Excel Macros examples

Brief Look at Variables

You can think of variables as memory containers that you can use in your procedures. There are different types of variables, each tasked with holding a specific type of data. Some of the common types of variables:

- ✓ **String:** Holds textual data
- ✓ **Integer:** Holds numeric data ranging from –32,768 to 32,767
- ✓ **Long:** Holds numeric data ranging from –2,147,483,648 to 2,147,483,647
- ✓ **Double:** Holds floating point numeric data
- ✓ **Variant:** Holds any kind of data
- ✓ **Boolean:** Holds binary data that returns True or False
- ✓ **Object:** Holds an actual object from the Excel object mode

The term used for creating a variable in a macro is declaring a variable. You do so by entering Dim (an abbreviation for dimension), the name of your variable, and then the type. For instance:

- Dim MyText as String
- Dim MyNumber as Integer
- Dim MyWorksheet as Worksheet

Macro Examples

Macro 1: Creating a New Workbook

You may sometimes want or need to create a new workbook in an automated way. For instance, you may need to copy data from a table and paste it into a newly created workbook. The following macro copies a range of cells from the active sheet and pastes the data into a new workbook.

```
Sub Macro1()
        'Step 1 Copy the data
         Sheets("Example 1").Range("B4:C15").Copy
        'Step 2 Create a new workbook
         Workbooks.Add
        'Step 3 Paste the data
         ActiveSheet.Paste Destination:=Range("A1")
        'Step 4 Turn off application alerts
        Application.DisplayAlerts = False
        'Step 5 Save the newly created workbook
        ActiveWorkbook.SaveAs _
        Filename:="C:\Temp\MyNewBook.xlsx"
        'Step 6 Turn application alerts back on
         Application.DisplayAlerts = True
End Sub
```

Komentár od [MH1]: Your own sheet name.

Komentár od [MH2]: Your own range.

Here’s how this macro works:

1. In Step 1, we simply copy the data that ranges from cells B4 to C15. The thing to note here is that you are specifying both the sheet and the range by name. This is a best practice when you are working with multiple workbooks open at one time.
2. We are using the Add method of the Workbook object to create a new workbook. This is equivalent to manually clicking File➔New➔Blank Document in the Excel Ribbon.
3. In this step, you use the Paste method to send the data you copied to cell A1 of the new workbook. Pay attention to the fact that the code refers to the ActiveSheet object. When you add a workbook, the new workbook immediately gains focus, becoming the active workbook. This is the same behavior you would see if you were to add a workbook manually.
4. In Step 4 of the code, we set the DisplayAlerts method to False, effectively turning off Excel’s warnings. We do this because in the next step of the code, we save the newly created workbook. We may run this macro multiple times, in which case Excel attempts to save the file multiple times. What happens when you try to save a workbook multiple times? That’s right — Excel warns you that there is already a file out there with that name and then asks if you want to overwrite the previously existing file. Because your goal is to automate the creation of the new workbook, you want to suppress that warning.
5. In Step 5, we save the file by using the SaveAs method. Note that we are entering the full path of the save location, including the final filename.
6. Because we turned application alters off in Step 4, we need to turn them back on. If we don’t, Excel continues to suppress all warnings for the life of the current session.

Macro 2: Protect a Worksheet on Workbook Close

Sometimes you need to send your workbook out into the world with specific worksheets protected. If you find that you’re constantly protecting and unprotecting sheets before distributing your workbooks, this macro can help you.

This code is triggered by the workbook’s BeforeClose event. When you try to close the workbook, this event fires, running the code within. The macro automatically protects the specified sheet with the given password, and then saves the workbook.

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)
        ‘Step 1: Protect the sheet with a password
         Sheets(“Sheet1”).Protect Password:=”RED”
        ‘Step 2: Save the workbook
        ActiveWorkbook.Save
End Sub
```

1. In Step 1, we are explicitly specifying which sheet we want to protect — Sheet1, in this case. We are also providing the password argument, Password:=RED. This defines the password needed to remove the protection. This password argument is completely optional. If you omit this altogether, the sheet will still be protected, but you won’t need a password to unprotect it. Also, be aware that Excel passwords are case-sensitive, so you’ll want pay attention to the exact password and capitalization that you are using.

2. Step 2 tells Excel to save the workbook. If we don't save the workbook, the sheet protection we just applied won't be in effect the next time the workbook is opened.

Macro 3: Create a Backup of a Current Workbook with Today's Date

You should know that making backups of your work is important. Now you can have a macro do it for you. This simple macro saves your workbook to a new file with today's date as part of the name.

The trick to this macro is piecing together the new filename. The new filename has three pieces: the path, today's date, and the original filename.

The path is captured by using the Path property of the ThisWorkbook object. Today's date is grabbed with the Date function.

You'll notice that we are formatting the date (Format(Date, "mm-dd-yy")). This is because by default, the Date function returns mm/dd/yyyy. We use hyphens instead of forward slashes because the forward slashes would cause the file save to fail. (Windows does not allow forward slashes in filenames.)

The last piece of the new filename is the original filename. We use the Name property of the ThisWorkbook object to capture that:

```
Sub Macro3()
        'Step 1: Save workbook with new filename
                ThisWorkbook.SaveCopyAs _
         Filename:=ThisWorkbook.Path & "\" & _
        Format(Date, "mm-dd-yy") & " " & _
        ThisWorkbook.Name
End Sub
```

In the one and only step, the macro builds a new filename and uses the SaveCopyAs method to save the file.

Macro 24: Create a New Workbook for Each Worksheet

Many Excel analysts need to parse their workbooks into separate books per worksheet tab. In other words, they need to create a new workbook for each of the worksheets in their existing workbook. You can imagine what an ordeal this would be if you were to do it manually. The following macro helps automate that task.

In this macro, you are looping the worksheets, copying each sheet, and then sending the copy to a new workbook that is created on the fly. The thing to note here is that the newly created workbooks are being saved in the same directory as your original workbook, with the same filename as the copied sheet (wb.SaveAs ThisWorkbook.Path & "\" & ws.Name).

```
Sub Macro4()
        'Step 1: Declare all the variables.
                Dim ws As Worksheet
                Dim wb As Workbook
        'Step 2: Start the looping through sheets
                For Each ws In ThisWorkbook.Worksheets
        'Step 3: Create new workbook and save it.
                Set wb = Workbooks.Add
```

```
		wb.SaveAs ThisWorkbook.Path & "\" & ws.Name
	'Step 4: Copy the target sheet to the new workbook
		ws.Copy Before:=wb.Worksheets(1)
		wb.Close SaveChanges:=True
	'Step 5: Loop back around to the next worksheet
		Next ws
End Sub
```

Not all valid worksheet names translate to valid filenames. Windows has specific rules that prevent you from naming files with certain characters. You cannot use these characters when naming a file: backslash (\), forward slash (/), colon (:), asterisk (*), question mark (?), pipe (|), double quote ("), greater than (>), and less than (<). The twist is that you can use a few of these restricted characters in your sheet names; specifically, double quote, pipe (|), greater than (>), and less than (<).
As you're running this macro, naming the newly created files to match the sheet name may cause an error. For instance, the macro throws an error when creating a new file from a sheet called May| Revenue (because of the pipe character). To make a long story short, avoid naming your worksheets with these restricted characters.

1. Step 1 declares two object variables. The ws variable creates a memory container for each worksheet the macro loops through. The wb variable creates the container for the new workbooks we create.
2. In Step 2, the macro starts looping through the sheets. The use of the ThisWorkbook object ensures that the active sheet that is being copied is from the workbook the code is in, not the new workbook that is created.
3. In Step 3, we create the new workbook and save it. We save this new book in the same path as the original workbook (ThisWorkbook). The filename is set to be the same name as the currently active sheet.
4. Step 4 copies the currently active sheet and uses the Before parameter to send it to the new book as the first tab.
5. Step 5 loops back to get the next sheet. After all of the sheets are evaluated, the macro ends.

Macro 5: Create a Table of Contents for Your Worksheets

Outside of sorting worksheets, creating a table of contents for the worksheets in a workbook is the most commonly requested Excel macro. The reason is probably not lost on you. We often have to work with files that have more worksheet tabs than can easily be seen or navigated. A table of contents definitely helps. The following macro not only creates a list of worksheet names in the workbook, but it also ads hyperlinks so that you can easily jump to a sheet with a simple click.

