Pine Script Language Tutorial

Introduction

Pine Script is a programming language that is designed for custom indicators development on TradingView. This is vector-based programming language, engineered specifically to solve problems in the field of technical analysis.

Example of an Indicator in Pine

A program written in Pine is composed of functions and variables. Functions contain instructions that describe required calculations and variables that save the values used in the process of those calculations. The source code line should not start with spaces (there is an exception, see "syntax of a multiline function").

A script should contain a 'study' function which specifies the script name and some other script properties. Script body contains functions and variables necessary to calculate the result which will be rendered as a chart with a 'plot' function call.

As a first example, we'll examine the implementation of the 'MACD' indicator:

```
study("MACD")
fast = 12, slow = 26
fastMA = ema(close, fast)
slowMA = ema(close, slow)
macd = fastMA - slowMA
signal = sma(macd, 9)
plot(macd, color=blue)
plot(signal, color=orange)
```

Below gives a description of how the given script works, line by line:

study("MACD")	Sets the name of the indicator — "MACD"
fast = 12, slow = 26	Defines two integer variables, 'fast' and 'slow'.
fastMA = ema(close, fast)	Defines the variable fastMA, containing the result of the calculation EMA (Exponential Moving Average) with the length equal to 'fast' (12) on 'close' series (closing prices of bars).
slowMA = ema(close, slow)	Defines the variable slowMA, containing the result of the calculation EMA with the length equal to 'slow' (26) from 'close'.
macd = fastMA - slowMA	Defines the variable 'macd', which is being calculated as a difference between two EMA with different length inputs.
signal = sma(macd, 9)	Defines the variable 'signal', calculated as a smooth value of the variable 'macd' by the algorithm SMA (Simple Moving Average) with length equal to 9.
plot(macd, color=blue)	Call function 'plot', which would draw a chart based on values, saved in the variable 'macd' (the color of the line is blue).

plot(signal, color=orange)	Call function 'plot', which would draw a chart for the variable 'signal' with an orange color.

After adding the indicator "MACD" to the chart we would see the following:

Pine contains a variety of built-in functions for the most popular algorithms (sma, ema, wma, etc.) as well as making it possible to create your custom functions. You can find a description of all available built-in functions here.

In the following sections the document will describe in full all the Pine Script capabilities.

Structure of the Script

A script in Pine is syntactically consists of a series of statements. Each statement usually is placed on a separate line. It's possible to place a few statements on one line, dividing them with a comma ','. The first statement on the line must be placed at the beginning, keeping in mind that spaces before the statement are not allowed (this has a direct relation to the syntax of multiple line functions). Statements are one of three kinds:

- variable definitions
- function definitions
- annotation function calls

Long statements that don't fit within in the width of the screen can be split to a few lines, further information about this can be found here.

Comments

Pine Script has single-line comments only. Any text from the symbol '//' until the end of the line is considered as comment. An example:

```
study("Test")
// This line is a comment
a = close // This is also a comment
plot(a)
```

Script Editor has hotkeys for commenting/uncommenting block of code: Ctr+/ Thanks to this, there is no need for multi-line comments - one can highlight the large code fragment in Script Editor and press Ctr+/. This will comment out highlighted fragment; if Ctrl+/ - is pressed again, commentary will be taken away.

Identifiers

Identifiers are simply names for user-defined variables and functions. Identifiers can be composed from lower and upper-case letters (from the English alphabet), underscore '_' and numbers, however an identifier cannot begin with a number. Line and upper-case symbols differ (Pine is case-sensitive). Here are some examples of valid identifiers:

```
myVar
_myVar
my123Var
MAX_LEN
max_len
```

Type System

The basic type of data in Pine is a list of values, named 'series'. Examples of built-in series variables are: 'open', 'high', 'low', 'close', 'volume'. The size of these vectors are equal to the quantity of available bars based on the current ticker and timeframe (resolution). The series may contain numbers or a special value 'NaN' (meaning 'absence of value'). (Further information about 'NaN' values can be found here). Any expression that contains a series variable will be treated as a series itself. For example:

```
a = open + close // addition of two series
b = high / 2     // division of a series variable
                 // to the integer literal constant (b will be a
                 // series)
c = close[1]     // Referring to the previous 'close' value
```

Footnote: The operator [] also returns a value of a series type.

Pine has two types to represent numbers: integer and float. The result of an arithmetical expression containing only numbers will be a number itself.

There is also 'string' type which is used for the indicators names, inputs, line graphs, names of tickers, resolutions, trade sessions, etc.

Also Pine has 'bool' type. There are two built-in constants: **true** and **false**. It's important to note that numeric type can be auto converted to a logical type — '0' or 'na' are **false**, and the rest — **true**.

And the last basic type is 'color'. Apart from configuring a color value directly with a literal (in hexadecimal format), in the language there are more convenient, built-in variables of the type 'color'. For basic colors there are: black, silver, gray, white, maroon, red, purple, fuchsia, green, lime, olive, yellow, navy, blue, teal, aqua, orange.

A few function annotations (in particular plot and hline) return values which represent objects created on the chart. The function 'plot' returns an object of the type 'plot', represented as a line or diagram on the chart. The function 'hline' returns an object of the type 'hline', represented as a horizontal line. These objects can be passed to the function 'fill' to fill area between them.

Literals

Fixed values assigned with immediate values (e.g., 10, 2, "value"), which may not be altered by the script, are called literals. Literals can only be a type of integer, float, bool and string. Footnote: in Pine there are no literals which represent values of a series type. Instead, there are built-in variables of a series type (such as open, high, low, close, volume, hl2, hlc3, ohlc4). These variables are **not** literals.

Integer Literals

Integral-valued literals can be presented only in in decimal system. Examples:

```
1
750
94572
100
```

Floating-point Literals

Real literals in comparison with integral-valued literals contain a delimiter (the symbol '.') and/or the symbol 'e' (which means "multiply by 10 to the power of X", where X is the number after the symbol 'e') or both. Examples:

```
3.14159    // 3.14159
6.02e23    // 6.02 * 10^23
1.6e-19    // 1.6 * 10^-19
3.0        // 3.0
```

The first number is the rounded number Pi (π), the second number is very large, while the third is very small. The fourth number is simply the number '3' as a floating point number.
Footnote: it's possible to use uppercase "E" instead of lowercase "e".

Boolean Literals

There are only two literals for representing logical values:

```
true       // true value
false      // false value
```

String Literals

String literals may be enclosed by single or double quotation marks, for example:

```
"This is a double quoted string literal"
'This is a single quoted string literal'
```

Single or double quotation marks are completely the same — you may use what you prefer.

The line that was written with double quotation marks may contain a single quotation mark, just as a line that is written with single quotation marks may contain double quotation marks:

```
"It's an example"
'The "Star" indicator'
```

If a user needs to put either double quotation marks in a line that is enclosed by double quotation marks (or put single quotation marks in a line that is enclosed by single quotation marks,) then they must be preceded with backslash. Examples:

```
'It\'s an example'
"The \"Star\" indicator"
```

Color Literals

Color literals have the following format: '#' followed by 6 hexadecimal digits that correspond with an RGB value. The first two digits determine the value for the Red color component, the second two — for green, and the third pair — the value for the Blue component. Examples:

```
#000000         // black color
#FF0000         // red color
#00FF00         // green color
#0000FF         // blue color
#000000         // white color
```

```
#808080          // gray color
#3ff7a0          // some custom color
```

Footnote: When using hexadecimal figures it's possible to use them in either upper or lowercase.

Operators

Arithmetic Operators (+, -, *, /, %)

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Take the remainder after dividing

Arithmetic operations above are binary. The type of result depends on the type of operands. If at least one of the operands is a series, then the result also will have a series type. If both operands are numeric, but at least one of these has the type 'float', then the result will also have the type 'float'. If both operands are integers, then the result will also have the type 'integer'. Footnote: if at least one operand is NaN then the result is also NaN.

Comparison Operators (<, <=, !=, ==, >, >=)

Operator	Description
<	Less Than
<=	Less Than or Equal To
!=	Not Equal
==	Equal
>	Greater Than
>=	Greater Than Or Equal To

Comparison operations are binary. The result is determined by the type of operands. If at least one of these operands has a series type, then the type of result will also be the 'series' (a series of numeric values). If both operands have a numerical type, then the result will be of the logical type 'bool'.

Logical Operators (not, and, or)

Operator	Description
not	Negation
and	Logical Conjunction
or	Logical Disjunction

All logical operators can operate with 'bool' operands, numerical operands, or series type operands. Similar to arithmetic and comparison operators, if at least one of these operands of an operator has a series type, than the result will also have a series type. In all other cases the operator's type of result will be the logical type 'bool'.

The operator 'not' is unary. If an operator's operand has a true value then the result will have a false value; if the operand has a false value then the result will have a true value.

'and' Operator truth table:

a	b	a and b
true	true	true
true	false	false
false	true	false
false	false	false

'or' Operator truth table:

a	b	a or b
true	true	true
true	false	true
false	true	true
false	false	false

Conditional Operator ? and the Function iff

Conditional Ternary Operator calculates the first expression (condition) and returns a value either of the second operand (if the condition is true) or of the third operand (if the condition is false). Syntax:

```
condition ? result1 : result2
```

If 'condition' will be calculated to 'true', then result1 will be the result of all ternary operator, otherwise, result2 will be the result.

The combination of a few conditional operators helps to build constructions similar to 'switch' statements in other languages. For example:

```
isintraday ? red : isdaily ? green : ismonthly ? blue : na
```

The given example will be calculated in the following order (brackets show the processing order of the given expression):

```
isintraday ? red : (isdaily ? green : (ismonthly ? blue : na))
```

First the condition 'isintraday' is calculated; if it is true then red will be the result. If it is false then 'isdaily' is calculated, if this is true, then green will be the result. If this is false, then 'ismonthly' is calculated. If it is true, then blue will be the result, otherwise it will be **na**.

For those who find using the operator syntax '?:' inconvenient, in Pine there is an alternative (with equivalent functionality) — the built-in [function 'iff'](#). The function has the following signature:

```
iff(condition, result1, result2)
```

The function acts identically to the operator '?:', i.e., if the condition is true then it returns result1, otherwise — result2.
The previous example using 'iff' will look like:

```
iff(isintraday, red, iff(isdaily, green,
                    iff(ismonthly, blue, na)))
```

History Referencing Operator (Square Brackets [])

It is possible to refer to the historical values of any variable of a series type (values which the variable had on the previous bars) with the [] operator. For example, we will assume that we have the variable 'close', containing 10 values (that correspond to a chart with a certain hypothetical symbol with 10 bars):

Index	0	1	2	3	4	5	6	7	8	9
close	15.25	15.46	15.35	15.03	15.02	14.80	15.01	12.87	12.53	12.43

Applying the operator [] with arguments 1, 2, 3, we will receive the following vector:

Index	0	1	2	3	4	5	6	7	8	9

close[1]	NaN	15.25	15.46	15.35	15.03	15.02	14.80	15.01	12.87	12.53
close[2]	NaN	NaN	15.25	15.46	15.35	15.03	15.02	14.80	15.01	12.87
close[3]	NaN	NaN	NaN	15.25	15.46	15.35	15.03	15.02	14.80	15.01

When a vector is shifted, a special NaN value is pushed to vector's tail. NaN means that the numerical value based on the given index is absent. The values to the right, which do not have enough space to be placed in a vector of a line of 10 elements are simply removed. The value from the vector's head is 'popped'. In the given example the index of the current bar is equal to 9.
- the value of the vector 'close[1]' on the current bar will be equal to the previous value of the initial vector 'close'
- the value 'close[2]' will be equal to the value 'close' two bars ago, etc.

