Using Loops to Repeat Code

Why You Need Loops

The macro recording tools in Microsoft Word and Microsoft Excel make easy work of creating simple coding tasks, but running a recorded macro is just that – you do a job once. If you need to do the job again on another document, paragraph or worksheet you have to run the macro again. To do that automatically you need a code loop.

A loop is created by a set of code statements that cause a piece of code to be repeated over and over again, as many times as necessary. There are different kinds of loop to suit different situations. The code required is usually quite simple and, once mastered, will help you unleash the true power of VBA.

The examples shown here use Microsoft Excel, although the techniques are the same for any program that supports VBA.

Looping through a Collection (For Each…Next)

A program's object model contains a number of *collections* of objects. The Excel *Application* object contains a collection of workbooks. The *Workbook* object contains a collection of worksheets. The *Worksheet* object contains collections of cells, columns, rows etc.

VBA uses a *For Each…Next* loop to iterate through each member of a collection. The code starts by declaring a variable which represents the object you want to examine (i.e. the *member* of the collection). This example (*Listing 1*) loops through the worksheets in a workbook to determine if one with a particular name already exists. If not, it creates one with that name.

Listing 1

```
Sub AddSheet_1()
    Dim objSheet As Worksheet
    For Each objSheet In ActiveWorkbook.Sheets
        If objSheet.Name = "NewSheet" Then Exit Sub
    Next objSheet
    Worksheets.Add
    ActiveSheet.Name = "NewSheet"
End Sub
```

The statement *For Each* instructs the program that you wish the code to repeat as many times as there are objects in the collection (unless you subsequently tell it otherwise). When the statement *Next* is reached the macro is sent back to the beginning of the loop until the job is done.

The code inside the loop includes a simple *If Statement* that terminates the macro with the command *Exit Sub* if a sheet with the specified name is found. There is no point in continuing the loop to look at any remaining sheets if the one being sought has already been found. If the named worksheet is not found the loop will run its course and move on to the next part of the macro which adds then names a new worksheet.

You could include a slightly more elaborate *If Statement*. Here (*Listing 2*) a message is displayed if the named sheet is found, and the sheet activated (i.e. selected) before the macro exits:

Listing 2

```
Sub AddSheet_2()
    Dim objSheet As Worksheet
    For Each objSheet In ActiveWorkbook.Sheets
        If objSheet.Name = "NewSheet" Then
            MsgBox "A worksheet with that name already exists."
            Worksheets("NewSheet").Activate
            Exit Sub
        End If
    Next objSheet
    Worksheets.Add
    ActiveSheet.Name = "NewSheet"
End Sub
```

The next example (*Listing 3*) loops through a collection of cells looking for a cell containing a particular value (in this case today's date). If it finds one it displays a message the selects the cell and cancels the macro.

Listing 3

```
Sub FindCell_1 ()
    Dim objCell As Object
    For Each objCell In ActiveSheet.UsedRange.Cells
        If objCell.Value = Date Then
            objCell.Select
            MsgBox "Today's date was found in cell: " & objCell.Address
            Exit Sub
        End If
    Next objCell
    MsgBox "No cell containing today's date was found."
End Sub
```

The code specifies the *UsedRange* of the active sheet as the area to search. There are many different ways to specify the range. For example, You could specify a particular range:

```
    For Each objCell In ActiveSheet.Range("A1:C25").Cells
```

You might want the macro to work on a certain column (here the second column, column "B"):

```
    For Each objCell In ActiveSheet.Columns(2).Cells
```

Or you might prefer to select a block of cells and work on the selection:

```
    For Each objCell In Selection.Cells
```

Continuing the Loop

In each of the examples shown above the macro was terminated when the object was found. If you want the loop to continue and examine all the members of the collection, simply omit the *Exit Sub* statement. Remember that doing this might have an implication on the code that comes after the loop.

