matrix.median matrix.sort matrix.avg

matrix.mult() 4 overloads

The function returns a new matrix resulting from the product between the matrices `id1` and `id2`, or between an `id1` matrix and an `id2` scalar (a numerical value), or between an `id1` matrix and an `id2` vector (an array of values).

SYNTAX & OVERLOADS

```
matrix.mult(id1, id2) → array<int>
matrix.mult(id1, id2) → array<float>
```

```
matrix.mult(id1, id2) → matrix<int>
```

```
matrix.mult(id1, id2) → matrix<float>
```

ARGUMENTS

id1 (matrix<int>) First matrix object.

id2 (array<int>) Second matrix object, value or array.

Product of two matrices

EXAMPLE

```
//@version=5
indicator("`matrix.mult()` Example 1")

// For efficiency, execute this code only once.
if barstate.islastconfirmedhistory
    // Create a 6x2 matrix containing values `5`.
    var m1 = matrix.new<float>(6, 2, 5)
    // Create a 2x3 matrix containing values `4`.
    // Note that it must have the same quantity of rows as there are columns in the first
    var m2 = matrix.new<float>(2, 3, 4)
    // Create a new matrix from the multiplication of the two matrices.
    var m3 = matrix.mult(m1, m2)

    // Display using a table.
    var t = table.new(position.top_right, 1, 2, color.green)
    table.cell(t, 0, 0, "Product of two matrices:")
    table.cell(t, 0, 1, str.tostring(m3))
```

Product of a matrix and a scalar

EXAMPLE

```
//@version=5
indicator("`matrix.mult()` Example 2")

// For efficiency, execute this code only once.
if barstate.islastconfirmedhistory
    // Create a 2x3 matrix containing values `4`.
    var m1 = matrix.new<float>(2, 3, 4)

    // Create a new matrix from the product of the two matrices.
    scalar = 5
    var m2 = matrix.mult(m1, scalar)

    // Display using a table.
    var t = table.new(position.top_right, 5, 2, color.green)
    table.cell(t, 0, 0, "Matrix 1:")
    table.cell(t, 0, 1, str.tostring(m1))
    table.cell(t, 1, 1, "x")
    table.cell(t, 2, 0, "Scalar:")
    table.cell(t, 2, 1, str.tostring(scalar))
    table.cell(t, 3, 1, "=")
    table.cell(t, 4, 0, "Matrix 2:")
    table.cell(t, 4, 1, str.tostring(m2))
```

Product of a matrix and an array vector

EXAMPLE

```
//@version=5
indicator("`matrix.mult()` Example 3")

// For efficiency, execute this code only once.
if barstate.islastconfirmedhistory
    // Create a 2x3 matrix containing values `4`.
    var m1 = matrix.new<int>(2, 3, 4)

    // Create an array of three elements.
    var int[] a = array.from(1, 1, 1)

    // Create a new matrix containing the product of the `m1` matrix and the `a` array.
    var m3 = matrix.mult(m1, a)

    // Display using a table.
    var t = table.new(position.top_right, 5, 2, color.green)
    table.cell(t, 0, 0, "Matrix 1:")
    table.cell(t, 0, 1, str.tostring(m1))
    table.cell(t, 1, 1, "x")
    table.cell(t, 2, 0, "Value:")
    table.cell(t, 2, 1, str.tostring(a, " "))
    table.cell(t, 3, 1, "=")
    table.cell(t, 4, 0, "Matrix 3:")
    table.cell(t, 4, 1, str.tostring(m3))
```

RETURNS

A new matrix object containing the product of `id2` and `id1`.

SEE ALSO

matrix.new<type> matrix.sum matrix.diff

matrix.new<type>()

The function creates a new matrix object. A matrix is a two-dimensional data structure containing rows and columns. All elements in the matrix must be of the type specified in the type template ("<type>").

SYNTAX

```
matrix.new<type>(rows, columns, initial_value) → matrix<type>
```

ARGUMENTS

rows (series int) Initial row count of the matrix. Optional. The default value is 0.

columns (series int) Initial column count of the matrix. Optional. The default value is 0.

initial_value (<matrix_type>) Initial value of all matrix elements. Optional. The default is 'na'.

Create a matrix of elements with the same initial value

EXAMPLE

```
//@version=5
indicator("`matrix.new<type>()` Example 1")

// Create a 2x3 (2 rows x 3 columns) "int" matrix with values zero.
var m = matrix.new<int>(2, 3, 0)

// Display using a label.
if barstate.islastconfirmedhistory
    label.new(bar_index, high, str.tostring(m))
```

Create a matrix from array values

EXAMPLE

```
//@version=5
indicator("`matrix.new<type>()` Example 2")

// Function to create a matrix whose rows are filled with array values.
matrixFromArray(int rows, int columns, array<float> data) =>
    m = matrix.new<float>(rows, columns)
    for i = 0 to rows <= 0 ? na : rows - 1
        for j = 0 to columns <= 0 ? na : columns - 1
            matrix.set(m, i, j, array.get(data, i * columns + j))
    m
```

```
// Create a 3x3 matrix from an array of values.
var m1 = matrixFromArray(3, 3, array.from(1, 2, 3, 4, 5, 6, 7, 8, 9))
// Display using a label.
if barstate.islastconfirmedhistory
    label.new(bar_index, high, str.tostring(m1))
```

Create a matrix from an `input.text_area()` field

EXAMPLE

```
//@version=5
indicator("`matrix.new<type>()` Example 3")

// Function to create a matrix from a text string.
// Values in a row must be separated by a space. Each line is one row.
matrixFromInputArea(stringOfValues) =>
    var rowsArray = str.split(stringOfValues, "\n")
    var rows = array.size(rowsArray)
    var cols = array.size(str.split(array.get(rowsArray, 0), " "))
    var matrix = matrix.new<float>(rows, cols, na)
    row = 0
    for rowString in rowsArray
        col = 0
        values = str.split(rowString, " ")
        for val in values
            matrix.set(matrix, row, col, str.tonumber(val))
            col += 1
        row += 1
    matrix


stringInput = input.text_area("1 2 3\n4 5 6\n7 8 9")
var m = matrixFromInputArea(stringInput)

// Display using a label.
if barstate.islastconfirmedhistory
    label.new(bar_index, high, str.tostring(m))
```

Create matrix from random values

EXAMPLE

```
//@version=5
indicator("`matrix.new<type>()` Example 4")

// Function to create a matrix with random values (0.0 to 1.0).
matrixRandom(int rows, int columns)=>
    result = matrix.new<float>(rows, columns)
    for i = 0 to rows - 1
```

```
        for j = 0 to columns - 1
            matrix.set(result, i, j, math.random())
    result

// Create a 2x3 matrix with random values.
var m = matrixRandom(2, 3)

// Display using a label.
if barstate.islastconfirmedhistory
    label.new(bar_index, high, str.tostring(m))
```

RETURNS

The ID of the new matrix object.

SEE ALSO

matrix.set matrix.fill matrix.columns matrix.rows array.new<type>

matrix.pinv() 2 overloads

The function returns the pseudoinverse of a matrix.

SYNTAX & OVERLOADS

```
matrix.pinv(id) → matrix<float>
```

```
matrix.pinv(id) → matrix<int>
```

ARGUMENTS

id (matrix<int/float>) A matrix object.

EXAMPLE

```
//@version=5
indicator("`matrix.pinv()` Example")

// For efficiency, execute this code only once.
if barstate.islastconfirmedhistory
    // Create a 2x2 matrix.
    var m1 = matrix.new<int>(2, 2, na)
    // Fill the matrix with values.
    matrix.set(m1, 0, 0, 1)
    matrix.set(m1, 0, 1, 2)
    matrix.set(m1, 1, 0, 3)
    matrix.set(m1, 1, 1, 4)
    // Pseudoinverse of the matrix.
    var m2 = matrix.pinv(m1)

    // Display matrix elements.
    var t = table.new(position.top_right, 2, 2, color.green)
    table.cell(t, 0, 0, "Original Matrix:")
    table.cell(t, 0, 1, str.tostring(m1))
    table.cell(t, 1, 0, "Pseudoinverse matrix:")
    table.cell(t, 1, 1, str.tostring(m2))
```

RETURNS

A new matrix containing the pseudoinverse of the `id` matrix.

REMARKS

The function is calculated using a Moore-Penrose inverse formula based on singular-value decomposition of a matrix. For non-singular square matrices this function returns the result of matrix.inv.

SEE ALSO

matrix.new<type> matrix.set matrix.inv

matrix.pow() 2 overloads

The function calculates the product of the matrix by itself `power` times.

SYNTAX & OVERLOADS

```
matrix.pow(id, power) → matrix<float>
```

```
matrix.pow(id, power) → matrix<int>
```

ARGUMENTS

id (matrix<int/float>) A matrix object.

power (series int) The number of times the matrix will be multiplied by itself.

EXAMPLE

```
//@version=5
indicator("`matrix.pow()` Example")

// Display using a table.
if barstate.islastconfirmedhistory
```

```
    // Create a 2x2 matrix.
    var m1 = matrix.new<int>(2, 2, 2)
    // Calculate the power of three of the matrix.
    var m2 = matrix.pow(m1, 3)

    // Display matrix elements.
    var t = table.new(position.top_right, 2, 2, color.green)
    table.cell(t, 0, 0, "Original Matrix:")
    table.cell(t, 0, 1, str.tostring(m1))
    table.cell(t, 1, 0, "Matrix³:")
    table.cell(t, 1, 1, str.tostring(m2))
```

RETURNS

The product of the `id` matrix by itself `power` times.

SEE ALSO

matrix.new<type>   matrix.set   matrix.mult

matrix.rank()

The function calculates the rank of the matrix.

SYNTAX

```
matrix.rank(id) → series int
```

ARGUMENTS

id (any matrix type) A matrix object.

EXAMPLE

```
//@version=5
indicator("`matrix.rank()` Example")

// For efficiency, execute this code only once.
if barstate.islastconfirmedhistory
    // Create a 2x2 matrix.
    var m1 = matrix.new<int>(2, 2, na)
    // Fill the matrix with values.
    matrix.set(m1, 0, 0, 1)
    matrix.set(m1, 0, 1, 2)
    matrix.set(m1, 1, 0, 3)
    matrix.set(m1, 1, 1, 4)

    // Get the rank of the matrix.
    r = matrix.rank(m1)
    // Display matrix elements.
    var t = table.new(position.top_right, 2, 2, color.green)
    table.cell(t, 0, 0, "Matrix elements:")
    table.cell(t, 0, 1, str.tostring(m1))
    table.cell(t, 1, 0, "Rank of the matrix:")
    table.cell(t, 1, 1, str.tostring(r))
```

RETURNS

The rank of the `id` matrix.

SEE ALSO

matrix.new<type>   matrix.set   str.tostring

matrix.remove_col()

The function removes the column at `column` index of the `id` matrix and returns an array containing the removed column's values.

SYNTAX

```
matrix.remove_col(id, column) → array<type>
```

ARGUMENTS

id (any matrix type) A matrix object.

column (series int) The index of the column to be removed. Optional. The default value is matrix.columns.

EXAMPLE

```
//@version=5
indicator("matrix_remove_col", overlay = true)

// Create a 2x2 matrix with ones.
var matrixOrig = matrix.new<int>(2, 2, 1)

// Set values to the 'matrixOrig' matrix.
matrix.set(matrixOrig, 0, 1, 2)
matrix.set(matrixOrig, 1, 0, 3)
matrix.set(matrixOrig, 1, 1, 4)


// Create a copy of the 'matrixOrig' matrix.
matrixCopy = matrix.copy(matrixOrig)
```

```
    // Remove the first column from the `matrixCopy` matrix.
    arr = matrix.remove_col(matrixCopy, 0)

    // Display matrix elements.
    if barstate.islastconfirmedhistory
        var t = table.new(position.top_right, 3, 2, color.green)
        table.cell(t, 0, 0, "Original Matrix:")
        table.cell(t, 0, 1, str.tostring(matrixOrig))
        table.cell(t, 1, 0, "Removed Elements:")
        table.cell(t, 1, 1, str.tostring(arr))
        table.cell(t, 2, 0, "Result Matrix:")
        table.cell(t, 2, 1, str.tostring(matrixCopy))
```

RETURNS

An array containing the elements of the column removed from the `id` matrix.

REMARKS

Indexing of rows and columns starts at zero. It is far more efficient to declare matrices with explicit dimensions than to build them by adding or removing columns. Deleting a column is also much slower than deleting a row with the matrix.remove_row function.

SEE ALSO

matrix.new<type> matrix.set matrix.copy matrix.remove_row

matrix.remove_row()

The function removes the row at `row` index of the `id` matrix and returns an array containing the removed row's values.

SYNTAX

```
matrix.remove_row(id, row) → array<type>
```

ARGUMENTS

id (any matrix type) A matrix object.

row (series int) The index of the row to be deleted. Optional. The default value is matrix.rows.

EXAMPLE

```
    //@version=5
    indicator("matrix_remove_row", overlay = true)
```

```
// Create a 2x2 "int" matrix containing values `1`.
var matrixOrig = matrix.new<int>(2, 2, 1)

// Set values to the 'matrixOrig' matrix.
matrix.set(matrixOrig, 0, 1, 2)
matrix.set(matrixOrig, 1, 0, 3)
matrix.set(matrixOrig, 1, 1, 4)

// Create a copy of the 'matrixOrig' matrix.
matrixCopy = matrix.copy(matrixOrig)

// Remove the first row from the matrix `matrixCopy`.
arr = matrix.remove_row(matrixCopy, 0)

// Display matrix elements.
if barstate.islastconfirmedhistory
    var t = table.new(position.top_right, 3, 2, color.green)
    table.cell(t, 0, 0, "Original Matrix:")
    table.cell(t, 0, 1, str.tostring(matrixOrig))
    table.cell(t, 1, 0, "Removed Elements:")
    table.cell(t, 1, 1, str.tostring(arr))
    table.cell(t, 2, 0, "Result Matrix:")
    table.cell(t, 2, 1, str.tostring(matrixCopy))
```

RETURNS

An array containing the elements of the row removed from the `id` matrix.

REMARKS

Indexing of rows and columns starts at zero. It is far more efficient to declare matrices with explicit dimensions than to build them by adding or removing rows.

SEE ALSO

matrix.new<type> matrix.set matrix.copy matrix.remove_col

matrix.reshape()

The function rebuilds the `id` matrix to `rows` x `cols` dimensions.

SYNTAX

```
matrix.reshape(id, rows, columns) → void
```

ARGUMENTS

id (any matrix type) A matrix object.

rows (series int) The number of rows of the reshaped matrix.

columns (series int) The number of columns of the reshaped matrix.

EXAMPLE

```
//@version=5
indicator("`matrix.reshape()` Example")

// For efficiency, execute this code only once.
if barstate.islastconfirmedhistory
    // Create a 2x3 matrix.
    var m1 = matrix.new<float>(2, 3)
    // Fill the matrix with values.
    matrix.set(m1, 0, 0, 1)
    matrix.set(m1, 0, 1, 2)
    matrix.set(m1, 0, 2, 3)
    matrix.set(m1, 1, 0, 4)
    matrix.set(m1, 1, 1, 5)
    matrix.set(m1, 1, 2, 6)

    // Copy the matrix to a new one.
    var m2 = matrix.copy(m1)

    // Reshape the copy to a 3x2.
    matrix.reshape(m2, 3, 2)

    // Display using a table.
    var t = table.new(position.top_right, 2, 2, color.green)
    table.cell(t, 0, 0, "Original matrix:")
    table.cell(t, 0, 1, str.tostring(m1))
    table.cell(t, 1, 0, "Reshaped matrix:")
    table.cell(t, 1, 1, str.tostring(m2))
```

SEE ALSO

matrix.new<type> matrix.get matrix.set matrix.add_row matrix.add_col

matrix.reverse()

The function reverses the order of rows and columns in the matrix `id`. The first row and first column become the last, and the last become the first.

SYNTAX

```
matrix.reverse(id) → void
```

ARGUMENTS

id (any matrix type) A matrix object.

EXAMPLE

```
//@version=5
indicator("`matrix.reverse()` Example")

// For efficiency, execute this code only once.
if barstate.islastconfirmedhistory
    // Copy the matrix to a new one.
    var m1 = matrix.new<int>(2, 2, na)
    // Fill the matrix with values.
    matrix.set(m1, 0, 0, 1)
    matrix.set(m1, 0, 1, 2)
    matrix.set(m1, 1, 0, 3)
    matrix.set(m1, 1, 1, 4)

    // Copy matrix elements to a new matrix.
    var m2 = matrix.copy(m1)

    // Reverse the `m2` copy of the original matrix.
    matrix.reverse(m2)

    // Display using a table.
    var t = table.new(position.top_right, 2, 2, color.green)
    table.cell(t, 0, 0, "Original matrix:")
    table.cell(t, 0, 1, str.tostring(m1))
    table.cell(t, 1, 0, "Reversed matrix:")
    table.cell(t, 1, 1, str.tostring(m2))
```

SEE ALSO

matrix.new<type> matrix.set matrix.columns matrix.rows matrix.reshape

matrix.row()

The function creates a one-dimensional array from the elements of a matrix row.

SYNTAX

```
matrix.row(id, row) → array<type>
```

ARGUMENTS

id (any matrix type) A matrix object.

row (series int) Index of the required row.

EXAMPLE

```
//@version=5
indicator("`matrix.row()` Example", "", true)

// Create a 2x3 "float" matrix from `hlc3` values.
m = matrix.new<float>(2, 3, hlc3)

// Return an array with the values of the first row of the matrix.
a = matrix.row(m, 0)

// Plot the first value from the array `a`.
plot(array.get(a, 0))
```

RETURNS

An array ID containing the `row` values of the `id` matrix.

REMARKS

Indexing of rows starts at 0.

SEE ALSO

matrix.new<type> matrix.get array.get matrix.col matrix.rows

matrix.rows()

The function returns the number of rows in the matrix.

SYNTAX

```
matrix.rows(id) → series int
```

ARGUMENTS

id (any matrix type) A matrix object.

EXAMPLE

```
//@version=5
indicator("`matrix.rows()` Example")

// Create a 2x6 matrix with values `0`.
var m = matrix.new<int>(2, 6, 0)

// Get the quantity of rows in the matrix.
var x = matrix.rows(m)

// Display using a label.
```

```
if barstate.islastconfirmedhistory
    label.new(bar_index, high, "Rows: " + str.tostring(x) + "\n" + str.tostring(m))
```

RETURNS

The number of rows in the matrix `id`.

SEE ALSO

matrix.new<type>   matrix.get   matrix.set   matrix.columns   matrix.row

matrix.set()

The function assigns `value` to the element at the `row` and `column` of the `id` matrix.

SYNTAX

```
matrix.set(id, row, column, value) → void
```

ARGUMENTS

id (any matrix type) A matrix object.

row (series int) The row index of the element to be modified.

column (series int) The column index of the element to be modified.

value (series <type of the matrix's elements>) The new value to be set.

EXAMPLE

```
//@version=5
indicator("`matrix.set()` Example")

// Create a 2x3 "int" matrix containing values `4`.
m = matrix.new<int>(2, 3, 4)

// Replace the value of element at row 1 and column 2 with value `3`.
matrix.set(m, 0, 1, 3)

// Display using a label.
if barstate.islastconfirmedhistory
    label.new(bar_index, high, str.tostring(m))
```

SEE ALSO

matrix.new<type>   matrix.get   matrix.columns   matrix.rows

matrix.sort()

The function rearranges the rows in the `id` matrix following the sorted order of the values in the `column`.

SYNTAX

```
matrix.sort(id, column, order) → void
```

ARGUMENTS

id (matrix<int/float/string>) A matrix object to be sorted.

column (series int) Index of the column whose sorted values determine the new order of rows. Optional. The default value is 0.

order (series sort_order) The sort order. Possible values: order.ascending (default), order.descending.

EXAMPLE

```
//@version=5
indicator("`matrix.sort()` Example")

// For efficiency, execute this code only once.
if barstate.islastconfirmedhistory
    // Create a 2x2 matrix.
    var m1 = matrix.new<float>(2, 2, na)
    // Fill the matrix with values.
    matrix.set(m1, 0, 0, 3)
    matrix.set(m1, 0, 1, 4)
    matrix.set(m1, 1, 0, 1)
    matrix.set(m1, 1, 1, 2)

    // Copy the matrix to a new one.
    var m2 = matrix.copy(m1)
    // Sort the rows of `m2` using the default arguments (first column and ascending order
    matrix.sort(m2)

    // Display using a table.
    if barstate.islastconfirmedhistory
        var t = table.new(position.top_right, 2, 2, color.green)
        table.cell(t, 0, 0, "Original matrix:")
        table.cell(t, 0, 1, str.tostring(m1))
        table.cell(t, 1, 0, "Sorted matrix:")
        table.cell(t, 1, 1, str.tostring(m2))
```

SEE ALSO

matrix.new<type> matrix.max matrix.min matrix.avg

matrix.submatrix()

The function extracts a submatrix of the `id` matrix within the specified indices.

SYNTAX

```
matrix.submatrix(id, from_row, to_row, from_column, to_column) → matrix<type>
```

ARGUMENTS

id (any matrix type) A matrix object.

from_row (series int) Index of the row from which the extraction will begin (inclusive). Optional. The default value is 0.

to_row (series int) Index of the row where the extraction will end (non inclusive). Optional. The default value is matrix.rows.

from_column (series int) Index of the column from which the extraction will begin (inclusive). Optional. The default value is 0.

to_column (series int) Index of the column where the extraction will end (non inclusive). Optional. The default value is matrix.columns.

EXAMPLE

```
//@version=5
indicator("`matrix.submatrix()` Example")

// For efficiency, execute this code only once.
if barstate.islastconfirmedhistory
    // Create a 2x3 matrix matrix with values `0`.
    var m1 = matrix.new<int>(2, 3, 0)
    // Fill the matrix with values.
    matrix.set(m1, 0, 0, 1)
    matrix.set(m1, 0, 1, 2)
    matrix.set(m1, 0, 2, 3)
    matrix.set(m1, 1, 0, 4)
    matrix.set(m1, 1, 1, 5)
    matrix.set(m1, 1, 2, 6)

    // Create a 2x2 submatrix of the `m1` matrix.
    var m2 = matrix.submatrix(m1, 0, 2, 1, 3)

    // Display using a table.
    var t = table.new(position.top_right, 2, 2, color.green)
    table.cell(t, 0, 0, "Original Matrix:")
    table.cell(t, 0, 1, str.tostring(m1))
    table.cell(t, 1, 0, "Submatrix:")
    table.cell(t, 1, 1, str.tostring(m2))
```

RETURNS

A new matrix object containing the submatrix of the `id` matrix defined by the `from_row`, `to_row`, `from_column` and `to_column` indices.

REMARKS

Indexing of the rows and columns starts at zero.

SEE ALSO

matrix.new<type> matrix.set matrix.row matrix.col matrix.reshape

matrix.sum() 2 overloads

The function returns a new matrix resulting from the sum of two matrices `id1` and `id2`, or of an `id1` matrix and an `id2` scalar (a numerical value).

SYNTAX & OVERLOADS

```
matrix.sum(id1, id2) → matrix<int>
```

```
matrix.sum(id1, id2) → matrix<float>
```

ARGUMENTS

id1 (matrix<int>) First matrix object.

id2 (series int/float/matrix<int>) Second matrix object, or scalar value.

Sum of two matrices

EXAMPLE

```
//@version=5
indicator("`matrix.sum()` Example 1")

// For efficiency, execute this code only once.
if barstate.islastconfirmedhistory
    // Create a 2x3 matrix containing values `5`.
    var m1 = matrix.new<float>(2, 3, 5)
    // Create a 2x3 matrix containing values `4`.
    var m2 = matrix.new<float>(2, 3, 4)
    // Create a new matrix that sums matrices `m1` and `m2`.
    var m3 = matrix.sum(m1, m2)

    // Display using a table.
    var t = table.new(position.top_right, 1, 2, color.green)
    table.cell(t, 0, 0, "Sum of two matrices:")
    table.cell(t, 0, 1, str.tostring(m3))
```

Sum of a matrix and scalar

EXAMPLE

```
//@version=5
indicator("`matrix.sum()` Example 2")

// For efficiency, execute this code only once.
if barstate.islastconfirmedhistory
    // Create a 2x3 matrix with values `4`.
    var m1 = matrix.new<float>(2, 3, 4)

    // Create a new matrix containing the sum of the `m1` matrix with the "int" value `1`.
    var m2 = matrix.sum(m1, 1)

    // Display using a table.
    var t = table.new(position.top_right, 1, 2, color.green)
    table.cell(t, 0, 0, "Sum of a matrix and a scalar:")
    table.cell(t, 0, 1, str.tostring(m2))
```

RETURNS

A new matrix object containing the sum of `id2` and `id1`.

SEE ALSO

matrix.new<type> matrix.get matrix.set matrix.columns matrix.rows

matrix.swap_columns()

The function swaps the columns at the index `column1` and `column2` in the `id` matrix.

SYNTAX

```
matrix.swap_columns(id, column1, column2) → void
```

ARGUMENTS

id (any matrix type) A matrix object.

column1 (series int) Index of the first column to be swapped.

column2 (series int) Index of the second column to be swapped.

EXAMPLE

```
//@version=5
indicator("`matrix.swap_columns()` Example")

// For efficiency, execute this code only once.
if barstate.islastconfirmedhistory
    // Create a 2x2 matrix with 'na' values.
    var m1 = matrix.new<int>(2, 2, na)
    // Fill the matrix with values.
    matrix.set(m1, 0, 0, 1)
    matrix.set(m1, 0, 1, 2)
    matrix.set(m1, 1, 0, 3)
    matrix.set(m1, 1, 1, 4)

    // Copy the matrix to a new one.
    var m2 = matrix.copy(m1)

    // Swap the first and second columns of the matrix copy.
    matrix.swap_columns(m2, 0, 1)

    // Display using a table.
    var t = table.new(position.top_right, 2, 2, color.green)
    table.cell(t, 0, 0, "Original matrix:")
    table.cell(t, 0, 1, str.tostring(m1))
    table.cell(t, 1, 0, "Swapped columns in copy:")
    table.cell(t, 1, 1, str.tostring(m2))
```

REMARKS

Indexing of the rows and columns starts at zero.

SEE ALSO

matrix.new<type> matrix.get matrix.set matrix.columns matrix.rows

matrix.swap_rows()

The function swaps the rows at the index `row1` and `row2` in the `id` matrix.

SYNTAX

```
matrix.swap_rows(id, row1, row2) → void
```

ARGUMENTS

id (any matrix type) A matrix object.

row1 (series int) Index of the first row to be swapped.

row2 (series int) Index of the second row to be swapped.

EXAMPLE

```
//@version=5
indicator("`matrix.swap_rows()` Example")

// For efficiency, execute this code only once.
if barstate.islastconfirmedhistory
    // Create a 3x2 matrix with ‘na’ values.
    var m1 = matrix.new<int>(3, 2, na)
    // Fill the matrix with values.
    matrix.set(m1, 0, 0, 1)
    matrix.set(m1, 0, 1, 2)
    matrix.set(m1, 1, 0, 3)
    matrix.set(m1, 1, 1, 4)
    matrix.set(m1, 2, 0, 5)
    matrix.set(m1, 2, 1, 6)

    // Copy the matrix to a new one.
    var m2 = matrix.copy(m1)

    // Swap the first and second rows of the matrix copy.
    matrix.swap_rows(m2, 0, 1)

    // Display using a table.
    var t = table.new(position.top_right, 2, 2, color.green)
    table.cell(t, 0, 0, "Original matrix:")
    table.cell(t, 0, 1, str.tostring(m1))
    table.cell(t, 1, 0, "Swapped rows in copy:")
    table.cell(t, 1, 1, str.tostring(m2))
```

REMARKS

Indexing of the rows and columns starts at zero.

SEE ALSO

matrix.new<type> matrix.get matrix.set matrix.columns matrix.swap_columns

matrix.trace() 2 overloads

The function calculates the trace of a matrix (the sum of the main diagonal's elements).

SYNTAX & OVERLOADS

```
matrix.trace(id) → series float
```

```
matrix.trace(id) → series int
```

ARGUMENTS

id (matrix<int/float>) A matrix object.

EXAMPLE

```
//@version=5
indicator("`matrix.trace()` Example")

// For efficiency, execute this code only once.
if barstate.islastconfirmedhistory
    // Create a 2x2 matrix.
    var m1 = matrix.new<int>(2, 2, na)
    // Fill the matrix with values.
    matrix.set(m1, 0, 0, 1)
    matrix.set(m1, 0, 1, 2)
    matrix.set(m1, 1, 0, 3)
    matrix.set(m1, 1, 1, 4)

    // Get the trace of the matrix.
    tr = matrix.trace(m1)

    // Display matrix elements.
    var t = table.new(position.top_right, 2, 2, color.green)
    table.cell(t, 0, 0, "Matrix elements:")
    table.cell(t, 0, 1, str.tostring(m1))
    table.cell(t, 1, 0, "Trace of the matrix:")
    table.cell(t, 1, 1, str.tostring(tr))
```

RETURNS

The trace of the `id` matrix.

SEE ALSO

matrix.new<type> matrix.get matrix.set matrix.columns matrix.rows

matrix.transpose()

The function creates a new, transposed version of the `id`. This interchanges the row and column index of each element.

SYNTAX

```
matrix.transpose(id) → matrix<type>
```

ARGUMENTS

id (any matrix type) A matrix object.

EXAMPLE

```
//@version=5
indicator("`matrix.transpose()` Example")

// For efficiency, execute this code only once.
if barstate.islastconfirmedhistory
    // Create a 2x2 matrix.
    var m1 = matrix.new<float>(2, 2, na)
    // Fill the matrix with values.
    matrix.set(m1, 0, 0, 1)
    matrix.set(m1, 0, 1, 2)
    matrix.set(m1, 1, 0, 3)
    matrix.set(m1, 1, 1, 4)

    // Create a transpose of the matrix.
    var m2 = matrix.transpose(m1)

    // Display using a table.
    var t = table.new(position.top_right, 2, 2, color.green)
    table.cell(t, 0, 0, "Original matrix:")
    table.cell(t, 0, 1, str.tostring(m1))
    table.cell(t, 1, 0, "Transposed matrix:")
    table.cell(t, 1, 1, str.tostring(m2))
```

RETURNS

A new matrix containing the transposed version of the `id` matrix.

SEE ALSO

matrix.new<type> matrix.set matrix.columns matrix.rows matrix.reshape

matrix.reverse

max_bars_back()

Function sets the maximum number of bars that is available for historical reference of a given built-in or user variable. When operator '[]' is applied to a variable - it is a reference to a historical value of that variable.

If an argument of an operator '[]' is a compile time constant value (e.g. 'v[10]', 'close[500]') then there is no need to use 'max_bars_back' function for that variable. Pine Script® compiler will use that constant value as history buffer size.

If an argument of an operator '[]' is a value, calculated at runtime (e.g. 'v[i]' where 'i' - is a series variable) then Pine Script® attempts to autodetect the history buffer size at runtime. Sometimes it fails and the script crashes at runtime because it eventually refers to historical values that are out of the buffer. In that case you should use 'max_bars_back' to fix that problem manually.

SYNTAX

```
max_bars_back(var, num) → void
```

ARGUMENTS

var (series int/float/bool/color/label/line) Series variable identifier for which history buffer should be resized. Possible values are: 'open', 'high', 'low', 'close', 'volume', 'time', or any user defined variable id.

num (const int) History buffer size which is the number of bars that could be referenced for variable 'var'.

EXAMPLE

```
//@version=5
indicator("max_bars_back")
close_() => close
depth() => 400
d = depth()
v = close_()
max_bars_back(v, 500)
out = if bar_index > 0
    v[d]
else
    v
plot(out)
```

RETURNS

void

REMARKS

At the moment 'max_bars_back' cannot be applied to built-ins like 'hl2', 'hlc3', 'ohlc4'. Please use multiple 'max_bars_back' calls as workaround here (e.g. instead of a single ‘max_bars_back(hl2, 100)’ call you should call the function twice: ‘max_bars_back(high, 100), max_bars_back(low, 100)’).

If the indicator or strategy 'max_bars_back' parameter is used, all variables in the indicator are affected. This may result in excessive memory usage and cause runtime problems. When possible (i.e. when the cause is a variable rather than a function), please use the max_bars_back function instead.

SEE ALSO

indicator

minute() 2 overloads

SYNTAX & OVERLOADS

```
minute(time) → series int
```

```
minute(time, timezone) → series int
```

ARGUMENTS

time (series int) UNIX time in milliseconds.

RETURNS

Minute (in exchange timezone) for provided UNIX time.

REMARKS

UNIX time is the number of milliseconds that have elapsed since 00:00:00 UTC, 1 January 1970.

SEE ALSO

minute time year month dayofmonth dayofweek hour second

month() 2 overloads

SYNTAX & OVERLOADS

```
month(time) → series int
```

```
month(time, timezone) → series int
```

ARGUMENTS

time (series int) UNIX time in milliseconds.

RETURNS

Month (in exchange timezone) for provided UNIX time.

REMARKS

UNIX time is the number of milliseconds that have elapsed since 00:00:00 UTC, 1 January 1970.

Note that this function returns the month based on the time of the bar's open. For overnight sessions (e.g. EURUSD, where Monday session starts on Sunday, 17:00 UTC-4) this value can be lower by 1 than the month of the trading day.

SEE ALSO

month time year dayofmonth dayofweek hour minute second

na() 2 overloads

Tests if `x` is na.

SYNTAX & OVERLOADS

```
na(x) → series bool
```

```
na(x) → simple bool
```

ARGUMENTS

x (<arg_type>) Value to be tested.

EXAMPLE

```
//@version=5
indicator("na")
// Use the `na()` function to test for `na`.
plot(na(close[1]) ? close : close[1])
// ALTERNATIVE
// `nz()` also tests `close[1]` for `na`. It returns `close[1]` if it is not `na`, and `cl
plot(nz(close[1], close))
```

RETURNS

Returns true if `x` is na, false otherwise.

SEE ALSO

na fixnan nz

nz() 16 overloads

Replaces NaN values with zeros (or given value) in a series.

SYNTAX & OVERLOADS

```
nz(source) → simple color
```

```
nz(source) → simple int
```

```
nz(source) → series color
```

```
nz(source) → series int
```

```
nz(source) → simple float
```

```
nz(source) → series float
```

```
nz(source, replacement) → simple color
```

```
nz(source, replacement) → simple int
```

```
nz(source, replacement) → series color
```

```
nz(source, replacement) → series int
```

```
nz(source, replacement) → simple float
```

```
nz(source, replacement) → series float
nz(source) → simple bool
```

```
nz(source) → series bool
```

```
nz(source, replacement) → simple bool
```

```
nz(source, replacement) → series bool
```

ARGUMENTS

source (simple color) Series of values to process.

EXAMPLE

```
//@version=5
indicator("nz", overlay=true)
plot(nz(ta.sma(close, 100)))
```

RETURNS

The value of `source` if it is not `na`. If the value of `source` is `na`, returns zero, or the `replacement` argument when one is used.

SEE ALSO

na na fixnan

plot()

Plots a series of data on the chart.

SYNTAX

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

ARGUMENTS

series (series int/float) Series of data to be plotted. Required argument.

title (const string) Title of the plot.

color (series color) Color of the plot. You can use constants like 'color=color.red' or 'color=#ff001a' as well as complex expressions like 'color = close >= open ? color.green :

color.red'. Optional argument.

linewidth (input int) Width of the plotted line. Default value is 1. Not applicable to every style.

style (input plot_style) Type of plot. Possible values are: plot.style_line, plot.style_stepline, plot.style_stepline_diamond, plot.style_histogram, plot.style_cross, plot.style_area, plot.style_columns, plot.style_circles, plot.style_linebr, plot.style_areabr, plot.style_steplinebr. Default value is plot.style_line.

trackprice (input bool) If true then a horizontal price line will be shown at the level of the last indicator value. Default is false.

histbase (input int/float) The price value used as the reference level when rendering plot with plot.style_histogram, plot.style_columns or plot.style_area style. Default is 0.0.

offset (series int) Shifts the plot to the left or to the right on the given number of bars. Default is 0.

join (input bool) If true then plot points will be joined with line, applicable only to plot.style_cross and plot.style_circles styles. Default is false.

editable (const bool) If true then plot style will be editable in Format dialog. Default is true.

show_last (input int) If set, defines the number of bars (from the last bar back to the past) to plot on chart.

display (input plot_display) Controls where the plot's information is displayed. Display options support addition and subtraction, meaning that using `display.all - display.status_line` will display the plot's information everywhere except in the script's status line. `display.price_scale + display.status_line` will display the plot only in the price scale and status line. When `display` arguments such as `display.price_scale` have user-controlled chart settings equivalents, the relevant plot information will only appear when all settings allow for it. Possible values: display.none, display.pane, display.data_window, display.price_scale, display.status_line, display.all. Optional. The default is display.all.

format (input string) Determines whether the script formats the plot's values as prices, percentages, or volume values. The argument passed to this parameter supersedes the `format` parameter of the indicator, and strategy functions. Optional. The default is the `format` value used by the indicator/strategy function. Possible values: format.price, format.percent, format.volume.

precision (input int) The number of digits after the decimal point the plot's values show on the chart pane's y-axis, the script's status line, and the Data Window. Accepts a non-negative integer less than or equal to 16. The argument passed to this parameter supersedes the `precision` parameter of the indicator and strategy functions. When the function's `format` parameter uses format.volume, the `precision` parameter will not affect the result, as the decimal precision rules defined by format.volume supersede other precision settings. Optional. The default is the `precision` value used by the indicator/strategy function.

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

EXAMPLE

```
//@version=5
indicator("plot")
plot(high+low, title='Title', color=color.new(#00ffaa, 70), linewidth=2, style=plot.style_

// You may fill the background between any two plots with a fill() function:
p1 = plot(open)
p2 = plot(close)
fill(p1, p2, color=color.new(color.green, 90))
```

RETURNS

A plot object, that can be used in fill

SEE ALSO

plotshape plotchar plotarrow barcolor bgcolor fill

plotarrow()

Plots up and down arrows on the chart. Up arrow is drawn at every indicator positive value, down arrow is drawn at every negative value. If indicator returns na then no arrow is drawn. Arrows has different height, the more absolute indicator value the longer arrow is drawn.

SYNTAX

```
plotarrow(series, title, colorup, colordown, offset, minheight, maxheight, editable,
show_last, display, format, precision, force_overlay) → void
```

ARGUMENTS

series (series int/float) Series of data to be plotted as arrows. Required argument.

title (const string) Title of the plot.

colorup (series color) Color of the up arrows. Optional argument.

colordown (series color) Color of the down arrows. Optional argument.

offset (series int) Shifts arrows to the left or to the right on the given number of bars. Default is 0.

minheight (input int) Minimal possible arrow height in pixels. Default is 5.

maxheight (input int) Maximum possible arrow height in pixels. Default is 100.

editable (const bool) If true then plotarrow style will be editable in Format dialog. Default is true.

show_last (input int) If set, defines the number of arrows (from the last bar back to the past) to plot on chart.

display (input plot_display) Controls where the plot's information is displayed. Display options support addition and subtraction, meaning that using `display.all - display.status_line` will display the plot's information everywhere except in the script's status line. `display.price_scale + display.status_line` will display the plot only in the price scale and status line. When `display` arguments such as `display.price_scale` have user-controlled chart settings equivalents, the relevant plot information will only appear when all settings allow for it. Possible values: display.none, display.pane, display.data_window, display.price_scale, display.status_line, display.all. Optional. The default is display.all.

format (input string) Determines whether the script formats the plot's values as prices, percentages, or volume values. The argument passed to this parameter supersedes the `format` parameter of the indicator, and strategy functions. Optional. The default is the `format` value used by the indicator/strategy function. Possible values: format.price, format.percent, format.volume.

precision (input int) The number of digits after the decimal point the plot's values show on the chart pane's y-axis, the script's status line, and the Data Window. Accepts a non-negative integer less than or equal to 16. The argument passed to this parameter supersedes the `precision` parameter of the indicator and strategy functions. When the function's `format` parameter uses format.volume, the `precision` parameter will not affect the result, as the decimal precision rules defined by format.volume supersede other precision settings. Optional. The default is the `precision` value used by the indicator/strategy function.

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

EXAMPLE

```
//@version=5
indicator("plotarrow example", overlay=true)
codiff = close - open
plotarrow(codiff, colorup=color.new(color.teal,40), colordown=color.new(color.orange, 40))
```

REMARKS

Use plotarrow function in conjunction with 'overlay=true' indicator parameter!

SEE ALSO

plot plotshape plotchar barcolor bgcolor

plotbar()

Plots ohlc bars on the chart.

