Loops

Introduction

When loops are not needed

Pine Script™'s runtime and its built-in functions make loops unnecessary in many situations. Budding Pine Script™ programmers not yet familiar with the Pine Script™ runtime and built-ins who want to calculate the average of the

```
//@version=5
indicator("Inefficient MA", "", true)
MA_LENGTH = 10
sumOfCloses = 0.0
for offset = 0 to MA_LENGTH - 1
    sumOfCloses := sumOfCloses + close[offset]
inefficientMA = sumOfCloses / MA_LENGTH
plot(inefficientMA)
```

A for loop is unnecessary and inefficient to accomplish tasks like this in Pine Script™. This is how it should be done. This code is shorter *and* will run much faster because it does not use a loop and uses the ta.sma() built-in function to accomplish the task:

```
//@version=5
indicator("Efficient MA", "", true)
thePineMA = ta.sma(close, 10)
plot(thePineMA)
```

Counting the occurrences of a condition in the last bars is also a task which beginning Pine Script™ programmers often think must be done with a loop. To count the number of up bars in the last 10 bars, they will use:

```
//@version=5
indicator("Inefficient sum")
MA_LENGTH = 10
upBars = 0.0
for offset = 0 to MA_LENGTH - 1
    if close[offset] > open[offset]
        upBars := upBars + 1
plot(upBars)
```

The efficient way to write this in Pine Script™ (for the programmer because it saves time, to achieve the fastest-loading charts, and to share our common resources most equitably), is to use the math.sum() built-in function to accomplish the task:

```
//@version=5
indicator("Efficient sum")
upBars = math.sum(close > open ? 1 : 0, 10)
plot(upBars)
```

What's happening in there is:

- We use the ?: ternary operator to build an expression that yields 1 on up bars and 0 on other bars.
- We use the math.sum() built-in function to keep a running sum of that value for the last 10 bars.

When loops are necessary

Loops exist for good reason because even in Pine Script™, they are necessary in some cases. These cases typically include:

- The manipulation of arrays.
- Looking back in history to analyze bars using a reference value that can only be known on the current bar, e.g., to find how many past highs are higher than the high of the current bar. Since the current bar's high is only known on the bar the script is running on, a loop is necessary to go back in time and analyze past bars.
- Performing calculations on past bars that cannot be accomplished using Pine Script™'s built-in functions, like the Pearson correlation coefficient.

`for`

The for structure allows the repetitive execution of statements using a counter. Its syntax is:

```
[[<declaration_mode>] [<type>] <identifier> = ]for <identifier> = <expression> to <expr
    <local_block_loop>
```

where:

- Parts enclosed in square brackets (`[]`) can appear zero or one time, and those enclosed in curly braces (`{}`) can appear zero or more times.
- <declaration_mode> is the variable's declaration mode
- <type> is optional, as in almost all Pine Script™ variable declarations (see types)
- <identifier> is a variable's name
- <expression> can be a literal, a variable, an expression or a function call.
- <local_block_loop> consists of zero or more statements followed by a return value, which can be a tuple of values. It must be indented by four spaces or a tab. It can contain the `break` statement to exit the loop, or the `continue` statement to exit the current iteration and continue on with the next.

- The value assigned to the variable is the return value of the <local_block_loop>, i.e., the last value calculated on the loop's last iteration, or na if the loop is not executed.
- The identifier in `for <identifier>` is the loop's counter *initial value*.
- The expression in `= <expression>` is the *start value* of the counter.
- The expression in `to <expression>` is the *end value* of the counter. **It is only evaluated upon entry in the loop**.
- The expression in `by <expression>` is optional. It is the step by which the loop counter is increased or decreased on each iteration of the loop. Its default value is 1 when `start value < end value`. It is -1 when `start value > end value`. The step (+1 or -1) used as the default is determined by the start and end values.

This example uses a for statement to look back a user-defined amount of bars to determine how many bars have a high that is higher or lower than the high of the last bar on the chart. A for loop is necessary here, since the script only has access to the reference value on the chart's last bar. Pine Script™'s runtime cannot, here, be used to calculate on the fly, as the script is executing bar to bar:

```
//@version=5
indicator("`for` loop")
lookbackInput = input.int(50, "Lookback in bars", minval = 1, maxval = 4999)
higherBars = 0
lowerBars = 0
if barstate.islast
    var label lbl = label.new(na, na, "", style = label.style_label_left)
    for i = 1 to lookbackInput
        if high[i] > high
            higherBars += 1
        else if high[i] < high
            lowerBars += 1
    label.set_xy(lbl, bar_index, high)
    label.set_text(lbl, str.tostring(higherBars, "# higher bars\n") + str.tostring(lowe
```

