# Variable declarations

## Introduction

Variables are identifiers that hold values. They must be *declared* in your code before you use them. The syntax of variable declarations is:

```
[<declaration_mode>] [<type>] <identifier> = <expression> | <structure>
```

or

```
<tuple_declaration> = <function_call> | <structure>
```

where:

- `|` means "or", and parts enclosed in square brackets ( `[]` ) can appear zero or one time.
- <declaration_mode> is the variable's declaration mode. It can be var or varip, or nothing.
- <type> is optional, as in almost all Pine Script™ variable declarations (see types).
- <identifier> is the variable's name.
- <expression> can be a literal, a variable, an expression or a function call.
- <structure> can be an if, for, while or switch *structure*.
- <tuple_declaration> is a comma-separated list of variable names enclosed in square brackets ( `[]` ), e.g., `[ma, upperBand, lowerBand]` .

These are all valid variable declarations. The last one requires four lines:

```
BULL_COLOR = color.lime
i = 1
len = input(20, "Length")
float f = 10.5
closeRoundedToTick = math.round_to_mintick(close)
st = ta.supertrend(4, 14)
var barRange = float(na)
var firstBarOpen = open
varip float lastClose = na
[macdLine, signalLine, histLine] = ta.macd(close, 12, 26, 9)
plotColor = if close > open
    color.green
else
    color.red
```

**Note**
The above statements all contain the `=` assignment operator because they are **variable declarations**. When you see similar lines using the `:=` reassignment operator, the code is **reassigning** a value to a variable that was **already declared**. Those are **variable reassignments**. Be sure you understand the distinction as this is a common stumbling block for newcomers to Pine Script™. See the next Variable reassignment section for details.

The formal syntax of a variable declaration is:

```
<variable_declaration>
    [<declaration_mode>] [<type>] <identifier> = <expression> | <structure>
    |
    <tuple_declaration> = <function_call> | <structure>

<declaration_mode>
    var | varip

<type>
    int | float | bool | color | string | line | linefill | label | box | table | array
```

## Initialization with `na`

In most cases, an explicit type declaration is redundant because type is automatically inferred from the value on the right of the `=` at compile time, so the decision to use them is often a matter of preference. For example:

```
baseLine0 = na          // compile time error!
float baseLine1 = na    // OK
baseLine2 = float(na)   // OK
```

In the first line of the example, the compiler cannot determine the type of the `baseLine0` variable because na is a generic value of no particular type. The declaration of the `baseLine1` variable is correct because its float type is declared explicitly. The declaration of the `baseLine2` variable is also correct because its type can be derived from the expression `float(na)`, which is an explicit cast of the na value to the float type. The declarations of `baseLine1` and `baseLine2` are equivalent.

## Tuple declarations

Function calls or structures are allowed to return multiple values. When we call them and want to store the values they return, a *tuple declaration* must be used, which is a comma-separated set of one or more values enclosed in brackets. This allows us to declare multiple variables simultaneously. As an example, the ta.bb() built-in function for Bollinger bands returns three values:

```
[bbMiddle, bbUpper, bbLower] = ta.bb(close, 5, 4)
```

## Variable reassignment

A variable reassignment is done using the := reassignment operator. It can only be done after a variable has been first declared and given an initial value. Reassigning a new value to a variable is often necessary in calculations, and it is always necessary when a variable from the global scope must be assigned a new value from within a structure's local block, e.g.:

```
//@version=5
indicator("", "", true)
sensitivityInput = input.int(2, "Sensitivity", minval = 1, tooltip = "Higher values mal
ma = ta.sma(close, 20)
maUp = ta.rising(ma, sensitivityInput)
maDn = ta.falling(ma, sensitivityInput)

// On first bar only, initialize color to gray
var maColor = color.gray
if maUp
    // MA has risen for two bars in a row; make it lime.
    maColor := color.lime
else if maDn
    // MA has fallen for two bars in a row; make it fuchsia.
    maColor := color.fuchsia

plot(ma, "MA", maColor, 2)
```

Note that:

- We initialize `maColor` on the first bar only, so it preserves its value across bars.
- On every bar, the if statement checks if the MA has been rising or falling for the user-specified number of bars (the default is 2). When that happens, the value of `maColor` must be reassigned a new value from within the if local blocks. To do this, we use the := reassignment operator.
- If we did not use the := reassignment operator, the effect would be to initialize a new `maColor` local variable which would have the same name as that of the global scope, but actually be a very confusing independent entity that would persist only for the length of the local block, and then disappear without a trace.