SYNTAX

```
plotbar(open, high, low, close, title, color, editable, show_last, display, format,
precision, force_overlay) → void
```

ARGUMENTS

open (series int/float) Open series of data to be used as open values of bars. Required argument.

high (series int/float) High series of data to be used as high values of bars. Required argument.

low (series int/float) Low series of data to be used as low values of bars. Required argument.

close (series int/float) Close series of data to be used as close values of bars. Required argument.

title (const string) Title of the plotbar. Optional argument.

color (series color) Color of the ohlc bars. You can use constants like 'color=color.red' or 'color=#ff001a' as well as complex expressions like 'color = close >= open ? color.green : color.red'. Optional argument.

editable (const bool) If true then plotbar style will be editable in Format dialog. Default is true.

show_last (input int) If set, defines the number of bars (from the last bar back to the past) to plot on chart.

display (input plot_display) Controls where the plot's information is displayed. Display options support addition and subtraction, meaning that using `display.all - display.status_line` will display the plot's information everywhere except in the script's status line. `display.price_scale + display.status_line` will display the plot only in the price scale and status line. When `display` arguments such as `display.price_scale` have user-controlled chart settings equivalents, the relevant plot information will only appear when all settings allow for it. Possible values: display.none, display.pane, display.data_window, display.price_scale, display.status_line, display.all. Optional. The default is display.all.

format (input string) Determines whether the script formats the plot's values as prices, percentages, or volume values. The argument passed to this parameter supersedes the `format` parameter of the indicator, and strategy functions. Optional. The default is the `format` value used by the indicator/strategy function. Possible values: format.price, format.percent, format.volume.

precision (input int) The number of digits after the decimal point the plot's values show on the chart pane's y-axis, the script's status line, and the Data Window. Accepts a non-negative integer less than or equal to 16. The argument passed to this parameter supersedes the `precision` parameter of the indicator and strategy functions. When the function's `format` parameter uses format.volume, the `precision` parameter will not affect the result, as the decimal precision rules defined by format.volume supersede other precision settings. Optional. The default is the `precision` value used by the indicator/strategy function.

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

EXAMPLE

```
//@version=5
indicator("plotbar example", overlay=true)
plotbar(open, high, low, close, title='Title', color = open < close ? color.green : color.
```

REMARKS

Even if one value of open, high, low or close equal NaN then bar no draw.

The maximal value of open, high, low or close will be set as 'high', and the minimal value will be set as 'low'.

SEE ALSO

plotcandle

plotcandle()

Plots candles on the chart.

SYNTAX

```
plotcandle(open, high, low, close, title, color, wickcolor, editable, show_last,
bordercolor, display, format, precision, force_overlay) → void
```

ARGUMENTS

open (series int/float) Open series of data to be used as open values of candles. Required argument.

high (series int/float) High series of data to be used as high values of candles. Required argument.

low (series int/float) Low series of data to be used as low values of candles. Required argument.

close (series int/float) Close series of data to be used as close values of candles. Required argument.

title (const string) Title of the plotcandles. Optional argument.

color (series color) Color of the candles. You can use constants like 'color=color.red' or 'color=#ff001a' as well as complex expressions like 'color = close >= open ? color.green : color.red'. Optional argument.

wickcolor (series color) The color of the wick of candles. An optional argument.

editable (const bool) If true then plotcandle style will be editable in Format dialog. Default is true.

show_last (input int) If set, defines the number of candles (from the last bar back to the past) to plot on chart.

bordercolor (series color) The border color of candles. An optional argument.

display (input plot_display) Controls where the plot's information is displayed. Display options support addition and subtraction, meaning that using `display.all - display.status_line` will display the plot's information everywhere except in the script's status line. `display.price_scale + display.status_line` will display the plot only in the price scale and status line. When `display` arguments such as `display.price_scale` have user-controlled chart settings equivalents, the relevant plot information will only appear when all settings allow for it. Possible values: display.none, display.pane, display.data_window, display.price_scale, display.status_line, display.all. Optional. The default is display.all.

format (input string) Determines whether the script formats the plot's values as prices, percentages, or volume values. The argument passed to this parameter supersedes the `format` parameter of the indicator, and strategy functions. Optional. The default is the `format` value used by the indicator/strategy function. Possible values: format.price, format.percent, format.volume.

precision (input int) The number of digits after the decimal point the plot's values show on the chart pane's y-axis, the script's status line, and the Data Window. Accepts a non-negative integer less than or equal to 16. The argument passed to this parameter supersedes the `precision` parameter of the indicator and strategy functions. When the function's `format` parameter uses format.volume, the `precision` parameter will not affect the result, as the decimal precision rules defined by format.volume supersede other precision settings. Optional. The default is the `precision` value used by the indicator/strategy function.

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

EXAMPLE

```
//@version=5
indicator("plotcandle example", overlay=true)
plotcandle(open, high, low, close, title='Title', color = open < close ? color.green : col
```

REMARKS

Even if one value of open, high, low or close equal NaN then bar no draw.

The maximal value of open, high, low or close will be set as 'high', and the minimal value will be set as 'low'.

SEE ALSO

plotbar

plotchar()

Plots visual shapes using any given one Unicode character on the chart.

SYNTAX

```
plotchar(series, title, char, location, color, offset, text, textcolor, editable, size,
show_last, display, format, precision, force_overlay) → void
```

ARGUMENTS

series (series int/float/bool) Series of data to be plotted as shapes. Series is treated as a series of boolean values for all location values except location.absolute. Required argument.

title (const string) Title of the plot.

char (input string) Character to use as a visual shape.

location (input string) Location of shapes on the chart. Possible values are: location.abovebar, location.belowbar, location.top, location.bottom, location.absolute. Default value is location.abovebar.

color (series color) Color of the shapes. You can use constants like 'color=color.red' or 'color=#ff001a' as well as complex expressions like 'color = close >= open ? color.green : color.red'. Optional argument.

offset (series int) Shifts shapes to the left or to the right on the given number of bars. Default is 0.

text (const string) Text to display with the shape. You can use multiline text, to separate lines use '\n' escape sequence. Example: 'line one\nline two'.

textcolor (series color) Color of the text. You can use constants like 'textcolor=color.red' or 'textcolor=#ff001a' as well as complex expressions like 'textcolor = close >= open ? color.green : color.red'. Optional argument.

editable (const bool) If true then plotchar style will be editable in Format dialog. Default is true.

size (const string) Size of characters on the chart. Possible values are: size.auto, size.tiny, size.small, size.normal, size.large, size.huge. Default is size.auto.

show_last (input int) If set, defines the number of chars (from the last bar back to the past) to plot on chart.

display (input plot_display) Controls where the plot's information is displayed. Display options support addition and subtraction, meaning that using `display.all - display.status_line` will display the plot's information everywhere except in the script's status line. `display.price_scale + display.status_line` will display the plot only in the price scale and status line. When `display` arguments such as `display.price_scale` have user-controlled chart settings equivalents, the relevant plot information will only appear when all settings allow for it. Possible values: display.none, display.pane, display.data_window, display.price_scale, display.status_line, display.all. Optional. The default is display.all.

format (input string) Determines whether the script formats the plot's values as prices, percentages, or volume values. The argument passed to this parameter supersedes the `format` parameter of the indicator, and strategy functions. Optional. The default is the `format` value used by the indicator/strategy function. Possible values: format.price, format.percent, format.volume.

precision (input int) The number of digits after the decimal point the plot's values show on the chart pane's y-axis, the script's status line, and the Data Window. Accepts a non-negative integer less than or equal to 16. The argument passed to this parameter supersedes the `precision` parameter of the indicator and strategy functions. When the function's `format` parameter uses format.volume, the `precision` parameter will not affect the result, as the decimal precision rules defined by format.volume supersede other precision settings. Optional. The default is the `precision` value used by the indicator/strategy function.

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

EXAMPLE

```
//@version=5
indicator("plotchar example", overlay=true)
data = close >= open
plotchar(data, char='❄')
```

REMARKS

Use plotchar function in conjunction with 'overlay=true' indicator parameter!

SEE ALSO

plot plotshape plotarrow barcolor bgcolor

plotshape()

Plots visual shapes on the chart.

SYNTAX

```
plotshape(series, title, style, location, color, offset, text, textcolor, editable, size,
show_last, display, format, precision, force_overlay) → void
```

ARGUMENTS

series (series int/float/bool) Series of data to be plotted as shapes. Series is treated as a series of boolean values for all location values except location.absolute. Required argument.

title (const string) Title of the plot.

style (input string) Type of plot. Possible values are: shape.xcross, shape.cross, shape.triangleup, shape.triangledown, shape.flag, shape.circle, shape.arrowup, shape.arrowdown, shape.labelup, shape.labeldown, shape.square, shape.diamond. Default value is shape.xcross.

location (input string) Location of shapes on the chart. Possible values are:

location.abovebar, location.belowbar, location.top, location.bottom, location.absolute. Default value is location.abovebar.

color (series color) Color of the shapes. You can use constants like 'color=color.red' or 'color=#ff001a' as well as complex expressions like 'color = close >= open ? color.green : color.red'. Optional argument.

offset (series int) Shifts shapes to the left or to the right on the given number of bars. Default is 0.

text (const string) Text to display with the shape. You can use multiline text, to separate lines use '\n' escape sequence. Example: 'line one\nline two'.

textcolor (series color) Color of the text. You can use constants like 'textcolor=color.red' or 'textcolor=#ff001a' as well as complex expressions like 'textcolor = close >= open ? color.green : color.red'. Optional argument.

editable (const bool) If true then plotshape style will be editable in Format dialog. Default is true.

size (const string) Size of shapes on the chart. Possible values are: size.auto, size.tiny, size.small, size.normal, size.large, size.huge. Default is size.auto.

show_last (input int) If set, defines the number of shapes (from the last bar back to the past) to plot on chart.

display (input plot_display) Controls where the plot's information is displayed. Display options support addition and subtraction, meaning that using `display.all - display.status_line` will display the plot's information everywhere except in the script's status line. `display.price_scale + display.status_line` will display the plot only in the price scale and status line. When `display` arguments such as `display.price_scale` have user-controlled chart settings equivalents, the relevant plot information will only appear when all settings allow for it. Possible values: display.none, display.pane, display.data_window, display.price_scale, display.status_line, display.all. Optional. The default is display.all.

format (input string) Determines whether the script formats the plot's values as prices, percentages, or volume values. The argument passed to this parameter supersedes the `format` parameter of the indicator, and strategy functions. Optional. The default is the `format` value used by the indicator/strategy function. Possible values: format.price, format.percent, format.volume.

precision (input int) The number of digits after the decimal point the plot's values show on the chart pane's y-axis, the script's status line, and the Data Window. Accepts a non-negative integer less than or equal to 16. The argument passed to this parameter supersedes the `precision` parameter of the indicator and strategy functions. When the function's `format` parameter uses format.volume, the `precision` parameter will not affect the result, as the decimal precision rules defined by format.volume supersede other precision settings. Optional. The default is the `precision` value used by the indicator/strategy function.

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

EXAMPLE

```
//@version=5
indicator("plotshape example 1", overlay=true)
data = close >= open
plotshape(data, style=shape.xcross)
```

REMARKS

Use plotshape function in conjunction with 'overlay=true' indicator parameter!

SEE ALSO

plot plotchar plotarrow barcolor bgcolor

polyline.delete()

Deletes the specified polyline object. It has no effect if the `id` doesn't exist.

SYNTAX

```
polyline.delete(id) → void
```

ARGUMENTS

id (series polyline) The polyline ID to delete.

polyline.new()

Creates a new polyline instance and displays it on the chart, sequentially connecting all of the points in the `points` array with line segments. The segments in the drawing can be straight or curved depending on the `curved` parameter.

SYNTAX

```
polyline.new(points, curved, closed, xloc, line_color, fill_color, line_style, line_width,
force_overlay) → series polyline
```

ARGUMENTS

points (array<chart.point>) An array of chart.point objects for the drawing to sequentially connect.

curved (series bool) If true, the drawing will connect all points from the `points` array using curved line segments. Optional. The default is false.

closed (series bool) If true, the drawing will also connect the first point to the last point from the `points` array, resulting in a closed polyline. Optional. The default is false.

xloc (series string) Determines the field of the chart.point objects in the `points` array that the polyline will use for its x-coordinates. If xloc.bar_index, the polyline will use the `index` field from each point. If xloc.bar_time, it will use the `time` field. Optional. The default is xloc.bar_index.

line_color (series color) The color of the line segments. Optional. The default is color.blue.

fill_color (series color) The fill color of the polyline. Optional. The default is na.

line_style (series string) The style of the polyline. Possible values: line.style_solid, line.style_dotted, line.style_dashed, line.style_arrow_left, line.style_arrow_right, line.style_arrow_both. Optional. The default is line.style_solid.

line_width (series int) The width of the line segments, expressed in pixels. Optional. The default is 1.

force_overlay (const bool) If true, the drawing will display on the main chart pane, even when the script occupies a separate pane. Optional. The default is false.

EXAMPLE

```
//@version=5
indicator("Polylines example", overlay = true)

//@variable If `true`, connects all points in the polyline with curved line segments.
bool curvedInput = input.bool(false, "Curve Polyline")
//@variable If `true`, connects the first point in the polyline to the last point.
bool closedInput = input.bool(true,  "Close Polyline")
//@variable The color of the space filled by the polyline.
color fillcolor = input.color(color.new(color.blue, 90), "Fill Color")

// Time and price inputs for the polyline's points.
p1x = input.time(0,  "p1", confirm = true, inline = "p1")
p1y = input.price(0, "  ", confirm = true, inline = "p1")
p2x = input.time(0,  "p2", confirm = true, inline = "p2")
p2y = input.price(0, "  ", confirm = true, inline = "p2")
p3x = input.time(0,  "p3", confirm = true, inline = "p3")
p3y = input.price(0, "  ", confirm = true, inline = "p3")
p4x = input.time(0,  "p4", confirm = true, inline = "p4")
p4y = input.price(0, "  ", confirm = true, inline = "p4")
p5x = input.time(0,  "p5", confirm = true, inline = "p5")
p5y = input.price(0, "  ", confirm = true, inline = "p5")
```

```
    if barstate.islastconfirmedhistory
        //@variable An array of `chart.point` objects for the new polyline.
        var points = array.new<chart.point>()
        // Push new `chart.point` instances into the `points` array.
        points.push(chart.point.from_time(p1x, p1y))
        points.push(chart.point.from_time(p2x, p2y))
        points.push(chart.point.from_time(p3x, p3y))
        points.push(chart.point.from_time(p4x, p4y))
        points.push(chart.point.from_time(p5x, p5y))
        // Add labels for each `chart.point` in `points`.
        l1p1 = label.new(points.get(0), text = "p1", xloc = xloc.bar_time, color = na)
        l1p2 = label.new(points.get(1), text = "p2", xloc = xloc.bar_time, color = na)
        l2p1 = label.new(points.get(2), text = "p3", xloc = xloc.bar_time, color = na)
        l2p2 = label.new(points.get(3), text = "p4", xloc = xloc.bar_time, color = na)
        // Create a new polyline that connects each `chart.point` in the `points` array, start
        polyline.new(points, curved = curvedInput, closed = closedInput, fill_color = fillcolo
```

RETURNS

The ID of a new polyline object that a script can use in other `polyline.*()` functions.

SEE ALSO

chart.point.new

request.currency_rate()

Provides a daily rate that can be used to convert a value expressed in the `from` currency to another in the `to` currency.

SYNTAX

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

ARGUMENTS

from (series string) The currency in which the value to be converted is expressed. Possible values: a three-letter string with the currency code in the ISO 4217 format (e.g. "USD"), or one of the built-in variables that return currency codes, like syminfo.currency or currency.USD.

to (series string) The currency in which the value is to be converted. Possible values: a three-letter string with the currency code in the ISO 4217 format (e.g. "USD"), or one of the built-in variables that return currency codes, like syminfo.currency or currency.USD.

ignore_invalid_currency (series bool) Determines the behavior of the function if a conversion rate between the two currencies cannot be calculated: if false, the script will halt and return a runtime error; if true, the function will return na and execution will continue. Optional. The default is false.

EXAMPLE

```
//@version=5
indicator("Close in British Pounds")
rate = request.currency_rate(syminfo.currency, "GBP")
plot(close * rate)
```

REMARKS

If `from` and `to` arguments are equal, function returns 1. Please note that using this variable/function can cause indicator repainting.

request.dividends()

Requests dividends data for the specified symbol.

SYNTAX

```
request.dividends(ticker, field, gaps, lookahead, ignore_invalid_symbol, currency) →
series float
```

ARGUMENTS

ticker (series string) Symbol. Note that the symbol should be passed with a prefix. For example: "NASDAQ:AAPL" instead of "AAPL". Using syminfo.ticker will cause an error. Use syminfo.tickerid instead.

field (series string) Input string. Possible values include: dividends.net, dividends.gross. Default value is dividends.gross.

gaps (simple barmerge_gaps) Merge strategy for the requested data (requested data automatically merges with the main series OHLC data). Possible values: barmerge.gaps_on, barmerge.gaps_off. barmerge.gaps_on - requested data is merged with possible gaps (na values). barmerge.gaps_off - requested data is merged continuously without gaps, all the gaps are filled with the previous nearest existing values. Default value is barmerge.gaps_off.

lookahead (simple barmerge_lookahead) Merge strategy for the requested data position. Possible values: barmerge.lookahead_on, barmerge.lookahead_off. Default value is barmerge.lookahead_off starting from version 3. Note that behaviour is the same on real-time, and differs only on history.

ignore_invalid_symbol (input bool) An optional parameter. Determines the behavior of the function if the specified symbol is not found: if false, the script will halt and return a runtime error; if true, the function will return na and execution will continue. The default value is false.

currency (series string) Currency into which the symbol's currency-related dividends values (e.g. dividends.gross) are to be converted. The conversion rates used are based on the FX_IDC pairs' daily rates of the previous day (relative to the bar where the calculation is done). Optional. The default is syminfo.currency. Possible values: a three-letter string with the currency code in the ISO 4217 format (e.g. "USD") or one of the constants in the currency.* namespace, e.g. currency.USD.

EXAMPLE

```
//@version=5
indicator("request.dividends")
s1 = request.dividends("NASDAQ:BELFA")
plot(s1)
s2 = request.dividends("NASDAQ:BELFA", dividends.net, gaps=barmerge.gaps_on, lookahead=bar
plot(s2)
```

RETURNS

Requested series, or n/a if there is no dividends data for the specified symbol.

SEE ALSO

request.earnings request.splits request.security syminfo.tickerid

request.earnings()

Requests earnings data for the specified symbol.

SYNTAX

```
request.earnings(ticker, field, gaps, lookahead, ignore_invalid_symbol, currency) → series
float
```

ARGUMENTS

ticker (series string) Symbol. Note that the symbol should be passed with a prefix. For example: "NASDAQ:AAPL" instead of "AAPL". Using syminfo.ticker will cause an error. Use syminfo.tickerid instead.

field (series string) Input string. Possible values include: earnings.actual, earnings.estimate, earnings.standardized. Default value is earnings.actual.

gaps (simple barmerge_gaps) Merge strategy for the requested data (requested data automatically merges with the main series OHLC data). Possible values: barmerge.gaps_on, barmerge.gaps_off. barmerge.gaps_on - requested data is merged with possible gaps (na values). barmerge.gaps_off - requested data is merged continuously without gaps, all the gaps are filled with the previous nearest existing values. Default value is barmerge.gaps_off.

lookahead (simple barmerge_lookahead) Merge strategy for the requested data position. Possible values: barmerge.lookahead_on, barmerge.lookahead_off. Default value is barmerge.lookahead_off starting from version 3. Note that behavour is the same on real-time, and differs only on history.

ignore_invalid_symbol (input bool) An optional parameter. Determines the behavior of the function if the specified symbol is not found: if false, the script will halt and return a runtime error; if true, the function will return na and execution will continue. The default value is false.

currency (series string) Currency into which the symbol's currency-related earnings values (e.g. earnings.actual) are to be converted. The conversion rates used are based on the FX_IDC pairs' daily rates of the previous day (relative to the bar where the calculation is done). Optional. The default is syminfo.currency. Possible values: a three-letter string with the currency code in the ISO 4217 format (e.g. "USD") or one of the constants in the currency.* namespace, e.g. currency.USD.

EXAMPLE

```
//@version=5
indicator("request.earnings")
s1 = request.earnings("NASDAQ:BELFA")
plot(s1)
s2 = request.earnings("NASDAQ:BELFA", earnings.actual, gaps=barmerge.gaps_on, lookahead=ba
plot(s2)
```

RETURNS

Requested series, or n/a if there is no earnings data for the specified symbol.

SEE ALSO

request.dividends request.splits request.security syminfo.tickerid

request.economic()

Requests economic data for a symbol. Economic data includes information such as the state of a country's economy (GDP, inflation rate, etc.) or of a particular industry (steel production, ICU beds, etc.).

SYNTAX

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

ARGUMENTS

country_code (series string) The code of the country (e.g. "US") or the region (e.g. "EU") for which the economic data is requested. The Help Center article lists the countries and their codes. The countries for which information is available vary with metrics. The Help Center article for each metric lists the countries for which the metric is available.

field (series string) The code of the requested economic metric (e.g., "GDP"). The Help Center article lists the metrics and their codes.

gaps (simple barmerge_gaps) Specifies how the returned values are merged on chart bars. Possible values: barmerge.gaps_off, barmerge.gaps_on. With barmerge.gaps_on, a value only appears on the current chart bar when it first becomes available from the function's context, otherwise na is returned (thus a "gap" occurs). With barmerge.gaps_off, what would otherwise be gaps are filled with the latest known value returned, avoiding na values. Optional. The default is barmerge.gaps_off.

ignore_invalid_symbol (input bool) Determines the behavior of the function if the specified symbol is not found: if false, the script will halt and return a runtime error; if true, the function will return na and execution will continue. Optional. The default is false.

EXAMPLE

```
//@version=5
indicator("US GDP")
e = request.economic("US", "GDP")
plot(e)
```

RETURNS

Requested series.

REMARKS

Economic data can also be accessed from charts, just like a regular symbol. Use "ECONOMIC" as the exchange name and `{country_code}{field}` as the ticker. The name of US GDP data is thus "ECONOMIC:USGDP".

SEE ALSO

request.financial request.quandl

request.financial()

Requests financial series for symbol.

SYNTAX

```
request.financial(symbol, financial_id, period, gaps, ignore_invalid_symbol, currency) →
series float
```

ARGUMENTS

symbol (series string) Symbol. Note that the symbol should be passed with a prefix. For example: "NASDAQ:AAPL" instead of "AAPL".

financial_id (series string) Financial identifier. You can find the list of available ids via our Help Center.

period (series string) Reporting period. Possible values are "TTM", "FY", "FQ", "FH", "D".

gaps (simple barmerge_gaps) Merge strategy for the requested data (requested data automatically merges with the main series: OHLC data). Possible values include: barmerge.gaps_on, barmerge.gaps_off. barmerge.gaps_on - requested data is merged with possible gaps (na values). barmerge.gaps_off - requested data is merged continuously without gaps, all the gaps are filled with the previous, nearest existing values. Default value is barmerge.gaps_off.

ignore_invalid_symbol (input bool) An optional parameter. Determines the behavior of the function if the specified symbol is not found: if false, the script will halt and return a runtime error; if true, the function will return na and execution will continue. The default value is false.

currency (series string) Currency into which the symbol's financial metrics (e.g. Net Income) are to be converted. The conversion rates used are based on the FX_IDC pairs' daily rates of the previous day (relative to the bar where the calculation is done). Optional. The default is syminfo.currency. Possible values: a three-letter string with the currency code in the ISO 4217 format (e.g. "USD") or one of the constants in the currency.* namespace, e.g. currency.USD.

EXAMPLE

```
//@version=5
indicator("request.financial")
f = request.financial("NASDAQ:MSFT", "ACCOUNTS_PAYABLE", "FY")
plot(f)
```

RETURNS

Requested series.

SEE ALSO

request.security   syminfo.tickerid

request.quandl()

Requests Nasdaq Data Link (formerly Quandl) data for a symbol.

SYNTAX

```
request.quandl(ticker, gaps, index, ignore_invalid_symbol) → series float
```

ARGUMENTS

ticker (series string) Symbol. Note that the name of a time series and Quandl data feed should be divided by a forward slash. For example: "CFTC/SB_FO_ALL".

gaps (simple barmerge_gaps) Merge strategy for the requested data (requested data automatically merges with the main series: OHLC data). Possible values include: barmerge.gaps_on, barmerge.gaps_off. barmerge.gaps_on - requested data is merged with possible gaps (na values). barmerge.gaps_off - requested data is merged continuously without gaps, all the gaps are filled with the previous, nearest existing values. Default value is barmerge.gaps_off.

index (series int) A Quandl time-series column index.

ignore_invalid_symbol (input bool) An optional parameter. Determines the behavior of the function if the specified symbol is not found: if false, the script will halt and return a runtime error; if true, the function will return na and execution will continue. The default value is false.

EXAMPLE

```
//@version=5
indicator("request.quandl")
f = request.quandl("CFTC/SB_FO_ALL", barmerge.gaps_off, 0)
plot(f)
```

RETURNS

Requested series.

SEE ALSO

request.security   syminfo.tickerid

request.security()

Requests the result of an expression from a specified context (symbol and timeframe).

SYNTAX

```
request.security(symbol, timeframe, expression, gaps, lookahead, ignore_invalid_symbol,
currency, calc_bars_count) → series <type>
```

ARGUMENTS

symbol (series string) Symbol or ticker identifier of the requested data. Use an empty string or syminfo.tickerid to request data using the chart's symbol. To retrieve data with additional modifiers (extended sessions, dividend adjustments, non-standard chart types like Heikin Ashi and Renko, etc.), create a custom ticker ID for the request using the functions in the `ticker.*` namespace.

timeframe (series string) Timeframe of the requested data. Use an empty string or timeframe.period to request data from the chart's timeframe or the `timeframe` specified in the indicator function. To request data from a different timeframe, supply a valid timeframe string. See here to learn about specifying timeframe strings.

expression (variable, function, object, array, matrix, or map of series int/float/bool/string/color/enum, or a tuple of these) The expression to calculate and return from the requested context. It can accept a built-in variable like close, a user-defined variable, an expression such as `ta.change(close) / (high - low)`, a function call that does not use Pine Script® drawings, an object, a collection, or a tuple of expressions.

gaps (simple barmerge_gaps) Specifies how the returned values are merged on chart bars. Possible values: barmerge.gaps_on, barmerge.gaps_off. With barmerge.gaps_on a value only appears on the current chart bar when it first becomes available from the function's context, otherwise na is returned (thus a "gap" occurs). With barmerge.gaps_off what would otherwise be gaps are filled with the latest known value returned, avoiding na values. Optional. The default is barmerge.gaps_off.

lookahead (simple barmerge_lookahead) On historical bars only, returns data from the timeframe before it elapses. Possible values: barmerge.lookahead_on, barmerge.lookahead_off. Has no effect on realtime values. Optional. The default is barmerge.lookahead_off starting from Pine Script® v3. The default is barmerge.lookahead_on in v1 and v2. WARNING: Using barmerge.lookahead_on at timeframes higher than the chart's without offsetting the `expression` argument like in `close[1]` will introduce future leak in scripts, as the function will then return the `close` price before it is actually known in the current context. As is explained in the User Manual's page on Repainting this will produce misleading results.

ignore_invalid_symbol (input bool) Determines the behavior of the function if the specified symbol is not found: if false, the script will halt and throw a runtime error; if true, the function will return na and execution will continue. Optional. The default is false.

currency (series string) Currency into which values expressed in currency units (open, high, low, close, etc.) or expressions using such values are to be converted. The conversion rates used are based on the FX_IDC pairs' daily rates of the previous day (relative to the bar where the calculation is done). Possible values: a three-letter string with the currency code in the ISO 4217 format (e.g. "USD") or one of the constants in the currency.* namespace, e.g. currency.USD. Note that literal values such as `200` are not converted. Optional. The default is syminfo.currency.

calc_bars_count (simple int) If specified, the function will only request this number of values from the end of the symbol's history and calculate `expression` as if these values are the only available data, which might improve calculation speed in some cases. Optional. The default is 100,000, which is the limit for all non-professional TradingView plans.

EXAMPLE

```
//@version=5
indicator("Simple `request.security()` calls")
// Returns 1D close of the current symbol.
dailyClose = request.security(syminfo.tickerid, "1D", close)
plot(dailyClose)

// Returns the close of "AAPL" from the same timeframe as currently open on the chart.
aaplClose = request.security("AAPL", timeframe.period, close)
plot(aaplClose)
```

EXAMPLE

```
//@version=5
indicator("Advanced `request.security()` calls")
// This calculates a 10-period moving average on the active chart.
sma = ta.sma(close, 10)
// This sends the `sma` calculation for execution in the context of the "AAPL" symbol at a
aaplSma = request.security("AAPL", "240", sma)
plot(aaplSma)

// To avoid differences on historical and realtime bars, you can use this technique, which
indexHighTF = barstate.isrealtime ? 1 : 0
indexCurrTF = barstate.isrealtime ? 0 : 1
```

```
nonRepaintingClose = request.security(syminfo.tickerid, "1D", close[indexHighTF])[indexCur
plot(nonRepaintingClose, "Non-repainting close")

// Returns the 1H close of "AAPL", extended session included. The value is dividend-adjust
extendedTicker = ticker.modify("NASDAQ:AAPL", session = session.extended, adjustment = adj
aaplExtAdj = request.security(extendedTicker, "60", close)
plot(aaplExtAdj)

// Returns the result of a user-defined function.
// The `max` variable is mutable, but we can pass it to `request.security()` because it is
allTimeHigh(source) =>
    var max = source
    max := math.max(max, source)
allTimeHigh1D = request.security(syminfo.tickerid, "1D", allTimeHigh(high))

// By using a tuple `expression`, we obtain several values with only one `request.security
[open1D, high1D, low1D, close1D, ema1D] = request.security(syminfo.tickerid, "1D", [open,
plotcandle(open1D, high1D, low1D, close1D)
plot(ema1D)

// Returns an array containing the OHLC values of the chart's symbol from the 1D timeframe
ohlcArray = request.security(syminfo.tickerid, "1D", array.from(open, high, low, close))
plotcandle(array.get(ohlcArray, 0), array.get(ohlcArray, 1), array.get(ohlcArray, 2), arra
```

RETURNS

A result determined by `expression`.

REMARKS

Scripts using this function might calculate differently on historical and realtime bars, leading to repainting.

A single script can contain no more than 40 unique `request.*()` function calls. A call is unique only if it does not call the same function with the same arguments.

When using two calls to a `request.*()` function to evaluate the same expression from the same context with different `calc_bars_count` values, the second call requests the same number of historical bars as the first. For example, if a script calls `request.security("AAPL", "", close, calc_bars_count = 3)` after it calls `request.security("AAPL", "", close, calc_bars_count = 5)`, the second call also uses five bars of historical data, not three.

The symbol of a `request.()` call can be *inherited* if it is not specified precisely, i.e., if the `symbol` argument is an empty string or syminfo.tickerid. Similarly, the timeframe of a `request.()` call can be inherited if the `timeframe` argument is an empty string or timeframe.period. These values are normally taken from the chart that the script is running on. However, if `request.*()` function A is called from within the expression of `request.*()` function B, then function A can inherit the values from function B. See here for more information.

SEE ALSO

syminfo.ticker syminfo.tickerid timeframe.period ticker.new ticker.modify

request.security_lower_tf request.dividends request.earnings request.splits

request.financial request.quandl

request.security_lower_tf()

Requests the results of an expression from a specified symbol on a timeframe lower than or equal to the chart's timeframe. It returns an array containing one element for each lower-timeframe bar within the chart bar. On a 5-minute chart, requesting data using a `timeframe` argument of "1" typically returns an array with five elements representing the value of the `expression` on each 1-minute bar, ordered by time with the earliest value first.

SYNTAX

```
request.security_lower_tf(symbol, timeframe, expression, ignore_invalid_symbol, currency,
ignore_invalid_timeframe, calc_bars_count) → array<type>
```

ARGUMENTS

symbol (series string) Symbol or ticker identifier of the requested data. Use an empty string or syminfo.tickerid to request data using the chart's symbol. To retrieve data with additional modifiers (extended sessions, dividend adjustments, non-standard chart types like Heikin Ashi and Renko, etc.), create a custom ticker ID for the request using the functions in the `ticker.*` namespace.

timeframe (series string) Timeframe of the requested data. Use an empty string or timeframe.period to request data from the chart's timeframe or the `timeframe` specified in the indicator function. To request data from a different timeframe, supply a valid timeframe string. See here to learn about specifying timeframe strings.

expression (variable, object or function of series int/float/bool/string/color/enum, or a tuple of these) The expression to calculate and return from the requested context. It can accept a built-in variable like close, a user-defined variable, an expression such as `ta.change(close) / (high - low)`, a function call that does not use Pine Script® drawings, an object, or a tuple of expressions. Collections are not allowed unless they are within the fields of an object

ignore_invalid_symbol (series bool) Determines the behavior of the function if the specified symbol is not found: if false, the script will halt and throw a runtime error; if true, the function will return na and execution will continue. Optional. The default is false.

currency (series string) Currency into which values expressed in currency units (open, high, low, close, etc.) or expressions using such values are to be converted. The conversion rates used are based on the FX_IDC pairs' daily rates of the previous day (relative to the bar where the calculation is done). Possible values: a three-letter string with the currency code in the ISO 4217 format (e.g. "USD") or one of the constants in the currency.* namespace, e.g. currency.USD. Note that literal values such as `200` are not converted. Optional. The default is syminfo.currency.

ignore_invalid_timeframe (series bool) Determines the behavior of the function when the chart's timeframe is smaller than the `timeframe` used in the function call. If false, the script will halt and throw a runtime error. If true, the function will return na and execution will continue. Optional. The default is false.

calc_bars_count (simple int) If specified, the function will only request this number of values from the end of the symbol's history and calculate `expression` as if these values are the only available data, which might improve calculation speed in some cases. Optional. The default is 100,000, which is the limit for all non-professional TradingView plans.

EXAMPLE

```
//@version=5
indicator("`request.security_lower_tf()` Example", overlay = true)

// If the current chart timeframe is set to 120 minutes, then the `arrayClose` array will
arrClose = request.security_lower_tf(syminfo.tickerid, "60", close)

if bar_index == last_bar_index - 1
    label.new(bar_index, high, str.tostring(arrClose))
```

RETURNS

An array of a type determined by `expression`, or a tuple of these.

REMARKS

Scripts using this function might calculate differently on historical and realtime bars, leading to repainting.

Please note that spreads (e.g., "AAPL+MSFT*TSLA") do not always return reliable data with this function.

A single script can contain no more than 40 unique `request.*()` function calls. A call is unique only if it does not call the same function with the same arguments.

When using two calls to a `request.*()` function to evaluate the same expression from the same context with different `calc_bars_count` values, the second call requests the same number of historical bars as the first. For example, if a script calls `request.security("AAPL", "", close, calc_bars_count = 3)` after it calls `request.security("AAPL", "", close, calc_bars_count = 5)`, the second call also uses five bars of historical data, not three.

The symbol of a `request.()` call can be *inherited* if it is not specified precisely, i.e., if the `symbol` argument is an empty string or syminfo.tickerid. Similarly, the timeframe of a `request.()` call can be inherited if the `timeframe` argument is an empty string or timeframe.period. These values are normally taken from the chart that the script is running on. However, if `request.*()` function A is called from within the expression of `request.*()` function B, then function A can inherit the values from function B. See here for more information.

SEE ALSO

request.security syminfo.ticker syminfo.tickerid timeframe.period ticker.new

request.dividends request.earnings request.splits request.financial request.quandl

request.seed()

Requests data from a user-maintained GitHub repository and returns it as a series. An in-depth tutorial on how to add new data can be found here.

SYNTAX

```
request.seed(source, symbol, expression, ignore_invalid_symbol, calc_bars_count) → series
<type>
```

ARGUMENTS

source (series string) Name of the GitHub repository.

symbol (series string) Name of the file in the GitHub repository containing the data. The ".csv" file extension must not be included.

expression (<arg_expr_type>) An expression to be calculated and returned from the requested symbol's context. It can be a built-in variable like close, an expression such as `ta.sma(close, 100)`, a non-mutable variable previously calculated in the script, a function call that does not use Pine Script® drawings, an array, a matrix, or a tuple. Mutable variables are not allowed, unless they are enclosed in the body of a function used in the expression.

ignore_invalid_symbol (input bool) Determines the behavior of the function if the specified symbol is not found: if false, the script will halt and throw a runtime error; if true, the function will return na and execution will continue. Optional. The default is false.

calc_bars_count (simple int) If specified, the function will only request this number of values from the end of the symbol's history and calculate `expression` as if these values are the only available data, which might improve calculation speed in some cases. Optional. The default is 100,000, which is the limit for all non-professional TradingView plans.

EXAMPLE

```
//@version=5
indicator("BTC Development Activity")

[devAct, devActSMA] = request.seed("seed_crypto_santiment", "BTC_DEV_ACTIVITY", [close, ta

plot(devAct, "BTC Development Activity")
plot(devActSMA, "BTC Development Activity SMA10", color = color.yellow)
```

RETURNS

Requested series or tuple of series, which may include array/matrix IDs.

request.splits()

Requests splits data for the specified symbol.

SYNTAX

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

ARGUMENTS

ticker (series string) Symbol. Note that the symbol should be passed with a prefix. For example: "NASDAQ:AAPL" instead of "AAPL". Using syminfo.ticker will cause an error. Use syminfo.tickerid instead.

field (series string) Input string. Possible values include: splits.denominator, splits.numerator.

gaps (simple barmerge_gaps) Merge strategy for the requested data (requested data automatically merges with the main series OHLC data). Possible values: barmerge.gaps_on, barmerge.gaps_off. barmerge.gaps_on - requested data is merged with possible gaps (na values). barmerge.gaps_off - requested data is merged continuously without gaps, all the gaps are filled with the previous nearest existing values. Default value is barmerge.gaps_off.

lookahead (simple barmerge_lookahead) Merge strategy for the requested data position.

Possible values: barmerge.lookahead_on, barmerge.lookahead_off. Default value is barmerge.lookahead_off starting from version 3. Note that behaviour is the same on real-time, and differs only on history.

ignore_invalid_symbol (input bool) An optional parameter. Determines the behavior of the function if the specified symbol is not found: if false, the script will halt and return a runtime error; if true, the function will return na and execution will continue. The default value is false.

EXAMPLE

```
//@version=5
indicator("request.splits")
s1 = request.splits("NASDAQ:BELFA", splits.denominator)
plot(s1)
s2 = request.splits("NASDAQ:BELFA", splits.denominator, gaps=barmerge.gaps_on, lookahead=b
plot(s2)
```

RETURNS

Requested series, or n/a if there is no splits data for the specified symbol.

SEE ALSO

request.earnings request.dividends request.security syminfo.tickerid

runtime.error()

When called, causes a runtime error with the error message specified in the `message` argument.

SYNTAX

```
runtime.error(message) → void
```

ARGUMENTS

message (series string) Error message.

second() 2 overloads

SYNTAX & OVERLOADS

```
second(time) → series int
```

```
second(time, timezone) → series int
```

ARGUMENTS

time (series int) UNIX time in milliseconds.

RETURNS

Second (in exchange timezone) for provided UNIX time.

REMARKS

UNIX time is the number of milliseconds that have elapsed since 00:00:00 UTC, 1 January 1970.

SEE ALSO

second time year month dayofmonth dayofweek hour minute

str.contains() 3 overloads

Returns true if the `source` string contains the `str` substring, false otherwise.

SYNTAX & OVERLOADS

```
str.contains(source, str) → const bool
```

```
str.contains(source, str) → simple bool
```

```
str.contains(source, str) → series bool
```

ARGUMENTS

source (const string) Source string.

str (const string) The substring to search for.

EXAMPLE

```
//@version=5
indicator("str.contains")
// If the current chart is a continuous futures chart, e.g “BTC1!”, then the function will
var isFutures = str.contains(syminfo.tickerid, "!")
```

```
plot(isFutures ? 1 : 0)
```

RETURNS

True if the `str` was found in the `source` string, false otherwise.

SEE ALSO

str.pos str.match

str.endswith() 3 overloads

Returns true if the `source` string ends with the substring specified in `str`, false otherwise.

SYNTAX & OVERLOADS

```
str.endswith(source, str) → const bool
```

```
str.endswith(source, str) → simple bool
```

```
str.endswith(source, str) → series bool
```

ARGUMENTS

source (const string) Source string.

str (const string) The substring to search for.

