Learning Excel VBA

Using Loops in Your Code

Prepared By Daniel Lamarche

ComboProjects

ComboProjects
Understanding does matter

Using Loops in Your Code

By Daniel Lamarche (Last update June 2016).

Loops are pretty simple in concept however many new programmers find them difficult to tame. Like most beginner developer, the only way to master loops is to type 15 or 20 snippets of code involving loops of your own invention. This should do the trick.

What's a loop again?

The term Loop refers to a code structure that will be repeated a certain number of times. Some people will refer to individual occurrence of a loop as *iteration*.

VBA has essentially 2 types of loops. Each having variations on the theme of … looping. One thing that will help you to decide which type of loop to use is if you know in advance how many iterations the loop will do.

The Do While — Loop statement

A very commonly used type of loop is the Do While — Loop structure. Let's dive in and see the general look of a Do While — Loop.

```
Do While x < 10      ' Loop until x = 10 or as long as x is less than 10.
    x = x + 1
Loop
```

Explanation of each element

- **Do While** is the statement starting the loop. It's a bit like if someone would instruct you to 'take all their calls *while* they are in a meeting'.
- **x < 10** is a conditional statement that returns True or False. If the condition is True then the loop iterates one more time. When the condition becomes False the loop stops and execution goes to the statements after the Loop keyword.
- **x = x + 1** is the statement that will be executed with each iteration. In this case the variable 'x' is incremented by 1. Typically we call such operation a *Counter*.
- **Loop** instructs a re-evaluation of the condition at the top of the loop to see whether the condition is True (then run the loop again) or False (stop looping).

A Small Example

In a standard module type the Sub below. To indent hit the Tab key.

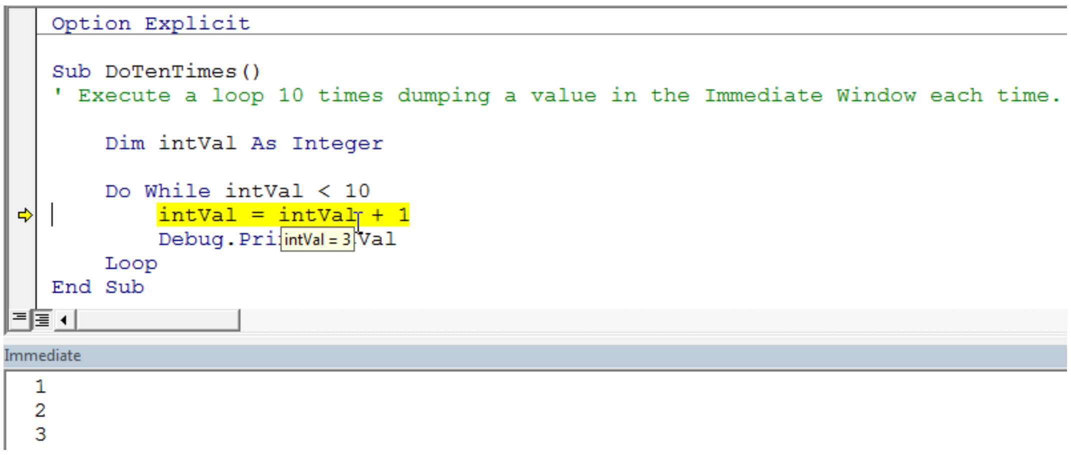
```
Sub DoTenTimes()
' Execute a loop 10 times dumping a value in the Immediate Window each time.

    Dim intVal As Integer

    Do While intVal < 10
        intVal = intVal + 1     ' Counter
        Debug.Print intVal      ' Print in the Immediate Window
    Loop
End Sub
```

Press Ctrl+G to open the Immediate Window if not already open. Move the insertion point anywhere in your Sub and hit F5 to run the Sub. The numbers 1 to 10 appear in the Immediate Window.

To see the Sub running in Break Mode (discussed in the *User Defined Functions* document) move the insertion point anywhere in the Sub then hit F8 a couple of times. Move the mouse pointer over the variable to see a screen tip displaying the current (or updated) value of the variable.



```
Option Explicit

Sub DoTenTimes()
' Execute a loop 10 times dumping a value in the Immediate Window each time.

    Dim intVal As Integer

    Do While intVal < 10
        intVal = intVal + 1
        Debug.Print intVal
    Loop
End Sub
```

```
1
2
3
```

Make sure you clearly understand (visually see) how does the loop know when to stop.