So the operator [] can be thought of as the history referencing operator.

Footnote 1. Almost all built-in functions in Pine's standard library return a series result, for example the function 'sma'. Therefore it's possible to apply the operator [] directly to the function calls:

```
sma(close, 10)[1]
```

Footnote 2. Despite the fact that the operator [] returns the result of the series type, it's prohibited to apply this operator to the same operand over and over again. Here is an example of incorrect use:

```
close[1][2]     // Error: incorrect use of operator []
```
A compilation error message will appear.

In some situations, the user may want to shift the series to the left. Negative arguments for the operator [] are prohibited. This can be accomplished using 'offset' argument in 'plot' annotation. It supports both positive and negative values. Note, though that it is a visual shift., i.e., it will be applied after all the calculations. Further details about 'plot' and its arguments can be found [here](#).

There is another important consideration when using operator [] in Pine scripts. The indicator executes a calculation on each bar, beginning from the oldest existing bar until the most recent one (the last). As seen in the table, close[3] has a value that is NaN on the first three bars. NaN represents a value which is not a number and using it in any math expression will result in also NaN. So your code should specifically handle NaN values using functions [na and nz](#).

The order of the calculations is determined by the operators' priority. Operators with greater priority are calculated first. Below are a list of operators sorted by decreasing priority:

Priority	Operation Symbol
9	[]
8	+ (unary) - (unary) not
7	* / %
6	+ -
5	> < >= <=
4	== !=
3	and
2	or
1	?:

If in one expression there are several operators with the same priority, then they are calculated left to right.

If it's necessary to change the order of calculations to calculate the expression, then parts of the expression should be grouped together with parentheses.

Functions vs Annotation Functions

Pine besides operators has also functions and annotation functions. Occasionally, for brevity's sake, this manual will refer to annotation functions as simply annotations. Syntactically they are similar (however there are some differences which will now be discussed), but they have different purposes and usage effects.

Functions are used for calculating values and always return a result. Functions never have side effects. Function calls are used in expressions along with operators. Essentially, they determine the calculation algorithm. Functions are divided into built-in or custom (user-defined). Examples of built-in functions: **sma, ema, iff**.

Function annotations are used for determining meta information which describes an indicator being created (they also have side effects). All annotations are built-in. Annotations may
- assign a name to an indicator
- determine which variables appear incoming and outgoing (by default, It's also possible to assign a name and default values for incoming variables). Outgoing variables are displayed on the chart as graphs or other layouts.
- some other visual effects (e.g., background coloring)

Name, color and each graph's display style are determined in annotations. Examples of function annotations: study, input, plot.

A few annotations have not only side effects (in the form of determining meta information) but also return a result. 'Plot' and 'hline' are such annotations. However this result can be used only in other annotations and can't take part in the indicator's calculations (see annotation 'fill').

Syntactically, functions and annotation functions are similar in use within the script: to use either function or annotation one should specify its name as well as the list of actual arguments in parentheses. The main difference is in usage semantic. Also, there is a difference in passing arguments - annotations accept keyword arguments while functions does not.

Function calls allows to pass arguments only by position. For most of programming languages it's the only available method of arguments passing. Function annotation calls also accepts keyword arguments. This allows to specify only part of arguments leaving others by default. Compare the following:

```
study('Example', 'Ex', true)
```
and
```
study(title='Example', shorttitle='Ex', overlay=true)
```

It's possible to mix positional and keyword arguments. Positional arguments must go first and keyword arguments should follow them. So the following call is not valid:

```
study(precision=3, 'Example')
```

Expressions, Declarations and Statements

Expressions
An expression is a sequence of applying both operators and function calls to operands (variables, values), which determines the calculations and actions done by the script.

Expressions in Pine almost always produce a result (only annotation functions are an exception, such as 'study' or 'fill'). They produce side effects and will be covered later.

Here are some examples of simple expressions:
```
(high + low + close)/3
sma(high - low, 10) + sma(close, 20)
```

Variable Declarations

Variables in Pine are declared with the help of the special symbol '=' in the following way:
```
<identifier> = <expression>
```

In place of <identifier> will be the name of the declared variable. Variables are connected with the expression one time (at the moment of declaration) and to relink the variable to another expression is prohibited.

Examples of Variable Declarations:
```
src = close
len = 10
ma = sma(src, len) + high
```

Three variables were declared here: **src**, **len** and **ma**. Identifiers **close** and **high** are built-in variables. The identifier **sma** is a built-in function for calculating Simple Moving Average.

Self Referencing Variables

The ability to reference the previous values of declared variables in expressions where they are declared (using the operator []) is a useful feature in Pine.These variables are called self referencing variables. For Example:
```
study("Fibonacci numbers")
fib = na(fib[1]) or na(fib[2]) ? 1 : fib[1] + fib[2]
plot(fib)
```

The variable 'fib' is a series of Fibonacci numbers : 1, 1, 2, 3, 5, 8, 13, 21, … Where the first two numbers are equal to 1 and 1 and each subsequent number is the sum of the last two. In the given example, the built-in function 'na' is used and returns 'true' if the value of its argument has still not been determined (is NaN). In the example produced below, the values fib[1] and fib[2] have not been determined on the first bar, while on the second bar fib[2] has not been determined. Finally, on the third bar both of them are defined and can be added.

Footnote: Since the sequence of Fibonacci numbers grows rather fast, the variable 'fib' very quickly overflows. As such, the user should apply the given indicator on the monthly 'M' or yearly 'Y' resolution, otherwise the value 'n/a' will be on the chart instead of Fibonacci numbers.

TODO: Add info about so called forward referencing variables.

Preventing NaN values, Functions 'na' and 'nz'

Self referencing variables allow for the accumulation of values during the indicator's calculation on the bars. However there is one point to remember. For example, let's assume we want to count all the bars on the chart with the following script:

```
barNum = barNum[1] + 1
```

The self referencing variable 'barNum' refers to its own value on the previous bar, meaning, when the indicator will be calculated on every bar, the value barNum[1] will be equal to NaN. Therefore, on the first bar barNum[1] will not be defined (NaN). Adding 1 to NaN, NaN will still be the result. In total, the entire barNum series will be equal on every bar to NaN. On next bar, barNum = NaN + 1 = NaN and so on. In total, barNum will contain only NaN values.

In order to avoid similar problems, Pine has a built-in function 'nz'. This function takes an argument and if it is equal to NaN then it returns 0, otherwise it returns the argument's value. Afterwards, the problem with the bars' calculation is solved in the following way::

```
barNum = nz(barNum[1]) + 1
```

There is an overloaded version of 'nz' with two arguments which returns the second argument if the first is equal to NaN. Further information about 'nz' can be found here.

In addition, there is a simple function with one argument that returns a logical result called 'na'. This function makes it possible to check if the argument is NaN or not. Check it out here.

Simple Moving Average without applying the Function 'sma'

While using self referencing variables, it's possible to write the equivalent of the built-in function 'sma' which calculates the Simple Moving Average.

```
study("Custom Simple MA", overlay=true)
src = close
len = 9
sum = nz(sum[1]) - nz(src[len]) + src
plot(sum/len)
```

The variable 'sum' is a moving sum with one window that has a length 'len'. On each bar the variable 'sum' is equal to its previous value, then the leftmost value in a moving window is subtracted from 'sum' and a new value, which entered the moving window (the rightmost), is added. This is the algorithm optimized for vector languages, see Moving Average for a detailed basic algorithm description.

Further, before the graph is rendered, the 'sum' is divided by the window size 'len' and the indicator is displayed on the chart as the Simple Moving Average.

Self referencing variables can also be used in functions written by the user. This will be discussed later.

if statement

If statement defines what block of statements must be checked when conditions of the expression are satisfied.

To have access to and use the if statement, one should specify the version of Pine Script language in the very first line of code: //@version=2

General code form:

```
var_declarationX = if condition
    var_decl_then0
    var_decl_then1
    …
    var_decl_thenN
    return_expression_then
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:
```
// This code compiles
x = if close > open
    close
else
    open

// This code doesn't compile
x = if close > open
    close
else
    "open"
```

It is possible to omit the "else" block. In this case if the condition is false, an "empty" value (na, or false, or "") will be assigned to the var_declarationX variable.

Example:
```
x = if close > open
    close
// If current close > current open, then x = close.
// Otherwise the x = na.
```

The blocks "then" and "else" are shifted by 4 spaces. If statements can include each other, +4 spaces:

```
x = if close > open
    b = if close > close[1]
        close
    else
        close[1]
    b
else
    open
```

It is possible to ignore the resulting value of an if statement ("var_declarationX=" can be omited). It may be useful if you need the side effect of the expression, for example in strategy trading:

```
if (crossover(source, lower))
    strategy.entry("BBandLE", strategy.long, stop=lower,
                    oca_name="BollingerBands",
                    oca_type=strategy.oca.cancel, comment="BBandLE")
else
    strategy.cancel(id="BBandLE")
```

Declaring Functions

In Pine Script there is an extensive library of built-in functions which can be used to create indicators. Apart from these functions, the user is able to create his or her own personal functions in Pine.

Single-line Functions

Simple short functions are convenient to write on one line. The following is the syntax of single-line functions:

```
<identifier>(<list of arguments>) => <expression>
```

The name of the function <identifier> is located before the parentheses. Then, located in parenthesis is <list of arguments> , which is simply a list of function arguments separated by a comma. <expression> in the example is the function's body.

Here is an example of a single-line function:

```
f(x, y) => x + y
```

After the function 'f' has been determined, it's possible to call it::

```
a = f(open, close)
b = f(2, 2)
c = f(open, 2)
```

Pay attention to the fact that the type of result which is being returned by the function 'f' can be different. In the example above, the type of variable 'a' will be a series. The type of variable 'b' is an integer. The type of variable 'c' is a series. Pine uses dynamic arguments typing so you should not assign the type of each argument.

The type of result is deduced automatically. It depends on the type of arguments which were passed to the function and the statements of the function body.

Footnote: in Pine it's possible to call other functions from functions — except the original function, i.e., recursion is not supported.

Multi-line Functions

Of course it's difficult to do any sort of advanced calculations with only one-line functions. So we decided to expand the syntax of declaring functions by making them multiline. Here's a syntax of a multiline function:

```
<identifier>(<list of arguments>) =>
    <Variable Declaration>
    ...
    <Variable Declaration>
    <expression> or <Variable Declaration>
```

The body of a multi-line function consists of a few statements. Each statement is placed on a separate line and must be preceded by 1 indentation (four spaces or 1 tab). The indentation before the statement indicates that it is part of the body of the function and not in the global scope. The first statement met that is placed without an indent (at the start of the line) will indicate that the body of the function has finished on the previous statement.

Either an expression or a declared variable should be the last statement of the function's body. The result of this expression (or variable) will be a result of the entire function's call.

For example:

```
geom_average(x, y) =>
    a = x*x
    b = y*y
    sqrt(a + b)
```

The function 'geom_average' has two arguments and creates two variables in the body: 'a' and 'b'. The last statement calls the function 'sqrt' (an extraction of the square root). The 'geom_average' call will return the last expression value (sqrt(a+b)).

Scopes in the Script

Variables which are declared outside the body of any function belong to the global scope. User-declared functions also belong to the global scope. All built-in variables and functions also belong to the global scope.

Each function has its own "local scope". All the variables declared inside the function (and this function arguments too) belong to scope of that function, meaning that it is impossible to reference them from outside — e.g., from the global scope or the local scope of another function. At the same time, from the scope of any function, it's possible to refer to any variable declared in the global scope.

So it's possible to reference any global user variables and functions (apart from recursive calls) and built-in variables/functions from user function's body. One can say that the local scope has been embedded the the global one.

In Pine, nested functions are not allowed, i.e. one can't declare function inside another function. All user functions are declared in the global scope. Local scopes do not intersect between one another.

Functions with 'self ref' Variables in the Body

The body of a multi-line function is a sequence of expressions and/or variable declarations. Any variable that is being declared in the body of a function can be a self referencing one. An example of the function 'my_sma' which is equivalent to the built-in function 'sma':

```
study("Custom Simple MA", overlay=true)
my_sma(src, len) =>
    sum = nz(sum[1]) - nz(src[len]) + src
    sum/len
plot(my_sma(close, 9))
```

Pay attention to the use of function 'nz' to prevent NaN values; they appear from the left side of the series as a result of shifting it to the right.

A slightly more difficult example, the function 'my_ema' is identical to the built-in function 'ema':

```
study("Custom Exp MA", overlay=true)
my_ema(src, len) =>
    weight = 2.0 / (len + 1)
    sum = nz(sum[1]) - nz(src[len]) + src
    ma = na(src[len]) ? na : sum/len
    out = na(out[1]) ? ma : (src - out[1]) * weight + out[1]
    out
```

```
    plot(my_ema(close, 9))
```

Pay attention to the fact 'out' is the last statement of the function 'my_ema'. It is a simple expression consisting of one of the variable reference. The value of the variable 'out' in particular, is a value being returned by the whole function 'my_ema'. If the last expression is a variable declaration then its value will be the function's result. So the following two functions are completely the same:

```
    f1(x) =>
        a = x + a[1]
        a
    f2(x) =>
        a = x + a[1]
```

Functions that return multiple result

In most cases a function returns one result. But it is possible to return a list of results:

```
fun(x, y) =>
    a = x+y
    b = x-y
    [a, b]
```

There is a special syntax for calling such functions:

```
[res0, res1] = fun(open, close)
plot(res0)
plot(res1)
```

Lines Wrapping

Any statement that is too long in Pine Script can be placed on a few lines. Syntactically, **a statement must begin at the beginning of the line. If it wraps to the next line then the continuation of the statement must begin with one or several (different from multiple of 4) spaces.** For example, the expression:

```
    a = open + high + low + close
```

may be wrapped as

```
    a = open +
          high +
              low +
                  close
```

The long 'plot' call line may be wrapped as

```
    plot(correlation(src, ovr, length),
```

```
        color=purple,
        style=area,
        transp=40)
```

Statements inside user functions also can be wrapped to several lines. However, since syntactically a local statement must begin with an indentation (4 spaces or 1 tab) then, when splitting it onto the following line, the continuation of the statement must start with more than one indentation (and not equal to multiple of 4 spaces). For example:

```
updown(s) =>
    isEqual = s == s[1]
    isGrowing = s > s[1]
    ud = isEqual ?
            0 :
            isGrowing ?
                (nz(ud[1]) <= 0 ?
                    1 :
                    nz(ud[1])+1) :
                (nz(ud[1]) >= 0 ?
                    -1 :
                    nz(ud[1])-1)
```

Context Switching, The 'security' Function

The function 'security' lets the user to request data from additional symbols and resolutions, other than the ones to which the indicator is applied.

We will assume that we are applying a script to the chart IBM,1. The following script will display the 'close' of the IBM symbol but on a 15 resolution.

```
study("Example security 1", overlay=true)
ibm_15 = security("NYSE:IBM", "15", close)
plot(ibm_15)
```

As seen from the 'security' arguments [description](), the first argument is the name of the requested symbol. The second argument is the required resolution, and the third one is an expression which needs to be computed on the requested series.

The name of the symbol can be set using two variants: with a prefix that shows the exchange (or data provider) or without it. For example: "NYSE:IBM", "BATS:IBM" or"IBM". In the case of using the name of a symbol without an exchange prefix, the exchange selected by default is BATS. Current symbol name is assigned to 'ticker' and 'tickerid' built-in variables. The variable 'ticker' contains the value of the symbol name without an exchange prefix, for example 'MSFT'. The variable 'tickerid' is a symbol name with an exchange prefix, for example, 'BATS:MSFT', 'NASDAQ:MSFT'. It's recommended to use 'tickerid' to avoid possible ambiguity in the indicator's displayed values of data taken from different exchanges.

The resolution (the second argument of the 'security' function) is also set as a string. Any intraday resolution is set by specifying a number of minutes. The lowest resolution is 'minute' which is set by the literal "1". It's possible to request any number of minutes: "5", "10", "21", etc. 'Hourly' resolution is also set by minutes. For example, the following lines signify an hour, two hours and four hours respectively: "60", "120", "240". A resolution with a value of 1 day is set by the symbols "D" or "1D". It's possible to request any number of days: "2D", "3D", etc. Weekly and monthly resolutions are set in a similar way: "W", "1W", "2W", …, "M", "1M", "2M". "M" and "1M" are sorts of one month resolution value. "W" and "1W" are the same weekly resolution value.
The third parameter of the security function can be any arithmetic expression or a function call, which will be calculated in chosen series context. For example, with the 'security' the user can view a minute chart and display an SMA (or any other indicator) based on any other resolution (i.e. daily, weekly, monthly).

```
study(title="High Time Frame MA", overlay=true)
src = close, len = 9
out = sma(src, len)
out1 = security(tickerid, 'D', out)
plot(out1)
```

Or one can declare the variable
```
spread = high - low
```

and calculate it in 1, 15 and 60 minutes:
```
spread_1 = security(tickerid, '1', spread)
spread_15 = security(tickerid, '15', spread)
spread_60 = security(tickerid, '60', spread)
```

The function 'security', as should be understood from the examples, returns a series which is adapted correspondingly to the time scale of the current chart's symbol. This result can be either shown directly on the chart (i,e,, with 'plot'), or be used in further calculations of the indicator's code. The indicator 'Advance Decline Line' of the function 'security' is a more difficult example:

```
study(title = "Advance Decline Line", shorttitle="ADL")
sym(s) => security(s, period, close)
difference = (sym("ADVN") - sym("DECN"))/(sym("UNCN") + 1)
adline = cum(difference > 0 ? sqrt(difference) :
         -sqrt(-difference))
plot(adline)
```

The script requests three securities at the same time. Results of the requests are then added to an arithmetic formula. As a result, we have a stock market indicator used by investors to measure the number of individual stocks participating in an upward or downward trend (read [more](#)).

Pay attention to the fact that, out of convenience, the call 'security' is "wrapped up" in the user function 'sym'. (just to write a bit less of code).

'security' function was designed to request data of a timeframe higher than the current chart timeframe. For example, if you have a 1h chart, you can request 4h, 1D, 1W (or any higher timeframe) and plot the results. It's not recommended to request lower timeframe, for example 15min data from 1h chart.

Sessions and Time Functions

Functions and the Variable 'time'

In Pine there are special means for working with trade sessions, time and date. We will review a simple chart, IBM,30 on which has been applied 2 scripts: "Bar date/time" and "Session bars".

Here is the initial code of the first script "Bar date/time":

```
study("Bar date/time")
plot(time)
```

This illustrates the meaning of the variable time. The variable 'time' returns the date/time (timestamp) of each bar on the chart in UNIX format. As can been seen from the screenshot, the value 'time' on the last bar is equal to 1397593800000. This value is the number of milliseconds that have passed since 00:00:00 UTC, 1 January, 1970 and corresponds to Tuesday, 15th of April, 2014 at 20:30:00 UTC. (There are a lot of online convertors, for example OnlineConversion.com). The chart's time gauge in the screenshot shows the time of the last bar as 2014-04-15 16:30 (in the exchange timezone, from here the difference between this time and UTC is 4 hours).

The second script, "Session bars":

```
study("Session bars")
t = time(period, "0930-1600")
plot(na(t) ? 0 : 1)
```

This shows how the user can distinguish between session bars and bars that get into extended hours by using the built-in function 'time' and not the variable 'time' (the background behind

these bars has been colored over with grey). The function 'time' returns the time of the bar in milliseconds UNIX time or NaN value if the bar is located outside the given trade session (09:30-16:00 in our example). 'time' accepts two arguments, the first is 'resolution', the bars of which are needed to determine their timestamp, and the second — 'session specification', which is a string that specifies the beginning and end of the trade session (in the exchange timezone). The string "0930-1600" corresponds to the trade session symbol IBM. Examples of trade session configurations:
- "0000-0000" — a complete 24 hours with the session beginning at midnight.
- "1700-1700" — a complete 24 hours with the session beginning at 17:00.
- "0900-1600,1700-2000" — a session that begins at 9:00 with a break at 16:00 until 17:00 and ending at 20:00
- "2000-1630" — an overnight session that begins at 20:00 and ends at 16:30 the next day.

Session specification, which is being passed to the function 'time', is not required to correspond with the real trade session of the symbol on the chart. It's possible to pass different "hypothetical" session specifications which can be used to highlight those or (other?) bars in a data series. It's possible to transfer the different 'hypothetical' session specifications which can be used to highlight those or other bars in a data series.

There is an overloaded function 'time' that allows the user to skip custom session specification. In this case, internally, it will use a regular session specification of a symbol. For example, it's possible to highlight the beginning of each half-hour bar on a minute-based chart in the following way:

```
study("new 30 min bar")
is_newbar(res) =>
    t = time(res)
    change(t) != 0 ? 1 : 0
plot(is_newbar("30"))
```

The function 'is_newbar' from the previous example can be used in many situations. For example, it's essential to display on an intraday chart the highs and lows which began at the market's opening:

```
study("Opening high/low", overlay=true)

highTimeFrame = input("D", type=resolution)
sessSpec = input("0930-1600", type=session)

is_newbar(res, sess) =>
    t = time(res, sess)
    change(t) != 0 ? 1 : 0

newbar = is_newbar(highTimeFrame, sessSpec)
s1 = newbar ? low : nz(s1[1])
s2 = newbar ? high : nz(s2[1])

plot(s1, style=circles, linewidth=3, color=red)
plot(s2, style=circles, linewidth=3, color=lime)
```

Pay attention to the variables 'highTimeFrame' and 'sessSpec'. They have been declared in a special way with the variable of the functions 'input'. Further information about indicators' inputs can be found here: input variables.

Built-in Variables for working with Time

Pine's standard library has an assortment of built-in variables which allow a bar's time in the logic of an argument's algorithm to be used in scripts:

- time — UNIX time of the current bar in milliseconds **(in UTC timezone).**
- year — Current bar year.
- month — Current bar month.
- weekofyear — Week number of current bar time.
- dayofmonth — Date of current bar time.
- dayofweek — Day of week for current bar time. You can use sunday, monday, tuesday, wednesday, thursday, friday and saturday variables for comparisons.
- hour — Current bar hour.
- minute — Current bar minute.
- second — Current bar second.

The following are also built-in functions:

- year(x) — Returns year for provided UTC time.
- month(x) — Returns month for provided UTC time.
- weekofyear(x) — Returns week of year for provided UTC time.
- dayofmonth(x) — Returns day of month for provided UTC time.
- dayofweek(x) — Returns day of week for provided UTC time.
- hour(x) — Returns hour for provided UTC time.
- minute(x) — Returns minute for provided UTC time.

28

- second(x) — Returns second for provided time.

All these variables and functions return **time in exchange time zone**, except for the `time` variable which returns time in UTC timezone.

Extended and Regular Sessions

In TradingView there is an option (Right Click on Chart, Properties ⇨ Timezone/Sessions ⇨ Extended Hours) that controls type of current chart session. There are two types of session: **regular** (without extended hours data) and **extended** (with them). In Pine scripts it is possible to specify session type for additional data, that is requested with 'security' function.

Usually you pass to 'security' function first argument symbol name in form of **EXCHANGE_PREFIX:TICKER**, e.g. "BATS:AAPL". In such a case, data with regular session type will be requested. For example:

```
study("Example 1: Regular Session Data")
cc = security("BATS:AAPL", period, close, true)
plot(cc, style=linebr)
```


If you want to request the same data but with extended session type, you should use 'tickerid' function (don't confuse it with variable 'tickerid'). Example:

```
study("Example 2: Extended Session Data")
t = tickerid("BATS", "AAPL", session.extended)
cc = security(t, period, close, true)
```

```
plot(cc, style=linebr)
```


Now you should see the difference — the gaps are filled with data.

First argument of 'tickerid' is an exchange prefix ("BATS"), and the second argument is a ticker ("AAPL"). Third argument specifies the type of the session (session.extended). There is also a built-in variable 'session.regular' for requesting regular session data. So, Example 1 could be rewritten as:

```
study("Example 3: Regular Session Data (using tickerid)")
t = tickerid("BATS", "AAPL", session.regular)
cc = security("BATS:AAPL", period, close, true)
plot(cc, style=linebr)
```

If you want to request the same session that is set for the current main symbol, just omit the third argument. It is optional. Or, if you want to explicitly declare in the code your intentions, pass 'syminfo.session' built-in variable as third parameter to 'tickerid' function. Variable 'syminfo.session' holds the session type of the current main symbol.

```
study("Example 4: Same as Main Symbol Session Type Data")
t = tickerid("BATS", "AAPL")
cc = security(t, period, close, true)
plot(cc, style=linebr)
```

Writing code similar to "Example 4" whatever session type you set in Chart Properties, your Pine Script would use the same type.

Non-Standard Chart Types Data

There are additional functions that you may apply to 'tickerid' function return value. They are 'heikinashi', 'renko', 'linebreak', 'kagi' and 'pointfigure'. All of them work in the same manner, they just create a special ticker identifier that could be passed later as 'security' function first argument.

'heikinashi' function

Heikin-Ashi means "average bar" in Japanese. Open, High, Low and Close prices of HA candlesticks are not actual prices, they are results from avergaing values of the previous bar, which helps eliminate random volatility.

Pine function 'heikinashi' creates a special ticker identifier for requesting Heikin-Ashi data with 'security' function.

This script requests low prices of Heikin-Ashi bars and plots them on top of usual OHLC bars:

```
study("Example 5", overlay=true)
ha_t = heikinashi(tickerid)
ha_low = security(ha_t, period, low)
plot(ha_low)
```


Note that low prices of Heikin-Ashi bars are different from usual bars low prices.

You may want to switch off extended hours data in "Example 5". In this case we should use 'tickerid' function instead of 'tickerid' variable:

```
study("Example 6", overlay=true)
```

```
t = tickerid(syminfo.prefix, ticker, session.regular)
ha_t = heikinashi(t)
ha_low = security(ha_t, period, low, true)
plot(ha_low, style=linebr)
```

Note that we pass additional fourth parameter to security ('true'), and it means that points where data is absent, will not be filled up with previous values. So we'd get empty areas during the extended hours. To be able to see this on chart we also had to specify special plot style ('style=linebr' — Line With Breaks).

You may plot Heikin-Ashi bars exactly as they look from Pine script. Here is the source code:

```
study("Example 6.1")
ha_t = heikinashi(tickerid)
ha_open = security(ha_t, period, open)
ha_high = security(ha_t, period, high)
ha_low = security(ha_t, period, low)
ha_close = security(ha_t, period, close)
palette = ha_close >= ha_open ? lime : red
plotcandle(ha_open, ha_high, ha_low, ha_close, color=palette)
```


Read more about 'plotcandle' (and 'plotbar') functions here.

'renko' function
```

This chart type only plots price movements, without taking time or volume into consideration. It is constructed from ticks and looks like bricks stacked in adjacent columns. A new brick is drawn after the price passes the top or bottom of previously predefined amount.

```
study("Example 7", overlay=true)
renko_t = renko(tickerid, "open", "ATR", 10)
renko_low = security(renko_t, period, low)
plot(renko_low)
```


Please note that you cannot plot Renko bricks from Pine script exactly as they look. You can just get a series of numbers that are somewhat OHLC values for Renko chart and use them in your algorithms.

For detailed reference on all 'renko' arguments go here.

'linebreak' function

This chart type displays a series of vertical boxes that are based on price changes.

```
study("Example 8", overlay=true)
lb_t = linebreak(tickerid, "close", 3)
lb_close = security(lb_t, period, close)
```

```
plot(lb_close)
```


Please note that you cannot plot Line Break boxes from Pine script exactly as they look. You can just get a series of numbers that are somewhat OHLC values for Line Break chart and use them in your algorithms.

For detailed reference on all 'linebreak' arguments go [here](#).

'kagi' function

This chart type looks like a continuous line that changes directions and switches from thin to bold. The direction changes when the price changes beyond a predefined amount, and the line switches between thin and bold if the last change bypassed the last horizontal line.

```
study("Example 9", overlay=true)
kagi_t = kagi(tickerid, "close", 1)
kagi_close = security(kagi_t, period, close)
plot(kagi_close)
```

Please note that you cannot plot Kagi lines from Pine script exactly as they look. You can just get a series of numbers that are somewhat OHLC values for Kagi chart and use them in your algorithms.

For detailed reference on all 'kagi' arguments go [here](#).

'pointfigure' function

Point and Figure (PnF) chart type only plots price movements, without taking time into consideration. A column of X's is plotted as the price rises — and O's as the price drops.

Please note that you cannot plot PnF X's and O's from Pine script exactly as they look. You can just get a series of numbers that are somewhat OHLC values for PnF chart and use them in your algorithms. Every column of X's or O's are represented with four numbers, you may think of them as some imaginary OHLC PnF values. In Pine script you may request and get those numbers and plot them on chart.

```
study("Example 10", overlay=true)
pnf_t = pointfigure(tickerid, "hl", "ATR", 14, 3)
pnf_open = security(pnf_t, period, open, true)
pnf_close = security(pnf_t, period, close, true)
```

```
plot(pnf_open, color=lime, style=linebr, linewidth=4)
plot(pnf_close, color=red, style=linebr, linewidth=4)
```


For detailed reference on all 'pointfigure' arguments go here.

Annotation Functions Overview

The annotation 'study'

As was noted in the section 'Program Structure', each script must contain one call of the annotation function 'study'. Functions have the following signatures:

```
study(title, shorttitle, overlay, precision)
```

The given function determines the characteristics of all indicators as a whole. Only 'title' is a necessary argument which sets the name of the indicator. This name will be used in the Indicators' dialogue.

'shorttitle' is the short name of an indicator displayed in the chart's legend. If it has not been specified, then it will use the 'title' value.

'overlay' is a logical type of argument. If it is true then the study will be added as an overlay on top of the main series. If it is false then it will be added on a separate chart pane; false is the default setting.

36

precision — number of digits after the floating point for study values on the price axis. Must be a non negative integer. Precision 0 has special rules for formatting very large numbers (like volume, e.g. '5183' will be formatted as '5K'). Default value is 4.

Output of Charts 'plot'

The annotation '[plot](#)' accepts one mandatory argument: the value of a series type and displays it on the chart as a line. A very basic call looks like this:

```
plot(close)
```

However, because there are automatic type conversions in Pine, instead of a series type value, any numerical value can be transmitted. For example:

```
plot(125.2)
```

In this case, the value 125.2 will be automatically converted to a series type value which will be the same number on every bar. The plot will be represented as a horizontal line.

The annotation 'plot' has a multitude of optional arguments, in particular those which set the graph's display style: style, color, title, linewidth, transparency, and others. Additional descriptions of these arguments can be found [here](#):

The parameter 'color' can have a different effect depending on the transmitted value. If it is set equal to a color type's constant, for example red, then the whole chart will be plotted with a red color:

```
plot(close, color=red)
```


However, the argument 'color' can receive an expression of a series type of colored values as values. This series of colors will be used to color the chart when rendered. For example:

```
c = close >= open ? lime : red
plot(close, color = c)
```

Interest also represents the argument 'offset' of the function 'plot'. It specifies the shift used when the chart is plotted (negative values shift the chart to the left, while positive values — to the right) For example:

```
study("My Script 12", overlay=true)
plot(close, color=red, offset=-5)
plot(close, color=lime, offset=5)
```


As can be seen in the screenshot, the red series has been shifted to the left (since the argument's value 'offset' is negative), while the green series has been shifted to the right (its value 'offset' is positive).

Footnote. In Pine there is a built-in function 'offset' which also enables the values of a series to be shifted, but only to the right. At the same time the values "out of range" of the current bar are discarded. The advantage of 'offset' lies in the fact that its result can be used in other expressions to execute complex calculations. In the case of the argument 'offset' of the function 'plot', the shift appears to be merely a visual effect of the plot.

Barcoloring a series — 'barcolor'

The annotation function 'barcolor' lets you specify a color for a bar dependent on the fulfilment of a certain condition. The following example script renders the inside and outside bars in different colors:

```
study("barcolor example", overlay=true)
isUp() => close > open
isDown() => close <= open
isOutsideUp() => high > high[1] and low < low[1] and isUp()
isOutsideDown() => high > high[1] and low < low[1] and isDown()
isInside() => high < high[1] and low > low[1]
barcolor(isInside() ? yellow : isOutsideUp() ? aqua : isOutsideDown()
? purple : na)
```

38

As you can see, when passing the na value, the colors stay the default chart color.