For example, the code in *Listing 3* stops after finding just the first cell containing today's date. The date might occur in several cells. The next example (*Listing 4*) keeps a note of each relevant cell's address by adding it to a string variable (preceded by a comma). Then, when the loop finishes, if the variable is empty (i.e. its length is zero) this shows that no cells were noted, but if cells were found a statement removes the leading comma from the string then uses it to specify a range of cells to be selected.

Listing 4

```
Sub FindCell_2()
    Dim objCell As Object
    Dim strRange As String
    For Each objCell In ActiveSheet.UsedRange.Cells
        If objCell.Value = Date Then
            strRange = strRange & "," & objCell.Address
        End If
    Next objCell
    If Len(strRange) = 0 Then
        MsgBox "No cell containing today's date was found."
    Else
        strRange = Right(strRange, Len(strRange) - 1)
        Range(strRange).Select
        MsgBox "Today's date appears in the selected cells"
    End If
End Sub
```

Exiting the Loop but Continuing the Macro

If you want your macro to continue after the loop, but you want to save time by having the code terminate the loop when the required object is found, you can do this without having to exit the macro by using the statement *Exit For* instead of *Exit Sub*. The command *Exit For* causes the macro to jump to the first line following the *Next…* statement.

Listing 5

```
Sub AddSheet_3()
    Dim objSheet As Worksheet
    Dim blnFound As Boolean
    For Each objSheet In ActiveWorkbook.Sheets
        If objSheet.Name = "NewSheet" Then
            blnFound = True
            Exit For
        End If
    Next objSheet
    If Not blnFound Then
        Worksheets.Add
        ActiveSheet.Name = "NewSheet"
    End If
    MsgBox "This workbook contains " & Worksheets.Count & " sheets."
End Sub
```

The macro in *Listing 5* uses the *Exit For* statement to terminate the loop if a suitable sheet is found but also notes this by setting the value of the boolean (True/False) variable to *True*. When the loop terminates, either because it ran its course or because a sheet was found, an *If Statement* checks the value of the variable and only creates a new sheet if the value is not *False*. The macro then continues – in this example displaying a message showing the number of sheets in the workbook.

Looping for Number of Times (For…Next)

If you know, or can find out, how many times you want a loop to run then you can use a *For…Next* loop. You might use this kind of loop to move down a column of cells performing a particular task.

The number of times the loop runs is specified using a start number (usually 1 or 0) and finish number. A counter variable (here named "i") keeps count of the loop's progress. The finish number can be hard-coded, for example:

```
For i = 1 To 25
```

Or, more usually, calculated or determined in some way and sometimes specified as a variable:

```
For i = 1 To Selection.Rows.Count
```

These examples look at the cells in column "A" and write a value in column "B". Their task is to determine whether the value in column "A" is odd or even. They do this by halving it and testing if the result is a whole number by comparing it with the integer of the same calculation. If it is so, the original value must be an even one. They determine how many times to run the loop by counting the number of rows in the *CurrentRegion* of cell "A1" (i.e. the area adjacent to that cell containing values).

The first example (*Listing 6*) selects each cell as it moves down the column, referring to the selected cell as the *ActiveCell*.

Listing 6

```
Sub OddOrEven_1()
    Dim intRowCount As Integer
    Dim i As Integer
    intRowCount = Range("A1").CurrentRegion.Rows.Count
    Range("A1").Select
    For i = 1 To intRowCount
        If ActiveCell.Value / 2 = Int(ActiveCell.Value / 2) Then
            ActiveCell.Offset(0, 1).Value = "Even"
        Else
            ActiveCell.Offset(0, 1).Value = "Odd"
        End If
        ActiveCell.Offset(1, 0).Select
    Next i
End Sub
```

Although perfectly serviceable, this syntax is inefficient and slow because each cell has to be selected. A more efficient way is to refer to the cell in question using the *Offset* property of the starting cell as shown in the next example (*Listing 7*). The *Row* component of the *Offset* property is supplied by the value of the counter "i":