This example uses a loop in its `checkLinesForBreaches()` function to go through an array of pivot lines and delete them when price crosses them. A loop is necessary here because all the lines in each of the `hiPivotLines` and `loPivotLines` arrays must be checked on each bar, and there is no Pine Script™ built-in that can do this for us:

```
//@version=5
MAX_LINES_COUNT = 100
indicator("Pivot line breaches", "", true, max_lines_count = MAX_LINES_COUNT)

color hiPivotColorInput  = input(color.new(color.lime, 0), "High pivots")
color loPivotColorInput  = input(color.new(color.fuchsia, 0), "Low pivots")
int   pivotLegsInput     = input.int(5, "Pivot legs")
int   qtyOfPivotsInput   = input.int(50, "Quantity of last pivots to remember", minval
int   maxLineLengthInput = input.int(400, "Maximum line length in bars", minval = 2)

// ————— Queues a new element in an array and de-queues its first element.
qDq(array, qtyOfElements, arrayElement) =>
    array.push(array, arrayElement)
    if array.size(array) > qtyOfElements
        // Only deqeue if array has reached capacity.
        array.shift(array)

// —————— Loop through an array of lines, extending those that price has not crossed an
checkLinesForBreaches(arrayOfLines) =>
    int qtyOfLines = array.size(arrayOfLines)
    // Don't loop in case there are no lines to check because "to" value will be `na` t
    for lineNo = 0 to (qtyOfLines > 0 ? qtyOfLines - 1 : na)
        // Need to check that array size still warrants a loop because we may have dele
        if lineNo < array.size(arrayOfLines)
            line  currentLine    = array.get(arrayOfLines, lineNo)
            float lineLevel      = line.get_price(currentLine, bar_index)
            bool  lineWasCrossed = math.sign(close[1] - lineLevel) != math.sign(close -
            bool  lineIsTooLong  = bar_index - line.get_x1(currentLine) > maxLineLength
            if lineWasCrossed or lineIsTooLong
                // Line stays on the chart but will no longer be extend on further bars
                array.remove(arrayOfLines, lineNo)
                // Force type of both local blocks to same type.
                int(na)
            else
                line.set_x2(currentLine, bar_index)
                int(na)

// Arrays of lines containing non-crossed pivot lines.
var line[] hiPivotLines = array.new_line(qtyOfPivotsInput)
var line[] loPivotLines = array.new_line(qtyOfPivotsInput)

// Detect new pivots.
float hiPivot = ta.pivothigh(pivotLegsInput, pivotLegsInput)
float loPivot = ta.pivotlow(pivotLegsInput, pivotLegsInput)

// Create new lines on new pivots.
if not na(hiPivot)
    line newLine = line.new(bar_index[pivotLegsInput], hiPivot, bar_index, hiPivot, col
    line.delete(qDq(hiPivotLines, qtyOfPivotsInput, newLine))
else if not na(loPivot)
    line newLine = line.new(bar_index[pivotLegsInput], loPivot, bar_index, loPivot, col
    line.delete(qDq(loPivotLines, qtyOfPivotsInput, newLine))

// Extend lines if they haven't been crossed by price.
checkLinesForBreaches(hiPivotLines)
checkLinesForBreaches(loPivotLines)
```

`while`

The while structure allows the repetitive execution of statements until a condition is false. Its syntax is:

```
[[<declaration_mode>] [<type>] <identifier> = ]while <expression>
    <local_block_loop>
```

where:

- Parts enclosed in square brackets (`[]`) can appear zero or one time.
- <declaration_mode> is the variable's declaration mode
- <type> is optional, as in almost all Pine Script™ variable declarations (see types)
- <identifier> is a variable's name
- <expression> can be a literal, a variable, an expression or a function call. It is evaluated at each iteration of the loop. When it evaluates to `true`, the loop executes. When it evaluates to `false` the loop stops. Note that evaluation of the expression is done before each iteration only. Changes to the expression's value inside the loop will only have an impact on the next iteration.
- <local_block_loop> consists of zero or more statements followed by a return value, which can be a tuple of values. It must be indented by four spaces or a tab. It can contain the `break` statement to exit the loop, or the `continue` statement to exit the current iteration and continue on with the next.
- The value assigned to the <identifier> variable is the return value of the <local_block_loop>, i.e., the last value calculated on the loop's last iteration, or na if the loop is not executed.

This is the first code example of the for section written using a while structure instead of a for one:

```
//@version=5
indicator("`for` loop")
lookbackInput = input.int(50, "Lookback in bars", minval = 1, maxval = 4999)
higherBars = 0
lowerBars = 0
if barstate.islast
    var label lbl = label.new(na, na, "", style = label.style_label_left)
    // Initialize the loop counter to its start value.
    i = 1
    // Loop until the `i` counter's value is <= the `lookbackInput` value.
    while i <= lookbackInput
        if high[i] > high
            higherBars += 1
        else if high[i] < high
            lowerBars += 1
        // Counter must be managed "manually".
        i += 1
    label.set_xy(lbl, bar_index, high)
    label.set_text(lbl, str.tostring(higherBars, "# higher bars\n") + str.tostring(lowe
```

Note that:

- The `i` counter must be incremented by one explicitly inside the while's local block.
- We use the += operator to add one to the counter. `lowerBars += 1` is equivalent to `lowerBars := lowerBars + 1`.

Let's calculate the factorial function using a while structure:

```
//@version=5
indicator("")
int n = input.int(10, "Factorial of", minval=0)

factorial(int val = na) =>
    int counter = val
    int fact = 1
    result = while counter > 0
        fact := fact * counter
        counter := counter - 1
        fact

// Only evaluate the function on the first bar.
var answer = factorial(n)
plot(answer)
```

Note that:

- We use input.int() for our input because we need to specify a `minval` value to protect our code. While input() also supports the input of “int” type values, it does not support the `minval` parameter.
- We have packaged our script’s functionality in a `factorial()` function which accepts as an argument the value whose factorial it must calculate. We have used `int val = na` to declare our function’s parameter, which says that if the function is called without an argument, as in `factorial()`, then the `val` parameter will initialize to na, which will prevent the execution of the while loop because its `counter > 0` expression will return na. The while structure will thus initialize the `result` variable to na. In turn, because the initialization of `result` is the return value of the our function’s local block, the function will return na.
- Note the last line of the while’s local block: `fact`. It is the local block’s return value, so the value it had on the while structure’s last iteration.
- Our initialization of `result` is not required; we do it for readability. We could just as well have used:

  ```
  while counter > 0
      fact := fact * counter
      counter := counter - 1
      fact
  ```