Background coloring − 'bgcolor'

Similar to the barcolor function, the bgcolor function changes the color of the background. Function will the color of that can be calculated in an expression, and an optional parameter transp – transparency from 0-100, which is 90 by default.

As an example, here's a script for coloring trading sessions (use it on EURUSD, 30 min resolution):

```
study("bgcolor example", overlay=true)
timeinrange(res, sess) => time(res, sess) != 0
premarket = #0050FF
regular = #0000FF
postmarket = #5000FF
notrading = na
sessioncolor = timeinrange("30", "0400-0930") ? premarket :
timeinrange("30", "0930-1600") ? regular : timeinrange("30",
"1600-2000") ? postmarket : notrading
bgcolor(sessioncolor, transp=75)
```

Inputs of the Indicator

'input' annotations make it possible to indicate which variables in the indicator's code are incoming. Widgets will be generated for the variables on the indicator's (properties/attributes) page in order to change the values via a more convenient way than modifying the script's source code. You can also specify the title of the input in the form of a short text string. The title is meant to explain the purpose of the input, and you can specify lowest and highest possible values for numerical inputs.

When the document is written, in Pine there are the following types of inputs:
- bool,
- integer,
- float,
- string,
- symbol,
- resolution,
- session,
- source.

The following examples show how to create, in code, each input and how its widgets look like.

```
b = input(title="On/Off", type=bool, defval=true)
plot(b ? open : na)
```

```
i = input(title="Offset", type=integer, defval=7, minval=-10,
maxval=10)
plot(offset(close, i))
```



```
f = input(title="Angle", type=float, defval=-0.5, minval=-3.14,
maxval=3.14, step=0.2)
plot(sin(f) > 0 ? close : open)
```



```
sym = input(title="Symbol", type=symbol, defval="SPY")
res = input(title="Resolution", type=resolution, defval="60")
plot(close, color=red)
plot(security(sym, res, close), color=green)
```


The input widget 'symbol' has a built-in symbol 'search' which is turned on automatically when the ticker's first symbols are entered.

```
s = input(title="Session", type=session, defval="24x7")
plot(time(period, s))
```



```
src = input(title="Source", type=source, defval=close)
ma = sma(src, 9)
plot(ma)
```

Find more information about indicator inputs in [Pine Reference](#).

Price levels 'hline'

The annotation function 'hline' renders a horizontal line at a given fixed price level. For example:

```
study(title="Chaikin Oscillator", shorttitle="Chaikin Osc")
short = input(3,minval=1), long = input(10,minval=1)
osc = ema(accdist, short) - ema(accdist, long)
plot(osc, color=red)
hline(0, title="Zero", color=gray, linestyle=dashed)
```


A number must be the first argument of 'hline'. Values of a type series are forbidden.
It's possible to create a few horizontal lines with the help of 'hline' and fill in the background between them with a translucent light using the function 'fill'.

Filling in the background between objects with 'fill'

The 'fill' annotation function lets you color the background between two series, or two horizontal lines (created with hline). The following example illustrates how it works:

```
study("fill Example")
p1 = plot(sin(high))
p2 = plot(cos(low))
p3 = plot(sin(close))
fill(p1, p3, color=red)
fill(p2, p3, color=blue)
h1 = hline(0)
h2 = hline(1.0)
h3 = hline(0.5)
h4 = hline(1.5)
fill(h1, h2, color=yellow)
fill(h3, h4, color=lime)
```


Footnote: Never execute a fill between 'plot' and 'hline'. However it's possible to display, with the help of 'plot', a series of the identical values (which will look like a horizontal line, similar to 'hline') and execute a fill between it and another plot. For example:

```
study("Fill example 2")
src = close, len = 10
ma = sma(src, len)
osc = 100 * (ma - src) / ma
p = plot(osc)
// NOTE: fill(p, hline(0)) wouldn't work, instead use this:
fill(p, plot(0))
```

Alert conditions

The annotation function [alertcondition](#) allows you to create custom alert conditions in Pine studies.
The function has the following signature:

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

'condition' is a series of boolean values that is used for alert. Available values: true, false. True means alert condition is met, alert should trigger. False means alert condition is not met, alert should not trigger. It is a required argument.
'tiile' is an optional argument that sets the name of the alert condition.
'message' is an optional argument that specifies text message to display when the alert fires.

Here is example of creating an alert condition:

```
//@version=2
study("Example of alertcondition")
src = input(close)
ma_1 = sma(src, 20)
ma_2 = sma(src, 10)
c = cross(ma_1, ma_2)
alertcondition(c, title='Red crosses blue', message='Red and blue
have crossed!')
plot(ma_1, color=red)
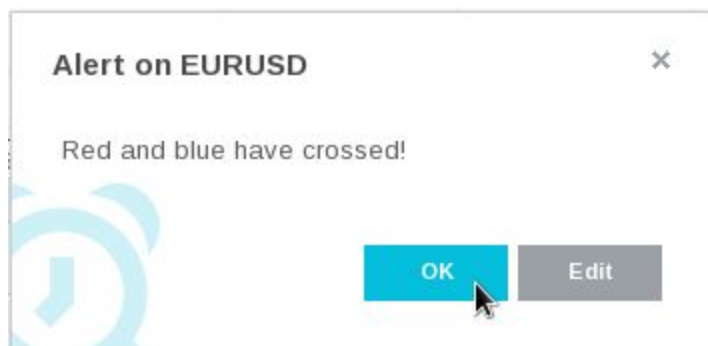plot(ma_2, color=blue)
```

The function creates alert condition that is available in Create Alert dialog. Please note, that alertcondition does NOT fire alerts from code automatically, it only gives you opportunity to create a custom condition for Create Alert dialog. Alerts must be still set manually. Also, an alert triggered based on a custom condition you created in Pine code is not displayed on a chart.

One script may include more than one alertcondition.

To create an alert based on alertcondition, one should apply a Pine code (study, strategy) with alertcontidion to current chart, open the Create Alert dialog, select the applied Pine code as main condition for the alert and choose the specific alert condition (implemented in the code itself).

When alert fires, you'll see the message:

Alert on EURUSD ✕

Red and blue have crossed!

OK | Edit

Shapes using plotshape, plotchar, plotarrow

In situations when you need to mark, highlight bars on a chart and not draw a usual plot in the form of a line, the function 'plot' may not be enough. Although it's possible to execute this kind of task by applying 'plot' in combination with the styles 'style=circles' or 'style=cross', it's recommended to use the functions 'plotshape', 'plotchar', 'plotarrow'.

The Function 'plotshape'

The function 'plotshape' is designed for displaying, under or above bars, a few icon-shapes. For example, the script below will draw an 'X' above all green bars:

```
study('plotshape example 1', overlay=true)
data = close >= open
plotshape(data, style=shape.xcross)
```

The first parameter, 'data', is considered as a series of logical values. The crosses are drawn on each true value and not on false values or 'na' values. It's possible to transfer a series of logical values in 'plotshape' to any series of numbers. It's possible to transfer any series of numbers to 'plotshape'. A '0' or 'na' is considered a false value, any other value is considered true.

By changing the value of the parameter 'style' is possible to change the form of the icon-shape. The available styles are:

Style Name	Display
shape.xcross	✕
shape.cross	✚
shape.circle	●
shape.triangleup	▲
shape.triangledown	▼
shape.flag	⚑
shape.arrowup	⬆
shape.arrowdown	⬇
shape.square	■
shape.diamond	◆
shape.labelup	⬆
shape.labeldown	⬇

The function 'plotshape' draws, by default, icons/shapes above bars.To set another position for the shapes, it's possible with the parameter 'location'. For example, the following script draws a green arrow 'shape.triangleup' above green bars and a red arrow 'shape.triangledown' above red bars:

```
study('plotshape example 2', overlay=true)
data = close >= open
plotshape(data, style=shape.triangleup,
          location=location.abovebar, color=green)
plotshape(not data, style=shape.triangledown,
          location=location.belowbar, color=red)
```


Possible values for the parameter 'location':
- location.abovebar — above a bar
- location.belowbar — below a bar
- location.top — top of a chart
- location.bottom — bottom of a chart
- location.absolute — any set/specified point on a chart

The value 'location.absolute' should be applied when the shapes need to be applied on the chart without being linked to the chart's bars or edges. A value of the first parameter of the function 'plotshape' is used as a 'y' coordinate. As such, the shape is displayed on null values and not only on the 'na' values.

The example 'plotshape example 2' also illustrates that the shapes' color can be set by the parameter 'color,' which can accept more than constant values of color constants.

Similar to the parameter 'color' of the function 'plot', it's possible to transfer expressions which will calculate the icon-shapes' color depending on conditions. For example:

```
study('plotshape example 3', overlay=true)
```

48

```
data = close >= open
plotshape(true, style=shape.flag, color=data ? green : red)
```


In the given example, the first parameter of the function 'plotshape' is equal to 'true' which means that the shape will be displayed on each bar. The color will be set by the condition: color=data ? green : red

The function 'plotshape' has other possibilities:
- Set the name of a displayed series of data using the parameter 'title'
- Shift a series of shapes to the left/right using the parameter 'offset'
- Set the transparency of shapes by the parameter 'transp'
- Parameter 'text' to display some short text above/below the shape. You may use '\n' to separate text lines

Function 'plotchar'

Plotchar's primary difference from 'plotshape' is in the way it assigns icon-shapes. In plotchar, it is set through the inline parameter 'char', allowing any encoding unicode symbol to be used (which are supported by the in-use font). For example:

```
study('plotchar example', overlay=true)
data = close >= open
plotchar(data, char='a')
```

By default, the parameter char accepts the value ★ ('BLACK STAR', U+2605). It's possible to use any letters, digits or various symbols, for example: ❤, ☀, €, ⚑, ❄, ◆, ↑, ↓.

Example of 'snowflakes' ❄:

```
study('plotchar example', overlay=true)
data = close >= open
plotchar(data, char='❄')
```


Like 'plotshape', the function 'plotchar' allows:
- Set a shape's color, with a constant or complex arithmetic expression
- Set a shape's location, the parameter 'location'

- Set the name of a displayed series of data using the parameter 'title'
- Shift a series of shapes left/right using the parameter 'offset'
- Set the transparency of shapes using the parameter 'transp'
- Parameter 'text' to display some short text above/below the shape. You may use '\n' to separate text lines

The Function 'plotarrow'

The function 'plotarrow' allows for up/down arrows to be displayed on the chart. The arrows' lengths are not the same on each bar and are calculated by the script code (depending on the conditions calculated).

The first series parameter of the function 'plotarrow' is used to place arrows on the chart, using the following logic:

- If a value series on the current bar is greater than 0, then an up arrow will be drawn, the length of the arrow proportionally to an absolute value.
- If a value series on the current bar is less than 0, then a down arrow will be drawn, the length of the arrow proportional to an absolute value.
- If a value series on the current bar is equal to 0 or 'na' then the arrow is not displayed.