Listing 7

```
Sub OddOrEven_2()
    Dim intRowCount As Integer
    Dim i As Integer
    intRowCount = Range("A1").CurrentRegion.Rows.Count
    For i = 0 To intRowCount - 1
        If Range("A1").Offset(i, 0).Value / 2 = _
            Int(Range("A1").Offset(i, 0).Value / 2) Then
            Range("A1").Offset(i, 1).Value = "Even"
        Else
            Range("A1").Offset(i, 1).Value = "Odd"
        End If
    Next i
End Sub
```

In addition to specifying the start and finish values of the counter variable you can declare a *Step* value if, for example you want your loop to proceed in units of more than one:

```
For i = 1 To 100 Step 2
```

Your loop can run in reverse order if the start an finish numbers are reversed. If you do this you must also declare a negative step value:

```
For i = 100 To 1 Step -1
```

Looping Until a Certain Condition is Satisfied (Do…Loop Until)

This method can be useful if you are unable to ascertain how many times to run the loop. It terminates the loop when a certain condition is satisfied. The first example (*Listing 8*) finds the location of the first empty cell in a column. It starts by selecting the first cell (in this case "A1") then selects the cell below and checks whether or not it is empty. If the selected cell empty the loop automatically terminates and leaves the empty cell selected.

Listing 8

```
Sub FindEmptyCell_1()
    Range("A1").Select
    Do
        ActiveCell.Offset(1, 0).Select
    Loop Until IsEmpty(ActiveCell.Value)
End Sub
```

The syntax can be expressed as:

```
Do
    'Code goes here
Loop Until <Condition>
```

…which does not apply the test to the start cell, or:

```
Do Until <Condition>
    'Code goes here
Loop
```

…which does apply the test to the start cell.

As mentioned earlier, selecting cells in a code loop is an inefficient way of working and could be slow if a lot of rows are to be processed. The next example (*Listing 9*) uses the loop to increase the value of a counter variable ("i") by one, then uses the final value of "i" to determine which cell gets selected when the loop finishes:

Listing 9

```
Sub FindEmptyCell_2()
    Dim i As Integer
    Do
        i = i + 1
    Loop Until IsEmpty(Range("A1").Offset(i, 0).Value)
    Range("A1").Offset(i, 0).Select
End Sub
```

Exiting the Loop

As with a *For…Next* loop you can have your code terminate a *Do…Loop* loop by using the statement *Exit Do*.

Looping Whilst a Certain Condition is Satisfied (Do While…Loop)

This is very similar to the last kind of loop. It causes the loop to run s long as a particular condition is satisfied and uses the syntax:

```
Do While <Condition>
    'Code goes here
Loop
```

Or:

```
Do
    'Code goes here
Loop While <Condition>
```

This example (*Listing 10*) simply uses the opposite of the condition used in *Listing 8*.

Listing 10

```
Sub FindEmptyCell_3()
    Range("A1").Select
    Do While Not IsEmpty(ActiveCell.Value)
        ActiveCell.Offset(1, 0).Select
    Loop
End Sub
```

These examples have all been used to look for an empty cell, but your condition can be any expression that can be evaluated and expressed. Your loop might be working down an ascending column of dates, either hard-coding a condition:

```
Do While ActiveCell.Value < #01/07/2007#
```

…or calculating it:

```
Do While ActiveCell.Value < DateAdd("m", 6, Date)
```

Alternative Syntax

You might see code using the syntax *While…Wend*. Although this syntax is supported by VBA it is not commonly used today, being a remnant of older Basic code. Here is an example (*Listing 11*).

Listing 11

```
Sub FindEmptyCell_4()
    Range("A1").Select
    While IsEmpty(ActiveCell.Value) = False
        ActiveCell.Offset(1, 0).Select
    Wend
End Sub
```