Another Example

The best thing to do now is to type another loop. This time one of the variables will *cumulate* instead of just counting.

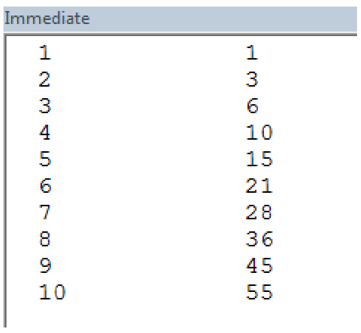
In a standard module type the Sub below. To indent hit the Tab key.

```
Sub CumulativeLoop()
' Execute a loop 10 times counting and cumulating each time.

    Dim intVal As Integer, intCumul As Integer

    Do While intVal < 10
        intVal = intVal + 1             ' Counter
        intCumul = intCumul + intVal    ' Cumulative
        Debug.Print intVal, intCumul    ' Print in the Immediate Window
    Loop
End Sub
```

Ensure the Immediate Window is open and empty. Move the insertion point anywhere in the CumulativeLoop procedure and hit F5 to run the Sub.



```
 1             1
 2             3
 3             6
 4             10
 5             15
 6             21
 7             28
 8             36
 9             45
 10            55
```

On the right is the result of running the loop. Do yourself a favour and set a break point on the Do While line (to add a break point press F9 while in that line) and run it again.

Step in your code (Hit F8) a couple of time and make *sure* you understand the difference the Counter (First column) and the Cumulative variables (Second column).

Change the value in the Do While line to something else and run it again. Before you proceed with the following material, you should be able to retype this last Sub by heart without consulting this handout.

Remove the break point (the burgundy line) by clicking the dot in the grey margin and save your work.

Looping Through Each Characters

As you might have figured out, using a Do While Loop requires that you have a mean of knowing the number of iterations *before* the loop starts. In the previous examples, it was obvious since *you* set the mark.

In this next example we will learn how to loop through each character of a string, print each character in the Immediate Window and report the number of character.

To make it more interesting the loop must omit spaces. For example using the string "Loops are essential" the sub will print 'Loopsareessential' in the Immediate Window and report that 17 characters were found. Ready?

```
Sub LoopInString(strString As String)
' Loop through each character and report information about the string.

    Dim strCurrentChar As String              ' Current char in the loop.
    Dim bytPos As Byte                        ' Current char position.
    Dim bytLength As Byte                     ' Total chars in original string.
    Dim bytCharOnly As Byte                   ' Count of chars only - no spaces.

    bytLength = Len(strString)                ' Total number of characters.
    Do While bytPos < bytLength               ' As long as bytPos < Length.
        bytPos = bytPos + 1                   ' Increment counter.
        ' Extract one character at a time.
        strCurrentChar = Mid(strString, bytPos, 1)
        If strCurrentChar <> " " Then         ' If current char not a space.
            Debug.Print strCurrentChar        ' Then print it.
            bytCharOnly = bytCharOnly + 1     ' Increment number of chars.
        End If
    Loop
    Debug.Print bytCharOnly                   ' Total chars only - no spaces.
End Sub
```

To be sure, this (rather dummy) example is an exercise in indentation, adding comments and understanding which variables are needed *before* we start typing the code.

Please be patient with yourself and type the procedure above. You don't absolutely need to type the comments although it would be good if you typed a couple of the important ones.

Before we explain…

Just before we briefly explain how the loop works type the following words on a piece of paper: "Quick Brown Fox" without the quotation marks. Below write the answer to each question. This is important! Start from the beginning and count slowly!

Count the number of character in the sentence: ______ (Of course you need to include spaces!)

Count the number of characters that are not spaces: ______

- To do this you had to use your pencil and move across one character at a time and count each character right?
- In the second question you counted again. This time IF it was a space you ignored it right?
- Bear with me one last time. Starting from the beginning write each character in a column counting each character except for spaces.