Here is a simple script that illustrates how 'plotarrow' function works:

```
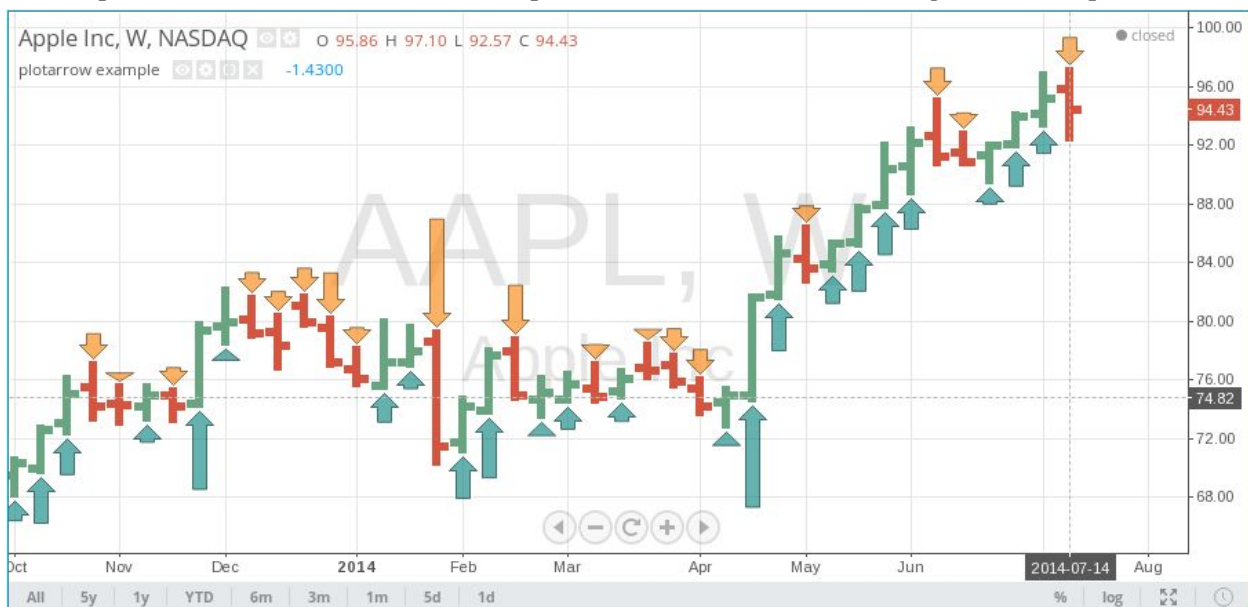study("plotarrow example", overlay=true)
codiff = close - open
plotarrow(codiff, colorup=teal, colordown=orange, transp=40)
```


As you can see, the more absolute value of the difference 'close - open' the longer the arrow. If 'close - open' is greater than zero, then an up arrow is rendered, otherwise (when 'close - open' is less than zero) we have a down arrow.

For another example, it's possible to take the indicator "Chaikin Oscillator" from the standard scripts and display it as an overlay above a series in the form of arrows using 'plotarrow' :

```
study("Chaikin Oscillator Arrows", overlay=true)
short = input(3,minval=1), long = input(10,minval=1)
osc = ema(accdist, short) - ema(accdist, long)
plotarrow(osc)
```


This screenshot shows the original "Chaikin Oscillator" alongside the script for better understanding.

As was stated earlier, the high of the arrow is chosen proportionally to the absolute value of the first series parameter of the function 'plotarrow'. The maximum and minimum possible sizes for the arrows (in pixels) are set by the parameters 'minheight' and 'maxheight' respectively.

Additionally, the function 'plotarrow' allows:
- Set the name of a displayed series of data using the parameter 'title'
- Set the color of an up arrow, parameter using 'colorup'
- Set the color of a down arrow and parameter using 'colordown'
- Shift a series of arrows left/right using the parameter 'offset'
- Set the transparency of shapes with the parameter 'transp'

It's important to note that 'colorup' and 'colordown' should receive a constant value of the type 'color'. Using expressions for determining color (as is done in plot, plotshape, plotchar) is not allowed.

Custom OHLC bars and candles

You may define your own custom bars and candles in Pine scripts. There are functions plotbar and plotcandle for that purpose. Here is a small example:

```
study("Example 1")
plotbar(open, high, low, close)
```


The script "Example 1" simply replicates bars of the current symbol. Nothing outstanding. We can paint them with green and red colors:

```
study("Example 2")
palette = close >= open ? lime : red
plotbar(open, high, low, close, color=palette)
```

The "Example 2" illustrates 'color' argument, which could be given constant values as red, lime, "#FF9090", as well as expressions that calculate color ('palette' variable) in runtime.

Function 'plotcandle' is similar to 'plotbar', it just plots candles instead of bars and have optional argument 'wickcolor'.

Both 'plotbar' and 'plotcandle' need four series arguments that will be used as bar/candle OHLC prices correspondingly. If for example one of the OHLC variables at some bar have NaN value, then the whole bar is not plotted. Example:

```
study("Example 3")
c = close > open ? na : close
plotcandle(open, high, low, c)
```

Of course you may calculate OHLC values without using available 'open', 'high', 'low' and 'close' values. For example you can plot "smoothed" candles:

```
study("Example 4")
len = input(9)
smooth(x) =>
    sma(x, len)
o = smooth(open)
h = smooth(high)
l = smooth(low)
c = smooth(close)
plotcandle(o, h, l, c)
```

You may get an interesting effect, if you plot OHLC values taken from higher timeframe. Let's say you want to plot daily bars on 60 minute chart:

```
// NOTE: add this script on intraday chart
study("Example 5")
higherRes = input("D", type=resolution)
is_newbar(res) =>
    t = time(res)
    change(t) != 0 ? 1 : 0
o = security(tickerid, higherRes, open)
h = security(tickerid, higherRes, high)
l = security(tickerid, higherRes, low)
c = security(tickerid, higherRes, close)
plotbar(is_newbar(higherRes) ? o : na, h, l, c, color=c >= o ? lime :
red)
```

Functions plotbar and plotcandle also have 'title' argument, so user can distinguish them in Styles tab of Format dialog.

Strategies

A **strategy** is a study that can send, modify and cancel orders (to buy/sell). Strategies allow you to perform **backtesting** (emulation of strategy trading on historical data) and **forwardtesting** (emulation of strategy trading on real-time data) according to your precoded algorithms.

A strategy written in Pine Script language has all the same capabilities as a Pine indicator. When you write a strategy code, it should start with the keyword "strategy", not "study". Strategies not only plot something, but also place, modify and cancel orders. They have access to essential strategy performance information through specific keywords. The same information is available for you on the "Strategy Tester" tab. Once a strategy is calculated on historical data, you can see hypothetical order fills.

A simple strategy example

```
//@version=2
strategy("test")
if n>4000
    strategy.entry("buy", strategy.long, 10,
                 when=strategy.position_size <= 0)
    strategy.entry("sell", strategy.short, 10,
                 when=strategy.position_size > 0)
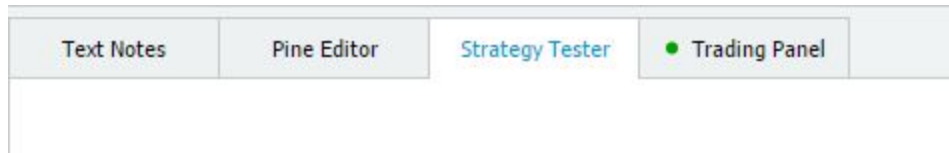plot(strategy.equity)
```

As soon as the script is compiled and applied to a chart, you can see filled order marks on it and how your balance was changing during backtesting (equity curve). It is a simplest strategy that buys and sells on every bar.

The first line **strategy("test")** states the code belongs to strategy type and its name is "test". **strategy.entry()** is a command to send "buy" and "sell" orders. **plot(strategy.equity)** plots the equity curve.

How to apply a strategy to the chart

To test your strategy, apply it to the chart. Use the symbol and time intervals that you want to test. You can use a built-in strategy from the "Indicators & Stratedies" dialog box, or write your own in Pine Editor.

Strategy perfomance report is displayed on the Strategy Tester tab on the bottom toolbar:

Backtesting and forwardtesting

On TradingView strategies are calculated on all available historical data on the chart and automatically continue calculation when real-time data comes in.

Both during historical and real-time calculation, code is calculated on bar closes by default.

If this is forwardtesting, code calculates on every tick in real-time. To enable this, check off the option "Recalculate On Every Tick" in settings or do it in script itself: **strategy(..., calc_on_every_tick=true, ...)**.

You can set the strategy to perform additional calculation after an order is filled. For this you need to check off "Recalculate After Order filled" in settings or do it in script itself: **strategy(…, calc_on_order_fills=true, ...)**.

Broker emulator

There is a broker emulator on TradingView for testing strategies. Unlike real trading, the emulator fills orders only at chart prices, that is why an order can be filled only on next tick in forwardtesting and on next bar in backtesting (or later) after strategy calculated.

As stated above, in backtesting strategy is calculated on bar's close. The following logic is used to emulate order fills:

1. If opening price of bar is closer to highest price of the same bar, the broker emulator assumes that intrabar price was moving this way: Open → High → Low → Close.
2. If opening price of bar is closer to lowest price of the same bar, the broker emulator assumes that intrabar price was moving this way: Open → Low → High → Close.
3. Broker emulator assumes that there were no gaps inside bar, meaning all intrabar prices are available for order execution.
4. If the option "Recalculate On Every Tick" in strategy properties is enabled (or **strategy(..., calc_on_every_tick=true, ...)** is specified in script), code is still calculated only on bar's close, following the above logic.

Here is the strategy demonstrating how orders are filled by the broker emulator:
```
strategy("History SAW demo", overlay=true, pyramiding=100,
        calc_on_order_fills=true)
strategy.entry("LE", strategy.long)
```

This code is calculated once per bar by default, on its close, however there is an additional calculation as soon as an order is filled. That is why you can see 4 filled orders on every bar: 2 orders on open, 1 order on high and 1 order on low. This is backtesting. If it were in real-time, orders would be executed on every new tick.

It is also possible to emulate order queue. The setting is called "Verify Price For Limit Orders" and can be found in strategy properties or set in script itself: **strategy(... backtest_fill_limits_assumption=X, ...)**. The specified value = number of points/pips (minimum price movements), default value = 0. A limit order is filled if current price is better (higher for sell orders, lower for buy orders) for the specified number of points/pips. The execution price still matches the limit order price.

Example:
backtest_fill_limits_assumption = 1. Minimum price movement = 0.25.
A buy limit order is placed at price 12.50.

Current price is 12.50.

The order cannot be filled at current price only because backtest_fill_limits_assumption = 1. To fill the order the price must become 0.25*1 lower. The order is put in queue.

Assume that the next tick comes at price 12.00. This price is 2 points lower, what means the condition "backtest_fill_limits_assumption = 1" is satisfied, so the order should be filled. The order is filled at 12.50 (original order price), even if the price is not available anymore.