RETURNS

True if the `source` string ends with the substring specified in `str`, false otherwise.

SEE ALSO

str.startswith

str.format() 2 overloads

Converts the formatting string and value(s) into a formatted string. The formatting string can contain literal text and one placeholder in curly braces {} for each value to be formatted. Each placeholder consists of the index of the required argument (beginning at 0) that will replace it, and an optional format specifier. The index represents the position of that argument in the str.format argument list.

SYNTAX & OVERLOADS

```
str.format(formatString, arg0, arg1, ...) → simple string
```

```
str.format(formatString, arg0, arg1, ...) → series string
```

ARGUMENTS

formatString (simple string) Format string.

arg0, arg1, ... (simple int/float/bool/string) Values to format.

EXAMPLE

```
//@version=5
indicator("str.format", overlay=true)
// The format specifier inside the curly braces accepts certain modifiers:
// - Specify the number of decimals to display:
s1 = str.format("{0,number,#.#}", 1.34) // returns: 1.3
label.new(bar_index, close, text=s1)
// - Round a float value to an integer:
s2 = str.format("{0,number,integer}", 1.34) // returns: 1
label.new(bar_index - 1, close, text=s2)
// - Display a number in currency:
s3 = str.format("{0,number,currency}", 1.34) // returns: $1.34
label.new(bar_index - 2, close, text=s3)
// - Display a number as a percentage:
s4 = str.format("{0,number,percent}", 0.5) // returns: 50%
label.new(bar_index - 3, close, text=s4)
// EXAMPLES WITH SEVERAL ARGUMENTS
// returns: Number 1 is not equal to 4
s5 = str.format("Number {0} is not {1} to {2}", 1, "equal", 4)
label.new(bar_index - 4, close, text=s5)
// returns: 1.34 != 1.3
s6 = str.format("{0} != {0, number, #.#}", 1.34)
label.new(bar_index - 5, close, text=s6)
// returns: 1 is equal to 1, but 2 is equal to 2
s7 = str.format("{0, number, integer} is equal to 1, but {1, number, integer} is equal to
label.new(bar_index - 6, close, text=s7)
// returns: The cash turnover amounted to $1,340,000.00
s8 = str.format("The cash turnover amounted to {0, number, currency}", 1340000)
label.new(bar_index - 7, close, text=s8)
// returns: Expected return is 10% - 20%
s9 = str.format("Expected return is {0, number, percent} - {1, number, percent}", 0.1, 0.
label.new(bar_index - 8, close, text=s9)
```

RETURNS

The formatted string.

REMARKS

By default, formatted numbers will display up to three decimals with no trailing zeros.

The string used as the `formatString` argument can contain single quote characters ('). However, one must pair all single quotes in that string to avoid unexpected formatting results.

Any curly braces within an unquoted pattern must be balanced. For example, "ab {0} de" and "ab '}' de" are valid patterns, but "ab {0'}' de", "ab } de" and "''{''" are not.

str.format_time()

Converts the `time` timestamp into a string formatted according to `format` and `timezone`.

SYNTAX

```
str.format_time(time, format, timezone) → series string
```

ARGUMENTS

time (series int) UNIX time, in milliseconds.

format (series string) A format string specifying the date/time representation of the `time` in the returned string. All letters used in the string, except those escaped by single quotation marks `'`, are considered formatting tokens and will be used as a formatting instruction. Refer to the Remarks section for a list of the most useful tokens. Optional. The default is "yyyy-MM-dd'T'HH:mm:ssZ", which represents the ISO 8601 standard.

timezone (series string) Allows adjusting the returned value to a time zone specified in either UTC/GMT notation (e.g., "UTC-5", "GMT+0530") or as an IANA time zone database name (e.g., "America/New_York"). Optional. The default is syminfo.timezone.

EXAMPLE

```
//@version=5
indicator("str.format_time")
if timeframe.change("1D")
    formattedTime = str.format_time(time, "yyyy-MM-dd HH:mm", syminfo.timezone)
    label.new(bar_index, high, formattedTime)
```

RETURNS

The formatted string.

REMARKS

The `M`, `d`, `h`, `H`, `m` and `s` tokens can all be doubled to generate leading zeros. For example, the month of January will display as `1` with `M`, or `01` with `MM`.

The most frequently used formatting tokens are:

y - Year. Use `yy` to output the last two digits of the year or `yyyy` to output all four. Year 2000 will be `00` with `yy` or `2000` with `yyyy`.

M - Month. Not to be confused with lowercase `m`, which stands for minute.

d - Day of the month.

a - AM/PM postfix.

h - Hour in the 12-hour format. The last hour of the day will be `11` in this format.

H - Hour in the 24-hour format. The last hour of the day will be `23` in this format.

m - Minute.

s - Second.

S - Fractions of a second.

Z - Timezone, the HHmm offset from UTC, preceded by either `+` or `-`.

str.length() 3 overloads

Returns an integer corresponding to the amount of chars in that string.

SYNTAX & OVERLOADS

```
str.length(string) → const int
```

```
str.length(string) → simple int
```

```
str.length(string) → series int
```

ARGUMENTS

string (const string) Source string.

RETURNS

The number of chars in source string.

str.lower() 3 overloads

Returns a new string with all letters converted to lowercase.

SYNTAX & OVERLOADS

```
str.lower(source) → const string
```

```
str.lower(source) → simple string
```

```
str.lower(source) → series string
```

ARGUMENTS

source (const string) String to be converted.

RETURNS

A new string with all letters converted to lowercase.

SEE ALSO

str.upper

str.match() 2 overloads

Returns the new substring of the `source` string if it matches a `regex` regular expression, an empty string otherwise.

SYNTAX & OVERLOADS

```
str.match(source, regex) → simple string
```

```
str.match(source, regex) → series string
```

ARGUMENTS

source (simple string) Source string.

regex (simple string) The regular expression to which this string is to be matched.

EXAMPLE

```
//@version=5
```

```
indicator("str.match")

s = input.string("It's time to sell some NASDAQ:AAPL!")

// finding first substring that matches regular expression "[\w]+:[\w]+"
var string tickerid = str.match(s, "[\\w]+:[\\w]+")

if barstate.islastconfirmedhistory
    label.new(bar_index, high, text = tickerid) // "NASDAQ:AAPL"
```

RETURNS

The new substring of the `source` string if it matches a `regex` regular expression, an empty string otherwise.

REMARKS

Function returns first occurrence of the regular expression in the `source` string.

The backslash "\" symbol in the `regex` string needs to be escaped with additional backslash, e.g. "\\d" stands for regular expression "\d".

SEE ALSO

str.contains str.substring

str.pos() 3 overloads

Returns the position of the first occurrence of the `str` string in the `source` string, 'na' otherwise.

SYNTAX & OVERLOADS

```
str.pos(source, str) → const int
```

```
str.pos(source, str) → simple int
```

```
str.pos(source, str) → series int
```

ARGUMENTS

source (const string) Source string.

str (const string) The substring to search for.

RETURNS

Position of the `str` string in the `source` string.

REMARKS

Strings indexing starts at 0.

SEE ALSO

str.contains str.match str.substring

str.repeat() 4 overloads

Constructs a new string containing the `source` string repeated `repeat` times with the `separator` injected between each repeated instance.

SYNTAX & OVERLOADS

```
str.repeat(source, repeat, separator) → const string
```

```
str.repeat(source, repeat, separator) → input string
```

```
str.repeat(source, repeat, separator) → simple string
```

```
str.repeat(source, repeat, separator) → series string
```

ARGUMENTS

source (const string) String to repeat.

repeat (const int) Number of times to repeat the `source` string. Must be greater than or equal to 0.

separator (const string) String to inject between repeated values. Optional. The default is empty string.

EXAMPLE

```
//@version=5
indicator("str.repeat")
repeat = str.repeat("?", 3, ",") // Returns "?,?,?"
label.new(bar_index,close,repeat)
```

REMARKS

Returns na if the `source` is na.

str.replace() 3 overloads

Returns a new string with the Nth occurrence of the `target` string replaced by the `replacement` string, where N is specified in `occurrence`.

SYNTAX & OVERLOADS

```
str.replace(source, target, replacement, occurrence) → const string
```

```
str.replace(source, target, replacement, occurrence) → simple string
```

```
str.replace(source, target, replacement, occurrence) → series string
```

ARGUMENTS

source (const string) Source string.

target (const string) String to be replaced.

replacement (const string) String to be inserted instead of the target string.

occurrence (const int) N-th occurrence of the target string to replace. Indexing starts at 0 for the first match. Optional. Default value is 0.

EXAMPLE

```
//@version=5
indicator("str.replace")
var source = "FTX:BTCUSD / FTX:BTCEUR"

// Replace first occurrence of "FTX" with "BINANCE" replacement string
var newSource = str.replace(source, "FTX",  "BINANCE", 0)

if barstate.islastconfirmedhistory
    // Display "BINANCE:BTCUSD / FTX:BTCEUR"
    label.new(bar_index, high, text = newSource)
```

RETURNS

Processed string.

SEE ALSO

str.replace_all str.match

str.replace_all() 2 overloads

Replaces each occurrence of the target string in the source string with the replacement string.

SYNTAX & OVERLOADS

```
str.replace_all(source, target, replacement) → simple string
```

```
str.replace_all(source, target, replacement) → series string
```

ARGUMENTS

source (simple string) Source string.

target (simple string) String to be replaced.

replacement (simple string) String to be substituted for each occurrence of target string.

RETURNS

Processed string.

str.split()

Divides a string into an array of substrings and returns its array id.

SYNTAX

```
str.split(string, separator) → array<string>
```

ARGUMENTS

string (series string) Source string.

separator (series string) The string separating each substring.

RETURNS

The id of an array of strings.

str.startswith() 3 overloads

Returns true if the `source` string starts with the substring specified in `str`, false otherwise.

SYNTAX & OVERLOADS

```
str.startswith(source, str) → const bool
```

```
str.startswith(source, str) → simple bool
```

```
str.startswith(source, str) → series bool
```

ARGUMENTS

source (const string) Source string.

str (const string) The substring to search for.

RETURNS

True if the `source` string starts with the substring specified in `str`, false otherwise.

SEE ALSO

str.endswith

str.substring() 6 overloads

Returns a new string that is a substring of the `source` string. The substring begins with the character at the index specified by `begin_pos` and extends to 'end_pos - 1' of the `source` string.

SYNTAX & OVERLOADS

```
str.substring(source, begin_pos) → const string
```

```
str.substring(source, begin_pos) → simple string
```

```
str.substring(source, begin_pos) → series string
```

```
str.substring(source, begin_pos, end_pos) → const string
```

```
str.substring(source, begin_pos, end_pos) → simple string
```

```
str.substring(source, begin_pos, end_pos) → series string
```

ARGUMENTS

source (const string) Source string from which to extract the substring.

begin_pos (const int) The beginning position of the extracted substring. It is inclusive (the extracted substring includes the character at that position).

EXAMPLE

```
//@version=5
indicator("str.substring", overlay = true)
sym= input.symbol("NASDAQ:AAPL")
pos = str.pos(sym, ":")  // Get position of ":" character
tkr= str.substring(sym, pos+1) // "AAPL"
if barstate.islastconfirmedhistory
    label.new(bar_index, high, text = tkr)
```

RETURNS

The substring extracted from the source string.

REMARKS

Strings indexing starts from 0. If `begin_pos` is equal to `end_pos`, the function returns an empty string.

SEE ALSO

str.contains str.pos str.match

str.tonumber() 4 overloads

Converts a value represented in `string` to its "float" equivalent.

SYNTAX & OVERLOADS

```
str.tonumber(string) → const float
```

```
str.tonumber(string) → input float
```

```
str.tonumber(string) → simple float
```

```
str.tonumber(string) → series float
```

ARGUMENTS

string (const string) String containing the representation of an integer or floating point value.

RETURNS

A "float" equivalent of the value in `string`. If the value is not a properly formed integer or floating point value, the function returns na.

str.tostring() 4 overloads

SYNTAX & OVERLOADS

```
str.tostring(value, format) → simple string
```

```
str.tostring(value, format) → series string
```

```
str.tostring(value) → simple string
```

```
str.tostring(value) → series string
```

ARGUMENTS

value (simple int/float) Value or array ID whose elements are converted to a string.

format (simple string) Format string. Accepts these format.* constants: format.mintick, format.percent, format.volume. Optional. The default value is '#.##########'.

RETURNS

The string representation of the `value` argument.

If the `value` argument is a string, it is returned as is.

When the `value` is na, the function returns the string "NaN".

REMARKS

The formatting of float values will also round those values when necessary, e.g. str.tostring(3.99, '#') will return "4".

To display trailing zeros, use '0' instead of '#'. For example, '#.000'.

When using format.mintick, the value will be rounded to the nearest number that can be divided by syminfo.mintick without the remainder. The string is returned with trailing zeros.

If the x argument is a string, the same string value will be returned.

Bool type arguments return "true" or "false".

When x is na, the function returns "NaN".

str.trim() 4 overloads

Constructs a new string with all consecutive whitespaces and other control characters (e.g., “\n”, “\t”, etc.) removed from the left and right of the `source`.

SYNTAX & OVERLOADS

```
str.trim(source) → const string
```

```
str.trim(source) → input string
```

```
str.trim(source) → simple string
```

```
str.trim(source) → series string
```

ARGUMENTS

source (const string) String to trim.

EXAMPLE

```
//@version=5
indicator("str.trim")
trim = str.trim("    abc     ") // Returns "abc"
label.new(bar_index,close,trim)
```

REMARKS

Returns an empty string ("") if the result is empty after the trim or if the `source` is na.

str.upper() 3 overloads

Returns a new string with all letters converted to uppercase.

SYNTAX & OVERLOADS

```
str.upper(source) → const string
```

```
str.upper(source) → simple string
```

```
str.upper(source) → series string
```

ARGUMENTS

source (const string) String to be converted.

RETURNS

A new string with all letters converted to uppercase.

SEE ALSO

str.lower

strategy()

This declaration statement designates the script as a strategy and sets a number of strategy-related properties.

SYNTAX

```
strategy(title, shorttitle, overlay, format, precision, scale, pyramiding,
calc_on_order_fills, calc_on_every_tick, max_bars_back, backtest_fill_limits_assumption,
default_qty_type, default_qty_value, initial_capital, currency, slippage, commission_type,
commission_value, process_orders_on_close, close_entries_rule, margin_long, margin_short,
explicit_plot_zorder, max_lines_count, max_labels_count, max_boxes_count, calc_bars_count,
risk_free_rate, use_bar_magnifier, fill_orders_on_standard_ohlc, max_polylines_count,
dynamic_requests, behind_chart) → void
```

ARGUMENTS

title (const string) The title of the script. It is displayed on the chart when no `shorttitle` argument is used, and becomes the publication's default title when publishing the script.

shorttitle (const string) The script's display name on charts. If specified, it will replace the `title` argument in most chart-related windows. Optional. The default is the argument used for `title`.

overlay (const bool) If true, the strategy will be displayed over the chart. If false, it will be added in a separate pane. Strategy-specific labels that display entries and exits will be displayed over the main chart regardless of this setting. Optional. The default is false.

format (const string) Specifies the formatting of the script's displayed values. Possible values: format.inherit, format.price, format.volume, format.percent. Optional. The default is format.inherit.

precision (const int) Specifies the number of digits after the floating point of the script's displayed values. Must be a non-negative integer no greater than 16. If `format` is set to format.inherit and `precision` is specified, the format will instead be set to format.price. When the function's `format` parameter uses format.volume, the `precision` parameter will not affect the result, as the decimal precision rules defined by format.volume supersede other precision settings. Optional. The default is inherited from the precision of the chart's symbol.

scale (const scale_type) The price scale used. Possible values: scale.right, scale.left, scale.none. The scale.none value can only be applied in combination with `overlay = true`. Optional. By default, the script uses the same scale as the chart.

pyramiding (const int) The maximum number of entries allowed in the same direction. If the value is 0, only one entry order in the same direction can be opened, and additional entry orders are rejected. This setting can also be changed in the strategy's "Settings/Properties" tab. Optional. The default is 0.

calc_on_order_fills (const bool) Specifies whether the strategy should be recalculated after an order is filled. If true, the strategy recalculates after an order is filled, as opposed to recalculating only when the bar closes. This setting can also be changed in the strategy's "Settings/Properties" tab. Optional. The default is false.

calc_on_every_tick (const bool) Specifies whether the strategy should be recalculated on each realtime tick. If true, when the strategy is running on a realtime bar, it will recalculate on each chart update. If false, the strategy only calculates when the realtime bar closes. The argument used does not affect strategy calculation on historical data. This setting can also be changed in the strategy's "Settings/Properties" tab. Optional. The default is false.

max_bars_back (const int) The length of the historical buffer the script keeps for every variable and function, which determines how many past values can be referenced using the `[]` history-referencing operator. The required buffer size is automatically detected by the Pine Script® runtime. Using this parameter is only necessary when a runtime error occurs because automatic detection fails. More information on the underlying mechanics of the historical buffer can be found in our Help Center. Optional. The default is 0.

backtest_fill_limits_assumption (const int) Limit order execution threshold in ticks. When it is used, limit orders are only filled if the market price exceeds the order's limit level by the specified number of ticks. Optional. The default is 0.

default_qty_type (const string) Specifies the units used for `default_qty_value`. Possible values are: strategy.fixed for contracts/shares/lots, strategy.cash for currency amounts, or strategy.percent_of_equity for a percentage of available equity. This setting can also be changed in the strategy's "Settings/Properties" tab. Optional. The default is strategy.fixed.

default_qty_value (const int/float) The default quantity to trade, in units determined by the argument used with the `default_qty_type` parameter. This setting can also be changed in the strategy's "Settings/Properties" tab. Optional. The default is 1.

initial_capital (const int/float) The amount of funds initially available for the strategy to trade, in units of `currency`. Optional. The default is 1000000.

currency (const string) Currency used by the strategy in currency-related calculations. Market positions are still opened by converting `currency` into the chart symbol's currency. The conversion rates used are based on the FX_IDC pairs' daily rates of the previous day (relative to the bar where the calculation is done). This setting can also be changed in the strategy's "Settings/Properties" tab. Optional. The default is currency.NONE, in which case the chart's currency is used. Possible values: one of the constants in the `currency.*` namespace, e.g. currency.USD.

slippage (const int) Slippage expressed in ticks. This value is added to or subtracted from the fill price of market/stop orders to make the fill price less favorable for the strategy. E.g., if syminfo.mintick is 0.01 and `slippage` is set to 5, a long market order will enter at 5 * 0.01 = 0.05 points above the actual price. This setting can also be changed in the strategy's "Settings/Properties" tab. Optional. The default is 0.

commission_type (const string) Determines what the number passed to the `commission_value` expresses: strategy.commission.percent for a percentage of the cash volume of the order, strategy.commission.cash_per_contract for currency per contract, strategy.commission.cash_per_order for currency per order. This setting can also be changed in the strategy's "Settings/Properties" tab. Optional. The default is strategy.commission.percent.

commission_value (const int/float) Commission applied to the strategy's orders in units determined by the argument passed to the `commission_type` parameter. This setting can also be changed in the strategy's "Settings/Properties" tab. Optional. The default is 0.

process_orders_on_close (const bool) When set to true, generates an additional attempt to execute orders after a bar closes and strategy calculations are completed. If the orders are market orders, the broker emulator executes them before the next bar's open. If the orders are price-dependent, they will only be filled if the price conditions are met. This option is useful if you wish to close positions on the current bar. This setting can also be changed in the strategy's "Settings/Properties" tab. Optional. The default is false.

close_entries_rule (const string) Determines the order in which trades are closed. Possible values are: "FIFO" (First-In, First-Out) if the earliest exit order must close the earliest entry order, or "ANY" if the orders are closed based on the `from_entry` parameter of the strategy.exit function. "FIFO" can only be used with stocks, futures and US forex (NFA Compliance Rule 2-43b), while "ANY" is allowed in non-US forex. Optional. The default is

"FIFO".

margin_long (const int/float) Margin long is the percentage of the purchase price of a security that must be covered by cash or collateral for long positions. Must be a non-negative number. The logic used to simulate margin calls is explained in the Help Center. This setting can also be changed in the strategy's "Settings/Properties" tab. Optional. The default is 0, in which case the strategy does not enforce any limits on position size.

margin_short (const int/float) Margin short is the percentage of the purchase price of a security that must be covered by cash or collateral for short positions. Must be a non-negative number. The logic used to simulate margin calls is explained in the Help Center. This setting can also be changed in the strategy's "Settings/Properties" tab. Optional. The default is 0, in which case the strategy does not enforce any limits on position size.

explicit_plot_zorder (const bool) Specifies the order in which the script's plots, fills, and hlines are rendered. If true, plots are drawn in the order in which they appear in the script's code, each newer plot being drawn above the previous ones. This only applies to `plot*()` functions, fill, and hline. Optional. The default is false.

max_lines_count (const int) The number of last line drawings displayed. Possible values: 1-500. Optional. The default is 50.

max_labels_count (const int) The number of last label drawings displayed. Possible values: 1-500. Optional. The default is 50.

max_boxes_count (const int) The number of last box drawings displayed. Possible values: 1-500. Optional. The default is 50.

calc_bars_count (const int) Limits the initial calculation of a script to the last number of bars specified. When specified, a "Calculated bars" field will be included in the "Calculation" section of the script's "Settings/Inputs" tab. Optional. The default is 0, in which case the script executes on all available bars.

risk_free_rate (const int/float) The risk-free rate of return is the annual percentage change in the value of an investment with minimal or zero risk. It is used to calculate the Sharpe and Sortino ratios. Optional. The default is 2.

use_bar_magnifier (const bool) When true, the Broker Emulator uses lower timeframe data during history backtesting to achieve more realistic results. Optional. The default is false. Only Premium accounts have access to this feature.

fill_orders_on_standard_ohlc (const bool) When true, forces strategies running on Heikin Ashi charts to fill orders using actual OHLC prices, for more realistic results. Optional. The default is false.

max_polylines_count (const int) The number of last polyline drawings displayed. Possible values: 1-100. The count is approximate; more drawings than the specified count may be displayed. Optional. The default is 50.

dynamic_requests (const bool) Specifies whether the script can dynamically call functions from the `request.*()` namespace. Dynamic `request.*()` calls are allowed within the local scopes of conditional structures (e.g., if), loops (e.g., for), and exported functions. Additionally, such calls allow "series" arguments for many of their parameters. Optional. The default is false. See the User Manual's Dynamic requests section for more information.

behind_chart (const bool) Controls whether the script's plots and drawings in the main chart pane appear behind the chart display (if true), or in front of it (if false). Optional. The default is true.

EXAMPLE

```
//@version=5
strategy("My strategy", overlay = true, margin_long = 100, margin_short = 100)

// Enter long by market if current open is greater than previous high.
if open > high[1]
    strategy.entry("Long", strategy.long, 1)
// Generate a full exit bracket (profit 10 points, loss 5 points per contract) from the er
strategy.exit("Exit", "Long", profit = 10, loss = 5)
```

REMARKS

You can learn more about strategies in our User Manual.

Every strategy script must have one strategy call.

Strategies using `calc_on_every_tick = true` parameter may calculate differently on historical and realtime bars, which causes repainting.

Strategies always use the chart's prices to enter and exit positions. Using them on non-standard chart types (Heikin Ashi, Renko, etc.) will produce misleading results, as their prices are synthetic. Backtesting on non-standard charts is thus not recommended.

SEE ALSO

indicator   library

strategy.cancel()

Cancels a pending or unfilled order with a specific identifier. If multiple unfilled orders share the same ID, calling this command with that ID as the `id` argument cancels all of them. If a script calls this command with an `id` representing the ID of a filled order, it has no effect.

This command is most useful when working with price-based orders (e.g., limit orders). Calls to this command can also cancel market orders, but only if they execute on the same ticks as the order placement commands.

SYNTAX

```
strategy.cancel(id) → void
```

ARGUMENTS

id (series string) The identifier of the unfilled order to cancel.

EXAMPLE

```
//@version=5
strategy(title = "Order cancellation demo")

conditionForBuy = open > high[1]
if conditionForBuy
    strategy.entry("Long", strategy.long, 1, limit = low) // Enter long using limit order
if not conditionForBuy
    strategy.cancel("Long") // Cancel the entry order with name "Long" if `conditionForBuy
```

strategy.cancel_all()

Cancels all pending or unfilled orders, regardless of their identifiers.

This command is most useful when working with price-based orders (e.g., limit orders). Calls to this command can also cancel market orders, but only if they execute on the same ticks as the order placement commands.

SYNTAX

```
strategy.cancel_all() → void
```

EXAMPLE

```
//@version=5
strategy(title = "Cancel all orders demo")
conditionForBuy1 = open > high[1]
if conditionForBuy1
    strategy.entry("Long entry 1", strategy.long, 1, limit = low) // Enter long using a li
conditionForBuy2 = conditionForBuy1 and open[1] > high[2]
float lowest2 = ta.lowest(low, 2)
if conditionForBuy2
    strategy.entry("Long entry 2", strategy.long, 1, limit = lowest2) // Enter long using
conditionForStopTrading = open < lowest2
if conditionForStopTrading
    strategy.cancel_all() // Cancel both limit orders if `conditionForStopTrading` is `tru
```

strategy.close()

Creates an order to exit from the part of a position opened by entry orders with a specific identifier. If multiple entries in the position share the same ID, the orders from this command apply to all those entries, starting from the first open trade, when its calls use that ID as the `id` argument.

This command always generates market orders. To exit from a position using price-based orders (e.g., stop-loss orders), use the strategy.exit command.

SYNTAX

```
strategy.close(id, comment, qty, qty_percent, alert_message, immediately, disable_alert) →
void
```

ARGUMENTS

id (series string) The entry identifier of the open trades to close.

comment (series string) Optional. Additional notes on the filled order. If the value is not an empty string, the Strategy Tester and the chart show this text for the order instead of the automatically generated exit identifier. The default is an empty string.

qty (series int/float) Optional. The number of contracts/lots/shares/units to close when an exit order fills. If specified, the command uses this value instead of `qty_percent` to determine the order size. The default is na, which means the order size depends on the `qty_percent` value.

qty_percent (series int/float) Optional. A value between 0 and 100 representing the percentage of the open trade quantity to close when an exit order fills. The percentage calculation depends on the total size of the open trades with the `id` entry identifier. The command ignores this parameter if the `qty` value is not na. The default is 100.

alert_message (series string) Optional. Custom text for the alert that fires when an order fills. If the "Message" field of the "Create Alert" dialog box contains the `{{strategy.order.alert_message}}` placeholder, the alert message replaces the placeholder with this text. The default is an empty string.

immediately (series bool) Optional. If true, the closing order executes on the same tick when the strategy places it, ignoring the strategy properties that restrict execution to the opening tick of the following bar. The default is false.

disable_alert (series bool) Optional. If true when the command creates an order, the strategy does not trigger an alert when that order fills. This parameter accepts a "series" value, meaning users can control which orders trigger alerts when they execute. The default is false.

EXAMPLE

```
//@version=5
strategy("Partial close strategy",  margin_long = 100, margin_short = 100)

// Calculate a 14-bar and 28-bar moving average of `close` prices.
float sma14 = ta.sma(close, 14)
float sma28 = ta.sma(close, 28)

// Place a market order to enter a long position when `sma14` crosses over `sma28`.
if ta.crossover(sma14, sma28)
    strategy.entry("My Long Entry ID", strategy.long)

// Place a market order to close the long trade when `sma14` crosses under `sma28`.
if ta.crossunder(sma14, sma28)
    strategy.close("My Long Entry ID", "50% market close", qty_percent = 50)

// Plot the position size.
plot(strategy.position_size)
```

REMARKS

When a position consists of several open trades and the `close_entries_rule` in the strategy declaration statement is "FIFO" (default), a strategy.close call exits from the position starting with the first open trade. This behavior applies even if the `id` value is the entry ID of different open trades. However, in that case, the maximum exit order size still depends on the trades opened by orders with the `id` identifier. For more information, see this section of our User Manual.

strategy.close_all() 2 overloads

Creates an order to close an open position completely, regardless of the identifiers of the entry orders that opened or added to it.

This command always generates market orders. To exit from a position using price-based orders (e.g., stop-loss orders), use the strategy.exit command.

SYNTAX & OVERLOADS

```
strategy.close_all(comment, alert_message) → void
strategy.close_all(comment, alert_message, immediately, disable_alert) → void
```

ARGUMENTS

comment (series string) Optional. Additional notes on the filled order. If the value is not an empty string, the Strategy Tester and the chart show this text for the order instead of the automatically generated exit identifier. The default is an empty string.

alert_message (series string) Optional. Custom text for the alert that fires when an order fills. If the "Message" field of the "Create Alert" dialog box contains the `{{strategy.order.alert_message}}` placeholder, the alert message replaces the placeholder with this text. The default is an empty string.

EXAMPLE

```
//@version=5
strategy("Multi-entry close strategy",  margin_long = 100, margin_short = 100)

// Calculate a 14-bar and 28-bar moving average of `close` prices.
float sma14 = ta.sma(close, 14)
float sma28 = ta.sma(close, 28)

// Place a market order to enter a long trade every time `sma14` crosses over `sma28`.
if ta.crossover(sma14, sma28)
    strategy.order("My Long Entry ID " + str.tostring(strategy.opentrades), strategy.long)

// Place a market order to close the entire position every 500 bars.
if bar_index % 500 == 0
    strategy.close_all()

// Plot the position size.
plot(strategy.position_size)
```

strategy.closedtrades.commission()

Returns the sum of entry and exit fees paid in the closed trade, expressed in strategy.account_currency.

SYNTAX

```
strategy.closedtrades.commission(trade_num) → series float
```

ARGUMENTS

trade_num (series int) The trade number of the closed trade. The number of the first trade is zero.

EXAMPLE

```
//@version=5
strategy("`strategy.closedtrades.commission` Example", commission_type = strategy.commissi

// Strategy calls to enter long trades every 15 bars and exit long trades every 20 bars.
if bar_index % 15 == 0
    strategy.entry("Long", strategy.long)
if bar_index % 20 == 0
    strategy.close("Long")

// Plot total fees for the latest closed trade.
plot(strategy.closedtrades.commission(strategy.closedtrades - 1))
```

SEE ALSO

strategy   strategy.opentrades.commission

strategy.closedtrades.entry_bar_index()

Returns the bar_index of the closed trade's entry.

SYNTAX

```
strategy.closedtrades.entry_bar_index(trade_num) → series int
```

ARGUMENTS

trade_num (series int) The trade number of the closed trade. The number of the first trade is zero.

EXAMPLE

```
//@version=5
strategy("strategy.closedtrades.entry_bar_index Example")
// Enter long trades on three rising bars; exit on two falling bars.
if ta.rising(close, 3)
    strategy.entry("Long", strategy.long)
if ta.falling(close, 2)
    strategy.close("Long")
// Function that calculates the average amount of bars in a trade.
avgBarsPerTrade() =>
    sumBarsPerTrade = 0
    for tradeNo = 0 to strategy.closedtrades - 1
        // Loop through all closed trades, starting with the oldest.
        sumBarsPerTrade += strategy.closedtrades.exit_bar_index(tradeNo) - strategy.closed
    result = nz(sumBarsPerTrade / strategy.closedtrades)
plot(avgBarsPerTrade())
```

SEE ALSO

strategy.closedtrades.exit_bar_index   strategy.opentrades.entry_bar_index

strategy.closedtrades.entry_comment()

Returns the comment message of the closed trade's entry, or na if there is no entry with this `trade_num`.

SYNTAX

```
strategy.closedtrades.entry_comment(trade_num) → series string
```

ARGUMENTS

trade_num (series int) The trade number of the closed trade. The number of the first trade is zero.

EXAMPLE

```
//@version=5
strategy("`strategy.closedtrades.entry_comment()` Example", overlay = true)

stopPrice = open * 1.01

longCondition = ta.crossover(ta.sma(close, 14), ta.sma(close, 28))

if (longCondition)
    strategy.entry("Long", strategy.long, stop = stopPrice, comment = str.tostring(stopPri
    strategy.exit("EXIT", trail_points = 1000, trail_offset = 0)

var testTable = table.new(position.top_right, 1, 3, color.orange, border_width = 1)

if barstate.islastconfirmedhistory or barstate.isrealtime
    table.cell(testTable, 0, 0, 'Last closed trade:')
    table.cell(testTable, 0, 1, "Order stop price value: " + strategy.closedtrades.entry_c
    table.cell(testTable, 0, 2, "Actual Entry Price: " + str.tostring(strategy.closedtrade
```

SEE ALSO

strategy strategy.entry strategy.closedtrades

strategy.closedtrades.entry_id()

Returns the id of the closed trade's entry.

SYNTAX

```
strategy.closedtrades.entry_id(trade_num) → series string
```

ARGUMENTS

trade_num (series int) The trade number of the closed trade. The number of the first trade is zero.

EXAMPLE

```
//@version=5
strategy("strategy.closedtrades.entry_id Example", overlay = true)

// Enter a short position and close at the previous to last bar.
if bar_index == 1
    strategy.entry("Short at bar #" + str.tostring(bar_index), strategy.short)
if bar_index == last_bar_index - 2
    strategy.close_all()

// Display ID of the last entry position.
if barstate.islastconfirmedhistory
    label.new(last_bar_index, high, "Last Entry ID is: " + strategy.closedtrades.entry_id(
```

RETURNS

Returns the id of the closed trade's entry.

REMARKS

The function returns na if trade_num is not in the range: 0 to strategy.closedtrades-1.

SEE ALSO

strategy.closedtrades.entry_bar_index strategy.closedtrades.entry_price

strategy.closedtrades.entry_time

strategy.closedtrades.entry_price()

Returns the price of the closed trade's entry.

SYNTAX

```
strategy.closedtrades.entry_price(trade_num) → series float
```

ARGUMENTS

trade_num (series int) The trade number of the closed trade. The number of the first trade is zero.

EXAMPLE

```
//@version=5
strategy("strategy.closedtrades.entry_price Example 1")

// Strategy calls to enter long trades every 15 bars and exit long trades every 20 bars.
if bar_index % 15 == 0
    strategy.entry("Long", strategy.long)
if bar_index % 20 == 0
    strategy.close("Long")

// Return the entry price for the latest  entry.
entryPrice = strategy.closedtrades.entry_price(strategy.closedtrades - 1)

plot(entryPrice, "Long entry price")
```

EXAMPLE

```
// Calculates the average profit percentage for all closed trades.
//@version=5
strategy("strategy.closedtrades.entry_price Example 2")

// Strategy calls to create single short and long trades
if bar_index == last_bar_index - 15
    strategy.entry("Long Entry",  strategy.long)
else if bar_index == last_bar_index - 10
    strategy.close("Long Entry")
    strategy.entry("Short", strategy.short)
else if bar_index == last_bar_index - 5
    strategy.close("Short")

// Calculate profit for both closed trades.
profitPct = 0.0
for tradeNo = 0 to strategy.closedtrades - 1
    entryP = strategy.closedtrades.entry_price(tradeNo)
```

```
        exitP = strategy.closedtrades.exit_price(tradeNo)
        profitPct += (exitP - entryP) / entryP * strategy.closedtrades.size(tradeNo) * 100

    // Calculate average profit percent for both closed trades.
    avgProfitPct = nz(profitPct / strategy.closedtrades)

    plot(avgProfitPct)
```

SEE ALSO

strategy.closedtrades.entry_price strategy.closedtrades.exit_price strategy.closedtrades.size

strategy.closedtrades

strategy.closedtrades.entry_time()

Returns the UNIX time of the closed trade's entry, expressed in milliseconds..

SYNTAX

```
strategy.closedtrades.entry_time(trade_num) → series int
```

ARGUMENTS

trade_num (series int) The trade number of the closed trade. The number of the first trade is zero.

EXAMPLE

```
    //@version=5
    strategy("strategy.closedtrades.entry_time Example", overlay = true)

    // Enter long trades on three rising bars; exit on two falling bars.
    if ta.rising(close, 3)
        strategy.entry("Long", strategy.long)
    if ta.falling(close, 2)
        strategy.close("Long")

    // Calculate the average trade duration
    avgTradeDuration() =>
        sumTradeDuration = 0
        for i = 0 to strategy.closedtrades - 1
            sumTradeDuration += strategy.closedtrades.exit_time(i) - strategy.closedtrades.ent
        result = nz(sumTradeDuration / strategy.closedtrades)

    // Display average duration converted to seconds and formatted using 2 decimal points
    if barstate.islastconfirmedhistory
        label.new(bar_index, high, str.tostring(avgTradeDuration() / 1000, "#.##") + " seconds
```

SEE ALSO

strategy.opentrades.entry_time strategy.closedtrades.exit_time time

strategy.closedtrades.exit_bar_index()

Returns the bar_index of the closed trade's exit.

SYNTAX

```
strategy.closedtrades.exit_bar_index(trade_num) → series int
```

ARGUMENTS

trade_num (series int) The trade number of the closed trade. The number of the first trade is zero.

EXAMPLE

```
//@version=5
strategy("strategy.closedtrades.exit_bar_index Example 1")

// Strategy calls to place a single short trade. We enter the trade at the first bar and e
if bar_index == 0
    strategy.entry("Short",  strategy.short)
if bar_index == last_bar_index - 10
    strategy.close("Short")

// Calculate the amount of bars since the last closed trade.
barsSinceClosed = strategy.closedtrades > 0 ? bar_index - strategy.closedtrades.exit_bar_i

plot(barsSinceClosed, "Bars since last closed trade")
```

EXAMPLE

```
// Calculates the average amount of bars per trade.
//@version=5
strategy("strategy.closedtrades.exit_bar_index Example 2")

// Enter long trades on three rising bars; exit on two falling bars.
if ta.rising(close, 3)
    strategy.entry("Long", strategy.long)
if ta.falling(close, 2)
```

```
    strategy.close("Long")

// Function that calculates the average amount of bars per trade.
avgBarsPerTrade() =>
    sumBarsPerTrade = 0
    for tradeNo = 0 to strategy.closedtrades - 1
        // Loop through all closed trades, starting with the oldest.
        sumBarsPerTrade += strategy.closedtrades.exit_bar_index(tradeNo) - strategy.closed
    result = nz(sumBarsPerTrade / strategy.closedtrades)

plot(avgBarsPerTrade())
```

SEE ALSO

bar_index   last_bar_index

strategy.closedtrades.exit_comment()

Returns the comment message of the closed trade's exit, or na if there is no entry with this `trade_num`.

SYNTAX

```
strategy.closedtrades.exit_comment(trade_num) → series string
```

ARGUMENTS

trade_num (series int) The trade number of the closed trade. The number of the first trade is zero.

EXAMPLE

```
//@version=5
strategy("`strategy.closedtrades.exit_comment()` Example", overlay = true)

longCondition = ta.crossover(ta.sma(close, 14), ta.sma(close, 28))
if (longCondition)
    strategy.entry("Long", strategy.long)
    strategy.exit("Exit", stop = open * 0.95, limit = close * 1.05, trail_points = 100, tr

exitStats() =>
    int slCount = 0
    int tpCount = 0
    int trailCount = 0

    if strategy.closedtrades > 0
        for i = 0 to strategy.closedtrades - 1
```

```
            switch strategy.closedtrades.exit_comment(i)
                "TP"    => tpCount    += 1
                "SL"    => slCount    += 1
                "TRAIL" => trailCount += 1
        [slCount, tpCount, trailCount]

    var testTable = table.new(position.top_right, 1, 4, color.orange, border_width = 1)

    if barstate.islastconfirmedhistory
        [slCount, tpCount, trailCount] = exitStats()
        table.cell(testTable, 0, 0, "Closed trades (" + str.tostring(strategy.closedtrades) +'
        table.cell(testTable, 0, 1, "Stop Loss: " + str.tostring(slCount))
        table.cell(testTable, 0, 2, "Take Profit: " + str.tostring(tpCount))
        table.cell(testTable, 0, 3, "Trailing Stop: " + str.tostring(trailCount))
```

SEE ALSO

strategy   strategy.exit   strategy.close   strategy.closedtrades

strategy.closedtrades.exit_id()

Returns the id of the closed trade's exit.

SYNTAX

```
strategy.closedtrades.exit_id(trade_num) → series string
```

ARGUMENTS

trade_num (series int) The trade number of the closed trade. The number of the first trade is zero.