If your answer was 13 then you understood the loop above. The only VBA terms that may be new to you are Len and Mid.

Len	The Len VBA function returns the number of characters in a string (including spaces).
Mid	The Mid VBA function allow you to extract a specified number of characters from a string starting at a specified position. Here's how it works: Mid(String, Position, How Many Chars)

Example: Mid("Worksheet", 4, 1) would return k since k is the 4th character and you specified that you wanted only 1 character. If you don't specify the number of character to return it will return all the characters starting at that position to the end of the string.

Try answering these expressions:

```
Mid("Finance", 3, 3)       _______
Mid("Workbook", 5)         _______
```

Creating Twelve Sheets and Typing On Each Sheet

In the next segment of this handout we will learn how to create 12 worksheets in a workbook (From Jan to Dec) and for each sheet, in cell A1, the loop will write: "Sales for the month of" with the fully spelled out month matching the worksheet tab. For example cell A1 of the first sheet will read "Sales for the month of January".

Create a workbook and save it as a Macro Enabled workbook under the name **Create Month.xlsm**.

Before we start

We need to know how to create a worksheet to the right of the active sheet. Let's start by experimenting with the Sheets collection a bit.

Split your screen in two so that Excel takes the left half and the VBA screen takes the right half. Ensure that the workbook has only 1 worksheet named Sheet1. In the Immediate window type the following expressions pressing Enter each time and write your observations:

```
?ActiveSheet.Name                        ______________________________
?Sheets(1).Name                          ______________________________
?Sheets.Count                            ______________________________
Sheets.Add                               ______________________________
?Sheets.Count                            ______________________________
Sheets("Sheet1").Select                  ______________________________
Sheets(2).Name = MonthName(4)            ______________________________
Sheets.Add After:=Sheets(2)              ______________________________
Sheets(1).Delete                         ______________________________
```

In the section above we briefly learned how to interact with worksheets in VBA.

Delete all sheets except the first one and rename it "Summary". Type the following expression in the Immediate Window and write your observation on the line below:

```
Sheets.Add(After:= Sheets(Sheets.Count)).Name = MonthName(1, True)
```

____

Now replace the 1 after MonthName with a 2 and hit Enter. Do it again with a 3. Continue until you reach 6. You just created the worksheets Jan to Jun without touching Excel!

The 'True' parameter at the end indicates that you want the abbreviated name of the month. Omit that parameter (or type False instead) to have the fully spelled out month name.

Did you note what happened upon creating a new sheet? Each time you pressed Enter after the statement above which worksheet became the active sheet? This is very important and convenient too!

The next procedure will use a loop to create 12 worksheets in a workbook and cell A1 of each of sheet will contain a text entry. For example the first sheet will have: Sales for the month of January. Make sure you only have one worksheet named "Summary".

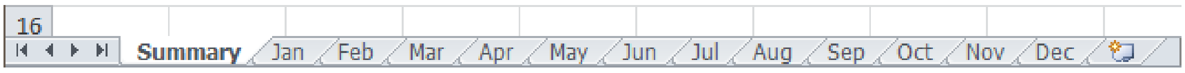
Type the procedure below in a module of your choice:

```
Sub CreateMonthSheets()
' Create 12 sheets from Jan to Dec.

    Dim bytMonth As Byte

    Application.ScreenUpdating = False      ' Freeze screen during execution.
    Do While bytMonth < 12
        bytMonth = bytMonth + 1             ' Increase month number each times.
        Sheets.Add(After:=Sheets(Sheets.Count)).Name = MonthName(bytMonth, True)
        With Range("A1")
            .Value = "Sales for the month of " & MonthName(bytMonth, False)
            .Font.Size = 16
        End With
    Loop
    Sheets("Summary").Select                ' Back to Summary sheet.
    Application.ScreenUpdating = True
End Sub
```

There you go! In 0.2 seconds!



Looping Until There Are No Entries Left

The last loop in this category will discuss how to move down column A until there are no more entries. The loop will stop automatically when there are no more cells to process. Type the following loop. We'll explain after.