Order placement commands

All keywords that are designed for strategies start with "**strategy.**" prefix. The following commands are used for placing orders: **strategy.entry**, **strategy.order** and **strategy.exit**:

- **strategy.entry** — this command places only entry orders. It is affected by pyramiding setting (is strategy properties) and by **strategy.risk.allow_entry_in** keyword. If there is an open market position when an opposite direction order is generated, the number of contracts/shares/lots/units will be increased by the number of currently open contracts (script equivalent: **strategy.position_size** + **quantity**). As the result, the size of market position to open will be equal to order size, specified in the command **strategy.entry**.
- **strategy.order** — this command places both entry and exit orders. It is not affected by pyramiding setting and by **strategy.risk.allow_entry_in** keyword. It allows you to create complex enter and exit order constructions when capabilities of the **strategy.entry** and **strategy.exit** are not enough.
- **strategy.exit** — this command allows you to exit a market position by an order or or form multiple exit order strategy (stop loss, profit target, trailing stop). All such orders are part of the same **strategy.oca.reduce** group. An exit order cannot be placed if there is no open market position or there is no active entry order (an exit order is bound to ID of an entry order). It is not possible to exit a position with a market order using the command **strategy.exit**. For this goal the following commands should be used: **strategy.close** or **strategy.close_all**. If number of contracts/shares/lots/units specified for the **strategy.exit** is less than the size of current open position, the exit will be partial. It is not possible to exit from the same entry order more than 1 time using the same exit order (ID), that allows you to create exit strategies with multiple levels. In case, when a market position was formed by multiple entry orders (pyramiding enabled), each exit orders is bound to each entry order individually.

Example 1:
```
//@version=2
strategy("revers demo")
if n > 4000
    strategy.entry("buy", strategy.long, 4, when=strategy.position_size <= 0)
    strategy.entry("sell", strategy.short, 6,
                   when=strategy.position_size > 0)
plot(strategy.equity)
```

The above strategy constantly reverses market position from +4 to -6, back and forth, what is shown by its plot.

Example 2:
```
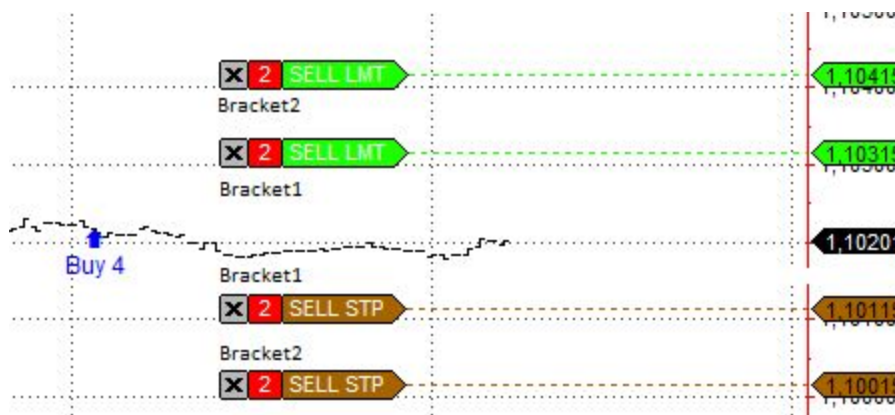strategy("exit once demo")
strategy.entry("buy", strategy.long, 4, when=strategy.position_size <= 0)
strategy.exit("bracket", "buy",  2, profit=10, stop=10)
```

This strategy demonstrates the case, when market position is never closed, because it uses exit order to close market position only partially and it cannot be used more than once. If you double the line for exiting, the strategy will close market position completely.

Example 3:
```
//@version=2
strategy("Partial exit demo")
if n > 4000
    strategy.entry("buy", strategy.long, 4, when=strategy.position_size <= 0)
strategy.exit("bracket1", "buy",  2, profit=10, stop=10)
strategy.exit("bracket2", "buy",  profit=20, stop=20)
```

This code generates 2 levels of brackets (2 take profit orders and 2 stop loss orders). Both levels are activated at the same time: first level to exit 2 contracts and the second one to exit all the rest.

The first take profit and stop loss orders (level 1) are in one OCA group. The other orders (level 2) are in another OCA group. It means that as soon as an order from level 1 is filled, the orders from level 2 are not cancelled, they stay active.

Every command placing an order has ID (string value) — unique order identifier. If an order with same ID is already placed (but not yet filled), current command modifies the existing order. If modification is not possible (conversion from buy to sell), the old order is cancelled, the new order is placed. **strategy.entry** and **strategy.order** work with the same IDs (they can modify the same entry order). **strategy.exit** works with other order IDs (it is possible to have an entry order and an exit order with the same ID).

To cancel a specific order (by its ID) the command **strategy.cancel(string id)** should be used. To cancel all pending orders the command **strategy.cancel_all()** should be used. Strategy orders are placed as soon as their conditions are satisfied and command is called in code. Broker emulator doesn't execute orders before next tick comes after the code was calculated, while in real trading with real broker, an order can be filled sooner. It means that if a market order is generated at close of current bar, it is filled at open oif next bar.

Example:
```
//@version=2
strategy("next bar open execution demo")
if n > 4000
    strategy.order("buy", strategy.long, when=strategy.position_size == 0)
    strategy.order("sell", strategy.short, when=strategy.position_size != 0)
```

If this code is applied to a chart, all orders are filled at open of every bar.

Conditions for order placement (**when**, pyramiding, **strategy.risk**) are checked when script is calculated. If all conditions are satisfied, the order is placed. If any condition is not satisfied, the order is not placed. It is important to cancel price orders (limit, stop and stop-limit orders).

Example (for MSFT 1D):
```
//@version=2
strategy("Priced Entry demo")
c = year > 2014 ? nz(c[1]) + 1 : 0
if c == 1
    strategy.entry("LE1", strategy.long, 2,
                   stop = high + 35 * syminfo.mintick)
    strategy.entry("LE2", strategy.long, 2,
                   stop = high + 2 * syminfo.mintick)
```

Even though pyramiding is disabled, these both orders are filled in backtesting, because when they are generated there is no open long market position. Both orders are placed and when price satisfies order execution, they both get executed. It is recommended to to put the orders in 1 OCA group by means of **strategy.oca.cancel**. in this case only one order is filled and the other one is cancelled. Here is the modified code:
```
//@version=2
strategy("Priced Entry demo")
c = year > 2014 ? nz(c[1]) + 1 : 0
if c == 1
    strategy.entry("LE1", strategy.long, 2,
                   stop = high + 35 * syminfo.mintick,
                   oca_type = strategy.oca.cancel, oca_name = "LE")
    strategy.entry("LE2", strategy.long, 2,
                   stop = high + 2 * syminfo.mintick,
```

```
                    oca_type = strategy.oca.cancel, oca_name = "LE")
```

If, for some reason, order placing conditions are not met when executing the command, the entry order will not be placed. For example, if pyramiding settings are set to 2, existing position already contains two entries and the strategy tries to place a third one, it will not be placed. Entry conditions are evaluated at the order generation stage and not at the execution stage. Therefore, if you submit two price type entries with pyramiding disabled, once one of them is executed the other will not be cancelled automatically. To avoid issues we recommend using OCA-Cancel groups for entries so when one entry order is filled the others are cancelled.

The same is true for price type exits - orders will be placed once their conditions are met (i.e.an entry order with the respective id is filled).

Example:
```
strategy("order place demo")
counter = nz(counter[1]) + 1
strategy.exit("bracket", "buy", profit=10, stop=10, when = counter == 1)
strategy.entry("buy", strategy.long, when=counter > 2)
```

If you apply this example to a chart, you can see that the exit order has been filled despite the fact that it had been generated only once before the entry order to be closed was placed. However, the next entry was not closed before the end of the calculation as the exit command has already been triggered.

Closing market position

Despite it is possible to exit from a specific entry in code, when orders are shown in the List of Trades on StrategyTester tab, they all are linked according FIFO (first in, first out) rule. If an entry order ID is not specified for an exit order in code, the exit order closes the first entry order that opened market position. Let's study the following example:

```
strategy("exit Demo", pyramiding=2, overlay=true)
strategy.entry("Buy1", strategy.long, 5,
               when = strategy.position_size == 0 and year > 2014)
strategy.entry("Buy2", strategy.long,
               10, stop = strategy.position_avg_price +
               strategy.position_avg_price*0.1,
               when = strategy.position_size == 5)
strategy.exit("bracket", loss=10, profit=10,
               when=strategy.position_size == 15)
```

The code given above places 2 orders sequentially: "Buy1" at market price and "Buy2" at 10% higher price (stop order). Exit order is placed only after entry orders have been filled. If you apply the code to a chart, you will see that each entry order is closed by exit order, though we

did not specify entry order ID to close in this line: `strategy.exit("bracket", loss=10,`
`profit=10, when=strategy.position_size == 15)`

Another example:
```
strategy("exit Demo", pyramiding=2, overlay=true)
strategy.entry("Buy1", strategy.long, 5, when = strategy.position_size == 0)
strategy.entry("Buy2", strategy.long,
               10, stop = strategy.position_avg_price +
               strategy.position_avg_price*0.1,
               when = strategy.position_size == 5)
strategy.close("Buy2",when=strategy.position_size == 15)
strategy.exit("bracket", "Buy1", loss=10, profit=10,
               when=strategy.position_size == 15)
plot(strategy.position_avg_price)
```

- It opens 5 contracts long position with the order "Buy1".
- It extends the long position by purchasing 10 more contracts at 10% higher price with the order "Buy2".
- The exit order (strategy.close) to sell 10 contracts (exit from "Buy2") is filled.

If you take a look at the plot, you can see that average entry price = "Buy2" execution price and our strategy closed exactly this entry order, while on the TradeList tab we can see that it closed the first "Buy1" order and half of the second "Buy2". It means that the no matter what entry order you specify for your strategy to close, the broker emulator will still close the the first one (according to FIFO rule). It works the same way when trading with through broker.

OCA groups
It is possible to put orders in 2 different OCA groups in Pine Script:
- **strategy.oca.cancel** - as soon as an order from group is filled (even partially) or cancelled, the other orders from the same group get cancelled. One should keep in mind that if order prices are the same or they are close, more than 1 order of the same group may be filled. This OCA group type is available only for entry orders because all exit orders are placed in **strategy.oca.reduce**.

Example:
```
//@version=2
strategy("oca_cancel demo")
if year > 2014 and year < 2016
    strategy.entry("LE", strategy.long, oca_type = strategy.oca.cancel,
                   oca_name="Entry")
    strategy.entry("SE", strategy.short, oca_type = strategy.oca.cancel,
                   oca_name="Entry")
```

You may think that this is a reverse strategy since pyramiding is not allowed, but in fact both order will get filled because they are market order, what means they are to be executed immediately at current price. The second order doesn't get cancelled because both are filled almost at the same moment and the system doesn't have time to process first order fill and cancel the second one before it gets executed. The same would happen if these were price orders with same or similar prices. Strategy places all orders (which are allowed according to market position, etc).

The strategy places all orders that do not contradict the rules (in our case market position is flat, therefore any entry order can be filled). At each tick calculation, firstly all orders with the satisfied conditions are executed and only then the orders from the group where an order was executed are cancelled.

- **strategy.oca.reduce** - this group type allows multiple orders within the group to be filled. As one of the orders within the group starts to be filled, the size of other orders is reduced by the filled contracts amount. It is very useful for the exit strategies. Once the price touches your take-profit order and it is being filled, the stop-loss is not cancelled but its amount is reduced by the filled contracts amount, thus protecting the rest of the open position.
- **strategy.oca.none** - the order is placed outside of the group (default value for the **strategy.order** and **strategy.entry** commands).