EXAMPLE

```
    //@version=5
    strategy("strategy.closedtrades.exit_id Example", overlay = true)

    // Strategy calls to create single short and long trades
    if bar_index == last_bar_index - 15
        strategy.entry("Long Entry",  strategy.long)
    else if bar_index == last_bar_index - 10
        strategy.entry("Short Entry", strategy.short)

    // When a new open trade is detected then we create the exit strategy corresponding with t
    // We detect the correct entry id by determining if a position is long or short based on t
    if ta.change(strategy.opentrades) != 0
        posSign = strategy.opentrades.size(strategy.opentrades - 1)
        strategy.exit(posSign > 0 ? "SL Long Exit" : "SL Short Exit", strategy.opentrades.entr
    // When a new closed trade is detected then we place a label above the bar with the exit i
    if ta.change(strategy.closedtrades) != 0
        msg = "Trade closed by: " + strategy.closedtrades.exit_id(strategy.closedtrades - 1)
        label.new(bar_index, high + (3 * ta.tr), msg)
```

RETURNS

Returns the id of the closed trade's exit.

REMARKS

The function returns na if trade_num is not in the range: 0 to strategy.closedtrades-1.

SEE ALSO

strategy.closedtrades.exit_bar_index strategy.closedtrades.exit_price

strategy.closedtrades.exit_time

strategy.closedtrades.exit_price()

Returns the price of the closed trade's exit.

SYNTAX

```
strategy.closedtrades.exit_price(trade_num) → series float
```

ARGUMENTS

trade_num (series int) The trade number of the closed trade. The number of the first trade is zero.

EXAMPLE

```
//@version=5
strategy("strategy.closedtrades.exit_price Example 1")

// We are creating a long trade every 5 bars
if bar_index % 5 == 0
    strategy.entry("Long",  strategy.long)
strategy.close("Long")

// Return the exit price from the latest closed trade.
exitPrice = strategy.closedtrades.exit_price(strategy.closedtrades - 1)

plot(exitPrice, "Long exit price")
```

EXAMPLE

```
// Calculates the average profit percentage for all closed trades.
//@version=5
strategy("strategy.closedtrades.exit_price Example 2")

// Strategy calls to create single short and long trades.
if bar_index == last_bar_index - 15
    strategy.entry("Long Entry",  strategy.long)
else if bar_index == last_bar_index - 10
    strategy.close("Long Entry")
    strategy.entry("Short", strategy.short)
else if bar_index == last_bar_index - 5
    strategy.close("Short")

// Calculate profit for both closed trades.
profitPct = 0.0
for tradeNo = 0 to strategy.closedtrades - 1
    entryP = strategy.closedtrades.entry_price(tradeNo)
    exitP = strategy.closedtrades.exit_price(tradeNo)
    profitPct += (exitP - entryP) / entryP * strategy.closedtrades.size(tradeNo) * 100

// Calculate average profit percent for both closed trades.
avgProfitPct = nz(profitPct / strategy.closedtrades)

plot(avgProfitPct)
```

SEE ALSO

strategy.closedtrades.entry_price

strategy.closedtrades.exit_time()

Returns the UNIX time of the closed trade's exit, expressed in milliseconds.

SYNTAX

```
strategy.closedtrades.exit_time(trade_num) → series int
```

ARGUMENTS

trade_num (series int) The trade number of the closed trade. The number of the first trade is zero.

EXAMPLE

```
//@version=5
strategy("strategy.closedtrades.exit_time Example 1")

// Enter long trades on three rising bars; exit on two falling bars.
if ta.rising(close, 3)
    strategy.entry("Long", strategy.long)
if ta.falling(close, 2)
    strategy.close("Long")

// Calculate the average trade duration.
avgTradeDuration() =>
    sumTradeDuration = 0
    for i = 0 to strategy.closedtrades - 1
        sumTradeDuration += strategy.closedtrades.exit_time(i) - strategy.closedtrades.ent
    result = nz(sumTradeDuration / strategy.closedtrades)

// Display average duration converted to seconds and formatted using 2 decimal points.
if barstate.islastconfirmedhistory
    label.new(bar_index, high, str.tostring(avgTradeDuration() / 1000, "#.##") + " seconds
```

EXAMPLE

```
// Reopens a closed trade after X seconds.
//@version=5
strategy("strategy.closedtrades.exit_time Example 2")

// Strategy calls to emulate a single long trade at the first bar.
if bar_index == 0
    strategy.entry("Long", strategy.long)

reopenPositionAfter(timeSec) =>
    if strategy.closedtrades > 0
        if time - strategy.closedtrades.exit_time(strategy.closedtrades - 1) >= timeSec *
            strategy.entry("Long", strategy.long)

// Reopen last closed position after 120 sec.
reopenPositionAfter(120)

if ta.change(strategy.opentrades) != 0
    strategy.exit("Long", stop = low * 0.9, profit = high * 2.5)
```

SEE ALSO

strategy.closedtrades.entry_time

strategy.closedtrades.max_drawdown()

Returns the maximum drawdown of the closed trade, i.e., the maximum possible loss during the trade, expressed in strategy.account_currency.

SYNTAX

```
strategy.closedtrades.max_drawdown(trade_num) → series float
```

ARGUMENTS

trade_num (series int) The trade number of the closed trade. The number of the first trade is zero.

EXAMPLE

```
//@version=5
strategy("`strategy.closedtrades.max_drawdown` Example")

// Strategy calls to enter long trades every 15 bars and exit long trades every 20 bars.
if bar_index % 15 == 0
    strategy.entry("Long", strategy.long)
if bar_index % 20 == 0
    strategy.close("Long")

// Get the biggest max trade drawdown value from all of the closed trades.
maxTradeDrawDown() =>
    maxDrawdown = 0.0
    for tradeNo = 0 to strategy.closedtrades - 1
        maxDrawdown := math.max(maxDrawdown, strategy.closedtrades.max_drawdown(tradeNo))
    result = maxDrawdown

plot(maxTradeDrawDown(), "Biggest max drawdown")
```

REMARKS

The function returns na if trade_num is not in the range: 0 to strategy.closedtrades - 1.

SEE ALSO

strategy.opentrades.max_drawdown strategy.max_drawdown

strategy.closedtrades.max_drawdown_percent()

Returns the maximum drawdown of the closed trade, i.e., the maximum possible loss during the trade, expressed as a percentage and calculated by formula: `Lowest Value During Trade / (Entry Price x Quantity) * 100`.

SYNTAX

```
strategy.closedtrades.max_drawdown_percent(trade_num) → series float
```

ARGUMENTS

trade_num (series int) The trade number of the closed trade. The number of the first trade is zero.

SEE ALSO

strategy.closedtrades.max_drawdown strategy.max_drawdown

strategy.closedtrades.max_runup()

Returns the maximum run up of the closed trade, i.e., the maximum possible profit during the trade, expressed in strategy.account_currency.

SYNTAX

```
strategy.closedtrades.max_runup(trade_num) → series float
```

ARGUMENTS

trade_num (series int) The trade number of the closed trade. The number of the first trade is zero.

EXAMPLE

```
//@version=5
strategy("`strategy.closedtrades.max_runup` Example")

// Strategy calls to enter long trades every 15 bars and exit long trades every 20 bars.
if bar_index % 15 == 0
    strategy.entry("Long", strategy.long)
if bar_index % 20 == 0
    strategy.close("Long")

// Get the biggest max trade runup value from all of the closed trades.
maxTradeRunUp() =>
    maxRunup = 0.0
    for tradeNo = 0 to strategy.closedtrades - 1
        maxRunup := math.max(maxRunup, strategy.closedtrades.max_runup(tradeNo))
    result = maxRunup

plot(maxTradeRunUp(), "Max trade runup")
```

SEE ALSO

strategy.opentrades.max_runup   strategy.max_runup

strategy.closedtrades.max_runup_percent()

Returns the maximum run-up of the closed trade, i.e., the maximum possible profit during the trade, expressed as a percentage and calculated by formula: `Highest Value During Trade / (Entry Price x Quantity) * 100`.

SYNTAX

```
strategy.closedtrades.max_runup_percent(trade_num) → series float
```

ARGUMENTS

trade_num (series int) The trade number of the closed trade. The number of the first trade is zero.

SEE ALSO

strategy.closedtrades.max_runup   strategy.max_runup

strategy.closedtrades.profit()

Returns the profit/loss of the closed trade, expressed in strategy.account_currency. Losses are expressed as negative values.

SYNTAX

```
strategy.closedtrades.profit(trade_num) → series float
```

ARGUMENTS

trade_num (series int) The trade number of the closed trade. The number of the first trade is zero.

EXAMPLE

```
//@version=5
strategy("`strategy.closedtrades.profit` Example")

// Strategy calls to enter long trades every 15 bars and exit long trades every 20 bars.
if bar_index % 15 == 0
```

```
    strategy.entry("Long", strategy.long)
if bar_index % 20 == 0
    strategy.close("Long")

// Calculate average gross profit by adding the difference between gross profit and commis
avgGrossProfit() =>
    sumGrossProfit = 0.0
    for tradeNo = 0 to strategy.closedtrades - 1
        sumGrossProfit += strategy.closedtrades.profit(tradeNo) - strategy.closedtrades.co
    result = nz(sumGrossProfit / strategy.closedtrades)

plot(avgGrossProfit(), "Average gross profit")
```

SEE ALSO

strategy.opentrades.profit strategy.closedtrades.commission

strategy.closedtrades.profit_percent()

Returns the profit/loss value of the closed trade, expressed as a percentage. Losses are expressed as negative values.

SYNTAX

```
strategy.closedtrades.profit_percent(trade_num) → series float
```

ARGUMENTS

trade_num (series int) The trade number of the closed trade. The number of the first trade is zero.

SEE ALSO

strategy.closedtrades.profit

strategy.closedtrades.size()

Returns the direction and the number of contracts traded in the closed trade. If the value is > 0, the market position was long. If the value is < 0, the market position was short.

SYNTAX

```
strategy.closedtrades.size(trade_num) → series float
```

ARGUMENTS

trade_num (series int) The trade number of the closed trade. The number of the first trade is zero.

EXAMPLE

```
//@version=5
strategy("`strategy.closedtrades.size` Example 1")

// We calculate the max amt of shares we can buy.
amtShares = math.floor(strategy.equity / close)
// Strategy calls to enter long trades every 15 bars and exit long trades every 20 bars
if bar_index % 15 == 0
    strategy.entry("Long", strategy.long, qty = amtShares)
if bar_index % 20 == 0
    strategy.close("Long")

// Plot the number of contracts traded in the last closed trade.
plot(strategy.closedtrades.size(strategy.closedtrades - 1), "Number of contracts traded")
```

EXAMPLE

```
// Calculates the average profit percentage for all closed trades.
//@version=5
strategy("`strategy.closedtrades.size` Example 2")

// Strategy calls to enter long trades every 15 bars and exit long trades every 20 bars.
if bar_index % 15 == 0
    strategy.entry("Long", strategy.long)
if bar_index % 20 == 0
    strategy.close("Long")


// Calculate profit for both closed trades.
profitPct = 0.0
for tradeNo = 0 to strategy.closedtrades - 1
    entryP = strategy.closedtrades.entry_price(tradeNo)
    exitP = strategy.closedtrades.exit_price(tradeNo)
    profitPct += (exitP - entryP) / entryP * strategy.closedtrades.size(tradeNo) * 100

// Calculate average profit percent for both closed trades.
avgProfitPct = nz(profitPct / strategy.closedtrades)

plot(avgProfitPct)
```

SEE ALSO

strategy.opentrades.size strategy.position_size strategy.closedtrades strategy.opentrades

strategy.convert_to_account()

Converts the value from the currency that the symbol on the chart is traded in (syminfo.currency) to the currency used by the strategy (strategy.account_currency).

SYNTAX

```
strategy.convert_to_account(value) → series float
```

ARGUMENTS

value (series int/float) The value to be converted.

EXAMPLE

```
//@version=5
strategy("`strategy.convert_to_account` Example 1", currency = currency.EUR)

plot(close, "Close price using default currency")
plot(strategy.convert_to_account(close), "Close price converted to strategy currency")
```

EXAMPLE

```
// Calculates the "Buy and hold return" using your account's currency.
//@version=5
strategy("`strategy.convert_to_account` Example 2", currency = currency.EUR)

dateInput = input.time(timestamp("20 Jul 2021 00:00 +0300"), "From Date", confirm = true)

buyAndHoldReturnPct(fromDate) =>
    if time >= fromDate
        money = close * syminfo.pointvalue
        var initialBal = strategy.convert_to_account(money)
        (strategy.convert_to_account(money) - initialBal) / initialBal * 100

plot(buyAndHoldReturnPct(dateInput))
```

SEE ALSO

strategy strategy.convert_to_symbol

strategy.convert_to_symbol()

Converts the value from the currency used by the strategy (strategy.account_currency) to the currency that the symbol on the chart is traded in (syminfo.currency).

SYNTAX

```
strategy.convert_to_symbol(value) → series float
```

ARGUMENTS

value (series int/float) The value to be converted.

EXAMPLE

```
//@version=5
strategy("`strategy.convert_to_symbol` Example", currency = currency.EUR)

// Calculate the max qty we can buy using current chart's currency.
calcContracts(accountMoney) =>
    math.floor(strategy.convert_to_symbol(accountMoney) / syminfo.pointvalue / close)

// Return max qty we can buy using 300 euros
qt = calcContracts(300)

// Strategy calls to enter long trades every 15 bars and exit long trades every 20 bars us
if bar_index % 15 == 0
    strategy.entry("Long", strategy.long, qty = qt)
if bar_index % 20 == 0
    strategy.close("Long")
```

SEE ALSO

strategy strategy.convert_to_account

strategy.default_entry_qty()

Calculates the default quantity, in units, of an entry order from strategy.entry or strategy.order if it were to fill at the specified `fill_price` value. The calculation depends on several strategy properties, including `default_qty_type`, `default_qty_value`, `currency`, and other parameters in the strategy function and their representation in the "Properties" tab of the strategy's settings.

SYNTAX

```
strategy.default_entry_qty(fill_price) → series float
```

ARGUMENTS

fill_price (series int/float) The fill price for which to calculate the default order quantity.

EXAMPLE

```
//@version=5
strategy("Supertrend Strategy", overlay = true, default_qty_type = strategy.percent_of_equ

//@variable The length of the ATR calculation.
atrPeriod = input(10, "ATR Length")
//@variable The ATR multiplier.
factor = input.float(3.0, "Factor", step = 0.01)
//@variable The tick offset of the stop order.
stopOffsetInput = input.int(100, "Tick offset for entry stop")

// Get the direction of the SuperTrend.
[_, direction] = ta.supertrend(factor, atrPeriod)

if ta.change(direction) < 0
    //@variable The stop price of the entry order.
    stopPrice = close + syminfo.mintick * stopOffsetInput
    //@variable The expected default fill quantity at the `stopPrice`. This value may not
    calculatedQty = strategy.default_entry_qty(stopPrice)
    strategy.entry("My Long Entry Id", strategy.long, stop = stopPrice)
    label.new(bar_index, stopPrice, str.format("Stop set at {0}\nExpected qty at {0}: {1}"

if ta.change(direction) > 0
    strategy.close_all()
```

REMARKS

This function does not consider open positions simulated by a strategy. For example, if a strategy script has an open position from a long order with a `qty` of 10 units, using the strategy.entry function to simulate a short order with a `qty` of 5 will prompt the script to sell 15 units to reverse the position. This function will still return 5 in such a case since it doesn't consider an open trade.

This value represents the default calculated quantity of an order.

Order placement commands can override the default value by explicitly passing a new `qty` value in the function call.

strategy.entry()

Creates a new order to open or add to a position. If an unfilled order with the same `id` exists, a call to this command modifies that order.

The resulting order's type depends on the `limit` and `stop` parameters. If the call does not contain `limit` or `stop` arguments, it creates a market order that executes on the next tick. If the call specifies a `limit` value but no `stop` value, it places a limit order that executes after the market price reaches the `limit` value or a better price (lower for buy orders and higher for sell orders). If the call specifies a `stop` value but no `limit` value, it places a stop order that executes after the market price reaches the `stop` value or a worse price (higher for buy orders and lower for sell orders). If the call contains `limit` and `stop` arguments, it creates a stop-limit order, which generates a limit order at the `limit` price only after the market price reaches the `stop` value or a worse price.

Orders from this command, unlike those from strategy.order, are affected by the `pyramiding` parameter of the strategy declaration statement. Pyramiding specifies the number of concurrent open entries allowed per position. For example, with `pyramiding = 3`, the strategy can have up to three open trades, and the command cannot create orders to open additional trades until at least one existing trade closes.

By default, when a strategy executes an order from this command in the opposite direction of the current market position, it reverses that position. For example, if there is an open long position of five shares, an order from this command with a `qty` of 5 and a `direction` of strategy.short triggers the sale of 10 shares to close the long position and open a new five-share short position. Users can change this behavior by specifying an allowed direction with the strategy.risk_allow_entry_in function.

SYNTAX

```
strategy.entry(id, direction, qty, limit, stop, oca_name, oca_type, comment,
alert_message, disable_alert) → void
```

ARGUMENTS

id (series string) The identifier of the order, which corresponds to an entry ID in the strategy's trades after the order fills. If the strategy opens a new position after filling the order, the order's ID becomes the strategy.position_entry_name value. Strategy commands can reference the order ID to cancel or modify pending orders and generate exit orders for specific open trades. The Strategy Tester and the chart display the order ID unless the command specifies a `comment` value.

direction (series strategy_direction) The direction of the trade. Possible values: strategy.long for a long trade, strategy.short for a short one.

qty (series int/float) Optional. The number of contracts/shares/lots/units in the resulting open trade when the order fills. The default is na, which means that the command uses the `default_qty_type` and `default_qty_value` parameters of the strategy declaration statement to determine the quantity.

limit (series int/float) Optional. The limit price of the order. If specified, the command creates a limit or stop-limit order, depending on whether the `stop` value is also specified. The default is na, which means the resulting order is not of the limit or stop-limit type.

stop (series int/float) Optional. The stop price of the order. If specified, the command creates a stop or stop-limit order, depending on whether the `limit` value is also specified. The default is na, which means the resulting order is not of the stop or stop-limit type.

oca_name (series string) Optional. The name of the order's One-Cancels-All (OCA) group. When a pending order with the same `oca_name` and `oca_type` parameters executes, that order affects all unfilled orders in the group. The default is an empty string, which means the order does not belong to an OCA group.

oca_type (input string) Optional. Specifies how an unfilled order behaves when another pending order with the same `oca_name` and `oca_type` values executes. Possible values: strategy.oca.cancel, strategy.oca.reduce, strategy.oca.none. The default is strategy.oca.none.

comment (series string) Optional. Additional notes on the filled order. If the value is not an empty string, the Strategy Tester and the chart show this text for the order instead of the specified `id`. The default is an empty string.

alert_message (series string) Optional. Custom text for the alert that fires when an order fills. If the "Message" field of the "Create Alert" dialog box contains the `{{strategy.order.alert_message}}` placeholder, the alert message replaces the placeholder with this text. The default is an empty string.

disable_alert (series bool) Optional. If true when the command creates an order, the strategy does not trigger an alert when that order fills. This parameter accepts a "series" value, meaning users can control which orders trigger alerts when they execute. The default is false.

EXAMPLE

```
//@version=5
strategy("Market order strategy", overlay = true, margin_long = 100, margin_short = 100)

// Calculate a 14-bar and 28-bar moving average of `close` prices.
float sma14 = ta.sma(close, 14)
float sma28 = ta.sma(close, 28)

// Place a market order to close the short trade and enter a long position when `sma14` cr
if ta.crossover(sma14, sma28)
    strategy.entry("My Long Entry ID", strategy.long)

// Place a market order to close the long trade and enter a short position when `sma14` cr
if ta.crossunder(sma14, sma28)
    strategy.entry("My Short Entry ID", strategy.short)
```

EXAMPLE

```
//@version=5
strategy("Limit order strategy", overlay=true, margin_long=100, margin_short=100)

//@variable The distance from the `close` price for each limit order.
float limitOffsetInput = input.int(100, "Limit offset, in ticks", 1) * syminfo.mintick

//@function Draws a label and line at the specified `price` to visualize a limit order's l
drawLimit(float price, bool isLong) =>
    color col = isLong ? color.blue : color.red
    label.new(
         bar_index, price, (isLong ? "Long" : "Short") + " limit order created",
         style = label.style_label_right, color = col, textcolor = color.white
     )
    line.new(bar_index, price, bar_index + 1, price, extend = extend.right, style = line.s

//@function Stops the `l` line from extending further.
method stopExtend(line l) =>
    l.set_x2(bar_index)
    l.set_extend(extend.none)

// Initialize two `line` variables to reference limit line IDs.
var line longLimit  = na
var line shortLimit = na

// Calculate a 14-bar and 28-bar moving average of `close` prices.
float sma14 = ta.sma(close, 14)
float sma28 = ta.sma(close, 28)

if ta.crossover(sma14, sma28)
    // Cancel any unfilled sell orders with the specified ID.
    strategy.cancel("My Short Entry ID")
    //@variable The limit price level. Its value is `limitOffsetInput` ticks below the cur
    float limitLevel = close - limitOffsetInput
    // Place a long limit order to close the short trade and enter a long position at the
    strategy.entry("My Long Entry ID", strategy.long, limit = limitLevel)
    // Make new drawings for the long limit and stop extending the `shortLimit` line.
    longLimit := drawLimit(limitLevel, isLong = true)
    shortLimit.stopExtend()

if ta.crossunder(sma14, sma28)
    // Cancel any unfilled buy orders with the specified ID.
    strategy.cancel("My Long Entry ID")
    //@variable The limit price level. Its value is `limitOffsetInput` ticks above the cur
    float limitLevel = close + limitOffsetInput
    // Place a short limit order to close the long trade and enter a short position at the
    strategy.entry("My Short Entry ID", strategy.short, limit = limitLevel)
    // Make new drawings for the short limit and stop extending the `shortLimit` line.
    shortLimit := drawLimit(limitLevel, isLong = false)
    longLimit.stopExtend()
```

strategy.exit()

Creates price-based orders to exit from an open position. If unfilled exit orders with the same `id` exist, calls to this command modify those orders. This command can generate more than one type of exit order, depending on the specified parameters. However, it does not create market orders. To exit from a position with a market order, use strategy.close or strategy.close_all.

If a call to this command contains a `profit` or `limit` argument, it creates take-profit orders to exit from applicable trades at the determined price levels or better values (higher for long trades and lower for short ones). If the call contains `loss` or `stop` arguments, it creates stop-loss orders to exit from applicable trades at the determined levels or worse values (lower for long trades and higher for short ones). Calling this command with `profit` or `limit` and `loss` or `stop` arguments creates an order bracket with both order types.

This command can create trailing stop orders when its call specifies a `trail_price` or `trail_points` argument and a `trail_offset` argument. A trailing stop order activates when the price moves `trail_points` ticks past the entry price or touches the `trail_price` level. Once activated, the stop follows `trail_offset` ticks behind the market price each time the trade's profit reaches a new high. The stop does not move when the trade does not achieve a new best value.

Each call to this command reserves a portion of the position to close until the strategy fills or cancels its orders. For example, if there is an open position of 50 contracts and a strategy.exit call specifies a `qty` of 20, that call's orders reserve 20 contracts out of the position. A second call can close a maximum of 30 contracts, even if its `qty` is 50 and one of its orders executes first. This behavior does not affect the orders from other commands, such as strategy.close or strategy.order.

If a call to this command occurs before a created entry order's execution, the strategy waits and does not create the exit orders until after the entry order executes.

SYNTAX

```
strategy.exit(id, from_entry, qty, qty_percent, profit, limit, loss, stop, trail_price,
trail_points, trail_offset, oca_name, comment, comment_profit, comment_loss,
comment_trailing, alert_message, alert_profit, alert_loss, alert_trailing, disable_alert)
→ void
```

ARGUMENTS

id (series string) The identifier of the orders, which corresponds to an exit ID in the strategy's trades after an order fills. Strategy commands can reference the order ID to cancel or modify pending exit orders. The Strategy Tester and the chart display the order ID unless the command includes a `comment*` argument that applies to the filled order.

from_entry (series string) Optional. The entry order ID of the trade to exit from. If there is more than one open trade with the specified entry ID, the command generates exit orders for all the entries from before or at the time of the call. The default is an empty string, which means the command generates exit orders for all open trades until the position closes.

qty (series int/float) Optional. The number of contracts/lots/shares/units to close when an exit order fills. If specified, the command uses this value instead of `qty_percent` to determine the order size. The exit orders reserve this quantity from the position, meaning other calls to this command cannot close this portion until the strategy fills or cancels those orders. The default is na, which means the order size depends on the `qty_percent` value.

qty_percent (series int/float) Optional. A value between 0 and 100 representing the percentage of the open trade quantity to close when an exit order fills. The exit orders reserve this percentage from the applicable open trades, meaning other calls to this command cannot close this portion until the strategy fills or cancels those orders. The percentage calculation depends on the total size of the applicable open trades without considering the reserved amount from other strategy.exit calls. The command ignores this parameter if the `qty` value is not na. The default is 100.

profit (series int/float) Optional. The take-profit distance, expressed in ticks. If specified, the command creates a limit order to exit the trade `profit` ticks away from the entry price in the favorable direction. The order executes at the calculated price or a better value. The command ignores this parameter if the `limit` value is not na. The default is na.

limit (series int/float) Optional. The take-profit price. If specified, the command ignores the `profit` parameter and creates a limit order to exit the trade at this price or a better value. The default is na.

loss (series int/float) Optional. The stop-loss distance, expressed in ticks. If specified, the command creates a stop order to exit the trade `loss` ticks away from the entry price in the unfavorable direction. The order executes at the calculated price or a worse value. The command ignores this parameter if the `stop` value is not na. The default is na.

stop (series int/float) Optional. The stop-loss price. If specified, the command ignores the `loss` parameter and creates a stop order to exit the trade at this price or worse value. The default is na.

trail_price (series int/float) Optional. The price of the trailing stop activation level. If specified, the command ignores the `trail_points` parameter and activates a trailing stop order when the market price reaches this value. The trailing stop's initial value is `trail_offset` ticks away from the activation level in the unfavorable direction. The default is na.

trail_points (series int/float) Optional. The trailing stop activation distance, expressed in ticks. If this value is positive, the command creates a trailing stop order when the market price moves `trail_points` ticks away from the trade's entry price in the favorable direction. If this value is negative, the command creates a trailing stop order when the market price moves the absolute `trail_points` ticks away from the entry price in the opposite direction of the trade. The command ignores this parameter if the `trail_price` value is not na. The default is na.

trail_offset (series int/float) Optional. The trailing stop offset. When the market price reaches the activation level determined by the `trail_price` or `trail_points` parameter, the command creates a trailing stop with an initial value `trail_offset` ticks away from that level in the unfavorable direction. After activation, the trailing stop moves toward the market price each time the trade's profit reaches a new high, maintaining the specified distance behind the best price. The default is na.

oca_name (series string) Optional. The name of the One-Cancels-All (OCA) group that the command's take-profit, stop-loss, and trailing stop orders belong to. All orders from this command are of the strategy.oca.reduce OCA type. When an order of this OCA type with the same `oca_name` executes, the strategy reduces the sizes of other unfilled orders in the OCA group by the filled quantity. The default is an empty string, which means the strategy assigns the OCA name automatically, and the resulting orders cannot reduce or be reduced by the orders from other commands.

comment (series string) Optional. Additional notes on the filled order. If the value is not an empty string, the Strategy Tester and the chart show this text for the order instead of the specified `id`. The command ignores this value if the call includes an argument for a `comment_*` parameter that applies to the order. The default is an empty string.

comment_profit (series string) Optional. Additional notes on the filled order. If the value is not an empty string, the Strategy Tester and the chart show this text for the order instead of the specified `id` or `comment`. This comment applies only to the command's take-profit orders created using the `profit` or `limit` parameter. The default is an empty string.

comment_loss (series string) Optional. Additional notes on the filled order. If the value is not an empty string, the Strategy Tester and the chart show this text for the order instead of the specified `id` or `comment`. This comment applies only to the command's stop-loss orders created using the `loss` or `stop` parameter. The default is an empty string.

comment_trailing (series string) Optional. Additional notes on the filled order. If the value is not an empty string, the Strategy Tester and the chart show this text for the order instead of the specified `id` or `comment`. This comment applies only to the command's trailing stop orders created using the `trail_price` or `trail_points` and `trail_offset` parameters. The default is an empty string.

alert_message (series string) Optional. Custom text for the alert that fires when an order fills. If the "Message" field of the "Create Alert" dialog box contains the `{{strategy.order.alert_message}}` placeholder, the alert message replaces the placeholder with this text. The command ignores this value if the call includes an argument for the other `alert_*` parameter that applies to the order. The default is an empty string.

alert_profit (series string) Optional. Custom text for the alert that fires when an order fills. If the "Message" field of the "Create Alert" dialog box contains the `{{strategy.order.alert_message}}` placeholder, the alert message replaces the placeholder with this text. This message applies only to the command's take-profit orders created using the `profit` or `limit` parameter. The default is an empty string.

alert_loss (series string) Optional. Custom text for the alert that fires when an order fills. If the "Message" field of the "Create Alert" dialog box contains the `{{strategy.order.alert_message}}` placeholder, the alert message replaces the placeholder with this text. This message applies only to the command's stop-loss orders created using the `loss` or `stop` parameter. The default is an empty string.

alert_trailing (series string) Optional. Custom text for the alert that fires when an order fills. If the "Message" field of the "Create Alert" dialog box contains the `{{strategy.order.alert_message}}` placeholder, the alert message replaces the placeholder with this text. This message applies only to the command's trailing stop orders created using the `trail_price` or `trail_points` and `trail_offset` parameters. The default is an empty string.

disable_alert (series bool) Optional. If true when the command creates an order, the strategy does not trigger an alert when that order fills. This parameter accepts a "series" value, meaning users can control which orders trigger alerts when they execute. The default is false.

EXAMPLE

```
//@version=5
strategy("Exit bracket strategy", overlay = true, margin_long = 100, margin_short = 100)

// Inputs that define the profit and loss amount of each trade as a tick distance from the
int profitDistanceInput = input.int(100, "Profit distance, in ticks", 1)
int lossDistanceInput   = input.int(100, "Loss distance, in ticks", 1)

// Variables to track the take-profit and stop-loss price.
var float takeProfit = na
var float stopLoss   = na

// Calculate a 14-bar and 28-bar moving average of `close` prices.
float sma14 = ta.sma(close, 14)
float sma28 = ta.sma(close, 28)
```

```
if ta.crossover(sma14, sma28) and strategy.opentrades == 0
    // Place a market order to enter a long position.
    strategy.entry("My Long Entry ID", strategy.long)
    // Place a take-profit and stop-loss order when the entry order fills.
    strategy.exit("My Long Exit ID", "My Long Entry ID", profit = profitDistanceInput, los

if ta.change(strategy.opentrades) == 1
    //@variable The long entry price.
    float entryPrice = strategy.opentrades.entry_price(0)
    // Update the `takeProfit` and `stopLoss` values.
    takeProfit := entryPrice + profitDistanceInput * syminfo.mintick
    stopLoss   := entryPrice - lossDistanceInput * syminfo.mintick

if ta.change(strategy.closedtrades) == 1
    // Reset the `takeProfit` and `stopLoss`.
    takeProfit := na
    stopLoss   := na

// Plot the `takeProfit` and `stopLoss`.
plot(takeProfit, "Take-profit level", color.green, 2, plot.style_linebr)
plot(stopLoss, "Stop-loss level", color.red, 2, plot.style_linebr)
```

EXAMPLE

```
//@version=5
strategy("Trailing stop strategy", overlay = true, margin_long = 100, margin_short = 100)

//@variable The distance required to activate the trailing stop.
float activationDistanceInput = input.int(100, "Trail activation distance, in ticks") * sy
//@variable The number of ticks the trailing stop follows behind the price as it reaches r
int trailDistanceInput = input.int(100, "Trail distance, in ticks")

//@function Draws a label and line at the specified `price` to visualize a trailing stop c
drawActivation(float price) =>
    label.new(
         bar_index, price, "Activation level", style = label.style_label_right,
         color = color.gray, textcolor = color.white
     )
    line.new(
         bar_index, price, bar_index + 1, price, extend = extend.right, style = line.style
     )

//@function Stops the `l` line from extending further.
method stopExtend(line l) =>
    l.set_x2(bar_index)
    l.set_extend(extend.none)

// The activation line, active trailing stop price, and active trailing stop flag.
var line activationLine     = na
var float trailingStopPrice = na
var bool isActive           = false
```

```
if bar_index % 100 == 0 and strategy.opentrades == 0
    trailingStopPrice := na
    isActive          := false
    // Place a market order to enter a long position.
    strategy.entry("My Long Entry ID", strategy.long)
    //@variable The activation level's price.
    float activationPrice = close + activationDistanceInput
    // Create a trailing stop order that activates the defined number of ticks above the e
    strategy.exit(
         "My Long Exit ID", "My Long Entry ID", trail_price = activationPrice, trail_offse
         oca_name = "Exit"
     )
    // Create new drawings at the `activationPrice`.
    activationLine := drawActivation(activationPrice)

// Logic for trailing stop visualization.
if strategy.opentrades == 1
    // Stop extending the `activationLine` when the stop activates.
    if not isActive and high > activationLine.get_price(bar_index)
        isActive := true
        activationLine.stopExtend()
    // Update the `trailingStopPrice` while the trailing stop is active.
    if isActive
        float offsetPrice = high - trailDistanceInput * syminfo.mintick
        trailingStopPrice := math.max(nz(trailingStopPrice, offsetPrice), offsetPrice)

// Close the trade with a market order if the trailing stop does not activate before the r
if not isActive and bar_index % 300 == 0
    strategy.close_all("Market close")

// Reset the `trailingStopPrice` and `isActive` flags when the trade closes, and stop exte
if ta.change(strategy.closedtrades) > 0
    if not isActive
        activationLine.stopExtend()
    trailingStopPrice := na
    isActive          := false

// Plot the `trailingStopPrice`.
plot(trailingStopPrice, "Trailing stop", color.red, 3, plot.style_linebr)
```

REMARKS

A single call to the strategy.exit command can generate exit orders for several entries in an open position, depending on the call's `from_entry` value. If the call does not include a `from_entry` argument, it creates exit orders for all open trades, even the ones opened after the call, until the position closes. See this section of our User Manual to learn more.

When a position consists of several open trades, and the `close_entries_rule` in the strategy declaration statement is "FIFO" (default), the orders from a strategy.exit call exit from the position starting with the first open trade. This behavior applies even if the `from_entry` value is the entry ID of different open trades. However, in that case, the maximum size of the exit orders still depends on the trades opened by orders with the `from_entry` ID. For more information, see this section of our User Manual.

If a strategy.exit call includes arguments for creating stop-loss and trailing stop orders, the command places only the order that is supposed to fill first, because both orders are of the "stop" type.

strategy.opentrades.commission()

Returns the sum of entry and exit fees paid in the open trade, expressed in strategy.account_currency.

SYNTAX

```
strategy.opentrades.commission(trade_num) → series float
```

ARGUMENTS

trade_num (series int) The trade number of the open trade. The number of the first trade is zero.

EXAMPLE

```
// Calculates the gross profit or loss for the current open position.
//@version=5
strategy("`strategy.opentrades.commission` Example", commission_type = strategy.commissior

// Strategy calls to enter long trades every 15 bars and exit long trades every 20 bars.
if bar_index % 15 == 0
    strategy.entry("Long", strategy.long)
if bar_index % 20 == 0
    strategy.close("Long")

// Calculate gross profit or loss for open positions only.
tradeOpenGrossPL() =>
    sumOpenGrossPL = 0.0
    for tradeNo = 0 to strategy.opentrades - 1
        sumOpenGrossPL += strategy.opentrades.profit(tradeNo) - strategy.opentrades.commis
    result = sumOpenGrossPL

plot(tradeOpenGrossPL())
```

SEE ALSO

strategy strategy.closedtrades.commission

strategy.opentrades.entry_bar_index()

Returns the bar_index of the open trade's entry.

SYNTAX

```
strategy.opentrades.entry_bar_index(trade_num) → series int
```

ARGUMENTS

trade_num (series int) The trade number of the open trade. The number of the first trade is zero.

EXAMPLE

```
// Wait 10 bars and then close the position.
//@version=5
strategy("`strategy.opentrades.entry_bar_index` Example")

barsSinceLastEntry() =>
    strategy.opentrades > 0 ? bar_index - strategy.opentrades.entry_bar_index(strategy.ope

// Enter a long position if there are no open positions.
if strategy.opentrades == 0
    strategy.entry("Long",  strategy.long)

// Close the long position after 10 bars.
if barsSinceLastEntry() >= 10
    strategy.close("Long")
```

SEE ALSO

strategy.closedtrades.entry_bar_index strategy.closedtrades.exit_bar_index

strategy.opentrades.entry_comment()

Returns the comment message of the open trade's entry, or na if there is no entry with this `trade_num`.

SYNTAX

```
strategy.opentrades.entry_comment(trade_num) → series string
```

ARGUMENTS

trade_num (series int) The trade number of the open trade. The number of the first trade is zero.

EXAMPLE

```
//@version=5
strategy("`strategy.opentrades.entry_comment()` Example", overlay = true)

stopPrice = open * 1.01

longCondition = ta.crossover(ta.sma(close, 14), ta.sma(close, 28))

if (longCondition)
    strategy.entry("Long", strategy.long, stop = stopPrice, comment = str.tostring(stopPri

var testTable = table.new(position.top_right, 1, 3, color.orange, border_width = 1)

if barstate.islastconfirmedhistory or barstate.isrealtime
    table.cell(testTable, 0, 0, 'Last entry stats')
    table.cell(testTable, 0, 1, "Order stop price value: " + strategy.opentrades.entry_com
    table.cell(testTable, 0, 2, "Actual Entry Price: " + str.tostring(strategy.opentrades.
```

SEE ALSO

strategy   strategy.entry   strategy.opentrades

strategy.opentrades.entry_id()

Returns the id of the open trade's entry.

SYNTAX

```
strategy.opentrades.entry_id(trade_num) → series string
```

ARGUMENTS

trade_num (series int) The trade number of the open trade. The number of the first trade is zero.

EXAMPLE

```
//@version=5
strategy("`strategy.opentrades.entry_id` Example", overlay = true)

// We enter a long position when 14 period sma crosses over 28 period sma.
// We enter a short position when 14 period sma crosses under 28 period sma.
```

```
longCondition = ta.crossover(ta.sma(close, 14), ta.sma(close, 28))
shortCondition = ta.crossunder(ta.sma(close, 14), ta.sma(close, 28))

// Strategy calls to enter a long or short position when the corresponding condition is me
if longCondition
    strategy.entry("Long entry at bar #" + str.tostring(bar_index), strategy.long)
if shortCondition
    strategy.entry("Short entry at bar #" + str.tostring(bar_index), strategy.short)

// Display ID of the latest open position.
if barstate.islastconfirmedhistory
    label.new(bar_index, high + (2 * ta.tr),  "Last opened position is \n " + strategy.ope
```

RETURNS

Returns the id of the open trade's entry.

REMARKS

The function returns na if trade_num is not in the range: 0 to strategy.opentrades-1.

SEE ALSO

strategy.opentrades.entry_bar_index   strategy.opentrades.entry_price

strategy.opentrades.entry_time

strategy.opentrades.entry_price()

Returns the price of the open trade's entry.

SYNTAX

```
strategy.opentrades.entry_price(trade_num) → series float
```

ARGUMENTS

trade_num (series int) The trade number of the open trade. The number of the first trade is zero.

EXAMPLE

```
//@version=5
strategy("strategy.opentrades.entry_price Example 1", overlay = true)

// Strategy calls to enter long trades every 15 bars and exit long trades every 20 bars.
if ta.crossover(close, ta.sma(close, 14))
    strategy.entry("Long", strategy.long)
```

```
// Return the entry price for the latest closed trade.
currEntryPrice = strategy.opentrades.entry_price(strategy.opentrades - 1)
currExitPrice = currEntryPrice * 1.05

if high >= currExitPrice
    strategy.close("Long")

plot(currEntryPrice, "Long entry price", style = plot.style_linebr)
plot(currExitPrice, "Long exit price", color.green, style = plot.style_linebr)
```

EXAMPLE

```
// Calculates the average price for the open position.
//@version=5
strategy("strategy.opentrades.entry_price Example 2", pyramiding = 2)

// Strategy calls to enter long trades every 15 bars and exit long trades every 20 bars.
if bar_index % 15 == 0
    strategy.entry("Long", strategy.long)
if bar_index % 20 == 0
    strategy.close("Long")

// Calculates the average price for the open position.
avgOpenPositionPrice() =>
    sumOpenPositionPrice = 0.0
    for tradeNo = 0 to strategy.opentrades - 1
        sumOpenPositionPrice += strategy.opentrades.entry_price(tradeNo) * strategy.opentr
    result = nz(sumOpenPositionPrice / strategy.opentrades)

plot(avgOpenPositionPrice())
```

SEE ALSO

strategy.closedtrades.exit_price

strategy.opentrades.entry_time()

Returns the UNIX time of the open trade's entry, expressed in milliseconds.