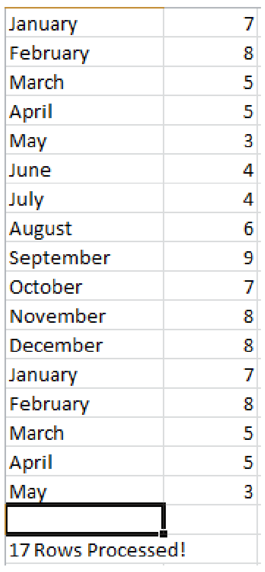
```
Sub LoopDown()
' Do something with each cell in Column A until it encounters a blank cell.

    Dim intCounter As Integer

    Range("A1").Select                      ' Start in cell A1
    Do While ActiveCell <> ""               ' Or:  Do While Not IsEmpty(ActiveCell)
        ActiveCell.Offset(0, 1) = Len(ActiveCell)
        ActiveCell.Offset(1, 0).Select      ' Move down one row
        intCounter = intCounter + 1
    Loop
    ActiveCell.Offset(1, 0) = intCounter & " Rows Processed!"
End Sub
```

Starting in A1 type January and drag the Fill Handle down a bit. About 15 cells will suffice. Then while the insertion point is somewhere in the sub hit F8 to step in the procedure.

The procedure is forced to start in cell A1 and writes in the cell next to each entry the number of characters in the current cell (called ActiveCell). The second line in the loop moves the active cell one row down.



January	7
February	8
March	5
April	5
May	3
June	4
July	4
August	6
September	9
October	7
November	8
December	8
January	7
February	8
March	5
April	5
May	3
17 Rows Processed!	

The condition for the loop to continue is that the active cell is *not* an empty string. So when the loop encounters a cell with no content the condition becomes False and the sub continues after the Loop keyword.

Optionnaly, we could have use the following systax:

```
Do While Not IsEmpty(ActiveCell)
    ...
Loop
```

The first time we encounter this type of expression we tend to read it twice. IsEmpty is a VBA function returning True or False whether the object refered to is indeed empty or not. If you read it: "Iterate as long as the ActiveCell is not Empty" it should help.

About the Offset property

The Offset property may be one of the most commonly used properties when coding Excel procedures. Fortunately it is extremely simple to use. Basically an offset is a distance from a known point. That distance is expressed in rows and columns from a known cell (here it's the active cell).

The syntax is *Range*.Offset(row, col) where a positive distance refers to a location across or down. A negative value refers to a location above or to the left.

You can use the Offset property to *refer* to a cell or to *do* something with that cell. For example:

intVal = ActiveCell.Offset(0, 1)	Stores the value of the cell immediately across in a variable.
ActiveCell.Offset(1, 0)=intVal	Writes in the cell immediately below the value of intVal.
Range("C7").Offset(3, 3).Select	Move the active cell to F10.

The For Each — Next statement

Another type of loop is the For Each — Next structure. Think of this statement as For Each *element* in a *collection* do something. After the loop goes through each element, it stops automatically.

Often you do not know how many elements there are in the collection. There is always a way you could find out but you may not bother. The classic use for a For Loop is with a selection of cells. This type of loop makes it extremely easy to visit each selected cell in a range and do whatever is required.

First Example of a For Each Loop

In a cell type "january" (in lowercase) and take note of the cell's address. Suppose the cell is A5 then type the following expression in the Immediate Window and look at your cell when hitting Enter.

```
Range("A5")=StrConv(Range("A5"), vbUpperCase)
```

Now change vbUpperCase to vbProperCase. Now try vbLowerCase. There you go. You now know how to change the case of any text entry using a VBA expression.

The VBA function StrConv stands for String Convert. It is mostly use to change the case of a string. It takes two arguments, a string and the desired format. It returns the updated string based of the parameter you supply.

Let’s see how we can apply what we’ve learned in a loop. Type the following Sub:

```
Sub Change2Upper()
' Change selection to uppercase.

    Dim rng As Range                        ' Object variable representing one cell

    Application.ScreenUpdating = False
    For Each rng In Selection               ' For each selected cell
        rng = StrConv(rng, vbUpperCase)     ' Update the entry to Uppercase
    Next rng                                ' Move to the next cell in selection
    Application.ScreenUpdating = True
End Sub
```

This amazing procedure can change tens of thousands entries in uppercase in fewer than 3 or 4 seconds. Copy that procedure two more times but use vbProperCase and vbLowerCase instead and you have a collection of subs that many would love to know about.