Every group has its own unique id (the same way as the orders have). If two groups have the same id, but different type, they will be considered different groups. Example:

```
//@version=2
strategy("My Script")
if year > 2014 and year < 2016
    strategy.entry("Buy", strategy.long, oca_name="My oca",
                oca_type=strategy.oca.reduce)
    strategy.exit("FromBy", "Buy", profit=100, loss=200, oca_name="My oca")
    strategy.entry("Sell", strategy.short, oca_name="My oca",
                oca_type=strategy.oca.cancel)
    strategy.order("Order", strategy.short, oca_name="My oca",
                oca_type=strategy.oca.none)
```

"Buy" and "Sell" will be placed in different groups as their type is different. "Order" will be outside of any group as its type is set to **strategy.oca.none.** Moreover, "Buy" will be placed in the exit group as exits are always placed in the **strategy.oca.reduce_size** type group.

Risk Management

It is not easy to create a universal profitable strategy. Usually, strategies are created for certain market patterns and can produce uncontrollable losses when applied to other data. Therefore stopping auto trading in time should things go bad is a serious issue. There is a special group of strategy commands to manage risks. They all start with the **strategy.risk.*** prefix.

You can combine any number of risks in any combination within one strategy. Every risk category command is calculated at every tick as well as at every order execution event regardless of the **calc_on_order_fills** strategy setting. There is no way to disable any risk rule in runtime from script. Regardless of where in the script the risk rule is located it will always be applied unless the line with the rule is deleted and the script is recompiled.

If on the next calculation any of the rules is triggered, no orders will be sent. Therefore if a strategy has several rules of the same type with different parameters, it will stop calculating when the rule with the most strict parameters is triggered. When a strategy is stopped all unexecuted orders are cancelled and then a market order is sent to close the position if it is not flat.

Furthermore, it is worth remembering that when using resolutions higher than 1 day, the whole bar is considered to be 1 day for the rules starting with prefix "**strategy.risk.max_intraday_**"

Example (MSFT 1):
```
//@version=2
strategy("multi risk demo", overlay=true, pyramiding=10,
        calc_on_order_fills = true)
if year > 2014
    strategy.entry("LE", strategy.long)
strategy.risk.max_intraday_filled_orders(5)
strategy.risk.max_intraday_filled_orders(2)
```

The position will be closed and trading will be stopped until the end of every trading session after two orders are executed within this session as the second rule is triggered earlier and is valid until the end of the trading session.

One should remember that the **strategy.risk.allow_entry_in** rule is applied to entries only so it will be possible to enter in a trade using the **strategy.order** command as this command is not an entry command per se. Moreover, when the **strategy.risk.allow_entry_in** rule is active, entries in a "prohibited trade" become exits instead of reverse trades.

Example (MSFT 1D):
```
//@version=2
strategy("allow_entry_in demo", overlay=true)
if year > 2014
    strategy.entry("LE", strategy.long, when=strategy.position_size <= 0)
    strategy.entry("SE", strategy.short, when=strategy.position_size > 0)
strategy.risk.allow_entry_in(strategy.direction.long)
```

As short entries are prohibited by the risk rules, instead of reverse trades long exit trades will be made.

Currency

TradingView strategies can operate in the currency different from the instrument currency. NetProfit and OpenProfit are recalculated in the account currency. Account currency is set in the strategy properties - the **Base Currency** drop-down list or in the script via the **strategy(..., currency=currency.*, ...)** keyword. At the same time, performance report values are calculated in the selected currency.

Trade profit (open or closed) is calculated based on the profit in the instrument currency multiplied by the cross-rate on the Close of the trading day previous to the bar where the strategy is calculated.

Example: we trade EURUSD, D and have selected EUR as the strategy currency. Our strategy buys and exits the position using 1 point profitTarget or stopLoss.

```
//@version=2
strategy("Currency test", currency=currency.EUR)
if year > 2014
    strategy.entry("LE", true, 1000)
    strategy.exit("LX", "LE", profit=1, loss=1)
profit = strategy.netprofit
plot(abs((profit - profit[1])*100), "1 point profit",
     color=blue, linewidth=2)
plot(1 / close[1], "prev usdeur", color=red)
```

After adding this strategy to the chart we can see that the plot lines are matching. This demonstrates that the rate to calculate the profit for every trade was based on the close of the previous day.

When trading on intra-day resolutions the cross-rate on the close of the trading day previous to the bar where the strategy is calculated will be used and it will not be changed during whole trading session.

When trading on resolutions higher than 1 day the cross-rate on the close of the trading day previous to the close of the bar where the strategy is calculated will be used. Let's say we trade on a weekly chart, then the cross rate on Thursday's session close will always be used to calculate the profits.

In real-time the yesterday's session close rate is used.

HOWTOs

Get real OHLC price on a Heikin Ashi chart

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

```
//@version=2
study("Visible OHLC", overlay=true)
c = close
plot(c)
```

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

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

```
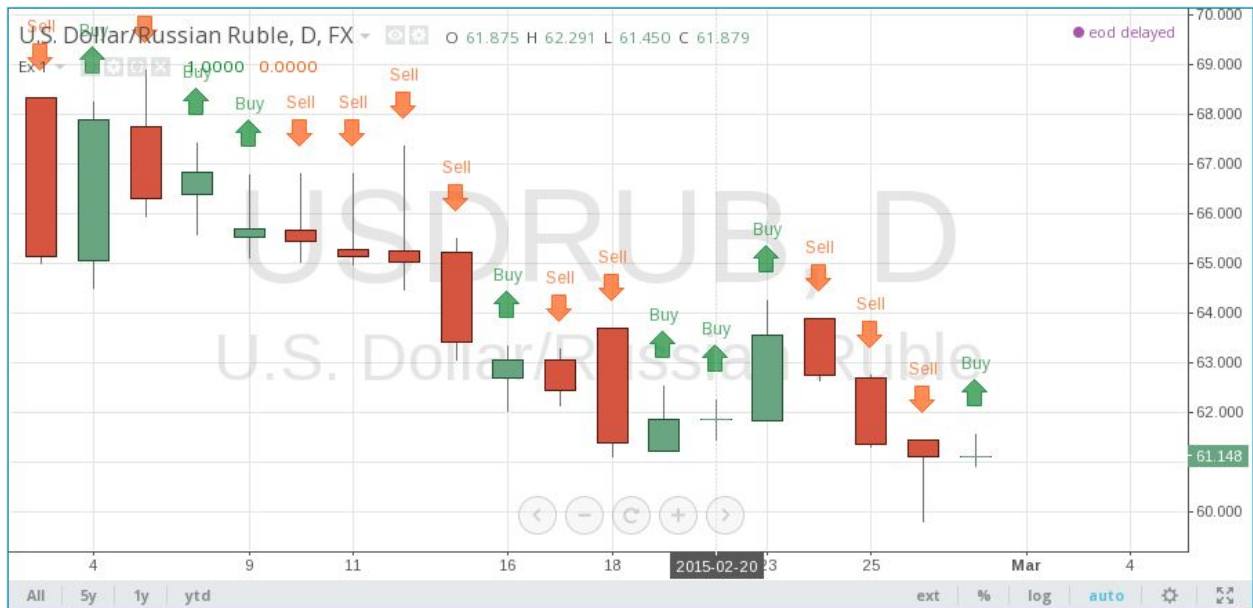//@version=2
study("Real OHLC", overlay=true)
t = tickerid(syminfo.prefix, ticker)
realC = security(t, period, close)
plot(realC)
```

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

Plot buy/sell arrows on the chart

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

```
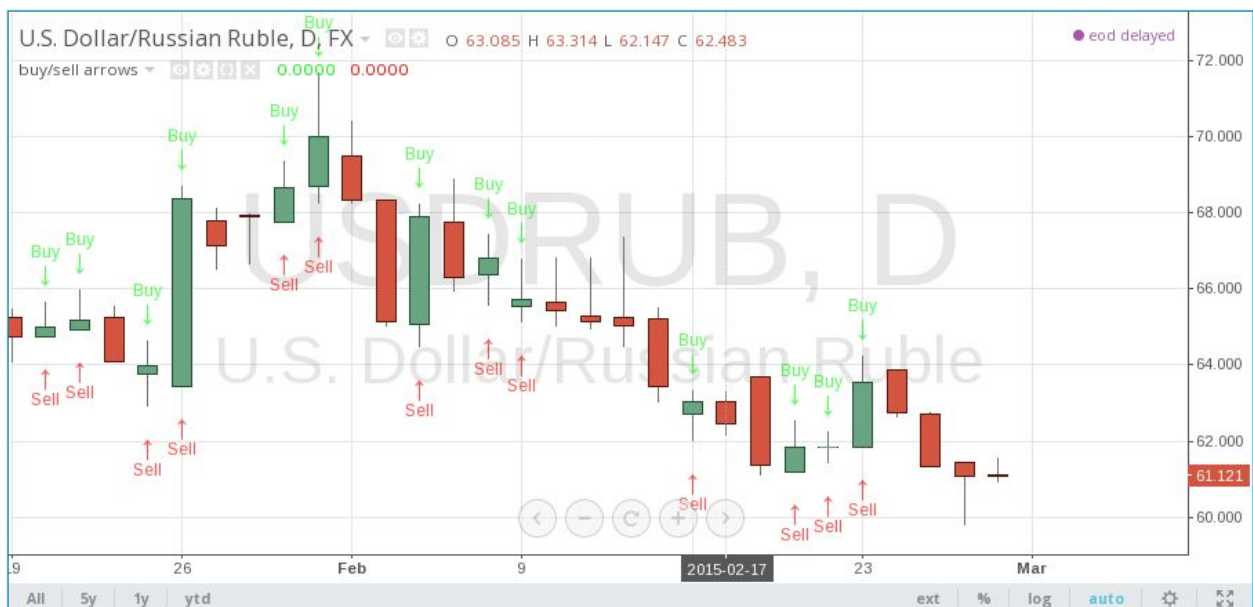study('Ex 1', overlay=true)
data = close >= open
plotshape(data, color=lime, style=shape.arrowup, text="Buy")
plotshape(not data, color=red, style=shape.arrowdown, text="Sell")
```

You may use plotchar function with any unicode character:

```
study('buy/sell arrows', overlay=true)
data = close >= open
plotchar(data, char='↓', color=lime, text="Buy")
plotchar(data, char='↑', location=location.belowbar, color=red,
text="Sell")
```

Where can I get more information?

First, additional descriptions of all built-in operators, variables, functions and annotation functions can be found here: https://www.tradingview.com/study-script-reference/

Second, write your own scripts by an example. Open new chart, push the 'Pine Script Editor' button on the toolbar. In the opened editor window click the 'New' button. You'll find lots of examples of TradingView standard indicators in the dropdown list.

Third, you can always ask questions on TradingView's support forum: https://getsatisfaction.com/tradingview

Finally, information about large releases and modifications to Pine Script (as well as other features) are regularly published in Trading View's blog: http://blog.tradingview.com