SYNTAX

```
strategy.opentrades.entry_time(trade_num) → series int
```

ARGUMENTS

trade_num (series int) The trade number of the open trade. The number of the first trade is zero.

EXAMPLE

```
//@version=5
strategy("strategy.opentrades.entry_time Example")

// Strategy calls to enter long trades every 15 bars and exit long trades every 20 bars.
if bar_index % 15 == 0
    strategy.entry("Long", strategy.long)
if bar_index % 20 == 0
    strategy.close("Long")

// Calculates duration in milliseconds since the last position was opened.
timeSinceLastEntry()=>
    strategy.opentrades > 0 ? (time - strategy.opentrades.entry_time(strategy.opentrades -

plot(timeSinceLastEntry() / 1000 * 60 * 60 * 24, "Days since last entry")
```

SEE ALSO

strategy.closedtrades.entry_time strategy.closedtrades.exit_time

strategy.opentrades.max_drawdown()

Returns the maximum drawdown of the open trade, i.e., the maximum possible loss during the trade, expressed in strategy.account_currency.

SYNTAX

```
strategy.opentrades.max_drawdown(trade_num) → series float
```

ARGUMENTS

trade_num (series int) The trade number of the open trade. The number of the first trade is zero.

EXAMPLE

```
//@version=5
strategy("strategy.opentrades.max_drawdown Example 1")

// Strategy calls to enter long trades every 15 bars and exit long trades every 20 bars.
if bar_index % 15 == 0
```

```
    strategy.entry("Long", strategy.long)
if bar_index % 20 == 0
    strategy.close("Long")

// Plot the max drawdown of the latest open trade.
plot(strategy.opentrades.max_drawdown(strategy.opentrades - 1), "Max drawdown of the lates
```

EXAMPLE

```
// Calculates the max trade drawdown value for all open trades.
//@version=5
strategy("`strategy.opentrades.max_drawdown` Example 2", pyramiding = 100)

// Strategy calls to enter long trades every 15 bars and exit long trades every 20 bars.
if bar_index % 15 == 0
    strategy.entry("Long", strategy.long)
if bar_index % 20 == 0
    strategy.close("Long")

// Get the biggest max trade drawdown value from all of the open trades.
maxTradeDrawDown() =>
    maxDrawdown = 0.0
    for tradeNo = 0 to strategy.opentrades - 1
        maxDrawdown := math.max(maxDrawdown, strategy.opentrades.max_drawdown(tradeNo))
    result = maxDrawdown

plot(maxTradeDrawDown(), "Biggest max drawdown")
```

REMARKS

The function returns na if trade_num is not in the range: 0 to strategy.closedtrades - 1.

SEE ALSO

strategy.closedtrades.max_drawdown strategy.max_drawdown

strategy.opentrades.max_drawdown_percent()

Returns the maximum drawdown of the open trade, i.e., the maximum possible loss during the trade, expressed as a percentage and calculated by formula: `Lowest Value During Trade / (Entry Price x Quantity) * 100`.

SYNTAX

```
strategy.opentrades.max_drawdown_percent(trade_num) → series float
```

ARGUMENTS

trade_num (series int) The trade number of the closed trade. The number of the first trade is zero.

SEE ALSO

strategy.opentrades.max_drawdown   strategy.max_drawdown

strategy.opentrades.max_runup()

Returns the maximum run up of the open trade, i.e., the maximum possible profit during the trade, expressed in strategy.account_currency.

SYNTAX

```
strategy.opentrades.max_runup(trade_num) → series float
```

ARGUMENTS

trade_num (series int) The trade number of the open trade. The number of the first trade is zero.

EXAMPLE

```
//@version=5
strategy("strategy.opentrades.max_runup Example 1")

// Strategy calls to enter long trades every 15 bars and exit long trades every 20 bars.
if bar_index % 15 == 0
    strategy.entry("Long", strategy.long)
if bar_index % 20 == 0
    strategy.close("Long")

// Plot the max runup of the latest open trade.
plot(strategy.opentrades.max_runup(strategy.opentrades - 1), "Max runup of the latest oper
```

EXAMPLE

```
// Calculates the max trade runup value for all open trades.
//@version=5
strategy("strategy.opentrades.max_runup Example 2", pyramiding = 100)

// Enter a long position every 30 bars.
if bar_index % 30 == 0
```

```
    strategy.entry("Long", strategy.long)

// Calculate biggest max trade runup value from all of the open trades.
maxOpenTradeRunUp() =>
    maxRunup = 0.0
    for tradeNo = 0 to strategy.opentrades - 1
        maxRunup := math.max(maxRunup, strategy.opentrades.max_runup(tradeNo))
    result = maxRunup

plot(maxOpenTradeRunUp(), "Biggest max runup of all open trades")
```

SEE ALSO

strategy.closedtrades.max_runup   strategy.max_drawdown

strategy.opentrades.max_runup_percent()

Returns the maximum run-up of the open trade, i.e., the maximum possible profit during the trade, expressed as a percentage and calculated by formula: `Highest Value During Trade / (Entry Price x Quantity) * 100`.

SYNTAX

```
strategy.opentrades.max_runup_percent(trade_num) → series float
```

ARGUMENTS

trade_num (series int) The trade number of the closed trade. The number of the first trade is zero.

SEE ALSO

strategy.opentrades.max_runup   strategy.max_runup

strategy.opentrades.profit()

Returns the profit/loss of the open trade, expressed in strategy.account_currency. Losses are expressed as negative values.

SYNTAX

```
strategy.opentrades.profit(trade_num) → series float
```

ARGUMENTS

trade_num (series int) The trade number of the open trade. The number of the first trade is zero.

EXAMPLE

```
// Returns the profit of the last open trade.
//@version=5
strategy("`strategy.opentrades.profit` Example 1", commission_type = strategy.commission.p

// Strategy calls to enter long trades every 15 bars and exit long trades every 20 bars.
if bar_index % 15 == 0
    strategy.entry("Long", strategy.long)
if bar_index % 20 == 0
    strategy.close("Long")

plot(strategy.opentrades.profit(strategy.opentrades - 1), "Profit of the latest open trade
```

EXAMPLE

```
// Calculates the profit for all open trades.
//@version=5
strategy("`strategy.opentrades.profit` Example 2", pyramiding = 5)

// Strategy calls to enter 5 long positions every 2 bars.
if bar_index % 2 == 0
    strategy.entry("Long", strategy.long, qty = 5)

// Calculate open profit or loss for the open positions.
tradeOpenPL() =>
    sumProfit = 0.0
    for tradeNo = 0 to strategy.opentrades - 1
        sumProfit += strategy.opentrades.profit(tradeNo)
    result = sumProfit

plot(tradeOpenPL(), "Profit of all open trades")
```

SEE ALSO

strategy.closedtrades.profit   strategy.openprofit   strategy.netprofit   strategy.grossprofit

strategy.opentrades.profit_percent()

Returns the profit/loss of the open trade, expressed as a percentage. Losses are expressed as negative values.

SYNTAX

```
strategy.opentrades.profit_percent(trade_num) → series float
```

ARGUMENTS

trade_num (series int) The trade number of the closed trade. The number of the first trade is zero.

SEE ALSO

strategy.opentrades.profit

strategy.opentrades.size()

Returns the direction and the number of contracts traded in the open trade. If the value is > 0, the market position was long. If the value is < 0, the market position was short.

SYNTAX

```
strategy.opentrades.size(trade_num) → series float
```

ARGUMENTS

trade_num (series int) The trade number of the open trade. The number of the first trade is zero.

EXAMPLE

```
//@version=5
strategy("`strategy.opentrades.size` Example 1")

// We calculate the max amt of shares we can buy.
amtShares = math.floor(strategy.equity / close)
// Strategy calls to enter long trades every 15 bars and exit long trades every 20 bars
if bar_index % 15 == 0
    strategy.entry("Long", strategy.long, qty = amtShares)
if bar_index % 20 == 0
    strategy.close("Long")

// Plot the number of contracts in the latest open trade.
plot(strategy.opentrades.size(strategy.opentrades - 1), "Amount of contracts in latest ope
```

EXAMPLE

```
// Calculates the average profit percentage for all open trades.
//@version=5
strategy("`strategy.opentrades.size` Example 2")

// Strategy calls to enter long trades every 15 bars and exit long trades every 20 bars.
if bar_index % 15 == 0
    strategy.entry("Long", strategy.long)
if bar_index % 20 == 0
    strategy.close("Long")

// Calculate profit for all open trades.
profitPct = 0.0
for tradeNo = 0 to strategy.opentrades - 1
    entryP = strategy.opentrades.entry_price(tradeNo)
    exitP = close
    profitPct += (exitP - entryP) / entryP * strategy.opentrades.size(tradeNo) * 100

// Calculate average profit percent for all open trades.
avgProfitPct = nz(profitPct / strategy.opentrades)
plot(avgProfitPct)
```

SEE ALSO

strategy.closedtrades.size   strategy.position_size   strategy.opentrades   strategy.closedtrades

strategy.order()

Creates a new order to open, add to, or exit from a position. If an unfilled order with the same `id` exists, a call to this command modifies that order.

The resulting order's type depends on the `limit` and `stop` parameters. If the call does not contain `limit` or `stop` arguments, it creates a market order that executes on the next tick. If the call specifies a `limit` value but no `stop` value, it places a limit order that executes after the market price reaches the `limit` value or a better price (lower for buy orders and higher for sell orders). If the call specifies a `stop` value but no `limit` value, it places a stop order that executes after the market price reaches the `stop` value or a worse price (higher for buy orders and lower for sell orders). If the call contains `limit` and `stop` arguments, it creates a stop-limit order, which generates a limit order at the `limit` price only after the market price reaches the `stop` value or a worse price.

Orders from this command, unlike those from strategy.entry, are not affected by the `pyramiding` parameter of the strategy declaration statement. Strategies can open any number of trades in the same direction with calls to this function.

This command does not automatically reverse open positions because it does not exclusively create entry orders like strategy.entry does. For example, if there is an open long position of five shares, an order from this command with a `qty` of 5 and a `direction` of strategy.short triggers the sale of five shares, which closes the position.

SYNTAX

```
strategy.order(id, direction, qty, limit, stop, oca_name, oca_type, comment,
alert_message, disable_alert) → void
```

ARGUMENTS

id (series string) The identifier of the order, which corresponds to an entry or exit ID in the strategy's trades after the order fills. If the strategy opens a new position after filling the order, the order's ID becomes the strategy.position_entry_name value. Strategy commands can reference the order ID to cancel or modify pending orders and generate exit orders for specific open trades. The Strategy Tester and the chart display the order ID unless the command specifies a `comment` value.

direction (series strategy_direction) The direction of the trade. Possible values: strategy.long for a long trade, strategy.short for a short one.

qty (series int/float) Optional. The number of contracts/shares/lots/units to trade when the order fills. The default is na, which means that the command uses the `default_qty_type` and `default_qty_value` parameters of the strategy declaration statement to determine the quantity.

limit (series int/float) Optional. The limit price of the order. If specified, the command creates a limit or stop-limit order, depending on whether the `stop` value is also specified. The default is na, which means the resulting order is not of the limit or stop-limit type.

stop (series int/float) Optional. The stop price of the order. If specified, the command creates a stop or stop-limit order, depending on whether the `limit` value is also specified. The default is na, which means the resulting order is not of the stop or stop-limit type.

oca_name (series string) Optional. The name of the order's One-Cancels-All (OCA) group. When a pending order with the same `oca_name` and `oca_type` parameters executes, that order affects all unfilled orders in the group. The default is an empty string, which means the order does not belong to an OCA group.

oca_type (input string) Optional. Specifies how an unfilled order behaves when another pending order with the same `oca_name` and `oca_type` values executes. Possible values: strategy.oca.cancel, strategy.oca.reduce, strategy.oca.none. The default is strategy.oca.none.

comment (series string) Optional. Additional notes on the filled order. If the value is not an empty string, the Strategy Tester and the chart show this text for the order instead of the specified `id`. The default is an empty string.

alert_message (series string) Optional. Custom text for the alert that fires when an order fills. If the "Message" field of the "Create Alert" dialog box contains the `{{strategy.order.alert_message}}` placeholder, the alert message replaces the placeholder with this text. The default is an empty string.

disable_alert (series bool) Optional. If true when the command creates an order, the strategy does not trigger an alert when that order fills. This parameter accepts a "series" value, meaning users can control which orders trigger alerts when they execute. The default is false.

EXAMPLE

```
//@version=5
strategy("Market order strategy", overlay = true, margin_long = 100, margin_short = 100)

// Calculate a 14-bar and 28-bar moving average of `close` prices.
float sma14 = ta.sma(close, 14)
float sma28 = ta.sma(close, 28)

// Place a market order to enter a long position when `sma14` crosses over `sma28`.
if ta.crossover(sma14, sma28) and strategy.position_size == 0
    strategy.order("My Long Entry ID", strategy.long)

// Place a market order to sell the same quantity as the long trade when `sma14` crosses u
// effectively closing the long position.
if ta.crossunder(sma14, sma28) and strategy.position_size > 0
    strategy.order("My Long Exit ID", strategy.short)
```

EXAMPLE

```
//@version=5
strategy("Limit and stop exit strategy", overlay = true, margin_long = 100, margin_short =

//@variable The distance from the long entry price for each short limit order.
float shortOffsetInput = input.int(200, "Sell limit/stop offset, in ticks", 1) * syminfo.m

//@function Draws a label and line at the specified `price` to visualize a limit order's l
drawLimit(float price, bool isLong, bool isStop = false) =>
    color col = isLong ? color.blue : color.red
    label.new(
         bar_index, price, (isLong ? "Long " : "Short ") + (isStop ? "stop" : "limit") + '
         style = label.style_label_right, color = col, textcolor = color.white
     )
    line.new(bar_index, price, bar_index + 1, price, extend = extend.right, style = line.s

//@function Stops the `l` line from extending further.
method stopExtend(line l) =>
    l.set_x2(bar_index)
    l.set_extend(extend.none)

// Initialize two `line` variables to reference limit and stop line IDs.
```

```
    var line profitLimit = na
    var line lossStop    = na

    // Calculate a 14-bar and 28-bar moving average of `close` prices.
    float sma14 = ta.sma(close, 14)
    float sma28 = ta.sma(close, 28)

    if ta.crossover(sma14, sma28) and strategy.position_size == 0
        // Place a market order to enter a long position.
        strategy.order("My Long Entry ID", strategy.long)

    if strategy.position_size > 0 and strategy.position_size[1] == 0
        //@variable The entry price of the long trade.
        float entryPrice = strategy.opentrades.entry_price(0)
        // Calculate short limit and stop levels above and below the `entryPrice`.
        float profitLevel = entryPrice + shortOffsetInput
        float lossLevel   = entryPrice - shortOffsetInput
        // Place short limit and stop orders at the `profitLevel` and `lossLevel`.
        strategy.order("Profit", strategy.short, limit = profitLevel, oca_name = "Bracket", oc
        strategy.order("Loss", strategy.short, stop = lossLevel, oca_name = "Bracket", oca_typ
        // Make new drawings for the `profitLimit` and `lossStop` lines.
        profitLimit := drawLimit(profitLevel, isLong = false)
        lossStop    := drawLimit(lossLevel, isLong = false, isStop = true)

    if ta.change(strategy.closedtrades) > 0
        // Stop extending the `profitLimit` and `lossStop` lines.
        profitLimit.stopExtend()
        lossStop.stopExtend()
```

strategy.risk.allow_entry_in()

This function can be used to specify in which market direction the strategy.entry function is allowed to open positions.

SYNTAX

```
strategy.risk.allow_entry_in(value) → void
```

ARGUMENTS

value (simple string) The allowed direction. Possible values: strategy.direction.all, strategy.direction.long, strategy.direction.short

EXAMPLE

```
    //@version=5
    strategy("strategy.risk.allow_entry_in")
```

```
strategy.risk.allow_entry_in(strategy.direction.long)
if open > close
    strategy.entry("Long", strategy.long)
// Instead of opening a short position with 10 contracts, this command will close long ent
if open < close
    strategy.entry("Short", strategy.short, qty = 10)
```

strategy.risk.max_cons_loss_days()

The purpose of this rule is to cancel all pending orders, close all open positions and stop placing orders after a specified number of consecutive days with losses. The rule affects the whole strategy.

SYNTAX

```
strategy.risk.max_cons_loss_days(count, alert_message) → void
```

ARGUMENTS

count (simple int) A required parameter. The allowed number of consecutive days with losses.

alert_message (simple string) An optional parameter which replaces the {{strategy.order.alert_message}} placeholder when it is used in the "Create Alert" dialog box's "Message" field.

EXAMPLE

```
//@version=5
strategy("risk.max_cons_loss_days Demo 1")
strategy.risk.max_cons_loss_days(3) // No orders will be placed after 3 days, if each day
plot(strategy.position_size)
```

strategy.risk.max_drawdown()

The purpose of this rule is to determine maximum drawdown. The rule affects the whole strategy. Once the maximum drawdown value is reached, all pending orders are cancelled, all open positions are closed and no new orders can be placed.

SYNTAX

```
strategy.risk.max_drawdown(value, type, alert_message) → void
```

ARGUMENTS

value (simple int/float) A required parameter. The maximum drawdown value. It is specified either in money (base currency), or in percentage of maximum equity. For % of equity the range of allowed values is from 0 to 100.

type (simple string) A required parameter. The type of the value. Please specify one of the following values: strategy.percent_of_equity or strategy.cash. Note: if equity drops down to zero or to a negative and the 'strategy.percent_of_equity' is specified, all pending orders are cancelled, all open positions are closed and no new orders can be placed for good.

alert_message (simple string) An optional parameter which replaces the {{strategy.order.alert_message}} placeholder when it is used in the "Create Alert" dialog box's "Message" field.

EXAMPLE

```
//@version=5
strategy("risk.max_drawdown Demo 1")
strategy.risk.max_drawdown(50, strategy.percent_of_equity) // set maximum drawdown to 50%
plot(strategy.position_size)
```

EXAMPLE

```
//@version=5
strategy("risk.max_drawdown Demo 2", currency = "EUR")
strategy.risk.max_drawdown(2000, strategy.cash)  // set maximum drawdown to 2000 EUR from
plot(strategy.position_size)
```

strategy.risk.max_intraday_filled_orders()

The purpose of this rule is to determine maximum number of filled orders per 1 day (per 1 bar, if chart resolution is higher than 1 day). The rule affects the whole strategy. Once the maximum number of filled orders is reached, all pending orders are cancelled, all open positions are closed and no new orders can be placed till the end of the current trading session.

SYNTAX

```
strategy.risk.max_intraday_filled_orders(count, alert_message) → void
```

ARGUMENTS

count (simple int) A required parameter. The maximum number of filled orders per 1 day.

alert_message (simple string) An optional parameter which replaces the {{strategy.order.alert_message}} placeholder when it is used in the "Create Alert" dialog box's "Message" field.

EXAMPLE

```
//@version=5
strategy("risk.max_intraday_filled_orders Demo")
strategy.risk.max_intraday_filled_orders(10) // After 10 orders are filled, no more strate
if open > close
    strategy.entry("buy", strategy.long)
if open < close
    strategy.entry("sell", strategy.short)
```

strategy.risk.max_intraday_loss()

The maximum loss value allowed during a day. It is specified either in money (base currency), or in percentage of maximum intraday equity (0 -100).

SYNTAX

```
strategy.risk.max_intraday_loss(value, type, alert_message) → void
```

ARGUMENTS

value (simple int/float) A required parameter. The maximum loss value. It is specified either in money (base currency), or in percentage of maximum intraday equity. For % of equity the range of allowed values is from 0 to 100.

type (simple string) A required parameter. The type of the value. Please specify one of the following values: strategy.percent_of_equity or strategy.cash. Note: if equity drops down to zero or to a negative and the strategy.percent_of_equity is specified, all pending orders are cancelled, all open positions are closed and no new orders can be placed for good.

alert_message (simple string) An optional parameter which replaces the {{strategy.order.alert_message}} placeholder when it is used in the "Create Alert" dialog box's "Message" field.

EXAMPLE

```
// Sets the maximum intraday loss using the strategy's equity value.
//@version=5
strategy("strategy.risk.max_intraday_loss Example 1", overlay = false, default_qty_type =

// Input for maximum intraday loss %.
lossPct = input.float(10)

// Set maximum intraday loss to our lossPct input
strategy.risk.max_intraday_loss(lossPct, strategy.percent_of_equity)

// Enter Short at bar_index zero.
if bar_index == 0
    strategy.entry("Short", strategy.short)

// Store equity value from the beginning of the day
eqFromDayStart = ta.valuewhen(ta.change(dayofweek) > 0, strategy.equity, 0)

// Calculate change of the current equity from the beginning of the current day.
eqChgPct = 100 * ((strategy.equity - eqFromDayStart) / strategy.equity)

// Plot it
plot(eqChgPct)
hline(-lossPct)
```

EXAMPLE

```
// Sets the maximum intraday loss using the strategy's cash value.
//@version=5
strategy("strategy.risk.max_intraday_loss Example 2", overlay = false)

// Input for maximum intraday loss in absolute cash value of the symbol.
absCashLoss = input.float(5)

// Set maximum intraday loss to `absCashLoss` in account currency.
strategy.risk.max_intraday_loss(absCashLoss, strategy.cash)

// Enter Short at bar_index zero.
if bar_index == 0
    strategy.entry("Short", strategy.short)

// Store the open price value from the beginning of the day.
beginPrice = ta.valuewhen(ta.change(dayofweek) > 0, open, 0)

// Calculate the absolute price change for the current period.
priceChg = (close - beginPrice)
```

```
hline(absCashLoss)
plot(priceChg)
```

SEE ALSO

strategy strategy.percent_of_equity strategy.cash

strategy.risk.max_position_size()

The purpose of this rule is to determine maximum size of a market position. The rule affects the following function: strategy.entry. The 'entry' quantity can be reduced (if needed) to such number of contracts/shares/lots/units, so the total position size doesn't exceed the value specified in 'strategy.risk.max_position_size'. If minimum possible quantity still violates the rule, the order will not be placed.

SYNTAX

```
strategy.risk.max_position_size(contracts) → void
```

ARGUMENTS

contracts (simple int/float) A required parameter. Maximum number of contracts/shares/lots/units in a position.

EXAMPLE

```
//@version=5
strategy("risk.max_position_size Demo", default_qty_value = 100)
strategy.risk.max_position_size(10)
if open > close
    strategy.entry("buy", strategy.long)
plot(strategy.position_size)  // max plot value will be 10
```

string() 4 overloads

Casts na to string

SYNTAX & OVERLOADS

```
string(x) → const string
string(x) → input string
```

```
string(x) → simple string
```

```
string(x) → series string
```

ARGUMENTS

x (const string) The value to convert to the specified type, usually na.

RETURNS

The value of the argument after casting to string.

SEE ALSO

float int bool color line label

syminfo.prefix() 2 overloads

Returns exchange prefix of the `symbol`, e.g. "NASDAQ".

SYNTAX & OVERLOADS

```
syminfo.prefix(symbol) → simple string
```

```
syminfo.prefix(symbol) → series string
```

ARGUMENTS

symbol (simple string) Symbol. Note that the symbol should be passed with a prefix. For example: "NASDAQ:AAPL" instead of "AAPL".

EXAMPLE

```
//@version=5
indicator("syminfo.prefix fun", overlay=true)
i_sym = input.symbol("NASDAQ:AAPL")
pref = syminfo.prefix(i_sym)
tick = syminfo.ticker(i_sym)
t = ticker.new(pref, tick, session.extended)
s = request.security(t, "1D", close)
plot(s)
```

RETURNS

Returns exchange prefix of the `symbol`, e.g. "NASDAQ".

REMARKS

The result of the function is used in the ticker.new/ticker.modify and request.security.

SEE ALSO

syminfo.tickerid syminfo.ticker syminfo.prefix syminfo.ticker ticker.new

syminfo.ticker() 2 overloads

Returns `symbol` name without exchange prefix, e.g. "AAPL".

SYNTAX & OVERLOADS

```
syminfo.ticker(symbol) → simple string
```

```
syminfo.ticker(symbol) → series string
```

ARGUMENTS

symbol (simple string) Symbol. Note that the symbol should be passed with a prefix. For example: "NASDAQ:AAPL" instead of "AAPL".

EXAMPLE

```
//@version=5
indicator("syminfo.ticker fun", overlay=true)
i_sym = input.symbol("NASDAQ:AAPL")
pref = syminfo.prefix(i_sym)
tick = syminfo.ticker(i_sym)
t = ticker.new(pref, tick, session.extended)
s = request.security(t, "1D", close)
plot(s)
```

RETURNS

Returns `symbol` name without exchange prefix, e.g. "AAPL".

REMARKS

The result of the function is used in the ticker.new/ticker.modify and request.security.

SEE ALSO

syminfo.tickerid syminfo.ticker syminfo.prefix syminfo.prefix ticker.new

ta.alma() 2 overloads

Arnaud Legoux Moving Average. It uses Gaussian distribution as weights for moving average.

SYNTAX & OVERLOADS

```
ta.alma(series, length, offset, sigma) → series float
```

```
ta.alma(series, length, offset, sigma, floor) → series float
```

ARGUMENTS

series (series int/float) Series of values to process.

length (series int) Number of bars (length).

offset (simple int/float) Controls tradeoff between smoothness (closer to 1) and responsiveness (closer to 0).

sigma (simple int/float) Changes the smoothness of ALMA. The larger sigma the smoother ALMA.

EXAMPLE

```
//@version=5
indicator("ta.alma", overlay=true)
plot(ta.alma(close, 9, 0.85, 6))

// same on pine, but much less efficient
pine_alma(series, windowsize, offset, sigma) =>
    m = offset * (windowsize - 1)
    //m = math.floor(offset * (windowsize - 1)) // Used as m when math.floor=true
    s = windowsize / sigma
    norm = 0.0
    sum = 0.0
    for i = 0 to windowsize - 1
        weight = math.exp(-1 * math.pow(i - m, 2) / (2 * math.pow(s, 2)))
        norm := norm + weight
        sum := sum + series[windowsize - i - 1] * weight
    sum / norm
plot(pine_alma(close, 9, 0.85, 6))
```

RETURNS

Arnaud Legoux Moving Average.

REMARKS

`na` values in the `source` series are included in calculations and will produce an `na` result.

SEE ALSO

`ta.sma` `ta.ema` `ta.rma` `ta.wma` `ta.vwma` `ta.swma`

ta.atr()

Function atr (average true range) returns the RMA of true range. True range is max(high - low, abs(high - close[1]), abs(low - close[1])).

SYNTAX

```
ta.atr(length) → series float
```

ARGUMENTS

length (simple int) Length (number of bars back).

EXAMPLE

```
//@version=5
indicator("ta.atr")
plot(ta.atr(14))

//the same on pine
pine_atr(length) =>
    trueRange = na(high[1])? high-low : math.max(math.max(high - low, math.abs(high - clos
    //true range can be also calculated with ta.tr(true)
    ta.rma(trueRange, length)

plot(pine_atr(14))
```

RETURNS

Average true range.

REMARKS

`na` values in the `source` series are ignored; the function calculates on the `length` quantity of non-`na` values.

SEE ALSO

ta.tr ta.rma

ta.barssince()

Counts the number of bars since the last time the condition was true.

SYNTAX

```
ta.barssince(condition) → series int
```

ARGUMENTS

condition (series bool) The condition to check for.

EXAMPLE

```
//@version=5
indicator("ta.barssince")
// get number of bars since last color.green bar
plot(ta.barssince(close >= open))
```

RETURNS

Number of bars since condition was true.

REMARKS

If the condition has never been met prior to the current bar, the function returns na.

Please note that using this variable/function can cause indicator repainting.

SEE ALSO

ta.lowestbars ta.highestbars ta.valuewhen ta.highest ta.lowest

ta.bb()

Bollinger Bands. A Bollinger Band is a technical analysis tool defined by a set of lines plotted two standard deviations (positively and negatively) away from a simple moving average (SMA) of the security's price, but can be adjusted to user preferences.

SYNTAX

```
ta.bb(series, length, mult) → [series float, series float, series float]
```

ARGUMENTS

series (series int/float) Series of values to process.

length (series int) Number of bars (length).

mult (simple int/float) Standard deviation factor.

EXAMPLE

```
//@version=5
indicator("ta.bb")

[middle, upper, lower] = ta.bb(close, 5, 4)
plot(middle, color=color.yellow)
plot(upper, color=color.yellow)
plot(lower, color=color.yellow)

// the same on pine
f_bb(src, length, mult) =>
    float basis = ta.sma(src, length)
    float dev = mult * ta.stdev(src, length)
    [basis, basis + dev, basis - dev]

[pineMiddle, pineUpper, pineLower] = f_bb(close, 5, 4)

plot(pineMiddle)
plot(pineUpper)
plot(pineLower)
```

RETURNS

Bollinger Bands.

REMARKS

`na` values in the `source` series are ignored; the function calculates on the `length` quantity of non-`na` values.

SEE ALSO

ta.sma ta.stdev ta.kc

ta.bbw()

Bollinger Bands Width. The Bollinger Band Width is the difference between the upper and the lower Bollinger Bands divided by the middle band.

SYNTAX

```
ta.bbw(series, length, mult) → series float
```

ARGUMENTS

series (series int/float) Series of values to process.

length (series int) Number of bars (length).

mult (simple int/float) Standard deviation factor.

EXAMPLE

```
//@version=5
indicator("ta.bbw")

plot(ta.bbw(close, 5, 4), color=color.yellow)

// the same on pine
f_bbw(src, length, mult) =>
    float basis = ta.sma(src, length)
    float dev = mult * ta.stdev(src, length)
    ((basis + dev) - (basis - dev)) / basis

plot(f_bbw(close, 5, 4))
```

RETURNS

Bollinger Bands Width.

REMARKS

`na` values in the `source` series are ignored; the function calculates on the `length` quantity of non-`na` values.

SEE ALSO

ta.bb ta.sma ta.stdev

ta.cci()

The CCI (commodity channel index) is calculated as the difference between the typical price of a commodity and its simple moving average, divided by the mean absolute deviation of the typical price. The index is scaled by an inverse factor of 0.015 to provide more readable numbers.

SYNTAX

```
ta.cci(source, length) → series float
```

ARGUMENTS

source (series int/float) Series of values to process.

length (series int) Number of bars (length).

RETURNS

Commodity channel index of source for length bars back.

REMARKS

`na` values in the `source` series are ignored.

ta.change() 6 overloads

Compares the current `source` value to its value `length` bars ago and returns the difference.

SYNTAX & OVERLOADS

```
ta.change(source) → series int
```

```
ta.change(source) → series float
```

```
ta.change(source, length) → series int
```

```
ta.change(source, length) → series float
```

```
ta.change(source) → series bool
```

```
ta.change(source, length) → series bool
```

ARGUMENTS

source (series int) Source series.

EXAMPLE

```
//@version=5
indicator('Day and Direction Change', overlay = true)
dailyBarTime = time('1D')
```

```
isNewDay = ta.change(dailyBarTime) != 0
bgcolor(isNewDay ? color.new(color.green, 80) : na)

isGreenBar = close >= open
colorChange = ta.change(isGreenBar)
plotshape(colorChange, 'Direction Change')
```

RETURNS

The difference between the values when they are numerical. When a 'bool' source is used, returns `true` when the current source is different from the previous source.

REMARKS

`na` values in the `source` series are included in calculations and will produce an `na` result.

SEE ALSO

ta.mom ta.cross

ta.cmo()

Chande Momentum Oscillator. Calculates the difference between the sum of recent gains and the sum of recent losses and then divides the result by the sum of all price movement over the same period.

SYNTAX

```
ta.cmo(series, length) → series float
```

ARGUMENTS

series (series int/float) Series of values to process.

length (series int) Number of bars (length).

EXAMPLE

```
//@version=5
indicator("ta.cmo")
plot(ta.cmo(close, 5), color=color.yellow)

// the same on pine
f_cmo(src, length) =>
    float mom = ta.change(src)
    float sm1 = math.sum((mom >= 0) ? mom : 0.0, length)
    float sm2 = math.sum((mom >= 0) ? 0.0 : -mom, length)
```

```
        100 * (sm1 - sm2) / (sm1 + sm2)

    plot(f_cmo(close, 5))
```

RETURNS

Chande Momentum Oscillator.

REMARKS

`na` values in the `source` series are ignored.

SEE ALSO

ta.rsi ta.stoch math.sum

ta.cog()

The cog (center of gravity) is an indicator based on statistics and the Fibonacci golden ratio.

SYNTAX

```
ta.cog(source, length) → series float
```

ARGUMENTS

source (series int/float) Series of values to process.

length (series int) Number of bars (length).

EXAMPLE

```
//@version=5
indicator("ta.cog", overlay=true)
plot(ta.cog(close, 10))

// the same on pine
pine_cog(source, length) =>
    sum = math.sum(source, length)
    num = 0.0
    for i = 0 to length - 1
        price = source[i]
        num := num + price * (i + 1)
    -num / sum

plot(pine_cog(close, 10))
```

RETURNS

Center of Gravity.

REMARKS

`na` values in the `source` series are ignored.

SEE ALSO

ta.stoch

ta.correlation()

Correlation coefficient. Describes the degree to which two series tend to deviate from their ta.sma values.

SYNTAX

```
ta.correlation(source1, source2, length) → series float
```

ARGUMENTS

source1 (series int/float) Source series.

source2 (series int/float) Target series.

length (series int) Length (number of bars back).

RETURNS

Correlation coefficient.

REMARKS

`na` values in the `source` series are ignored; the function calculates on the `length` quantity of non- `na` values.

SEE ALSO

request.security

ta.cross()

SYNTAX

```
ta.cross(source1, source2) → series bool
```

ARGUMENTS

source1 (series int/float) First data series.

source2 (series int/float) Second data series.

RETURNS

true if two series have crossed each other, otherwise false.

SEE ALSO

ta.change

ta.crossover()

The `source1`-series is defined as having crossed over `source2`-series if, on the current bar, the value of `source1` is greater than the value of `source2`, and on the previous bar, the value of `source1` was less than or equal to the value of `source2`.

SYNTAX

```
ta.crossover(source1, source2) → series bool
```

ARGUMENTS

source1 (series int/float) First data series.

source2 (series int/float) Second data series.

RETURNS

true if `source1` crossed over `source2` otherwise false.

ta.crossunder()

The `source1`-series is defined as having crossed under `source2`-series if, on the current bar, the value of `source1` is less than the value of `source2`, and on the previous bar, the value of `source1` was greater than or equal to the value of `source2`.

SYNTAX

```
ta.crossunder(source1, source2) → series bool
```

ARGUMENTS

source1 (series int/float) First data series.

source2 (series int/float) Second data series.

RETURNS

true if `source1` crossed under `source2` otherwise false.

ta.cum()

Cumulative (total) sum of `source`. In other words it's a sum of all elements of `source`.

SYNTAX

```
ta.cum(source) → series float
```

ARGUMENTS

source (series int/float) Source used for the calculation.

RETURNS

Total sum series.

SEE ALSO

math.sum

ta.dev()

Measure of difference between the series and it's ta.sma

SYNTAX

```
ta.dev(source, length) → series float
```

ARGUMENTS

source (series int/float) Series of values to process.

length (series int) Number of bars (length).

EXAMPLE

```
//@version=5
indicator("ta.dev")
plot(ta.dev(close, 10))

// the same on pine
```

```
pine_dev(source, length) =>
    mean = ta.sma(source, length)
    sum = 0.0
    for i = 0 to length - 1
        val = source[i]
        sum := sum + math.abs(val - mean)
    dev = sum/length
plot(pine_dev(close, 10))
```

RETURNS

Deviation of `source` for `length` bars back.

REMARKS

`na` values in the `source` series are ignored.

SEE ALSO

ta.variance ta.stdev

ta.dmi()

The dmi function returns the directional movement index.

SYNTAX

```
ta.dmi(diLength, adxSmoothing) → [series float, series float, series float]
```

ARGUMENTS

diLength (simple int) DI Period.

adxSmoothing (simple int) ADX Smoothing Period.

EXAMPLE

```
//@version=5
indicator(title="Directional Movement Index", shorttitle="DMI", format=format.price, preci
len = input.int(17, minval=1, title="DI Length")
lensig = input.int(14, title="ADX Smoothing", minval=1)
[diplus, diminus, adx] = ta.dmi(len, lensig)
plot(adx, color=color.red, title="ADX")
plot(diplus, color=color.blue, title="+DI")
plot(diminus, color=color.orange, title="-DI")
```

RETURNS

Tuple of three DMI series: Positive Directional Movement (+DI), Negative Directional Movement (-DI) and Average Directional Movement Index (ADX).

SEE ALSO

ta.rsi ta.tsi ta.mfi

ta.ema()

The ema function returns the exponentially weighted moving average. In ema weighting factors decrease exponentially. It calculates by using a formula: `EMA = alpha * source + (1 - alpha) * EMA[1]`, where `alpha = 2 / (length + 1)`.

SYNTAX

```
ta.ema(source, length) → series float
```

ARGUMENTS

source (series int/float) Series of values to process.

length (simple int) Number of bars (length).

EXAMPLE

```
//@version=5
indicator("ta.ema")
plot(ta.ema(close, 15))

//the same on pine
pine_ema(src, length) =>
    alpha = 2 / (length + 1)
    sum = 0.0
    sum := na(sum[1]) ? src : alpha * src + (1 - alpha) * nz(sum[1])
plot(pine_ema(close,15))
```

RETURNS

Exponential moving average of `source` with alpha = 2 / (length + 1).

REMARKS

Please note that using this variable/function can cause indicator repainting.

`na` values in the `source` series are ignored; the function calculates on the `length` quantity of non-`na` values.

SEE ALSO

ta.sma ta.rma ta.wma ta.vwma ta.swma ta.alma

ta.falling()

Test if the `source` series is now falling for `length` bars long.

SYNTAX

```
ta.falling(source, length) → series bool
```

ARGUMENTS

source (series int/float) Series of values to process.

length (series int) Number of bars (length).

RETURNS

true if current `source` value is less than any previous `source` value for `length` bars back, false otherwise.

REMARKS

`na` values in the `source` series are ignored; the function calculates on the `length` quantity of non-`na` values.

SEE ALSO

ta.rising

ta.highest() 2 overloads

Highest value for a given number of bars back.

SYNTAX & OVERLOADS

```
ta.highest(length) → series float
```

```
ta.highest(source, length) → series float
```

ARGUMENTS

length (series int) Number of bars (length).

RETURNS

Highest value in the series.

REMARKS

Two args version: `source` is a series and `length` is the number of bars back.

One arg version: `length` is the number of bars back. Algorithm uses high as a `source` series.

`na` values in the `source` series are ignored.

SEE ALSO

ta.lowest ta.lowestbars ta.highestbars ta.valuewhen ta.barssince

ta.highestbars() 2 overloads

Highest value offset for a given number of bars back.

SYNTAX & OVERLOADS

```
ta.highestbars(length) → series int
```

```
ta.highestbars(source, length) → series int
```

ARGUMENTS

length (series int) Number of bars (length).

RETURNS

Offset to the highest bar.

REMARKS

Two args version: `source` is a series and `length` is the number of bars back.

One arg version: `length` is the number of bars back. Algorithm uses high as a `source` series.

`na` values in the `source` series are ignored.

SEE ALSO

ta.lowest ta.highest ta.lowestbars ta.barssince ta.valuewhen

ta.hma()

The hma function returns the Hull Moving Average.

SYNTAX

```
ta.hma(source, length) → series float
```

ARGUMENTS

source (series int/float) Series of values to process.

length (simple int) Number of bars.

EXAMPLE

```
//@version=5
indicator("Hull Moving Average")
src = input(defval=close, title="Source")
length = input(defval=9, title="Length")
hmaBuildIn = ta.hma(src, length)
plot(hmaBuildIn, title="Hull MA", color=#674EA7)
```

RETURNS

Hull moving average of 'source' for 'length' bars back.

REMARKS

`na` values in the `source` series are ignored.

SEE ALSO

ta.ema ta.rma ta.wma ta.vwma ta.sma

ta.kc() 2 overloads

Keltner Channels. Keltner channel is a technical analysis indicator showing a central moving average line plus channel lines at a distance above and below.

SYNTAX & OVERLOADS

```
ta.kc(series, length, mult) → [series float, series float, series float]
```

```
ta.kc(series, length, mult, useTrueRange) → [series float, series float, series float]
```

ARGUMENTS

series (series int/float) Series of values to process.

length (simple int) Number of bars (length).

mult (simple int/float) Standard deviation factor.