The really cool line in this procedure is “For Each rng In Selection”. In this case the variable rng will take on the value of each selected cell and stop automatically after processing the last cell in the selection. Just like that!

Of course, the only prerequisite is that the user selected the correct cells prior to launching the Sub. But most users are familiar with making a selection *then* activating some command.

About Object Variables

By now you are familiar with data variables. They hold data like strings, numbers and dates (which are really numbers). Object variables on the other hand hold Excel objects.

For example instead of moving to a sheet to work on a specific area of that sheet you can use an object variable to refer to that area without ever going to that sheet or selecting the cells manually! In the case of the Change2Upper sub above, the variable rng refers to each individual cell of the selection. You can do any number of formatting of transformation on that object using the rng variable.

For instance you may want to change each cell selected to uppercase with a font size of 14 points then italic and centre alignment.

```
For Each rng In Selection                   ' For each selected cell.
    rng = StrConv(rng, vbUpperCase)         ' Update the entry to Uppercase.
    With rng
        .Font.Italic = True                 ' Format to your liking.
        .HorizontalAlignment = xlCenter
        .Font.Size = 14
    End With
Next rng                                    ' Move to the next cell in selection.
```

As you can see the design of the loop above is extremely similar to the previous one. Once you have done a couple loop of that sort you can focus on what you want the loop to do instead of how to make the loop work.

Visiting Each Worksheet

Earlier we learned that a For Each — Next loop works very well with collections. A workbook has a collection of worksheets. The next procedure shows the basic code required to loop through all the worksheets in a workbook.

```
Sub VisitEachSheet()
' Print the name of every worksheet in the Workbook.
    Dim wks As Worksheet
    For Each wks In Worksheets
        Debug.Print wks.Name
    Next wks
End Sub
```

The loop will run through all worksheets in the current workbook regardless of the number of sheets it contains. You don't even need to know how many worksheets are in that workbook.

Instead of the Debug.Print statement you can use any VBA statement that will accomplish the task you want to do. If you need to skip some sheets base on some condition, specify that condition inside the loop. For example:

```
For Each wks In Worksheets
    If Left(wks.Name,2) <> "PK" Then
        Debug.Print wks.Name
    End If
Next wks
```

Here the loop goes through each sheet but the action will be executed only if the worksheet's name does *not* start with the letters 'PK'. You can use any conditional statements that fit you requirements.

The For Next Statement

The For Next statement may look simpler than the For Each statement but both have different use in VBA. As seen before, the For Each statement is used for collection of objects (Selected cells, worksheets, charts and so on) the For Next statement uses good old data to control the loop.

As in the For Each type of loop the For Next loop requires that you know (or have a mean to know) the number of iterations required. The For Loop has the following form:

```
For intVar = 1 to intTotal
    Statements to run
Next intVar
```

Sound pretty simple. The variable intVar is initiated with the value of 1 and will increment by 1 every time the loop runs. Note that the variable controlling the loop is incremented by 1 automatically. The variable after the keyword For does not have to start with 1, you can use any initial value.

An important point here is that intTotal needs to have a known value before the loop starts. This is the nature of the For Loop.

In a module type the following Sub:

```
Sub ForLoop()
    Dim i As Integer

    For i = 1 To 5
        Debug.Print i * 2
    Next i
End Sub
```

Ensure that the insertion point is anywhere in the Sub and that the Immediate Window is open. Hit the F8 key until the loop is completely finished.

If you ask yourself how the loop knows that the variable next to the keyword For will increment by 1, the answer is because it is the default setting for that type of loop. You can change that value to suits your need. For example modify the loop above as shown:

```
Sub ForLoop()
    Dim i As Integer

    For i = 1 To 15 Step 2
        Debug.Print i * 2
    Next i
End Sub
```

Here the Step is 2 so the variable i increments by 2 until it reaches the maximum of 15.