All user-defined variables in Pine Script™ are *mutable*, which means their value can be changed using the := reassignment operator. Assigning a new value to a variable may change its *form* (see the page on Pine Script™'s type system for more information). A variable can be assigned a new value as many times as needed during the script's execution on one bar, so a script can contain any number of reassignments of one variable. A variable's declaration mode determines how new values assigned to a variable will be saved.

## Declaration modes

Understanding the impact that declaration modes have on the behavior of variables requires prior knowledge of Pine Script™'s execution model.

When you declare a variable, if a declaration mode is specified, it must come first. Three modes can be used:

- "On each bar", when none is specified

- var
- varip

## On each bar

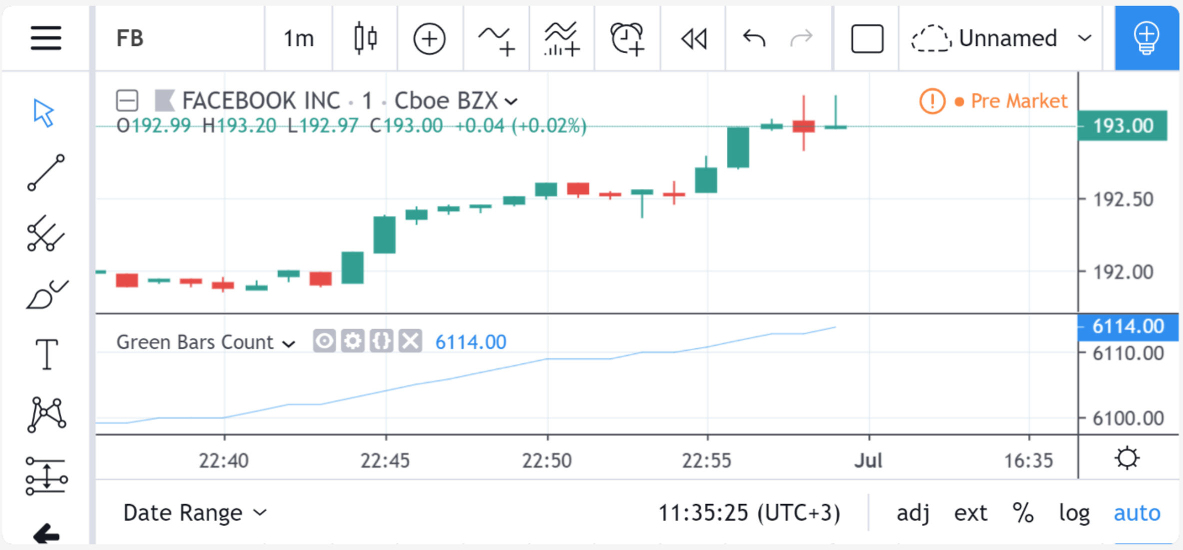
When no explicit declaration mode is specified, i.e. no var or varip keyword is used, the variable is declared and initialized on each bar, e.g., the following declarations from our first set of examples in this page's introduction:

```
BULL_COLOR = color.lime
i = 1
len = input(20, "Length")
float f = 10.5
closeRoundedToTick = math.round_to_mintick(close)
st = ta.supertrend(4, 14)
[macdLine, signalLine, histLine] = ta.macd(close, 12, 26, 9)
plotColor = if close > open
    color.green
else
    color.red
```

## `var`

When the var keyword is used, the variable is only initilized once, on the first bar if the declaration is in the global scope, or the first time the local block is executed if the declaration is inside a local block. After that, it will preserve its last value on successive bars, until we reassign a new value to it. This behavior is very useful in many cases where a variable's value must persist through the iterations of a script across successive bars. For example, suppose we'd like to count the number of green bars on the chart:

```
//@version=5
indicator("Green Bars Count")
var count = 0
isGreen = close >= open
if isGreen
    count := count + 1
plot(count)
```



Without the `var` modifier, variable `count` would be reset to zero (thus losing its value) every time a new bar update triggered a script recalculation.