EXAMPLE

```
//@version=5
indicator("ta.kc")

[middle, upper, lower] = ta.kc(close, 5, 4)
plot(middle, color=color.yellow)
plot(upper, color=color.yellow)
plot(lower, color=color.yellow)


// the same on pine
f_kc(src, length, mult, useTrueRange) =>
    float basis = ta.ema(src, length)
    float span = (useTrueRange) ? ta.tr : (high - low)
    float rangeEma = ta.ema(span, length)
    [basis, basis + rangeEma * mult, basis - rangeEma * mult]

[pineMiddle, pineUpper, pineLower] = f_kc(close, 5, 4, true)

plot(pineMiddle)
plot(pineUpper)
plot(pineLower)
```

RETURNS

Keltner Channels.

REMARKS

`na` values in the `source` series are ignored; the function calculates on the `length` quantity of non-`na` values.

SEE ALSO

ta.ema ta.atr ta.bb

ta.kcw() 2 overloads

Keltner Channels Width. The Keltner Channels Width is the difference between the upper and the lower Keltner Channels divided by the middle channel.

SYNTAX & OVERLOADS

```
ta.kcw(series, length, mult) → series float
```

```
ta.kcw(series, length, mult, useTrueRange) → series float
```

ARGUMENTS

series (series int/float) Series of values to process.

length (simple int) Number of bars (length).

mult (simple int/float) Standard deviation factor.

EXAMPLE

```
//@version=5
indicator("ta.kcw")

plot(ta.kcw(close, 5, 4), color=color.yellow)

// the same on pine
f_kcw(src, length, mult, useTrueRange) =>
    float basis = ta.ema(src, length)
    float span = (useTrueRange) ? ta.tr : (high - low)
    float rangeEma = ta.ema(span, length)

    ((basis + rangeEma * mult) - (basis - rangeEma * mult)) / basis

plot(f_kcw(close, 5, 4, true))
```

RETURNS

Keltner Channels Width.

REMARKS

`na` values in the `source` series are ignored; the function calculates on the `length` quantity of non- `na` values.

SEE ALSO

ta.kc ta.ema ta.atr ta.bb

ta.linreg()

Linear regression curve. A line that best fits the prices specified over a user-defined time period. It is calculated using the least squares method. The result of this function is calculated using the formula: linreg = intercept + slope * (length - 1 - offset), where intercept and slope are the values calculated with the least squares method on `source` series.

SYNTAX

```
ta.linreg(source, length, offset) → series float
```

ARGUMENTS

source (series int/float) Source series.

length (series int) Number of bars (length).

offset (simple int) Offset.

RETURNS

Linear regression curve.

REMARKS

`na` values in the `source` series are included in calculations and will produce an `na` result.

ta.lowest() 2 overloads

Lowest value for a given number of bars back.

SYNTAX & OVERLOADS

```
ta.lowest(length) → series float
```

```
ta.lowest(source, length) → series float
```

ARGUMENTS

length (series int) Number of bars (length).

RETURNS

Lowest value in the series.

REMARKS

Two args version: `source` is a series and `length` is the number of bars back.

One arg version: `length` is the number of bars back. Algorithm uses low as a `source` series.

`na` values in the `source` series are ignored.

SEE ALSO

ta.highest ta.lowestbars ta.highestbars ta.valuewhen ta.barssince

ta.lowestbars() 2 overloads

Lowest value offset for a given number of bars back.

SYNTAX & OVERLOADS

```
ta.lowestbars(length) → series int
```

```
ta.lowestbars(source, length) → series int
```

ARGUMENTS

length (series int) Number of bars back.

RETURNS

Offset to the lowest bar.

REMARKS

Two args version: `source` is a series and `length` is the number of bars back.

One arg version: `length` is the number of bars back. Algorithm uses low as a `source` series.

`na` values in the `source` series are ignored.

SEE ALSO

ta.lowest ta.highest ta.highestbars ta.barssince ta.valuewhen

ta.macd()

MACD (moving average convergence/divergence). It is supposed to reveal changes in the strength, direction, momentum, and duration of a trend in a stock's price.

SYNTAX

```
ta.macd(source, fastlen, slowlen, siglen) → [series float, series float, series float]
```

ARGUMENTS

source (series int/float) Series of values to process.

fastlen (simple int) Fast Length parameter.

slowlen (simple int) Slow Length parameter.

siglen (simple int) Signal Length parameter.

EXAMPLE

```
//@version=5
indicator("MACD")
[macdLine, signalLine, histLine] = ta.macd(close, 12, 26, 9)
plot(macdLine, color=color.blue)
plot(signalLine, color=color.orange)
plot(histLine, color=color.red, style=plot.style_histogram)
```

If you need only one value, use placeholders '_' like this:

EXAMPLE

```
//@version=5
indicator("MACD")
[_, signalLine, _] = ta.macd(close, 12, 26, 9)
plot(signalLine, color=color.orange)
```

RETURNS

Tuple of three MACD series: MACD line, signal line and histogram line.

REMARKS

`na` values in the `source` series are ignored; the function calculates on the `length` quantity of non-`na` values.

SEE ALSO

ta.sma ta.ema

ta.max()

Returns the all-time high value of `source` from the beginning of the chart up to the current bar.

SYNTAX

```
ta.max(source) → series float
```

ARGUMENTS

source (series int/float) Source used for the calculation.

REMARKS

na occurrences of `source` are ignored.

ta.median() 2 overloads

Returns the median of the series.

SYNTAX & OVERLOADS

```
ta.median(source, length) → series int
```

```
ta.median(source, length) → series float
```

ARGUMENTS

source (series int) Series of values to process.

length (series int) Number of bars (length).

RETURNS

The median of the series.

REMARKS

`na` values in the `source` series are ignored; the function calculates on the `length` quantity of non- `na` values.

ta.mfi()

Money Flow Index. The Money Flow Index (MFI) is a technical oscillator that uses price and volume for identifying overbought or oversold conditions in an asset.

SYNTAX

```
ta.mfi(series, length) → series float
```

ARGUMENTS

series (series int/float) Series of values to process.

length (series int) Number of bars (length).

EXAMPLE

```
//@version=5
indicator("Money Flow Index")

plot(ta.mfi(hlc3, 14), color=color.yellow)

// the same on pine
pine_mfi(src, length) =>
    float upper = math.sum(volume * (ta.change(src) <= 0.0 ? 0.0 : src), length)
    float lower = math.sum(volume * (ta.change(src) >= 0.0 ? 0.0 : src), length)
    mfi = 100.0 - (100.0 / (1.0 + upper / lower))
    mfi

plot(pine_mfi(hlc3, 14))
```

RETURNS

Money Flow Index.

REMARKS

`na` values in the `source` series are ignored; the function calculates on the `length` quantity of non-`na` values.

SEE ALSO

ta.rsi math.sum

ta.min()

Returns the all-time low value of `source` from the beginning of the chart up to the current bar.

SYNTAX

```
ta.min(source) → series float
```

ARGUMENTS

source (series int/float) Source used for the calculation.

REMARKS

na occurrences of `source` are ignored.

ta.mode() 2 overloads

Returns the mode of the series. If there are several values with the same frequency, it returns the smallest value.

SYNTAX & OVERLOADS

```
ta.mode(source, length) → series int
```

```
ta.mode(source, length) → series float
```

ARGUMENTS

source (series int) Series of values to process.

length (series int) Number of bars (length).

RETURNS

The most frequently occurring value from the `source`. If none exists, returns the smallest value instead.

REMARKS

`na` values in the `source` series are ignored; the function calculates on the `length` quantity of non-`na` values.

ta.mom()

Momentum of `source` price and `source` price `length` bars ago. This is simply a difference: source - source[length].

SYNTAX

```
ta.mom(source, length) → series float
```

ARGUMENTS

source (series int/float) Series of values to process.

length (series int) Offset from the current bar to the previous bar.

RETURNS

Momentum of `source` price and `source` price `length` bars ago.

REMARKS

`na` values in the `source` series are included in calculations and will produce an `na` result.

SEE ALSO

ta.change

ta.percentile_linear_interpolation()

Calculates percentile using method of linear interpolation between the two nearest ranks.

SYNTAX

```
ta.percentile_linear_interpolation(source, length, percentage) → series float
```

ARGUMENTS

source (series int/float) Series of values to process (source).

length (series int) Number of bars back (length).

percentage (simple int/float) Percentage, a number from range 0..100.

RETURNS

P-th percentile of `source` series for `length` bars back.

REMARKS

Note that a percentile calculated using this method will NOT always be a member of the input data set.

`na` values in the `source` series are included in calculations and will produce an `na` result.

SEE ALSO

ta.percentile_nearest_rank

ta.percentile_nearest_rank()

Calculates percentile using method of Nearest Rank.

SYNTAX

```
ta.percentile_nearest_rank(source, length, percentage) → series float
```

ARGUMENTS

source (series int/float) Series of values to process (source).

length (series int) Number of bars back (length).

percentage (simple int/float) Percentage, a number from range 0..100.

RETURNS

P-th percentile of `source` series for `length` bars back.

REMARKS

Using the Nearest Rank method on lengths less than 100 bars back can result in the same number being used for more than one percentile.

A percentile calculated using the Nearest Rank method will always be a member of the input data set.

The 100th percentile is defined to be the largest value in the input data set.

`na` values in the `source` series are ignored.

SEE ALSO

ta.percentile_linear_interpolation

ta.percentrank()

Percent rank is the percents of how many previous values was less than or equal to the current value of given series.

SYNTAX

```
ta.percentrank(source, length) → series float
```

ARGUMENTS

source (series int/float) Series of values to process.

length (series int) Number of bars (length).

RETURNS

Percent rank of `source` for `length` bars back.

REMARKS

`na` values in the `source` series are included in calculations and will produce an `na` result.

ta.pivot_point_levels()

Calculates the pivot point levels using the specified `type` and `anchor`.

SYNTAX

```
ta.pivot_point_levels(type, anchor, developing) → array<float>
```

ARGUMENTS

type (series string) The type of pivot point levels. Possible values: "Traditional", "Fibonacci", "Woodie", "Classic", "DM", "Camarilla".

anchor (series bool) The condition that triggers the reset of the pivot point calculations. When true, calculations reset; when false, results calculated at the last reset persist.

developing (series bool) If false, the values are those calculated the last time the anchor condition was true. They remain constant until the anchor condition becomes true again. If true, the pivots are developing, i.e., they constantly recalculate on the data developing between the point of the last anchor (or bar zero if the anchor condition was never true) and the current bar. Optional. The default is false.

EXAMPLE

```
//@version=5
indicator("Weekly Pivots", max_lines_count=500, overlay=true)
timeframe = "1W"
typeInput = input.string("Traditional", "Type", options=["Traditional", "Fibonacci", "Wooc
weekChange = timeframe.change(timeframe)
pivotPointsArray = ta.pivot_point_levels(typeInput, weekChange)
if weekChange
    for pivotLevel in pivotPointsArray
        line.new(time, pivotLevel, time + timeframe.in_seconds(timeframe) * 1000, pivotLev
```

RETURNS

An `array<float>` with numerical values representing 11 pivot point levels: [P, R1, S1, R2, S2, R3, S3, R4, S4, R5, S5]. Levels absent from the specified `type` return na values (e.g., "DM" only calculates P, R1, and S1).

REMARKS

The `developing` parameter cannot be `true` when `type` is set to "Woodie", because the Woodie calculation for a period depends on that period's open, which means that the pivot value is either available or unavailable, but never developing. If used together, the indicator will return a runtime error.

ta.pivothigh() 2 overloads

This function returns price of the pivot high point. It returns 'NaN', if there was no pivot high point.

SYNTAX & OVERLOADS

```
ta.pivothigh(leftbars, rightbars) → series float
```

```
ta.pivothigh(source, leftbars, rightbars) → series float
```

ARGUMENTS

leftbars (series int/float) Left strength.

rightbars (series int/float) Right strength.

EXAMPLE

```
//@version=5
indicator("PivotHigh", overlay=true)
leftBars = input(2)
rightBars=input(2)
ph = ta.pivothigh(leftBars, rightBars)
plot(ph, style=plot.style_cross, linewidth=3, color= color.red, offset=-rightBars)
```

RETURNS

Price of the point or 'NaN'.

REMARKS

If parameters 'leftbars' or 'rightbars' are series you should use max_bars_back function for the 'source' variable.

ta.pivotlow() 2 overloads

This function returns price of the pivot low point. It returns 'NaN', if there was no pivot low point.

SYNTAX & OVERLOADS

```
ta.pivotlow(leftbars, rightbars) → series float
ta.pivotlow(source, leftbars, rightbars) → series float
```

ARGUMENTS

leftbars (series int/float) Left strength.

rightbars (series int/float) Right strength.

EXAMPLE

```
//@version=5
indicator("PivotLow", overlay=true)
leftBars = input(2)
rightBars=input(2)
pl = ta.pivotlow(close, leftBars, rightBars)
plot(pl, style=plot.style_cross, linewidth=3, color= color.blue, offset=-rightBars)
```

RETURNS

Price of the point or 'NaN'.

REMARKS

If parameters 'leftbars' or 'rightbars' are series you should use max_bars_back function for the 'source' variable.

ta.range() 2 overloads

Returns the difference between the min and max values in a series.

SYNTAX & OVERLOADS

```
ta.range(source, length) → series int
```

```
ta.range(source, length) → series float
```

ARGUMENTS

source (series int) Series of values to process.

length (series int) Number of bars (length).

RETURNS

The difference between the min and max values in the series.

REMARKS

`na` values in the `source` series are ignored; the function calculates on the `length` quantity of non-`na` values.

ta.rising()

Test if the `source` series is now rising for `length` bars long.

SYNTAX

```
ta.rising(source, length) → series bool
```

ARGUMENTS

source (series int/float) Series of values to process.

length (series int) Number of bars (length).

RETURNS

true if current `source` is greater than any previous `source` for `length` bars back, false otherwise.

REMARKS

`na` values in the `source` series are ignored.

SEE ALSO

ta.falling

ta.rma()

Moving average used in RSI. It is the exponentially weighted moving average with alpha = 1 / length.

SYNTAX

```
ta.rma(source, length) → series float
```

ARGUMENTS

source (series int/float) Series of values to process.

length (simple int) Number of bars (length).

EXAMPLE

```
//@version=5
indicator("ta.rma")
plot(ta.rma(close, 15))

//the same on pine
pine_rma(src, length) =>
    alpha = 1/length
    sum = 0.0
    sum := na(sum[1]) ? ta.sma(src, length) : alpha * src + (1 - alpha) * nz(sum[1])
plot(pine_rma(close, 15))
```

RETURNS

Exponential moving average of `source` with alpha = 1 / `length`.

REMARKS

`na` values in the `source` series are ignored; the function calculates on the `length` quantity of non-`na` values.

SEE ALSO

ta.sma ta.ema ta.wma ta.vwma ta.swma ta.alma ta.rsi

ta.roc()

Calculates the percentage of change (rate of change) between the current value of `source` and its value `length` bars ago.

It is calculated by the formula: 100 * change(src, length) / src[length].

SYNTAX

```
ta.roc(source, length) → series float
```

ARGUMENTS

source (series int/float) Series of values to process.

length (series int) Number of bars (length).

RETURNS

The rate of change of `source` for `length` bars back.

REMARKS

`na` values in the `source` series are included in calculations and will produce an `na` result.

ta.rsi()

Relative strength index. It is calculated using the `ta.rma()` of upward and downward changes of `source` over the last `length` bars.

SYNTAX

```
ta.rsi(source, length) → series float
```

ARGUMENTS

source (series int/float) Series of values to process.

length (simple int) Number of bars (length).

EXAMPLE

```
//@version=5
indicator("ta.rsi")
plot(ta.rsi(close, 7))

// same on pine, but less efficient
pine_rsi(x, y) =>
    u = math.max(x - x[1], 0) // upward ta.change
    d = math.max(x[1] - x, 0) // downward ta.change
    rs = ta.rma(u, y) / ta.rma(d, y)
    res = 100 - 100 / (1 + rs)
    res

plot(pine_rsi(close, 7))
```

RETURNS

Relative strength index.

REMARKS

`na` values in the `source` series are ignored; the function calculates on the `length` quantity of non-`na` values.

SEE ALSO

ta.rma

ta.sar()

Parabolic SAR (parabolic stop and reverse) is a method devised by J. Welles Wilder, Jr., to find potential reversals in the market price direction of traded goods.

SYNTAX

```
ta.sar(start, inc, max) → series float
```

ARGUMENTS

start (simple int/float) Start.

inc (simple int/float) Increment.

max (simple int/float) Maximum.

EXAMPLE

```
//@version=5
indicator("ta.sar")
plot(ta.sar(0.02, 0.02, 0.2), style=plot.style_cross, linewidth=3)

// The same on Pine Script®
pine_sar(start, inc, max) =>
    var float result = na
    var float maxMin = na
    var float acceleration = na
    var bool isBelow = na
    bool isFirstTrendBar = false

    if bar_index == 1
        if close > close[1]
            isBelow := true
            maxMin := high
            result := low[1]
        else
            isBelow := false
            maxMin := low
            result := high[1]
        isFirstTrendBar := true
        acceleration := start

    result := result + acceleration * (maxMin - result)

    if isBelow
        if result > low
            isFirstTrendBar := true
            isBelow := false
            result := math.max(high, maxMin)
            maxMin := low
            acceleration := start
    else
```

```
        if result < high
            isFirstTrendBar := true
            isBelow := true
            result := math.min(low, maxMin)
            maxMin := high
            acceleration := start

    if not isFirstTrendBar
        if isBelow
            if high > maxMin
                maxMin := high
                acceleration := math.min(acceleration + inc, max)
        else
            if low < maxMin
                maxMin := low
                acceleration := math.min(acceleration + inc, max)

    if isBelow
        result := math.min(result, low[1])
        if bar_index > 1
            result := math.min(result, low[2])

    else
        result := math.max(result, high[1])
        if bar_index > 1
            result := math.max(result, high[2])

    result

plot(pine_sar(0.02, 0.02, 0.2), style=plot.style_cross, linewidth=3)
```

RETURNS

Parabolic SAR.

ta.sma()

The sma function returns the moving average, that is the sum of last y values of x, divided by y.

SYNTAX

```
ta.sma(source, length) → series float
```

ARGUMENTS

source (series int/float) Series of values to process.

length (series int) Number of bars (length).

EXAMPLE

```
//@version=5
indicator("ta.sma")
plot(ta.sma(close, 15))

// same on pine, but much less efficient
pine_sma(x, y) =>
    sum = 0.0
    for i = 0 to y - 1
        sum := sum + x[i] / y
    sum
plot(pine_sma(close, 15))
```

RETURNS

Simple moving average of `source` for `length` bars back.

REMARKS

`na` values in the `source` series are ignored.

SEE ALSO

ta.ema ta.rma ta.wma ta.vwma ta.swma ta.alma

ta.stdev()

SYNTAX

```
ta.stdev(source, length, biased) → series float
```

ARGUMENTS

source (series int/float) Series of values to process.

length (series int) Number of bars (length).

biased (series bool) Determines which estimate should be used. Optional. The default is true.

EXAMPLE

```
//@version=5
indicator("ta.stdev")
plot(ta.stdev(close, 5))

//the same on pine
```

```
isZero(val, eps) => math.abs(val) <= eps

SUM(fst, snd) =>
    EPS = 1e-10
    res = fst + snd
    if isZero(res, EPS)
        res := 0
    else
        if not isZero(res, 1e-4)
            res := res
        else
            15

pine_stdev(src, length) =>
    avg = ta.sma(src, length)
    sumOfSquareDeviations = 0.0
    for i = 0 to length - 1
        sum = SUM(src[i], -avg)
        sumOfSquareDeviations := sumOfSquareDeviations + sum * sum

    stdev = math.sqrt(sumOfSquareDeviations / length)
plot(pine_stdev(close, 5))
```

RETURNS

Standard deviation.

REMARKS

If `biased` is true, function will calculate using a biased estimate of the entire population, if false - unbiased estimate of a sample.

`na` values in the `source` series are ignored; the function calculates on the `length` quantity of non-`na` values.

SEE ALSO

ta.dev ta.variance

ta.stoch()

Stochastic. It is calculated by a formula: 100 * (close - lowest(low, length)) / (highest(high, length) - lowest(low, length)).

SYNTAX

```
ta.stoch(source, high, low, length) → series float
```

ARGUMENTS

source (series int/float) Source series.

high (series int/float) Series of high.

low (series int/float) Series of low.

length (series int) Length (number of bars back).

RETURNS

Stochastic.

REMARKS

`na` values in the `source` series are ignored.

SEE ALSO

ta.cog

ta.supertrend()

The Supertrend Indicator. The Supertrend is a trend following indicator.

SYNTAX

```
ta.supertrend(factor, atrPeriod) → [series float, series float]
```

ARGUMENTS

factor (series int/float) The multiplier by which the ATR will get multiplied.

atrPeriod (simple int) Length of ATR.

EXAMPLE

```
//@version=5
indicator("Pine Script® Supertrend")

[supertrend, direction] = ta.supertrend(3, 10)
plot(direction < 0 ? supertrend : na, "Up direction", color = color.green, style=plot.styl
plot(direction > 0 ? supertrend : na, "Down direction", color = color.red, style=plot.styl

// The same on Pine Script®
pine_supertrend(factor, atrPeriod) =>
    src = hl2
    atr = ta.atr(atrPeriod)
    upperBand = src + factor * atr
    lowerBand = src - factor * atr
    prevLowerBand = nz(lowerBand[1])
    prevUpperBand = nz(upperBand[1])
    lowerBand := lowerBand > prevLowerBand or close[1] < prevLowerBand ? lowerBand : prevL
    upperBand := upperBand < prevUpperBand or close[1] > prevUpperBand ? upperBand : prevU
    int _direction = na
    float superTrend = na
    prevSuperTrend = superTrend[1]
    if na(atr[1])
        _direction := 1
    else if prevSuperTrend == prevUpperBand
        _direction := close > upperBand ? -1 : 1
    else
        _direction := close < lowerBand ? 1 : -1
    superTrend := _direction == -1 ? lowerBand : upperBand
    [superTrend, _direction]

[Pine_Supertrend, pineDirection] = pine_supertrend(3, 10)
plot(pineDirection < 0 ? Pine_Supertrend : na, "Up direction", color = color.green, style=
plot(pineDirection > 0 ? Pine_Supertrend : na, "Down direction", color = color.red, style=
```

RETURNS

Tuple of two supertrend series: supertrend line and direction of trend. Possible values are 1 (down direction) and -1 (up direction).

SEE ALSO

ta.macd

ta.swma()

Symmetrically weighted moving average with fixed length: 4. Weights: [1/6, 2/6, 2/6, 1/6].

SYNTAX

```
ta.swma(source) → series float
```

ARGUMENTS

source (series int/float) Source series.

EXAMPLE

```
//@version=5
indicator("ta.swma")
plot(ta.swma(close))

// same on pine, but less efficient
pine_swma(x) =>
```

```
    x[3] * 1 / 6 + x[2] * 2 / 6 + x[1] * 2 / 6 + x[0] * 1 / 6
plot(pine_swma(close))
```

RETURNS

Symmetrically weighted moving average.

REMARKS

`na` values in the `source` series are included in calculations and will produce an `na` result.

SEE ALSO

ta.sma ta.ema ta.rma ta.wma ta.vwma ta.alma

ta.tr()

SYNTAX

```
ta.tr(handle_na) → series float
```

ARGUMENTS

handle_na (simple bool) How NaN values are handled. if true, and previous day's close is NaN then tr would be calculated as current day high-low. Otherwise (if false) tr would return NaN in such cases. Also note, that ta.atr uses ta.tr(true).

RETURNS

True range. It is math.max(high - low, math.abs(high - close[1]), math.abs(low - close[1])).

REMARKS

ta.tr(false) is exactly the same as ta.tr.

SEE ALSO

ta.tr ta.atr

ta.tsi()

True strength index. It uses moving averages of the underlying momentum of a financial instrument.

SYNTAX

```
ta.tsi(source, short_length, long_length) → series float
```

ARGUMENTS

source (series int/float) Source series.

short_length (simple int) Short length.

long_length (simple int) Long length.

RETURNS

True strength index. A value in range [-1, 1].

REMARKS

`na` values in the `source` series are ignored; the function calculates on the `length` quantity of non-`na` values.

ta.valuewhen() 4 overloads

Returns the value of the `source` series on the bar where the `condition` was true on the nth most recent occurrence.

SYNTAX & OVERLOADS

```
ta.valuewhen(condition, source, occurrence) → series color
```

```
ta.valuewhen(condition, source, occurrence) → series int
```

```
ta.valuewhen(condition, source, occurrence) → series float
```

```
ta.valuewhen(condition, source, occurrence) → series bool
```

ARGUMENTS

condition (series bool) The condition to search for.

source (series color) The value to be returned from the bar where the condition is met.

occurrence (simple int) The occurrence of the condition. The numbering starts from 0 and goes back in time, so '0' is the most recent occurrence of `condition`, '1' is the second most recent and so forth. Must be an integer >= 0.

EXAMPLE

```
//@version=5
indicator("ta.valuewhen")
slow = ta.sma(close, 7)
fast = ta.sma(close, 14)
// Get value of `close` on second most recent cross
plot(ta.valuewhen(ta.cross(slow, fast), close, 1))
```

REMARKS

This function requires execution on every bar. It is not recommended to use it inside a for or while loop structure, where its behavior can be unexpected. Please note that using this function can cause indicator repainting.

SEE ALSO

ta.lowestbars ta.highestbars ta.barssince ta.highest ta.lowest

ta.variance()

Variance is the expectation of the squared deviation of a series from its mean (ta.sma), and it informally measures how far a set of numbers are spread out from their mean.

SYNTAX

```
ta.variance(source, length, biased) → series float
```

ARGUMENTS

source (series int/float) Series of values to process.

length (series int) Number of bars (length).

biased (series bool) Determines which estimate should be used. Optional. The default is true.

RETURNS

Variance of `source` for `length` bars back.

REMARKS

If `biased` is true, function will calculate using a biased estimate of the entire population, if false - unbiased estimate of a sample.

`na` values in the `source` series are ignored; the function calculates on the `length` quantity of non- `na` values.

SEE ALSO

ta.dev ta.stdev

ta.vwap() 3 overloads

Volume weighted average price.

SYNTAX & OVERLOADS

```
ta.vwap(source) → series float
```

```
ta.vwap(source, anchor) → series float
```

```
ta.vwap(source, anchor, stdev_mult) → [series float, series float, series float]
```

ARGUMENTS

source (series int/float) Source used for the VWAP calculation.

EXAMPLE

```
//@version=5
indicator("Simple VWAP")
vwap = ta.vwap(open)
plot(vwap)
```

EXAMPLE

```
//@version=5
indicator("Advanced VWAP")
vwapAnchorInput = input.string("Daily", "Anchor", options = ["Daily", "Weekly", "Monthly"]
stdevMultiplierInput = input.float(1.0, "Standard Deviation Multiplier")
anchorTimeframe = switch vwapAnchorInput
    "Daily"   => "1D"
    "Weekly"  => "1W"
    "Monthly" => "1M"
anchor = timeframe.change(anchorTimeframe)
[vwap, upper, lower] = ta.vwap(open, anchor, stdevMultiplierInput)
plot(vwap)
plot(upper, color = color.green)
plot(lower, color = color.green)
```

RETURNS

A VWAP series, or a tuple [vwap, upper_band, lower_band] if `stdev_mult` is specified.

REMARKS

Calculations only begin the first time the anchor condition becomes true. Until then, the function returns na.

SEE ALSO

ta.vwap

ta.vwma()

The vwma function returns volume-weighted moving average of `source` for `length` bars back. It is the same as: sma(source * volume, length) / sma(volume, length).

SYNTAX

```
ta.vwma(source, length) → series float
```

ARGUMENTS

source (series int/float) Series of values to process.

length (series int) Number of bars (length).

EXAMPLE

```
//@version=5
indicator("ta.vwma")
plot(ta.vwma(close, 15))

// same on pine, but less efficient
pine_vwma(x, y) =>
    ta.sma(x * volume, y) / ta.sma(volume, y)
plot(pine_vwma(close, 15))
```

RETURNS

Volume-weighted moving average of `source` for `length` bars back.

REMARKS

`na` values in the `source` series are ignored.

SEE ALSO

ta.sma ta.ema ta.rma ta.wma ta.swma ta.alma

ta.wma()

The wma function returns weighted moving average of `source` for `length` bars back. In wma weighting factors decrease in arithmetical progression.

SYNTAX

```
ta.wma(source, length) → series float
```

ARGUMENTS

source (series int/float) Series of values to process.

length (series int) Number of bars (length).

EXAMPLE

```
//@version=5
indicator("ta.wma")
plot(ta.wma(close, 15))

// same on pine, but much less efficient
pine_wma(x, y) =>
    norm = 0.0
    sum = 0.0
    for i = 0 to y - 1
        weight = (y - i) * y
        norm := norm + weight
        sum := sum + x[i] * weight
    sum / norm
plot(pine_wma(close, 15))
```

RETURNS

Weighted moving average of `source` for `length` bars back.

REMARKS

`na` values in the `source` series are ignored.

SEE ALSO

ta.sma ta.ema ta.rma ta.vwma ta.swma ta.alma

ta.wpr()

Williams %R. The oscillator shows the current closing price in relation to the high and low of the past 'length' bars.

SYNTAX

```
ta.wpr(length) → series float
```

ARGUMENTS

length (series int) Number of bars.

EXAMPLE

```
//@version=5
indicator("Williams %R", shorttitle="%R", format=format.price, precision=2)
plot(ta.wpr(14), title="%R", color=color.new(#ff6d00, 0))
```

RETURNS

Williams %R.

REMARKS

`na` values in the `source` series are ignored.

SEE ALSO

ta.mfi ta.cmo

table()

Casts na to table

SYNTAX

```
table(x) → series table
```

ARGUMENTS

x (series table) The value to convert to the specified type, usually na.

RETURNS

The value of the argument after casting to table.

SEE ALSO

float int bool color string line label

table.cell()

The function defines a cell in the table and sets its attributes.

SYNTAX

```
table.cell(table_id, column, row, text, width, height, text_color, text_halign,
text_valign, text_size, bgcolor, tooltip, text_font_family) → void
```

ARGUMENTS

table_id (series table) A table object.

column (series int) The index of the cell's column. Numbering starts at 0.

row (series int) The index of the cell's row. Numbering starts at 0.

text (series string) The text to be displayed inside the cell. Optional. The default is empty string.

width (series int/float) The width of the cell as a % of the indicator's visual space. Optional. By default, auto-adjusts the width based on the text inside the cell. Value 0 has the same effect.

height (series int/float) The height of the cell as a % of the indicator's visual space. Optional. By default, auto-adjusts the height based on the text inside of the cell. Value 0 has the same effect.

text_color (series color) The color of the text. Optional. The default is color.black.

text_halign (series string) The horizontal alignment of the cell's text. Optional. The default value is text.align_center. Possible values: text.align_left, text.align_center, text.align_right.

text_valign (series string) The vertical alignment of the cell's text. Optional. The default value is text.align_center. Possible values: text.align_top, text.align_center, text.align_bottom.

text_size (series string) The size of the text. An optional parameter, the default value is size.normal. Possible values: size.auto, size.tiny, size.small, size.normal, size.large, size.huge.

bgcolor (series color) The background color of the text. Optional. The default is no color.

tooltip (series string) The tooltip to be displayed inside the cell. Optional.

text_font_family (series string) The font family of the text. Optional. The default value is font.family_default. Possible values: font.family_default, font.family_monospace.

REMARKS

This function does not create the table itself, but defines the table's cells. To use it, you first need to create a table object with table.new.

Each table.cell call overwrites all previously defined properties of a cell. If you call table.cell twice in a row, e.g., the first time with text='Test Text', and the second time with text_color=color.red but without a new text argument, the default value of the 'text' being an empty string, it will overwrite 'Test Text', and your cell will display an empty string. If you want, instead, to modify any of the cell's properties, use the table.cell_set_*() functions.

A single script can only display one table in each of the possible locations. If table.cell is used on several bars to change the same attribute of a cell (e.g. change the background color of the cell to red on the first bar, then to yellow on the second bar), only the last change will be reflected in the table, i.e., the cell's background will be yellow. Avoid unnecessary setting of cell properties by enclosing function calls in an if barstate.islast block whenever possible, to restrict their execution to the last bar of the series.

SEE ALSO

table.cell_set_bgcolor table.cell_set_height table.cell_set_text table.cell_set_text_color table.cell_set_text_halign table.cell_set_text_size table.cell_set_text_valign table.cell_set_width table.cell_set_tooltip

table.cell_set_bgcolor()

The function sets the background color of the cell.

SYNTAX

```
table.cell_set_bgcolor(table_id, column, row, bgcolor) → void
```

ARGUMENTS

table_id (series table) A table object.

column (series int) The index of the cell's column. Numbering starts at 0.

row (series int) The index of the cell's row. Numbering starts at 0.

bgcolor (series color) The background color of the cell.

SEE ALSO

table.cell_set_height table.cell_set_text table.cell_set_text_color table.cell_set_text_halign table.cell_set_text_size table.cell_set_text_valign

table.cell_set_width table.cell_set_tooltip

table.cell_set_height()

The function sets the height of cell.

SYNTAX

```
table.cell_set_height(table_id, column, row, height) → void
```

ARGUMENTS

table_id (series table) A table object.

column (series int) The index of the cell's column. Numbering starts at 0.

row (series int) The index of the cell's row. Numbering starts at 0.

height (series int/float) The height of the cell as a % of the chart window. Passing 0 auto-adjusts the height based on the text inside of the cell.

SEE ALSO

table.cell_set_bgcolor table.cell_set_text table.cell_set_text_color

table.cell_set_text_halign table.cell_set_text_size table.cell_set_text_valign

table.cell_set_width table.cell_set_tooltip

table.cell_set_text()

The function sets the text in the specified cell.

SYNTAX

```
table.cell_set_text(table_id, column, row, text) → void
```

ARGUMENTS

table_id (series table) A table object.

column (series int) The index of the cell's column. Numbering starts at 0.

row (series int) The index of the cell's row. Numbering starts at 0.

text (series string) The text to be displayed inside the cell.

EXAMPLE

```
//@version=5
indicator("TABLE example")
var tLog = table.new(position = position.top_left, rows = 1, columns = 2, bgcolor = color.
table.cell(tLog, row = 0, column = 0, text = "sometext", text_color = color.blue)
table.cell_set_text(tLog, row = 0, column = 0, text = "sometext")
```

SEE ALSO

table.cell_set_bgcolor table.cell_set_height table.cell_set_text_color table.cell_set_text_halign table.cell_set_text_size table.cell_set_text_valign table.cell_set_width table.cell_set_tooltip

table.cell_set_text_color()

The function sets the color of the text inside the cell.

SYNTAX

```
table.cell_set_text_color(table_id, column, row, text_color) → void
```

ARGUMENTS

table_id (series table) A table object.

column (series int) The index of the cell's column. Numbering starts at 0.

row (series int) The index of the cell's row. Numbering starts at 0.

text_color (series color) The color of the text.

SEE ALSO

table.cell_set_bgcolor table.cell_set_height table.cell_set_text table.cell_set_text_halign table.cell_set_text_size table.cell_set_text_valign table.cell_set_width table.cell_set_tooltip

table.cell_set_text_font_family()

The function sets the font family of the text inside the cell.

SYNTAX

```
table.cell_set_text_font_family(table_id, column, row, text_font_family) → void
```

ARGUMENTS

table_id (series table) A table object.

column (series int) The index of the cell's column. Numbering starts at 0.

row (series int) The index of the cell's row. Numbering starts at 0.

text_font_family (series string) The font family of the text. Possible values: font.family_default, font.family_monospace.

EXAMPLE

```
//@version=5
indicator("Example of setting the table cell font")
var t = table.new(position.top_left, rows = 1, columns = 1)
table.cell(t, 0, 0, "monospace", text_color = color.blue)
table.cell_set_text_font_family(t, 0, 0, font.family_monospace)
```

SEE ALSO

table.new font.family_default font.family_monospace

table.cell_set_text_halign()

The function sets the horizontal alignment of the cell's text.

SYNTAX

```
table.cell_set_text_halign(table_id, column, row, text_halign) → void
```

ARGUMENTS

table_id (series table) A table object.

column (series int) The index of the cell's column. Numbering starts at 0.

row (series int) The index of the cell's row. Numbering starts at 0.

text_halign (series string) The horizontal alignment of a cell's text. Possible values: text.align_left, text.align_center, text.align_right.

SEE ALSO

table.cell_set_bgcolor table.cell_set_height table.cell_set_text table.cell_set_text_color table.cell_set_text_size table.cell_set_text_valign table.cell_set_width table.cell_set_tooltip

table.cell_set_text_size()

The function sets the size of the cell's text.

SYNTAX

```
table.cell_set_text_size(table_id, column, row, text_size) → void
```

ARGUMENTS

table_id (series table) A table object.

column (series int) The index of the cell's column. Numbering starts at 0.

row (series int) The index of the cell's row. Numbering starts at 0.

text_size (series string) The size of the text. Possible values: size.auto, size.tiny, size.small, size.normal, size.large, size.huge.

SEE ALSO

table.cell_set_bgcolor table.cell_set_height table.cell_set_text table.cell_set_text_color table.cell_set_text_halign table.cell_set_text_valign table.cell_set_width table.cell_set_tooltip

table.cell_set_text_valign()

The function sets the vertical alignment of a cell's text.

SYNTAX

```
table.cell_set_text_valign(table_id, column, row, text_valign) → void
```

ARGUMENTS

table_id (series table) A table object.

column (series int) The index of the cell's column. Numbering starts at 0.

row (series int) The index of the cell's row. Numbering starts at 0.

text_valign (series string) The vertical alignment of the cell's text. Possible values: text.align_top, text.align_center, text.align_bottom.

SEE ALSO

table.cell_set_bgcolor table.cell_set_height table.cell_set_text table.cell_set_text_color table.cell_set_text_halign table.cell_set_text_size table.cell_set_width table.cell_set_tooltip

table.cell_set_tooltip()

The function sets the tooltip in the specified cell.

SYNTAX

```
table.cell_set_tooltip(table_id, column, row, tooltip) → void
```

ARGUMENTS

table_id (series table) A table object.

column (series int) The index of the cell's column. Numbering starts at 0.

row (series int) The index of the cell's row. Numbering starts at 0.

tooltip (series string) The tooltip to be displayed inside the cell.

EXAMPLE

```
//@version=5
indicator("TABLE example")
var tLog = table.new(position = position.top_left, rows = 1, columns = 2, bgcolor = color.
table.cell(tLog, row = 0, column = 0, text = "sometext", text_color = color.blue)
table.cell_set_tooltip(tLog, row = 0, column = 0, tooltip = "sometext")
```

SEE ALSO

table.cell_set_bgcolor table.cell_set_height table.cell_set_text_color table.cell_set_text_halign table.cell_set_text_size table.cell_set_text_valign table.cell_set_width table.cell_set_text

table.cell_set_width()

The function sets the width of the cell.

SYNTAX

```
table.cell_set_width(table_id, column, row, width) → void
```

ARGUMENTS

table_id (series table) A table object.

column (series int) The index of the cell's column. Numbering starts at 0.

row (series int) The index of the cell's row. Numbering starts at 0.

width (series int/float) The width of the cell as a % of the chart window. Passing 0 auto-adjusts the width based on the text inside of the cell.

SEE ALSO

table.cell_set_bgcolor table.cell_set_height table.cell_set_text table.cell_set_text_color table.cell_set_text_halign table.cell_set_text_size table.cell_set_text_valign table.cell_set_tooltip

table.clear()

The function removes a cell or a sequence of cells from the table. The cells are removed in a rectangle shape where the start_column and start_row specify the top-left corner, and end_column and end_row specify the bottom-right corner.

SYNTAX

```
table.clear(table_id, start_column, start_row, end_column, end_row) → void
```

ARGUMENTS

table_id (series table) A table object.

start_column (series int) The index of the column of the first cell to delete. Numbering starts at 0.

start_row (series int) The index of the row of the first cell to delete. Numbering starts at 0.

end_column (series int) The index of the column of the last cell to delete. Optional. The default is the argument used for start_column. Numbering starts at 0.

end_row (series int) The index of the row of the last cell to delete. Optional. The default is the argument used for start_row. Numbering starts at 0.