Run the Sub in Break Mode as you did before using the F8 key. In the Immediate Window you will see the numbers: 2, 6, 10, 14, 18, 22, 26 and 30 printed. The Step keyword specifies the increment of the variable until it reaches the upper limit of 15. So the output is the numbers 1, 3, 5, 7, 9, 11, 13, 15 multiplied by 2.

The Step keyword can even have a negative value so the loop variable decreases until it reaches the value after the keyword To. In the following example:

```
For i = 30 To 5 Step -5
    Debug.Print i * 2
Next i
```

The Variable i takes on the values: 30, 25, 20, 15, 10 and 5. When it reached the value 5 it iterates one last time.

Iterating in an Array

Using a For Loop is most useful when working with arrays. Below the array aryFruits is initialised to hold 5 strings indexed 0, 1, 2, 3 and 4. Type this Sub and test it in Break Mode like you did before.

```
Sub IterArray()
    Dim aryFruits(4) As String
    Dim i As Integer

    aryFruits(0) = "Apple"  ' First item in an array is item 0.
    aryFruits(1) = "Kiwi"
    aryFruits(2) = "Orange"
    aryFruits(3) = "Grape"
    aryFruits(4) = "Cherry"

    ' UBound returns the number of elements in the array.
    For i = 0 To UBound(aryFruits())
        Debug.Print aryFruits(i)
    Next i
End Sub
```

Two Dimensions Loop

At this point all our loops were linear as it were. They produced only *one* column of data in the Immediate Window. It is not rare that you need to create loops that examine two dimensional data. For example if you were to scan a table of values where all the states are at the top of each column and the products at the beginning of each row. That would have to be a two dimensions table right?

The general idea here can be expressed like this: For each column of a table scan each row below and do something with them. Of course the loop then moves to the next column and scan the rows again until there a no more column.

The next procedure uses the concept discussed above

```
Sub TwoDimLoop()
' Create a twelve month table with 10 random values under each month.

    Dim i As Byte, j As Byte
    Dim rngStart As Range

    Set rngStart = ActiveCell

    For i = 1 To 12                     ' Twelve months columns (outer loop).
        rngStart.Offset(0, i - 1) = MonthName(i, False)
        For j = 1 To 10                 ' Ten rows each column (inner loop).
            ' The expression Int((100 * Rnd) + 1) below returns
            ' a random integer between 1 and 100 inclusive.
            rngStart.Offset(j, i - 1) = Int((100 * Rnd) + 1)
        Next j       ' Next row.
    Next i           ' Next column.
    Columns("A:L").ColumnWidth = 12     ' Adjust column widths to 12 points
    Range(ActiveCell.Offset(1, 0), _    ' Remember a space before the underscore!!
          ActiveCell.Offset(1, 0).End(xlDown).End(xlToRight)).Clear
End Sub
```

Split your screen in two with the VBE Screen on the right then click any cell in your worksheet. Position the insertion point anywhere in your sub and hold your finger on the F8 key. You can see how the inner loop (the j loop) interacts with the outer loop (the i loop). When you're tired of pressing F8 hit F5 to finish the procedure. Once the sub is done it erases (or Clear) all the values under the months.

The continuation character

In the last instruction we use the Continuation character (the underscore). This simple feature allows you with split a long expression in multiple lines. The underscore instructs VBA that the expression continues onto the next.

If your expression is very long (as in the case of a long concatenation) you can split the expression in as many segments as required.

When using the Continuation character do not forget to put a space in front of it. If you forget you will have a nasty Compile Error from VBA.

Finally you cannot use the Continuation character just anywhere. A typical place is after a comma or after an ampersand (&) in a long concatenation. For example:

```
Msg = "Are you sure that you want to delete the customer " & strClient & _
      " from the list? There are still " & intItems & _
      " items that are not shipped?"
```

To be sure, you will encounter the Continuation character many times in sample code on the Web.

Conclusion

You will regularly need a loop in your procedures. This document explored some common use of loops using practical example. Remember that to master loops; one has to be patient and type a good dozen of short loops without notes. It can only get easier.

Daniel Lamarche

ComboProjects
Understanding does matter