Declaring variables on the first bar only is often useful to manage drawings more efficiently. Suppoose we want to extend the last bar's close line to the right of the right chart. We could write:

```
//@version=5
indicator("Inefficient version", "", true)
closeLine = line.new(bar_index - 1, close, bar_index, close, extend = extend.right, wi
line.delete(closeLine[1])
```

but this is inefficient because we are creating and deleting the line on each historical bar and on each update in the realtime bar. It is more efficient to use:

```
//@version=5
indicator("Efficient version", "", true)
var closeLine = line.new(bar_index - 1, close, bar_index, close, extend = extend.right,
if barstate.islast
    line.set_xy1(closeLine, bar_index - 1, close)
    line.set_xy2(closeLine, bar_index, close)
```

Note that:

- We initialize `closeLine` on the first bar only, using the var declaration mode
- We restrict the execution of the rest of our code to the chart's last bar by enclosing our code that updates the line in an if barstate.islast structure.

There is a very slight penalty performance for using the var declaration mode. For that reason, when declaring constants, it is preferable not to use var if performance is a concern, unless the initialization involves calculations that take longer than the maintenance penalty, e.g., functions with complex code or string manipulations.

## `varip`

Understanding the behavior of variables using the varip declaration mode requires prior knowledge of Pine Script™'s execution model and bar states.

The varip keyword can be used to declare variables that escape the *rollback process*, which is explained in the page on Pine Script™'s execution model.

Whereas scripts only execute once at the close of historical bars, when a script is running in realtime, it executes every time the chart's feed detects a price or volume update. At every realtime update, Pine Script™'s runtime normally resets the values of a script's variables to their last committed value, i.e., the value they held when the previous bar closed. This is generally handy, as each realtime script execution starts from a known state, which simplifies script logic.

Sometimes, however, script logic requires code to be able to save variable values **between different executions** in the realtime bar. Declaring variables with varip makes that possible. The "ip" in varip stands for *intrabar persist*.

Let's look at the following code, which does not use varip:

```
//@version=5
indicator("")
int updateNo = na
if barstate.isnew
    updateNo := 1
else
    updateNo := updateNo + 1

plot(updateNo, style = plot.style_circles)
```

On historical bars, barstate.isnew is always true, so the plot shows a value of “1” because the `else` part of the if structure is never executed. On realtime bars, barstate.isnew is only true when the script first executes on the bar’s “open”. The plot will then briefly display “1” until subsequent executions occur. On the next executions during the realtime bar, the second branch of the if statement is executed because barstate.isnew is no longer true. Since `updateNo` is initialized to na at each execution, the `updateNo + 1` expression yields na, so nothing is plotted on further realtime executions of the script.

If we now use varip to declare the `updateNo` variable, the script behaves very differently:

```
//@version=5
indicator("")
varip int updateNo = na
if barstate.isnew
    updateNo := 1
else
    updateNo := updateNo + 1

plot(updateNo, style = plot.style_circles)
```

The difference now is that `updateNo` tracks the number of realtime updates that occur on each realtime bar. This can happen because the varip declaration allows the value of `updateNo` to be preserved between realtime updates; it is no longer rolled back at each realtime execution of the script. The test on barstate.isnew allows us to reset the update count when a new realtime bar comes in.

Because varip only affects the behavior of your code in the realtime bar, it follows that backtest results on strategies designed using logic based on varip variables will not be able to reproduce that behavior on historical bars, which will invalidate test results on them. This also entails that plots on historical bars will not be able to reproduce the script’s behavior in realtime.

Operators Conditional structures

Options v: v5