EXAMPLE

```
//@version=5
indicator("A donut", overlay=true)
if barstate.islast
    colNum = 8, rowNum = 8
    padding = "◯"
    donutTable = table.new(position.middle_right, colNum, rowNum)
    for c = 0 to colNum - 1
        for r = 0 to rowNum - 1
            table.cell(donutTable, c, r, text=padding, bgcolor=#face6e, text_color=color.r
    table.clear(donutTable, 2, 2, 5, 5)
```

SEE ALSO

table.delete table.new

table.delete()

The function deletes a table.

SYNTAX

```
table.delete(table_id) → void
```

ARGUMENTS

table_id (series table) A table object.

EXAMPLE

```
//@version=5
indicator("table.delete example")
var testTable = table.new(position = position.top_right, columns = 2, rows = 1, bgcolor =
if barstate.islast
    table.cell(table_id = testTable, column = 0, row = 0, text = "Open is " + str.tostring
    table.cell(table_id = testTable, column = 1, row = 0, text = "Close is " + str.tostrin
if barstate.isrealtime
    table.delete(testTable)
```

SEE ALSO

table.new table.clear

table.merge_cells()

The function merges a sequence of cells in the table into one cell. The cells are merged in a rectangle shape where the start_column and start_row specify the top-left corner, and end_column and end_row specify the bottom-right corner.

SYNTAX

```
table.merge_cells(table_id, start_column, start_row, end_column, end_row) → void
```

ARGUMENTS

table_id (series table) A table object.

start_column (series int) The index of the column of the first cell to merge. Numbering starts at 0.

start_row (series int) The index of the row of the first cell to merge. Numbering starts at 0.

end_column (series int) The index of the column of the last cell to merge. Numbering starts at 0.

end_row (series int) The index of the row of the last cell to merge. Numbering starts at 0.

EXAMPLE

```
//@version=5
indicator("table.merge_cells example")
SMA50  = ta.sma(close, 50)
SMA100 = ta.sma(close, 100)
SMA200 = ta.sma(close, 200)
if barstate.islast
    maTable = table.new(position.bottom_right, 3, 3, bgcolor = color.gray, border_width =
    // Header
    table.cell(maTable, 0, 0, text = "SMA Table")
    table.merge_cells(maTable, 0, 0, 2, 0)
    // Cell Titles
    table.cell(maTable, 0, 1, text = "SMA 50")
    table.cell(maTable, 1, 1, text = "SMA 100")
    table.cell(maTable, 2, 1, text = "SMA 200")
    // Values
    table.cell(maTable, 0, 2, bgcolor = color.white, text = str.tostring(SMA50))
    table.cell(maTable, 1, 2, bgcolor = color.white, text = str.tostring(SMA100))
    table.cell(maTable, 2, 2, bgcolor = color.white, text = str.tostring(SMA200))
```

REMARKS

This function will merge cells, even if their properties are not yet defined with table.cell.

The resulting merged cell inherits all of its values from the cell located at `start_column`:`start_row`, except width and height. The width and height of the resulting merged cell are based on the width/height of other cells in the neighboring columns/rows and cannot be set manually.

To modify the merged cell with any of the `table.cell_set_*` functions, target the cell at the `start_column`:`start_row` coordinates.

An attempt to merge a cell that has already been merged will result in an error.

SEE ALSO

table.delete table.new

table.new()

The function creates a new table.

SYNTAX

```
table.new(position, columns, rows, bgcolor, frame_color, frame_width, border_color,
border_width, force_overlay) → series table
```

ARGUMENTS

position (series string) Position of the table. Possible values are: position.top_left, position.top_center, position.top_right, position.middle_left, position.middle_center, position.middle_right, position.bottom_left, position.bottom_center, position.bottom_right.

columns (series int) The number of columns in the table.

rows (series int) The number of rows in the table.

bgcolor (series color) The background color of the table. Optional. The default is no color.

frame_color (series color) The color of the outer frame of the table. Optional. The default is no color.

frame_width (series int) The width of the outer frame of the table. Optional. The default is 0.

border_color (series color) The color of the borders of the cells (excluding the outer frame). Optional. The default is no color.

border_width (series int) The width of the borders of the cells (excluding the outer frame). Optional. The default is 0.

force_overlay (const bool) If true, the drawing will display on the main chart pane, even when the script occupies a separate pane. Optional. The default is false.

EXAMPLE

```
//@version=5
indicator("table.new example")
var testTable = table.new(position = position.top_right, columns = 2, rows = 1, bgcolor =
if barstate.islast
    table.cell(table_id = testTable, column = 0, row = 0, text = "Open is " + str.tostring
    table.cell(table_id = testTable, column = 1, row = 0, text = "Close is " + str.tostrin
```

RETURNS

The ID of a table object that can be passed to other table.*() functions.

REMARKS

This function creates the table object itself, but the table will not be displayed until its cells are populated. To define a cell and change its contents or attributes, use table.cell and other table.cell_*() functions.

One table.new call can only display one table (the last one drawn), but the function itself will be recalculated on each bar it is used on. For performance reasons, it is wise to use table.new in conjunction with either the var keyword (so the table object is only created on the first bar) or in an if barstate.islast block (so the table object is only created on the last bar).

SEE ALSO

table.cell table.clear table.delete table.set_bgcolor table.set_border_color

table.set_border_width table.set_frame_color table.set_frame_width table.set_position

table.set_bgcolor()

The function sets the background color of a table.

SYNTAX

```
table.set_bgcolor(table_id, bgcolor) → void
```

ARGUMENTS

table_id (series table) A table object.

bgcolor (series color) The background color of the table. Optional. The default is no color.

SEE ALSO

table.clear table.delete table.new table.set_border_color table.set_border_width table.set_frame_color table.set_frame_width table.set_position

table.set_border_color()

The function sets the color of the borders (excluding the outer frame) of the table's cells.

SYNTAX

```
table.set_border_color(table_id, border_color) → void
```

ARGUMENTS

table_id (series table) A table object.

border_color (series color) The color of the borders. Optional. The default is no color.

SEE ALSO

table.clear table.delete table.new table.set_frame_color table.set_border_width table.set_bgcolor table.set_frame_width table.set_position

table.set_border_width()

The function sets the width of the borders (excluding the outer frame) of the table's cells.

SYNTAX

```
table.set_border_width(table_id, border_width) → void
```

ARGUMENTS

table_id (series table) A table object.

border_width (series int) The width of the borders. Optional. The default is 0.

SEE ALSO

table.clear table.delete table.new table.set_frame_color table.set_frame_width table.set_bgcolor table.set_border_color table.set_position

table.set_frame_color()

The function sets the color of the outer frame of a table.

SYNTAX

```
table.set_frame_color(table_id, frame_color) → void
```

ARGUMENTS

table_id (series table) A table object.

frame_color (series color) The color of the frame of the table. Optional. The default is no color.

SEE ALSO

table.clear table.delete table.new table.set_border_color table.set_border_width table.set_bgcolor table.set_frame_width table.set_position

table.set_frame_width()

The function set the width of the outer frame of a table.

SYNTAX

```
table.set_frame_width(table_id, frame_width) → void
```

ARGUMENTS

table_id (series table) A table object.

frame_width (series int) The width of the outer frame of the table. Optional. The default is 0.

SEE ALSO

table.clear table.delete table.new table.set_frame_color table.set_border_width table.set_bgcolor table.set_border_color table.set_position

table.set_position()

The function sets the position of a table.

SYNTAX

```
table.set_position(table_id, position) → void
```

ARGUMENTS

table_id (series table) A table object.

position (series string) Position of the table. Possible values are: position.top_left, position.top_center, position.top_right, position.middle_left, position.middle_center, position.middle_right, position.bottom_left, position.bottom_center, position.bottom_right.

SEE ALSO

table.clear table.delete table.new table.set_bgcolor table.set_border_color table.set_border_width table.set_frame_color table.set_frame_width

ticker.heikinashi()

Creates a ticker identifier for requesting Heikin Ashi bar values.

SYNTAX

```
ticker.heikinashi(symbol) → simple string
```

ARGUMENTS

symbol (simple string) Symbol ticker identifier.

EXAMPLE

```
//@version=5
indicator("ticker.heikinashi", overlay=true)
heikinashi_close = request.security(ticker.heikinashi(syminfo.tickerid), timeframe.period,

heikinashi_aapl_60_close = request.security(ticker.heikinashi("AAPL"), "60", close)
plot(heikinashi_close)
plot(heikinashi_aapl_60_close)
```

RETURNS

String value of ticker id, that can be supplied to request.security function.

SEE ALSO

syminfo.tickerid syminfo.ticker request.security ticker.renko ticker.linebreak ticker.kagi ticker.pointfigure

ticker.inherit()

Constructs a ticker ID for the specified `symbol` with additional parameters inherited from the ticker ID passed into the function call, allowing the script to request a symbol's data using the same modifiers that the `from_tickerid` has, including extended session, dividend adjustment, currency conversion, non-standard chart types, back-adjustment, settlement-as-close, etc.

SYNTAX

```
ticker.inherit(from_tickerid, symbol) → simple string
```

ARGUMENTS

from_tickerid (simple string) The ticker ID to inherit modifiers from.

symbol (simple string) The symbol to construct the new ticker ID for.

EXAMPLE

```
//@version=5
indicator("ticker.inherit")

//@variable A "NASDAQ:AAPL" ticker ID with Extender Hours enabled.
tickerExtHours = ticker.new("NASDAQ", "AAPL", session.extended)
//@variable A Heikin Ashi ticker ID for "NASDAQ:AAPL" with Extended Hours enabled.
HAtickerExtHours = ticker.heikinashi(tickerExtHours)
//@variable The "NASDAQ:MSFT" symbol with no modifiers.
testSymbol = "NASDAQ:MSFT"
//@variable A ticker ID for "NASDAQ:MSFT" with inherited Heikin Ashi and Extended Hours mo
testSymbolHAtickerExtHours = ticker.inherit(HAtickerExtHours, testSymbol)

//@variable The `close` price requested using "NASDAQ:MSFT" with inherited modifiers.
secData = request.security(testSymbolHAtickerExtHours, "60", close, ignore_invalid_symbol
//@variable The `close` price requested using "NASDAQ:MSFT" without modifiers.
compareData = request.security(testSymbol, "60", close, ignore_invalid_symbol = true)
```

```
plot(secData, color = color.green)
plot(compareData)
```

REMARKS

If the constructed ticker ID inherits a modifier that doesn't apply to the symbol (e.g., if the `from_tickerid` has Extended Hours enabled, but no such option is available for the `symbol`), the script will ignore the modifier when requesting data using the ID.

ticker.kagi()

Creates a ticker identifier for requesting Kagi values.

SYNTAX

```
ticker.kagi(symbol, reversal) → simple string
```

ARGUMENTS

symbol (simple string) Symbol ticker identifier.

reversal (simple int/float) Reversal amount (absolute price value).

EXAMPLE

```
//@version=5
indicator("ticker.kagi", overlay=true)
kagi_tickerid = ticker.kagi(syminfo.tickerid, 3)
kagi_close = request.security(kagi_tickerid, timeframe.period, close)
plot(kagi_close)
```

RETURNS

String value of ticker id, that can be supplied to request.security function.

SEE ALSO

syminfo.tickerid syminfo.ticker request.security ticker.heikinashi ticker.renko ticker.linebreak ticker.pointfigure

ticker.linebreak()

Creates a ticker identifier for requesting Line Break values.

SYNTAX

```
ticker.linebreak(symbol, number_of_lines) → simple string
```

ARGUMENTS

symbol (simple string) Symbol ticker identifier.

number_of_lines (simple int) Number of line.

EXAMPLE

```
//@version=5
indicator("ticker.linebreak", overlay=true)
linebreak_tickerid = ticker.linebreak(syminfo.tickerid, 3)
linebreak_close = request.security(linebreak_tickerid, timeframe.period, close)
plot(linebreak_close)
```

RETURNS

String value of ticker id, that can be supplied to request.security function.

SEE ALSO

syminfo.tickerid syminfo.ticker request.security ticker.heikinashi ticker.renko ticker.kagi ticker.pointfigure

ticker.modify()

Creates a ticker identifier for requesting additional data for the script.

SYNTAX

```
ticker.modify(tickerid, session, adjustment, backadjustment, settlement_as_close) → simple
string
```

ARGUMENTS

tickerid (simple string) Symbol name with exchange prefix, e.g. 'BATS:MSFT', 'NASDAQ:MSFT' or tickerid with session and adjustment from the ticker.new function.

session (simple string) Session type. Optional argument. Possible values: session.regular, session.extended. Session type of the current chart is syminfo.session. If session is not given, then syminfo.session value is used.

adjustment (simple string) Adjustment type. Optional argument. Possible values: adjustment.none, adjustment.splits, adjustment.dividends. If adjustment is not given, then default adjustment value is used (can be different depending on particular instrument).

backadjustment (simple backadjustment) Specifies whether past contract data on continuous futures symbols is back-adjusted. This setting only affects the data from symbols with this option available on their charts. Optional. The default is backadjustment.inherit, meaning that the modified ticker ID inherits the setting from the ticker ID passed to the `tickerid` parameter, or it inherits the symbol's default if the `tickerid` does not specify this setting. Possible values: backadjustment.inherit, backadjustment.on, backadjustment.off.

settlement_as_close (simple settlement) Specifies whether a futures symbol's close value represents the actual closing price or the settlement price on "1D" and higher timeframes. This setting only affects the data from symbols with this option available on their charts. Optional. The default is settlement_as_close.inherit, meaning that the modified ticker ID inherits the setting from the `tickerid` passed into the function, or it inherits the chart symbol's default if the `tickerid` does not specify this setting. Possible values: settlement_as_close.inherit, settlement_as_close.on, settlement_as_close.off.

EXAMPLE

```
//@version=5
indicator("ticker_modify", overlay=true)
t1 = ticker.new(syminfo.prefix, syminfo.ticker, session.regular, adjustment.splits)
c1 = request.security(t1, "D", close)
t2 = ticker.modify(t1, session.extended)
c2 = request.security(t2, "2D", close)
plot(c1)
plot(c2)
```

RETURNS

String value of ticker id, that can be supplied to request.security function.

SEE ALSO

syminfo.tickerid syminfo.ticker syminfo.session session.extended session.regular ticker.heikinashi adjustment.none adjustment.splits adjustment.dividends backadjustment.inherit backadjustment.on backadjustment.off settlement_as_close.inherit settlement_as_close.on settlement_as_close.off

ticker.new()

Creates a ticker identifier for requesting additional data for the script.

SYNTAX

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

ARGUMENTS

prefix (simple string) Exchange prefix. For example: 'BATS', 'NYSE', 'NASDAQ'. Exchange prefix of main series is syminfo.prefix.

ticker (simple string) Ticker name. For example 'AAPL', 'MSFT', 'EURUSD'. Ticker name of the main series is syminfo.ticker.

session (simple string) Session type. Optional argument. Possible values: session.regular, session.extended. Session type of the current chart is syminfo.session. If session is not given, then syminfo.session value is used.

adjustment (simple string) Adjustment type. Optional argument. Possible values: adjustment.none, adjustment.splits, adjustment.dividends. If adjustment is not given, then default adjustment value is used (can be different depending on particular instrument).

backadjustment (simple backadjustment) Specifies whether past contract data on continuous futures symbols is back-adjusted. This setting only affects the data from symbols with this option available on their charts. Optional. The default is backadjustment.inherit, meaning that the new ticker ID inherits the symbol's default setting. Possible values: backadjustment.inherit, backadjustment.on, backadjustment.off.

settlement_as_close (simple settlement) Specifies whether a futures symbol's close value represents the actual closing price or the settlement price on "1D" and higher timeframes. This setting only affects the data from symbols with this option available on their charts. Optional. The default is settlement_as_close.inherit, meaning that the new ticker ID inherits the chart symbol's default setting. Possible values: settlement_as_close.inherit, settlement_as_close.on, settlement_as_close.off.

EXAMPLE

```
//@version=5
indicator("ticker.new", overlay=true)
t = ticker.new(syminfo.prefix, syminfo.ticker, session.regular, adjustment.splits)
t2 = ticker.heikinashi(t)
c = request.security(t2, timeframe.period, low, barmerge.gaps_on)
plot(c, style=plot.style_linebr)
```

RETURNS

String value of ticker id, that can be supplied to request.security function.

REMARKS

You may use return value of ticker.new function as input argument for ticker.heikinashi, ticker.renko, ticker.linebreak, ticker.kagi, ticker.pointfigure functions.

SEE ALSO

syminfo.tickerid syminfo.ticker syminfo.session session.extended session.regular ticker.heikinashi adjustment.none adjustment.splits adjustment.dividends backadjustment.inherit backadjustment.on backadjustment.off settlement_as_close.inherit settlement_as_close.on settlement_as_close.off

ticker.pointfigure()

Creates a ticker identifier for requesting Point & Figure values.

SYNTAX

```
ticker.pointfigure(symbol, source, style, param, reversal) → simple string
```

ARGUMENTS

symbol (simple string) Symbol ticker identifier.

source (simple string) The source for calculating Point & Figure. Possible values are: 'hl', 'close'.

style (simple string) Box Size Assignment Method: 'ATR', 'Traditional'.

param (simple int/float) ATR Length if `style` is equal to 'ATR', or Box Size if `style` is equal to 'Traditional'.

reversal (simple int) Reversal amount.

EXAMPLE

```
//@version=5
indicator("ticker.pointfigure", overlay=true)
pnf_tickerid = ticker.pointfigure(syminfo.tickerid, "hl", "Traditional", 1, 3)
pnf_close = request.security(pnf_tickerid, timeframe.period, close)
plot(pnf_close)
```

RETURNS

String value of ticker id, that can be supplied to request.security function.

SEE ALSO

syminfo.tickerid syminfo.ticker request.security ticker.heikinashi ticker.renko ticker.linebreak ticker.kagi

ticker.renko()

Creates a ticker identifier for requesting Renko values.

SYNTAX

```
ticker.renko(symbol, style, param, request_wicks, source) → simple string
```

ARGUMENTS

symbol (simple string) Symbol ticker identifier.

style (simple string) Box Size Assignment Method: 'ATR', 'Traditional'.

param (simple int/float) ATR Length if `style` is equal to 'ATR', or Box Size if `style` is equal to 'Traditional'.

request_wicks (simple bool) Specifies if wick values are returned for Renko bricks. When true, high and low values requested from a symbol using the ticker formed by this function will include wick values when they are present. When false, high and low will always be equal to either open or close. Optional. The default is false. A detailed explanation of how Renko wicks are calculated can be found in our Help Center.

source (simple string) The source used to calculate bricks. Optional. Possible values: "Close", "OHLC". The default is "Close".

EXAMPLE

```
//@version=5
indicator("ticker.renko", overlay=true)
renko_tickerid = ticker.renko(syminfo.tickerid, "ATR", 10)
renko_close = request.security(renko_tickerid, timeframe.period, close)
plot(renko_close)
```

EXAMPLE

```
//@version=5
indicator("Renko candles", overlay=false)
renko_tickerid = ticker.renko(syminfo.tickerid, "ATR", 10)
[renko_open, renko_high, renko_low, renko_close] = request.security(renko_tickerid, timefr
plotcandle(renko_open, renko_high, renko_low, renko_close, color = renko_close > renko_ope
```

RETURNS

String value of ticker id, that can be supplied to request.security function.

SEE ALSO

syminfo.tickerid syminfo.ticker request.security ticker.heikinashi ticker.linebreak ticker.kagi ticker.pointfigure

ticker.standard()

Creates a ticker to request data from a standard chart that is unaffected by modifiers like extended session, dividend adjustment, currency conversion, and the calculations of non-standard chart types: Heikin Ashi, Renko, etc. Among other things, this makes it possible to retrieve standard chart values when the script is running on a non-standard chart.

SYNTAX

```
ticker.standard(symbol) → simple string
```

ARGUMENTS

symbol (simple string) A ticker ID to be converted into its standard form. Optional. The default is syminfo.tickerid.

EXAMPLE

```
//@version=5
indicator("ticker.standard", overlay = true)
// This script should be run on a non-standard chart such as HA, Renko...

// Requests data from the chart type the script is running on.
chartTypeValue = request.security(syminfo.tickerid, "1D", close)

// Request data from the standard chart type, regardless of the chart type the script is r
standardChartValue = request.security(ticker.standard(syminfo.tickerid), "1D", close)

// This will not use a standard ticker ID because the `symbol` argument contains only the
standardChartValue2 = request.security(ticker.standard(syminfo.ticker), "1D", close)
```

```
plot(chartTypeValue)
plot(standardChartValue, color = color.green)
```

RETURNS

A string representing the ticker of a standard chart in the "prefix:ticker" format. If the `symbol` argument does not contain the prefix and ticker information, the function returns the supplied argument as is.

SEE ALSO

request.security

time() 3 overloads

The time function returns the UNIX time of the current bar for the specified timeframe and session or NaN if the time point is out of session.

SYNTAX & OVERLOADS

```
time(timeframe, bars_back) → series int
```

```
time(timeframe, session, bars_back) → series int
```

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

ARGUMENTS

timeframe (series string) Timeframe. An empty string is interpreted as the current timeframe of the chart.

bars_back (series int) If specified, the function returns the timestamp from the bar N bars back relative to the current bar on the specified timeframe. Passing a negative number from -1 to -500 allows the function to request the expected time of a future bar. Optional. The default is 0.

EXAMPLE

```
//@version=5
indicator("Time", overlay=true)
// Try this on chart AAPL,1
timeinrange(res, sess) => not na(time(res, sess, "America/New_York")) ? 1 : 0
plot(timeinrange("1", "1300-1400"), color=color.red)
```

```
// This plots 1.0 at every start of 10 minute bar on a 1 minute chart:
newbar(res) => ta.change(time(res)) == 0 ? 0 : 1
plot(newbar("10"))
```

While setting up a session you can specify not just the hours and minutes but also the days of the week that will be included in that session.

If the days aren't specified, the session is considered to have been set from Sunday (1) to Saturday (7), i.e. "1100-2000" is the same as "1100-1200:1234567".

You can change that by specifying the days. For example, on a symbol that is traded seven days a week with the 24-hour trading session the following script will not color Saturdays and Sundays:

EXAMPLE

```
//@version=5
indicator("Time", overlay=true)
t1 = time(timeframe.period, "0000-0000:23456")
bgcolor(not na(t1) ? color.new(color.blue, 90) : na)
```

One `session` argument can include several different sessions, separated by commas. For example, the following script will highlight the bars from 10:00 to 11:00 and from 14:00 to 15:00 (workdays only):

EXAMPLE

```
//@version=5
indicator("Time", overlay=true)
t1 = time(timeframe.period, "1000-1100,1400-1500:23456")
bgcolor(not na(t1) ? color.new(color.blue, 90) : na)
```

RETURNS

UNIX time.

REMARKS

UNIX time is the number of milliseconds that have elapsed since 00:00:00 UTC, 1 January 1970.

SEE ALSO

`time`

time_close() 3 overloads

Returns the UNIX time of the current bar's close for the specified timeframe and session, or na if the time point is outside the session. On tick charts and price-based charts such as Renko, line break, Kagi, point & figure, and range, this function returns an na timestamp for the latest realtime bar (because the future closing time is unpredictable), but a valid timestamp for any previous bar.

SYNTAX & OVERLOADS

```
time_close(timeframe, bars_back) → series int
```

```
time_close(timeframe, session, bars_back) → series int
```

```
time_close(timeframe, session, timezone, bars_back) → series int
```

ARGUMENTS

timeframe (series string) Resolution. An empty string is interpreted as the current resolution of the chart.

bars_back (series int) If specified, the function returns the timestamp from the bar N bars back relative to the current bar on the specified timeframe. Passing a negative number from -1 to -500 allows the function to request the expected time of a future bar. Optional. The default is 0.

EXAMPLE

```
//@version=5
indicator("Time", overlay=true)
t1 = time_close(timeframe.period, "1200-1300", "America/New_York")
bgcolor(not na(t1) ? color.new(color.blue, 90) : na)
```

RETURNS

UNIX time.

REMARKS

UNIX time is the number of milliseconds that have elapsed since 00:00:00 UTC, 1 January 1970.

SEE ALSO

time_close

timeframe.change()

Detects changes in the specified `timeframe` .

SYNTAX

```
timeframe.change(timeframe) → series bool
```

ARGUMENTS

timeframe (series string) String formatted according to the User manual's timeframe string specifications.

EXAMPLE

```
//@version=5
// Run this script on an intraday chart.
indicator("New day started", overlay = true)
// Highlights the first bar of the new day.
isNewDay = timeframe.change("1D")
bgcolor(isNewDay ? color.new(color.green, 80) : na)
```

RETURNS

Returns true on the first bar of a new `timeframe` , false otherwise.

timeframe.from_seconds() 2 overloads

Converts a number of seconds into a valid timeframe string.

SYNTAX & OVERLOADS

```
timeframe.from_seconds(seconds) → simple string
```

```
timeframe.from_seconds(seconds) → series string
```

ARGUMENTS

seconds (simple int) The number of seconds in the timeframe.

EXAMPLE

```
//@version=5
indicator("HTF Close", "", true)
int chartTf = timeframe.in_seconds()
string tfTimes5 = timeframe.from_seconds(chartTf * 5)
float htfClose = request.security(syminfo.tickerid, tfTimes5, close)
plot(htfClose)
```

RETURNS

A timeframe string compliant with timeframe string specifications.

REMARKS

If no valid timeframe exists for the quantity of seconds supplied, the next higher valid timeframe will be returned. Accordingly, one second or less will return "1S", 2-5 seconds will return "5S", and 604,799 seconds (one second less than 7 days) will return "7D".

If the seconds exactly represent two or more valid timeframes, the one with the larger base unit will be used. Thus 604,800 seconds (7 days) returns "1W", not "7D".

All values above 31,622,400 (366 days) return "12M".

SEE ALSO

timeframe.in_seconds   request.security   request.security_lower_tf

timeframe.in_seconds() 2 overloads

Converts a timeframe string into seconds.

SYNTAX & OVERLOADS

```
timeframe.in_seconds(timeframe) → simple int
```

```
timeframe.in_seconds(timeframe) → series int
```

ARGUMENTS

timeframe (simple string) Timeframe string in timeframe string specifications format. Optional. The default is timeframe.period.

EXAMPLE

```
//@version=5
indicator("`timeframe_in_seconds()`")
```

```
// Get a user-selected timeframe.
tfInput = input.timeframe("1D")

// Convert it into an "int" number of seconds.
secondsInTf = timeframe.in_seconds(tfInput)

plot(secondsInTf)
```

RETURNS

The "int" representation of the number of seconds in the timeframe string.

REMARKS

When the timeframe is "1M" or more, calculations use 2628003 as the number of seconds in one month, which represents 30.4167 (365/12) days.

SEE ALSO

input.timeframe timeframe.period timeframe.from_seconds

timestamp() 5 overloads

Function timestamp returns UNIX time of specified date and time.

SYNTAX & OVERLOADS

```
timestamp(dateString) → const int
```

```
timestamp(year, month, day, hour, minute, second) → simple int
```

```
timestamp(year, month, day, hour, minute, second) → series int
```

```
timestamp(timezone, year, month, day, hour, minute, second) → simple int
```

```
timestamp(timezone, year, month, day, hour, minute, second) → series int
```

ARGUMENTS

dateString (const string) A string containing the date and, optionally, the time and time zone. Its format must comply with either the IETF RFC 2822 or ISO 8601 standards ("DD MMM YYYY hh:mm:ss ±hhmm" or "YYYY-MM-DDThh:mm:ss±hh:mm", so "20 Feb 2020" or "2020-02-20"). If no time is supplied, "00:00" is used. If no time zone is supplied, GMT+0 will be used. Note that this diverges from the usual behavior of the function where it returns time in the exchange's timezone.

EXAMPLE

```
//@version=5
indicator("timestamp")
plot(timestamp(2016, 01, 19, 09, 30), linewidth=3, color=color.green)
plot(timestamp(syminfo.timezone, 2016, 01, 19, 09, 30), color=color.blue)
plot(timestamp(2016, 01, 19, 09, 30), color=color.yellow)
plot(timestamp("GMT+6", 2016, 01, 19, 09, 30))
plot(timestamp(2019, 06, 19, 09, 30, 15), color=color.lime)
plot(timestamp("GMT+3", 2019, 06, 19, 09, 30, 15), color=color.fuchsia)
plot(timestamp("Feb 01 2020 22:10:05"))
plot(timestamp("2011-10-10T14:48:00"))
plot(timestamp("04 Dec 1995 00:12:00 GMT+5"))
```

RETURNS

UNIX time.

REMARKS

UNIX time is the number of milliseconds that have elapsed since 00:00:00 UTC, 1 January 1970.

SEE ALSO

time

weekofyear() 2 overloads

SYNTAX & OVERLOADS

```
weekofyear(time) → series int
```

```
weekofyear(time, timezone) → series int
```

ARGUMENTS

time (series int) UNIX time in milliseconds.

RETURNS

Week of year (in exchange timezone) for provided UNIX time.

REMARKS

UNIX time is the number of milliseconds that have elapsed since 00:00:00 UTC, 1 January 1970.

Note that this function returns the week based on the time of the bar's open. For overnight sessions (e.g. EURUSD, where Monday session starts on Sunday, 17:00) this value can be lower by 1 than the week of the trading day.

SEE ALSO

weekofyear time year month dayofmonth dayofweek hour minute second

year() 2 overloads

SYNTAX & OVERLOADS

```
year(time) → series int
```

```
year(time, timezone) → series int
```

ARGUMENTS

time (series int) UNIX time in milliseconds.

RETURNS

Year (in exchange timezone) for provided UNIX time.

REMARKS

UNIX time is the number of milliseconds that have elapsed since 00:00:00 UTC, 1 January 1970.

Note that this function returns the year based on the time of the bar's open. For overnight sessions (e.g. EURUSD, where Monday session starts on Sunday, 17:00 UTC-4) this value can be lower by 1 than the year of the trading day.

SEE ALSO

year time month dayofmonth dayofweek hour minute second

Keywords

and

Logical AND. Applicable to boolean expressions.

SYNTAX

```
expr1 and expr2
```

RETURNS

Boolean value, or series of boolean values.

enum

This keyword allows the creation of an enumeration, enum for short. Enums are unique constructs that hold groups of predefined constants.

Each field in an enum has a `const string` title. Scripts can access the fields in an enum using dot notation, similar to accessing the fields of a user-defined type.

Each field represents a value of the `enumName` enum. Scripts can declare each field in an `enum` with an optional `const string` title. If a field's title is not specified, its title is the string representation of its name. Use str.tostring on an enum field to retrieve its title.

SYNTAX

```
[export ]enum <enumName>
<field_1> [= <title_1>]
<field_2> [= <title_2>]
...
<field_N> [= <title_N>]
```

One can use an enum to quickly create a dropdown input with the help of the input.enum function. The options that appear in the dropdown represent the titles of the enum fields.

EXAMPLE

```
//@version=5
indicator("Session highlight", overlay = true)

//@enum       Contains fields with popular timezones as titles.
//@field exch  Has an empty string as the title to represent the chart timezone.
enum tz
    utc  = "UTC"
    exch = ""
    ny   = "America/New_York"
    chi  = "America/Chicago"
    lon  = "Europe/London"
    tok  = "Asia/Tokyo"
```

```
//@variable The session string.
selectedSession = input.session("1200-1500", "Session")
//@variable The selected timezone. The input's dropdown contains the fields in the `tz` en
selectedTimezone = input.enum(tz.utc, "Session Timezone")

//@variable Is `true` if the current bar's time is in the specified session.
bool inSession = false
if not na(time("", selectedSession, str.tostring(selectedTimezone)))
    inSession := true

// Highlight the background when `inSession` is `true`.
bgcolor(inSession ? color.new(color.green, 90) : na, title = "Active session highlight")
```

Additionally, one can use an enum in a collection's type template to restrict the values it will allow as elements. When used inside a type template, the collection will only accept fields that belong to the specified enum.

EXAMPLE

```
//@version=5
indicator("Map with enum keys")

//@enum        Contains fields with titles representing ticker IDs.
//@field aapl  Has an Apple ticker ID as its title.
//@field tsla  Has a Tesla ticker ID as its title.
//@field amzn  Has an Amazon ticker ID as its title.
enum symbols
    aapl = "NASDAQ:AAPL"
    tsla = "NASDAQ:TSLA"
    amzn = "NASDAQ:AMZN"

//@variable A map that accepts fields from the `symbols` enum as keys and "float" values.
map<symbols, float> data = map.new<symbols, float>()
// Put key-value pairs into the `data` map.
data.put(symbols.aapl, request.security(str.tostring(symbols.aapl), timeframe.period, clos
data.put(symbols.tsla, request.security(str.tostring(symbols.tsla), timeframe.period, clos
data.put(symbols.amzn, request.security(str.tostring(symbols.amzn), timeframe.period, clos
// Plot the value from the `data` map accessed by the `symbols.aapl` key.
plot(data.get(symbols.aapl))
```

export

Used in libraries to prefix the declaration of functions or user-defined type definitions that will be available from other scripts importing the library.

EXAMPLE

```
//@version=5
//@description Library of debugging functions.
library("Debugging_library", overlay = true)
//@function Displays a string as a table cell for debugging purposes.
//@param txt String to display.
//@returns Void.
export print(string txt) =>
    var table t = table.new(position.middle_right, 1, 1)
    table.cell(t, 0, 0, txt, bgcolor = color.yellow)
// Using the function from inside the library to show an example on the published chart.
// This has no impact on scripts using the library.
print("Library Test")
```

REMARKS

Each library must have at least one exported function or user-defined type (UDT).

Exported functions cannot use variables from the global scope if they are arrays, mutable variables (reassigned with `:=`), or variables of 'input' form.

Exported functions cannot use `request.*()` functions.

Exported functions must explicitly declare each parameter's type and all parameters must be used in the function's body. By default, all arguments passed to exported functions are of the series form, unless they are explicitly specified as simple in the function's signature.

The @description, @function, @param, @type, @field, and @returns compiler annotations are used to automatically generate the library's description and release notes, and in the Pine Script® Editor's tooltips.

SEE ALSO

library import simple series type

for

The 'for' structure allows the repeated execution of a number of statements:

SYNTAX

```
[var_declaration =] for counter = from_num to to_num [by step_num]
    statements | continue | break
    return_expression
```

var_declaration - An optional variable declaration that will be assigned the value of the loop's return_expression.

counter - A variable holding the value of the loop's counter, which is incremented/decremented by 1 or by the step_num value on each iteration of the loop.

from_num - The starting value of the counter. "series int/float" values/expressions are allowed.

to_num - The end value of the counter. When the counter becomes greater than to_num (or less than to_num in cases where from_num > to_num) the loop is broken. "series int/float" values/expressions are allowed, but they are evaluated only on the loop's first iteration.

step_num - The increment/decrement value of the counter. It is optional. The default value is +1 or -1, depending on which of from_num or to_num is the greatest. When a value is used, the counter is also incremented/decremented depending on which of from_num or to_num is the greatest, so the +/- sign of step_num is optional.

statements | continue | break - Any number of statements, or the 'continue' or 'break' keywords, indented by 4 spaces or a tab.

return_expression - The loop's return value which is assigned to the variable in var_declaration if one is present. If the loop exits because of a 'continue' or 'break' keyword, the loop's return value is that of the last variable assigned a value before the loop's exit.

continue - A keyword that can only be used in loops. It causes the next iteration of the loop to be executed.

break - A keyword that exits the loop.

EXAMPLE

```
//@version=5
indicator("for")
// Here, we count the quantity of bars in a given 'lookback' length which closed above the
qtyOfHigherCloses(lookback) =>
    int result = 0
    for i = 1 to lookback
        if close[i] > close
            result += 1
    result
plot(qtyOfHigherCloses(14))
```

EXAMPLE

```
//@version=5
indicator("`for` loop with a step")
```

```
a = array.from(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
sum = 0.0

for i = 0 to 9 by 5
    // Because the step is set to 5, we are adding only the first (0) and the sixth (5) va
    sum += array.get(a, i)

plot(sum)
```

SEE ALSO

for...in while

for...in

The `for...in` structure allows the repeated execution of a number of statements for each element in an array. It can be used with either one argument: `array_element`, or with two: `[index, array_element]`. The second form doesn't affect the functionality of the loop. It tracks the current iteration's index in the tuple's first variable.

SYNTAX

```
[var_declaration =] for array_element in array_id
    statements | continue | break
    return_expression

[var_declaration =] for [index, array_element] in array_id
    statements | continue | break
    return_expression
```

var_declaration - An optional variable declaration that will be assigned the value of the loop's `return_expression`.

index - An optional variable that tracks the current iteration's index. Indexing starts at 0. The variable is immutable in the loop's body. When used, it must be included in a tuple also containing `array_element`.

array_element - A variable containing each successive array element to be processed in the loop. The variable is immutable in the loop's body.

array_id - The ID of the array over which the loop is iterated.

statements | continue | break - Any number of statements, or the 'continue' or 'break' keywords, indented by 4 spaces or a tab.

return_expression - The loop's return value assigned to the variable in `var_declaration`, if one is present. If the loop exits because of a 'continue' or 'break' keyword, the loop's return value is that of the last variable assigned a value before the loop's exit.

continue - A keyword that can only be used in loops. It causes the next iteration of the loop to be executed.

break - A keyword that exits the loop.

Scripts can modify arrays and matrices while iterating over their elements with this structure. However, maps cannot change while looping through their key-value pairs. To modify a map within a `for...in` loop, iterate over the key-value pairs of a copy or over the elements in its map.keys array.

Here, we use the single-argument form of `for...in` to determine on each bar how many of the bar's OHLC values are greater than the SMA of 'close' values:

EXAMPLE

```
//@version=5
indicator("for...in")
// Here we determine on each bar how many of the bar's OHLC values are greater than the SM
float[] ohlcValues = array.from(open, high, low, close)
qtyGreaterThan(value, array) =>
    int result = 0
    for currentElement in array
        if currentElement > value
            result += 1
        result
plot(qtyGreaterThan(ta.sma(close, 20), ohlcValues))
```

Here, we use the two-argument form of for...in to set the values of our `isPos` array to `true` when their corresponding value in our `valuesArray` array is positive:

EXAMPLE

```
//@version=5
indicator("for...in")
var valuesArray = array.from(4, -8, 11, 78, -16, 34, 7, 99, 0, 55)
var isPos = array.new_bool(10, false)

for [index, value] in valuesArray
    if value > 0
        array.set(isPos, index, true)

if barstate.islastconfirmedhistory
    label.new(bar_index, high, str.tostring(isPos))
```

Iterate through matrix rows as arrays.

EXAMPLE

```
//@version=5
indicator("`for ... in` matrix Example")

// Create a 2x3 matrix with values `4`.
matrix1 = matrix.new<int>(2, 3, 4)

sum = 0.0
// Loop through every row of the matrix.
for rowArray in matrix1
    // Sum values of the every row
    sum += array.sum(rowArray)

plot(sum)
```

SEE ALSO

for while array.sum array.min array.max

if

If statement defines what block of statements must be executed when conditions of the expression are satisfied.

To have access to and use the if statement, one should specify the version >= 2 of Pine Script® language in the very first line of code, for example: //@version=5

The 4th version of Pine Script® Language allows you to use "else if" syntax.

General code form:

SYNTAX

```
var_declarationX = if condition
    var_decl_then0
    var_decl_then1
    …
    var_decl_thenN
else if [optional block]
    var_decl_else0
    var_decl_else1
    …
    var_decl_elseN
else
    var_decl_else0
    var_decl_else1
    …
    var_decl_elseN
    return_expression_else
```

where

var_declarationX — this variable gets the value of the if statement

condition — if the condition is true, the logic from the block 'then' (var_decl_then0, var_decl_then1, etc.) is used.

If the condition is false, the logic from the block 'else' (var_decl_else0, var_decl_else1, etc.) is used.

return_expression_then, **return_expression_else** — the last expression from the block then or from the block else will return the final value of the statement. If declaration of the variable is in the end, its value will be the result.

The type of returning value of the if statement depends on return_expression_then and return_expression_else type (their types must match: it is not possible to return an integer value from then, while you have a string value in else block).

EXAMPLE

```
//@version=5
indicator("if")
// This code compiles
x = if close > open
    close
else
    open

// This code doesn’t compile
// y = if close > open
//     close
// else
//     "open"
plot(x)
```

It is possible to omit the `else` block. In this case if the condition is false, an “empty” value (na, false, or “”) will be assigned to the var_declarationX variable:

EXAMPLE

```
//@version=5
indicator("if")
x = if close > open
    close
// If current close > current open, then x = close.
// Otherwise the x = na.
plot(x)
```

It is possible to use either multiple “else if” blocks or none at all. The blocks “then”, “else if”, “else” are shifted by four spaces:

EXAMPLE

```
//@version=5
indicator("if")
x = if open > close
    5
else if high > low
    close
else
    open
plot(x)
```

It is possible to ignore the resulting value of an `if` statement (“var_declarationX=“ can be omitted). It may be useful if you need the side effect of the expression, for example in strategy trading:

EXAMPLE

```
//@version=5
strategy("if")
if (ta.crossover(high, low))
    strategy.entry("BBandLE", strategy.long, stop=low, oca_name="BollingerBands", oca_type
else
    strategy.cancel(id="BBandLE")
```

If statements can include each other:

EXAMPLE

```
//@version=5
indicator("if")
float x = na
if close > open
    if close > close[1]
        x := close
    else
        x := close[1]
else
    x := open
plot(x)
```

import

Used to load an external library into a script and bind its functions to a namespace. The importing script can be an indicator, a strategy, or another library. A library must be published (privately or publicly) before it can be imported.

SYNTAX

```
import {username}/{libraryName}/{libraryVersion} as {alias}
```

ARGUMENTS

username (literal string) User name of the library's author.

libraryName (literal string) Name of the imported library, which corresponds to the `title` argument used by the author in his library script.

libraryVersion (literal int) Version number of the imported library.

alias (literal string) A non-numeric identifier used as a namespace to refer to the library's functions. Optional. The default is the `libraryName` string.

EXAMPLE

```
//@version=5
indicator("num_methods import")
// Import the first version of the username’s "num_methods" library and assign it to the '
import username/num_methods/1 as m
// Call the “sinh()” function from the imported library
y = m.sinh(3.14)
// Plot value returned by the "sinh()" function",
plot(y)
```

REMARKS

Using an alias that replaces a built-in namespace such as math.* or strategy.* is allowed, but if the library contains function names that shadow Pine Script®'s built-in functions, the built-ins will become unavailable. The same version of a library can only be imported once. Aliases must be distinct for each imported library. When calling library functions, casting their arguments to types other than their declared type is not allowed. An import statement cannot use 'as' or 'import' as `username`, `libraryName`, or `alias` identifiers.

SEE ALSO

library export

method

This keyword is used to prefix a function declaration, indicating it can then be invoked using dot notation by appending its name to a variable of the type of its first parameter and omitting that first parameter. Alternatively, functions declared as methods can also be invoked like normal user-defined functions. In that case, an argument must be supplied for its first parameter.

The first parameter of a method declaration must be explicitly typified.

SYNTAX

```
[export] method <functionName>(<paramType> <paramName> [= <defaultValue>], …) =>
    <functionBlock>
```

EXAMPLE

```
//@version=5
indicator("")

var prices = array.new<float>()

//@function Pushes a new value into the array and removes the first one if the resulting a
method maintainArray(array<float> id, maxSize, value) =>
    id.push(value)
    if id.size() > maxSize
        id.shift()

prices.maintainArray(50, close)
// The method can also be called like a function, without using dot notation.
// In this case an argument must be supplied for its first parameter.
// maintainArray(prices, 50, close)

// This calls the `array.avg()` built-in using dot notation with the `prices` array.
// It is possible because built-in functions belonging to some namespaces that are a speci
// can be invoked with method notation when the function's first parameter is an ID of tha
// Those namespaces are: `array`, `matrix`, `line`, `linefill`, `label`, `box`, and `table
plot(prices.avg())
```

not

Logical negation (NOT). Applicable to boolean expressions.

SYNTAX

```
not expr1
```

RETURNS

Boolean value, or series of boolean values.

or

Logical OR. Applicable to boolean expressions.

SYNTAX

```
expr1 or expr2
```

RETURNS

Boolean value, or series of boolean values.

switch

The switch operator transfers control to one of the several statements, depending on the values of a condition and expressions.

SYNTAX

```
[variable_declaration = ] switch expression
    value1 => local_block
    value2 => local_block
    …
    => default_local_block

[variable_declaration = ] switch
    condition1 => local_block
    condition2 => local_block
    …
    => default_local_block
```

Switch with an expression:

EXAMPLE

```
//@version=5
```

```
indicator("Switch using an expression")

string i_maType = input.string("EMA", "MA type", options = ["EMA", "SMA", "RMA", "WMA"])

float ma = switch i_maType
    "EMA" => ta.ema(close, 10)
    "SMA" => ta.sma(close, 10)
    "RMA" => ta.rma(close, 10)
    // Default used when the three first cases do not match.
    => ta.wma(close, 10)

plot(ma)
```

Switch without an expression:

EXAMPLE

```
//@version=5
strategy("Switch without an expression", overlay = true)

bool longCondition  = ta.crossover( ta.sma(close, 14), ta.sma(close, 28))
bool shortCondition = ta.crossunder(ta.sma(close, 14), ta.sma(close, 28))

switch
    longCondition  => strategy.entry("Long ID", strategy.long)
    shortCondition => strategy.entry("Short ID", strategy.short)
```

RETURNS

The value of the last expression in the local block of statements that is executed.

REMARKS

Only one of the `local_block` instances or the `default_local_block` can be executed. The `default_local_block` is introduced with the `=>` token alone and is only executed when none of the preceding blocks are executed. If the result of the `switch` statement is assigned to a variable and a `default_local_block` is not specified, the statement returns `na` if no `local_block` is executed. When assigning the result of the `switch` statement to a variable, all `local_block` instances must return the same type of value.

SEE ALSO

if ?:

type

This keyword allows the declaration of user-defined types (UDT) from which scripts can instantiate objects. UDTs are composite types that contain an arbitrary number of fields of any built-in or user-defined type, including the defined UDT itself. The syntax to define a UDT is:

SYNTAX

```
[export ]type <UDT_identifier>
    [varip ]<field_type> <field_name> [= <value>]
    …
```

Once a UDT is defined, scripts can instantiate objects from it with the `UDT_identifier.new()` construct. When creating a new type instance, the fields of the resulting object will initialize with the default values from the UDT's definition. Any type fields without specified defaults will initialize as na. Alternatively, users can pass initial values as arguments in the `*.new()` method to override the type's defaults. For example, `newFooObject = foo.new(x = true)` assigns a new `foo` object to the `newFooObject` variable with its `x` field initialized using a value of true.

Field declarations can include the varip keyword, in which case the field values persist between successive script iterations on the same bar.

For more information see the User Manual's sections on defining UDTs and using objects.

Libraries can export UDTs. See theLibraries page of our User Manual to learn more.

EXAMPLE

```
//@version=5
indicator("Multi Time Period Chart", overlay = true)

timeframeInput = input.timeframe("1D")

type bar
    float o = open
    float h = high
    float l = low
    float c = close
    int   t = time

drawBox(bar b, right) =>
    bar s = bar.new()
    color boxColor = b.c >= b.o ? color.green : color.red
    box.new(b.t, b.h, right, b.l, boxColor, xloc = xloc.bar_time, bgcolor = color.new(box

updateBox(box boxId, bar b) =>
    color boxColor = b.c >= b.o ? color.green : color.red
    box.set_border_color(boxId, boxColor)
    box.set_bgcolor(boxId, color.new(boxColor, 90))
    box.set_top(boxId, b.h)
    box.set_bottom(boxId, b.l)
    box.set_right(boxId, time)

secBar = request.security(syminfo.tickerid, timeframeInput, bar.new())

if not na(secBar)
    // To avoid a runtime error, only process data when an object exists.
    if not barstate.islast
        if timeframe.change(timeframeInput)
            // On historical bars, draw a new box in the past when the HTF closes.
            drawBox(secBar, time[1])
    else
        var box lastBox = na
        if na(lastBox) or timeframe.change(timeframeInput)
            // On the last bar, only draw a new current box the first time we get there or
            lastBox := drawBox(secBar, time)
        else
            // On other chart updates, use setters to modify the current box.
            updateBox(lastBox, secBar)
```

var

var is the keyword used for assigning and one-time initializing of the variable.

Normally, a syntax of assignment of variables, which doesn't include the keyword var, results in the value of the variable being overwritten with every update of the data. Contrary to that, when assigning variables with the keyword var, they can "keep the state" despite the data updating, only changing it when conditions within if-expressions are met.

SYNTAX

```
var variable_name = expression
```

where:

variable_name - any name of the user's variable that's allowed in Pine Script® (can contain capital and lowercase Latin characters, numbers, and underscores (_), but can't start with a number).

expression - any arithmetic expression, just as with defining a regular variable. The expression will be calculated and assigned to a variable once.

EXAMPLE

```
//@version=5
indicator("Var keyword example")
var a = close
var b = 0.0
var c = 0.0
var green_bars_count = 0
if close > open
    var x = close
    b := x
    green_bars_count := green_bars_count + 1
    if green_bars_count >= 10
        var y = close
        c := y
plot(a)
plot(b)
plot(c)
```

The variable 'a' keeps the closing price of the first bar for each bar in the series.

The variable 'b' keeps the closing price of the first "green" bar in the series.

The variable 'c' keeps the closing price of the tenth "green" bar in the series.

varip

varip (var intrabar persist) is the keyword used for the assignment and one-time initialization of a variable or a field of a user-defined type. It's similar to the var keyword, but variables and fields declared with varip retain their values between executions of the script on the same bar.

SYNTAX

```
varip [<variable_type> ]<variable_name> = <expression>

[export ]type <UDT_identifier>
    varip <field_type> <field_name> [= <value>]
```

where:

variable_type - An optional fundamental type (int, float, bool, color, string) or a user-defined type, or an array or matrix of one of those types. Special types are not compatible with this keyword.

variable_name - A valid identifier. The variable can also be an object created from a UDT.

expression - Any arithmetic expression, just as when defining a regular variable. The expression will be calculated and assigned to the variable only once, on the first bar.

UDT_identifier, field_type, field_name, value - Constructs related to user-defined types as described in the type section.

EXAMPLE

```
//@version=5
indicator("varip")
varip int v = -1
v := v + 1
plot(v)
```

With var, `v` would equal the value of the bar_index. On historical bars, where the script calculates only once per chart bar, the value of `v` is the same as with var. However, on realtime bars, the script will evaluate the expression on each new chart update, producing a different result.

EXAMPLE

```
//@version=5
indicator("varip with types")
type barData
    int index = -1
    varip int ticks = -1

var currBar = barData.new()
currBar.index += 1
currBar.ticks += 1

// Will be equal to bar_index on all bars
plot(currBar.index)
// In real time, will increment per every tick on the chart
plot(currBar.ticks)
```

The same += operation applied to both the `index` and `ticks` fields results in different real-time values because `ticks` increases on every chart update, while `index` only does so once per bar. Note how the `currBar` object does not use the varip keyword. The `ticks` field of the object can increment on every tick, but the reference itself is defined once and then stays unchanged. If we were to declare `currBar` using varip, the behavior of `index` would remain unchanged because while the reference to the type instance would persist between chart updates, the `index` field of the object would not.

REMARKS

When using varip to declare variables in strategies that may execute more than once per historical chart bar, the values of such variables are preserved across successive iterations of the script on the same bar.

The effect of varip eliminates the rollback of variables before each successive execution of a script on the same bar.

while

The `while` statement allows the conditional iteration of a local code block.

SYNTAX

```
variable_declaration = while condition
    …
    continue
    …
    break
    …
    return_expression
```

where:

variable_declaration - An optional variable declaration. The `return expression` can provide the initialization value for this variable.

condition - when true, the local block of the `while` statement is executed. When false, execution of the script resumes after the `while` statement.

continue - The `continue` keyword causes the loop to branch to its next iteration.

break - The `break` keyword causes the loop to terminate. The script's execution resumes after the `while` statement.

return_expression - An optional line providing the `while` statement's returning value.

EXAMPLE

```
//@version=5
indicator("while")
// This is a simple example of calculating a factorial using a while loop.
int i_n = input.int(10, "Factorial Size", minval=0)
int counter   = i_n
int factorial = 1
while counter > 0
    factorial := factorial * counter
    counter   := counter - 1

plot(factorial)
```

REMARKS

The local code block after the initial `while` line must be indented with four spaces or a tab. For the `while` loop to terminate, the boolean expression following `while` must eventually become false, or a `break` must be executed.

Types

array

Keyword used to explicitly declare the "array" type of a variable or a parameter. Array objects (or IDs) can be created with the array.new<type>, array.from function.

EXAMPLE

```
//@version=5
indicator("array", overlay=true)
array<float> a = na
a := array.new<float>(1, close)
plot(array.get(a, 0))
```

REMARKS

Array objects are always of "series" form.

SEE ALSO

var line label table box array.new<type> array.from

bool

Keyword used to explicitly declare the "bool" (boolean) type of a variable or a parameter. "Bool" variables can have values true, false or na.

EXAMPLE

```
//@version=5
indicator("bool")
bool b = true    // Same as `b = true`
b := na
plot(b ? open : close)
```

REMARKS

Explicitly mentioning the type in a variable declaration is optional, except when it is initialized with na. Learn more about Pine Script® types in the User Manual page on the Type System.

SEE ALSO

var varip int float color string true false

box

Keyword used to explicitly declare the "box" type of a variable or a parameter. Box objects (or IDs) can be created with the box.new function.

EXAMPLE

```
//@version=5
indicator("box")
// Empty `box1` box ID.
var box box1 = na
// `box` type is unnecessary because `box.new()` returns a "box" type.
var box2 = box.new(na, na, na, na)
box3 = box.new(time, open, time + 60 * 60 * 24, close, xloc=xloc.bar_time)
```

REMARKS

Box objects are always of "series" form.

SEE ALSO

var line label table box.new

chart.point

Keyword to explicitly declare the type of a variable or parameter as `chart.point`. Scripts can produce `chart.point` instances using the chart.point.from_time, chart.point.from_index, chart.point.now, and chart.point.new functions.

FIELDS

index (series int) The x-coordinate of the point, expressed as a bar index value.

time (series float) The x-coordinate of the point, expressed as a UNIX time value, in milliseconds.

price (series float) The y-coordinate of the point.

SEE ALSO

polyline

color

Keyword used to explicitly declare the "color" type of a variable or a parameter.

EXAMPLE

```
//@version=5
indicator("color", overlay = true)

color textColor = color.green
color labelColor = #FF000080  // Red color (FF0000) with 50% transparency (80 which is hal
if barstate.islastconfirmedhistory
    label.new(bar_index, high, text = "Label", color = labelColor, textcolor = textColor)

// When declaring variables with color literals, built-in constants(color.green) or functi
c = color.rgb(0,255,0,0)
plot(close, color = c)
```

REMARKS

Color literals have the following format: #RRGGBB or #RRGGBBAA. The letter pairs represent 00 to FF hexadecimal values (0 to 255 in decimal) where RR, GG and BB pairs are the values for the color's red, green and blue components. AA is an optional value for the color's transparency (or alpha component) where 00 is invisible and FF opaque. When no AA pair is supplied, FF is used. The hexadecimal letters can be upper or lower case.

Explicitly mentioning the type in a variable declaration is optional, except when it is initialized with na. Learn more about Pine Script® types in the User Manual page on the Type System.

SEE ALSO

var varip int float string color.rgb color.new

const

The `const` keyword explicitly assigns the "const" type qualifier to variables and the parameters of non-exported functions. Variables and parameters with the "const" qualifier reference values established at compile time that never change in the script's execution.

In variable declarations, the compiler can usually infer the qualified type automatically based on the values assigned to a variable, and it can automatically change a variable's qualifier to a stronger one when necessary. The type qualifier hierarchy is "const" < "input" < "simple" < "series", where "const" is the weakest.

Explicitly declaring a variable with the `const` keyword restricts the type qualifier to "const", meaning the variable cannot accept a value with a stronger qualifier (e.g., "input"), nor can the value assigned to the variable change at any point in the script's execution.

When using this keyword to specify the type qualifier, one must also use a type keyword to declare the allowed type.

SYNTAX

```
[method ]<functionName>([const <paramType> ]<paramName>[ = <defaultValue>])

[var/varip ]const <variableType> <variableName> = <variableValue>
```

EXAMPLE

```
//@version=5
indicator("custom plot title")

//@function Concatenates two "const string" values.
concatStrings(const string x, const string y) =>
    const string result = x + y

//@variable The title of the plot.
const string myTitle = concatStrings("My ", "Plot")

plot(close, myTitle)
```

EXAMPLE

```
//@version=5
indicator("can't assign input to const")

//@variable A variable declared as "const float" that attempts to assign the result of `ir
//          This declaration causes an error. The "input float" qualified type is stronger
const float myVar = input.float(2.0)

plot(myVar)
```

REMARKS

To learn more, see our User Manual's section on type qualifiers.

SEE ALSO

simple series

float

Keyword used to explicitly declare the "float" (floating point) type of a variable or a parameter.

EXAMPLE

```
//@version=5
indicator("float")
float f = 3.14    // Same as `f = 3.14`
f := na
plot(f)
```

REMARKS

Explicitly mentioning the type in a variable declaration is optional, except when it is initialized with na. Learn more about Pine Script® types in the User Manual page on the Type System.

SEE ALSO

var varip int bool color string

int

Keyword used to explicitly declare the "int" (integer) type of a variable or a parameter.

EXAMPLE

```
//@version=5
indicator("int")
int i = 14    // Same as `i = 14`
i := na
plot(i)
```

REMARKS

Explicitly mentioning the type in a variable declaration is optional, except when it is initialized with na. Learn more about Pine Script® types in the User Manual page on the Type System.

SEE ALSO

var varip float bool color string

label

Keyword used to explicitly declare the "label" type of a variable or a parameter. Label objects (or IDs) can be created with the label.new function.

EXAMPLE

```
//@version=5
indicator("label")
// Empty `label1` label ID.
var label label1 = na
// `label` type is unnecessary because `label.new()` returns "label" type.
var label2 = label.new(na, na, na)
if barstate.islastconfirmedhistory
    label3 = label.new(bar_index, high, text = "label3 text")
```

REMARKS

Label objects are always of "series" form.

SEE ALSO

var line box label.new

line

Keyword used to explicitly declare the "line" type of a variable or a parameter. Line objects (or IDs) can be created with the line.new function.

EXAMPLE

```
//@version=5
indicator("line")
// Empty `line1` line ID.
var line line1 = na
// `line` type is unnecessary because `line.new()` returns "line" type.
var line2 = line.new(na, na, na, na)
line3 = line.new(bar_index - 1, high, bar_index, high, extend = extend.right)
```

REMARKS

Line objects are always of "series" form.

SEE ALSO

var label box line.new

linefill

Keyword used to explicitly declare the "linefill" type of a variable or a parameter. Linefill objects (or IDs) can be created with the linefill.new function.

EXAMPLE

```
//@version=5
indicator("linefill", overlay=true)
// Empty `linefill1` line ID.
var linefill linefill1 = na
// `linefill` type is unnecessary because `linefill.new()` returns "linefill" type.
var linefill2 = linefill.new(na, na, na)

if barstate.islastconfirmedhistory
    line1 = line.new(bar_index - 10, high+1, bar_index, high+1, extend = extend.right)
    line2 = line.new(bar_index - 10, low+1, bar_index, low+1, extend = extend.right)
    linefill3 = linefill.new(line1, line2, color = color.new(color.green, 80))
```

REMARKS

Linefill objects are always of "series" form.

SEE ALSO

var line label table box linefill.new

map

Keyword used to explicitly declare the "map" type of a variable or a parameter. Map objects (or IDs) can be created with the map.new<type,type> function.

EXAMPLE

```
//@version=5
indicator("map", overlay=true)
map<int, float> a = na
a := map.new<int, float>()
a.put(bar_index, close)
label.new(bar_index, a.get(bar_index), "Current close")
```

REMARKS

Map objects are always of series form.

SEE ALSO

map.new<type,type>

matrix

Keyword used to explicitly declare the "matrix" type of a variable or a parameter. Matrix objects (or IDs) can be created with the matrix.new<type> function.

EXAMPLE

```
//@version=5
indicator("matrix example")

// Create `m1` matrix of `int` type.
matrix<int> m1 = matrix.new<int>(2, 3, 0)

// `matrix<int>` is unnecessary because the `matrix.new<int>()` function returns an `int`
m2 = matrix.new<int>(2, 3, 0)

// Display matrix using a label.
if barstate.islastconfirmedhistory
    label.new(bar_index, high, str.tostring(m2))
```

REMARKS

Matrix objects are always of "series" form.

SEE ALSO

var matrix.new<type> array

polyline

Keyword to explicitly declare the type of a variable or parameter as `polyline`. Scripts can produce `polyline` instances using the polyline.new function.

SEE ALSO

chart.point

series

The `series` keyword explicitly assigns the "series" type qualifier to variables and function parameters. Variables and parameters that use the "series" qualifier can reference values that change throughout a script's execution.

Explicit use of the `series` keyword when declaring the parameters of a library's exported functions is typically unnecessary, as the compiler can usually automatically detect whether a parameter is compatible with "series" or "simple" qualified values. By default, all exported function parameters are qualified as "series" wherever possible.

In variable declarations, the compiler can usually infer the qualified type automatically based on the values assigned to a variable, and it can automatically change a variable's qualifier to a stronger one when necessary. The type qualifier hierarchy is "const" < "input" < "simple" < "series", where "series" is the strongest.

Explicitly declaring a variable with the `series` keyword restricts the type qualifier to "series", meaning the script cannot pass its value to any variable or function parameter that requires a value with a weaker qualifier ("const", "input", or "simple").

When using this keyword to specify the type qualifier, one must also use a type keyword to declare the allowed type.

SYNTAX

```
export [method ]<functionName>([[series ]<paramType>] <paramName>[ = <defaultValue>])

[method ]<functionName>([series <paramType> ]<paramName>[ = <defaultValue>])

[var/varip ]series <variableType> <variableName> = <variableValue>
```

EXAMPLE

```
//@version=5
//@description A library with custom functions.
library("CustomFunctions", overlay = true)

//@function Finds the highest `source` value over `length` bars, filtered by the `cond` co
export conditionalHighest(series float source, series bool cond, series int length) =>
    //@variable The highest `source` value from when the `cond` was `true` over `length` b
    series float result = na
    // Loop to find the highest value.
    for i = 0 to length - 1
        if cond[i]
            value   = source[i]
            result := math.max(nz(result, value), value)
    // Return the `result`.
    result

//@variable Is `true` once every five bars.
```

```
series bool condition = bar_index % 5 == 0

//@variable The highest `close` value from every fifth bar over the last 100 bars.
series float hiValue = conditionalHighest(close, condition, 100)

plot(hiValue)
bgcolor(condition ? color.new(color.teal, 80) : na)
```

EXAMPLE

```
//@version=5
indicator("series variable not allowed")

//@variable A variable declared as "series int" with a value of 5.
series int myVar = 5

// This call causes an error.
// The `histbase` accepts "input int/float". It can't accept the stronger "series int" qua
plot(close, style = plot.style_histogram, histbase = myVar)
```

REMARKS

To learn more, see our User Manual's section on type qualifiers.

SEE ALSO

simple const

simple

The `simple` keyword explicitly assigns the "simple" type qualifier to variables and function parameters. Variables and parameters that use the "simple" qualifier can reference values established at the beginning of a script's execution that do not change later.

To restrict the parameters in a library's exported functions to only allow values with a "simple" or weaker type qualifier, using the `simple` keyword when declaring parameters is often necessary, as libraries automatically qualify all parameters as "series" wherever possible by default. Explicitly restricting functions to accept "simple" arguments also allows them to return "simple" values in some cases, depending on the operations they execute, making them usable with the parameters of built-in functions that do not allow "series" arguments.

In variable declarations, the compiler can usually infer the qualified type automatically based on the values assigned to a variable, and it can automatically change a variable's qualifier to a stronger one when necessary. The type qualifier hierarchy is "const" < "input" <

"simple" < "series", where "simple" is stronger than "input" and "const".

Explicitly declaring a variable with the `simple` keyword restricts the type qualifier to "simple", meaning the script cannot pass its value to any variable or function parameter that requires a value with a weaker qualifier ("const" or "input"). Additionally, one cannot assign a "series" value to a variable explicitly declared with the `simple` keyword.

When using this keyword to specify the type qualifier, one must also use a type keyword to declare the allowed type.

SYNTAX

```
export [method ]<functionName>([[simple ]<paramType>] <paramName>[ = <defaultValue>])

[method ]<functionName>([simple <paramType> ]<paramName>[ = <defaultValue>])

[var/varip ]simple <variableType> <variableName> = <variableValue></variableValue>
```

EXAMPLE

```
//@version=5
//@description A library with custom functions.
library("CustomFunctions", overlay = true)

//@function          Calculates the length values for a ribbon of four EMAs by multiplying
//@param baseLength The initial EMA length. Requires "simple int" because you can't use "s
//@returns           A tuple of length values.
export ribbonLengths(simple int baseLength) =>
    simple int length1 = baseLength
    simple int length2 = baseLength * 2
    simple int length3 = baseLength * 3
    simple int length4 = baseLength * 4
    [length1, length2, length3, length4]

// Get a tuple of "simple int" length values.
[len1, len2, len3, len4] = ribbonLengths(14)

// Plot four EMAs using the values from the tuple.
plot(ta.ema(close, len1), "EMA 1", color = color.red)
plot(ta.ema(close, len2), "EMA 1", color = color.orange)
plot(ta.ema(close, len3), "EMA 1", color = color.green)
plot(ta.ema(close, len4), "EMA 1", color = color.blue)
```

EXAMPLE

```
//@version=5
indicator("can't change simple to series")
```

```
//@variable A variable declared as "simple float" with a value of 5.0.
simple float myVar = 5.0

// This reassignment causes an error.
// The `close` variable returns a "series float" value. Since `myVar` is restricted to "si
// change its qualifier to "series".
myVar := close

plot(myVar)
```

REMARKS

To learn more, see our User Manual's section on type qualifiers.

SEE ALSO

series const

string

Keyword used to explicitly declare the "string" type of a variable or a parameter.

EXAMPLE

```
//@version=5
indicator("string")
string s = "Hello World!"    // Same as `s = "Hello world!"`
// string s = na // same as ""
plot(na, title=s)
```

REMARKS

Explicitly mentioning the type in a variable declaration is optional, except when it is initialized with na. Learn more about Pine Script® types in the User Manual page on the Type System.

SEE ALSO

var varip int float bool str.tostring str.format

table

Keyword used to explicitly declare the "table" type of a variable or a parameter. Table objects (or IDs) can be created with the table.new function.

EXAMPLE

```
//@version=5
indicator("table")
// Empty `table1` table ID.
var table table1 = na
// `table` type is unnecessary because `table.new()` returns "table" type.
var table2 = table.new(position.top_left, na, na)

if barstate.islastconfirmedhistory
    var table3 = table.new(position = position.top_right, columns = 1, rows = 1, bgcolor =
    table.cell(table_id = table3, column = 0, row = 0, text = "table3 text")
```

REMARKS

Table objects are always of "series" form.

SEE ALSO

var line label box table.new

Operators

-

Subtraction or unary minus. Applicable to numerical expressions.

SYNTAX

```
expr1 - expr2
```

RETURNS

Returns integer or float value, or series of values:

Binary `-` returns expr1 minus expr2.

Unary `-` returns the negation of expr.

REMARKS

You may use arithmetic operators with numbers as well as with series variables. In case of usage with series the operators are applied elementwise.

-=

Subtraction assignment. Applicable to numerical expressions.

SYNTAX

```
expr1 -= expr2
```

EXAMPLE

```
//@version=5
indicator("-=")
// Equals to expr1 = expr1 - expr2.
a = 2
b = 3
a -= b
// Result: a = -1.
plot(a)
```

RETURNS

Integer or float value, or series of values.

:=

Reassignment operator. It is used to assign a new value to a previously declared variable.

SYNTAX

```
<var_name> := <new_value>
```

EXAMPLE

```
//@version=5
indicator("My script")

myVar = 10

if close > open
    // Modifies the existing global scope `myVar` variable by changing its value from 10 t
    myVar := 20
    // Creates a new `myVar` variable local to the `if` condition and unreachable from the
    // Does not affect the `myVar` declared in global scope.
    myVar = 30

plot(myVar)
```

!=

Not equal to. Applicable to expressions of any type.

SYNTAX

```
expr1 != expr2
```

RETURNS

Boolean value, or series of boolean values.

?:

Ternary conditional operator.

SYNTAX

```
expr1 ? expr2 : expr3
```

EXAMPLE

```
//@version=5
indicator("?:")
// Draw circles at the bars where open crosses close
s2 = ta.cross(open, close) ? math.avg(open,close) : na
plot(s2, style=plot.style_circles, linewidth=2, color=color.red)

// Combination of ?: operators for 'switch'-like logic
c = timeframe.isintraday ? color.red : timeframe.isdaily ? color.green : timeframe.isweekl
plot(hl2, color=c)
```

RETURNS

expr2 if expr1 is evaluated to true, expr3 otherwise. Zero value (0 and also NaN, +Infinity, -Infinity) is considered to be false, any other value is true.

REMARKS

Use na for 'else' branch if you do not need it.

You can combine two or more ?: operators to achieve the equivalent of a 'switch'-like statement (see examples above).

You may use arithmetic operators with numbers as well as with series variables. In case of usage with series the operators are applied elementwise.

SEE ALSO

na

[]

Series subscript. Provides access to previous values of series expr1. expr2 is the number of bars back, and must be numerical. Floats will be rounded down.

SYNTAX

```
expr1[expr2]
```

EXAMPLE

```
//@version=5
indicator("[]")
// [] can be used to "save" variable value between bars
a = 0.0 // declare `a`
a := a[1] // immediately set current value to the same as previous. `na` in the beginning
if high == low // if some condition - change `a` value to another
    a := low
plot(a)
```

RETURNS

A series of values.

SEE ALSO

math.floor

*

Multiplication. Applicable to numerical expressions.

SYNTAX

```
expr1 * expr2
```

RETURNS

Integer or float value, or series of values.

*=

Multiplication assignment. Applicable to numerical expressions.

SYNTAX

```
expr1 *= expr2
```

EXAMPLE

```
//@version=5
indicator("*=")
// Equals to expr1 = expr1 * expr2.
a = 2
b = 3
a *= b
// Result: a = 6.
plot(a)
```

RETURNS

Integer or float value, or series of values.

/

Division. Applicable to numerical expressions.

SYNTAX

```
expr1 / expr2
```

RETURNS

Integer or float value, or series of values.

/=

Division assignment. Applicable to numerical expressions.

SYNTAX

```
expr1 /= expr2
```

EXAMPLE

```
//@version=5
indicator("/=")
// Equals to expr1 = expr1 / expr2.
a = 3
b = 3
a /= b
// Result: a = 1.
plot(a)
```

RETURNS

Integer or float value, or series of values.

%

Modulo (integer remainder). Applicable to numerical expressions.

SYNTAX

```
expr1 % expr2
```

RETURNS

Integer or float value, or series of values.

REMARKS

In Pine Script®, when the integer remainder is calculated, the quotient is truncated, i.e. rounded towards the lowest absolute value. The resulting value will have the same sign as the dividend.

Example: `-1 % 9 = -1 - 9 * int(-1/9) = -1 - 9 * int(-0.111) = -1 - 9 * 0 = -1.`

%=

Modulo assignment. Applicable to numerical expressions.

SYNTAX

```
expr1 %= expr2
```

EXAMPLE

```
//@version=5
indicator("%=")
// Equals to expr1 = expr1 % expr2.
a = 3
b = 3
a %= b
// Result: a = 0.
plot(a)
```

RETURNS

Integer or float value, or series of values.

+

Addition or unary plus. Applicable to numerical expressions or strings.

SYNTAX

```
expr1 + expr2
```

RETURNS

Binary `+` for strings returns concatenation of expr1 and expr2

For numbers returns integer or float value, or series of values:

Binary `+` returns expr1 plus expr2.

Unary `+` returns expr (does nothing added just for the symmetry with the unary - operator).

REMARKS

You may use arithmetic operators with numbers as well as with series variables. In case of usage with series the operators are applied elementwise.

+=

Addition assignment. Applicable to numerical expressions or strings.

SYNTAX

```
expr1 += expr2
```

EXAMPLE

```
//@version=5
indicator("+=")
// Equals to expr1 = expr1 + expr2.
a = 2
b = 3
a += b
// Result: a = 5.
plot(a)
```

RETURNS

For strings returns concatenation of expr1 and expr2. For numbers returns integer or float value, or series of values.

REMARKS

You may use arithmetic operators with numbers as well as with series variables. In case of usage with series the operators are applied elementwise.

<

Less than. Applicable to numerical expressions.

SYNTAX

```
expr1 < expr2
```

RETURNS

Boolean value, or series of boolean values.

<=

Less than or equal to. Applicable to numerical expressions.

SYNTAX

```
expr1 <= expr2
```

RETURNS

Boolean value, or series of boolean values.

==

Equal to. Applicable to expressions of any type.

SYNTAX

```
expr1 == expr2
```

RETURNS

Boolean value, or series of boolean values.

=>

The '=>' operator is used in user-defined function declarations and in switch statements.

The function declaration syntax is:

SYNTAX

```
<identifier>([<parameter_name>[=<default_value>]], ...) =>
    <local_block>
    <function_result>
```

A <local_block> is zero or more Pine Script® statements.

The <function_result> is a variable, an expression, or a tuple.

EXAMPLE

```
//@version=5
indicator("=>")
// single-line function
f1(x, y) => x + y
// multi-line function
f2(x, y) =>
    sum = x + y
    sumChange = ta.change(sum, 10)
    // Function automatically returns the last expression used in it
plot(f1(30, 8) + f2(1, 3))
```

REMARKS

You can learn more about user-defined functions in the User Manual's pages on Declaring functions and Libraries.

>

Greater than. Applicable to numerical expressions.

SYNTAX

```
expr1 > expr2
```

RETURNS

Boolean value, or series of boolean values.

>=

Greater than or equal to. Applicable to numerical expressions.

SYNTAX

```
expr1 >= expr2
```

RETURNS

Boolean value, or series of boolean values.

Annotations

@description

Sets a custom description for scripts that use the library declaration statement. The text provided with this annotation will be used to pre-fill the "Description" field in the publication dialogue.

EXAMPLE

```
//@version=5
// @description Provides a tool to quickly output a label on the chart.
library("MyLibrary")

// @function Outputs a label with `labelText` on the bar's high.
// @param labelText (series string) The text to display on the label.
// @returns Drawn label.
export drawLabel(string labelText) =>
    label.new(bar_index, high, text = labelText)
```

@enum

If placed above an enum declaration, it adds a custom description for the enum. The Pine Editor's autosuggest uses this description and displays it when a user hovers over the enum name. When used in library scripts, the descriptions of all enums using the export keyword will pre-fill the "Description" field in the publication dialogue.

EXAMPLE

```
//@version=5
indicator("Session highlight", overlay = true)

//@enum        Contains fields with popular timezones as titles.
//@field exch  Has an empty string as the title to represent the chart timezone.
enum tz
    utc  = "UTC"
    exch = ""
    ny   = "America/New_York"
    chi  = "America/Chicago"
    lon  = "Europe/London"
    tok  = "Asia/Tokyo"

//@variable The session string.
selectedSession = input.session("1200-1500", "Session")
//@variable The selected timezone. The input's dropdown contains the fields in the `tz` er
selectedTimezone = input.enum(tz.utc, "Session Timezone")

//@variable Is `true` if the current bar's time is in the specified session.
bool inSession = false
if not na(time("", selectedSession, str.tostring(selectedTimezone)))
    inSession := true

// Highlight the background when `inSession` is `true`.
bgcolor(inSession ? color.new(color.green, 90) : na, title = "Active session highlight")
```

@field

If placed above a type or enum declaration, it adds a custom description for a field of the type/enum. After the annotation, users should specify the field name, followed by its description.

The Pine Editor's autosuggest uses this description and displays it when a user hovers over the type/enum or field name. When used in library scripts, the descriptions of all types/enums using the export keyword will pre-fill the "Description" field in the publication dialogue.

EXAMPLE

```
//@version=5
indicator("New high over the last 20 bars", overlay = true)

//@type A point on a chart.
//@field index The index of the bar where the point is located, i.e., its `x` coordinate.
//@field price The price where the point is located, i.e., its `y` coordinate.
type Point
    int index
    float price

//@variable If the current `high` is the highest over the last 20 bars, returns a new `Poi
Point highest = na

if ta.highestbars(high, 20) == 0
    highest := Point.new(bar_index, high)
    label.new(highest.index, highest.price, str.tostring(highest.price))
```

@function

If placed above a function declaration, it adds a custom description for the function.

The Pine Editor's autosuggest uses this description and displays it when a user hovers over the function name. When used in library scripts, the descriptions of all functions using the export keyword will pre-fill the "Description" field in the publication dialogue.

EXAMPLE

```
//@version=5
// @description Provides a tool to quickly output a label on the chart.
library("MyLibrary")
```

```
// @function Outputs a label with `labelText` on the bar's high.
// @param labelText (series string) The text to display on the label.
// @returns Drawn label.
export drawLabel(string labelText) =>
    label.new(bar_index, high, text = labelText)
```

@param

If placed above a function declaration, it adds a custom description for a function parameter. After the annotation, users should specify the parameter name, then its description.

The Pine Editor's autosuggest uses this description and displays it when a user hovers over the function name. When used in library scripts, the descriptions of all functions using the export keyword will pre-fill the "Description" field in the publication dialogue.

EXAMPLE

```
//@version=5
// @description Provides a tool to quickly output a label on the chart.
library("MyLibrary")

// @function Outputs a label with `labelText` on the bar's high.
// @param labelText (series string) The text to display on the label.
// @returns Drawn label.
export drawLabel(string labelText) =>
    label.new(bar_index, high, text = labelText)
```

@returns

If placed above a function declaration, it adds a custom description for what that function returns.

The Pine Editor's autosuggest uses this description and displays it when a user hovers over the function name. When used in library scripts, the descriptions of all functions using the export keyword will pre-fill the "Description" field in the publication dialogue.

EXAMPLE

```
//@version=5
// @description Provides a tool to quickly output a label on the chart.
library("MyLibrary")
```

```
// @function Outputs a label with `labelText` on the bar's high.
// @param labelText (series string) The text to display on the label.
// @returns Drawn label.
export drawLabel(string labelText) =>
    label.new(bar_index, high, text = labelText)
```

@strategy_alert_message

If used within a strategy script, it provides a default message to pre-fill the "Message" field in the alert creation dialogue.

EXAMPLE

```
//@version=5
strategy("My strategy", overlay=true, margin_long=100, margin_short=100)
//@strategy_alert_message Strategy alert on symbol {{ticker}}

longCondition = ta.crossover(ta.sma(close, 14), ta.sma(close, 28))
if (longCondition)
    strategy.entry("My Long Entry Id", strategy.long)
strategy.exit("Exit", "My Long Entry Id", profit = 10 / syminfo.mintick, loss = 10 / symir
```

@type

If placed above a type declaration, it adds a custom description for the type.

The Pine Editor's autosuggest uses this description and displays it when a user hovers over the type name. When used in library scripts, the descriptions of all types using the export keyword will pre-fill the "Description" field in the publication dialogue.

EXAMPLE

```
//@version=5
indicator("New high over the last 20 bars", overlay = true)

//@type A point on a chart.
//@field index The index of the bar where the point is located, i.e., its `x` coordinate.
//@field price The price where the point is located, i.e., its `y` coordinate.
type Point
    int index
    float price
```

```
//@variable If the current `high` is the highest over the last 20 bars, returns a new `Poi
Point highest = na

if ta.highestbars(high, 20) == 0
    highest := Point.new(bar_index, high)
    label.new(highest.index, highest.price, str.tostring(highest.price))
```

@variable

If placed above a variable declaration, it adds a custom description for the variable.

The Pine Editor's autosuggest uses this description and displays it when a user hovers over the variable name.

EXAMPLE

```
//@version=5
indicator("New high over the last 20 bars", overlay = true)

//@type A point on a chart.
//@field index The index of the bar where the point is located, i.e., its `x` coordinate.
//@field price The price where the point is located, i.e., its `y` coordinate.
type Point
    int index
    float price

//@variable If the current `high` is the highest over the last 20 bars, returns a new `Poi
Point highest = na

if ta.highestbars(high, 20) == 0
    highest := Point.new(bar_index, high)
    label.new(highest.index, highest.price, str.tostring(highest.price))
```

@version=

Specifies the Pine Script® version that the script will use. The number in this annotation should not be confused with the script's version number, which updates on every saved change to the code.

EXAMPLE

```
//@version=5
indicator("Pine v5 Indicator")
plot(close)
```

EXAMPLE

```
//This indicator has no version annotation, so it will try to use v1.
//Pine Script® v1 has no function named `indicator()`, so the script will not compile.
indicator("Pine v1 Indicator")
plot(close)
```

REMARKS

The version should always be specified. Otherwise, for compatibility reasons, the script will be compiled using Pine Script® v1, which lacks most of the newer features and is bound to confuse. This annotation can be anywhere within a script, but we recommend placing it at the top of the code for readability.