```
Sub Macro5 ()
	'Step 1: Declare Variables
		Dim i As Long
	'Step 2: Delete Previous TOC if Exists
		On Error Resume Next
		Application.DisplayAlerts = False
		Sheets("Table Of Contents").Delete
```

```
            Application.DisplayAlerts = True
            On Error GoTo 0
    'Step 3: Add a new TOC sheet as the first sheet
            ThisWorkbook.Sheets.Add _
            Before:=ThisWorkbook.Worksheets(1)
            ActiveSheet.Name = "Table Of Contents"
    'Step 4: Start the i Counter
             For i = 1 To Sheets.Count
    'Step 5: Select Next available row
            ActiveSheet.Cells(i, 1).Select
    'Step 6: Add Sheet Name and Hyperlink
            ActiveSheet.Hyperlinks.Add _
            Anchor:=ActiveSheet.Cells(i, 1), _
            Address:="", _
             SubAddress:="'" & Sheets(i).Name & "'!A1", _
            TextToDisplay:=Sheets(i).Name
    'Step 7: Loop back increment i
            Next i
End Sub
```

1. Step 1 declares an integer variable called i to serve as the counter as the macro iterates through the sheets. Note that this macro is not looping through the sheets the way previous macros in this Part did. In previous macros, we looped through the worksheets collection and selected each worksheet there. In this procedure, we are using a counter (our i variable). The main reason is because we not only have to keep track of the sheets, but we also have to manage to enter each sheet name on a new row into a table of contents. The idea is that as the counter progresses through the sheets, it also serves to move the cursor down in the table of contents so each new entry goes on a new row.
2. Step 2 essentially attempts to delete any previous sheet called Table of Contents. Because there may not be any Table of Contents sheet to delete, we have to start Step 2 with the On Error Resume Next error handler. This tells Excel to continue the macro if an error is encountered here. We then delete the Table of Contents sheet using the DisplayAlerts method, which effectively turns off Excel's warnings so we don't have to confirm the deletion. Finally, we reset the error handler to trap all errors again by entering On Error GoTo 0.
3. In Step 3, we add a new sheet to the workbook using the Before argument to position the new sheet as the first sheet. We then name the sheet Table of Contents. As we mentioned previously in this Part, when you add a new worksheet, it automatically becomes the active sheet. Because this new sheet has the focus throughout the procedure, any references to ActiveSheet in this code refer to the Table of Contents sheet.
4. Step 4 starts the i counter at 1 and ends it at the maximum count of all sheets in the workbook. Again, instead of looping through the Worksheets collection like we've done in previous macros, we are simply using the i counter as an index number that we can pass to the Sheets object. When the maximum number is reached, the macro ends.

5. Step 5 selects the corresponding row in the Table of Contents sheet. That is to say, if the I counter is on 1, it selects the first row in the Table of Contents sheet. If the i counter is at 2, it selects the second row, and so on. We are able to do this using the Cells item. The Cells item provides an extremely handy way of selecting ranges through code. It requires only relative row and column positions as parameters. So Cells(1,1) translates to row 1, column 1 (or cell A1). Cells(5, 3) translates to row 5, column 3 (or cell C5). The numeric parameters in the Cells item are particularly handy when you want to loop through a series of rows or columns using an incrementing index number.
6. Step 6 uses the Hyperlinks.Add method to add the sheet name and hyperlinks to the selected cell. This step feeds the Hyperlinks.Add method the parameters it needs to build out the hyperlinks.
7. The last step in the macro loops back to increment the i counter to the next count. When the i counter reaches a number that equals the count of worksheets in the workbook, the macro ends.

Macro 6: Highlight the Active Row and Column

When looking at a table of numbers, it would be nice if Excel automatically highlighted the row and column you're on. This effect gives your eyes a lead line up and down the column as well as left and right across the row. The following macro enables that effect with just a simple double-click. When the macro is in place, Excel highlights the row and column for the cell that is active, greatly improving your ability to view and edit a large grid.

```
Private Sub Worksheet_BeforeDoubleClick(ByVal Target As Range, Cancel As Boolean)
        'Step 1: Declare Variables
                Dim strRange As String
        'Step2: Build the range string
                strRange = Target.Cells.Address & "," & _
                Target.Cells.EntireColumn.Address & "," & _
                Target.Cells.EntireRow.Address
        'Step 3: Pass the range string to a Range
                 Range(strRange).Select
End Sub
```

1. We first declare an object called strRange. This creates a memory container we can use to build a range string.
2. A range string is nothing more than the address for a range. "A1" is a range string that points to cell A1. "A1:G5" is also a range string; this points to a range of cells encompassing cells A1 to G5. In Step 2, we are building a range string that encompasses the double-clicked cell (called Target in this macro), the entire active row, and the entire active column. The Address properties for these three ranges are captured and pieced together into the strange variable.
3. In Step 3, we feed the strRange variable as the address for a Range.Select statement. This is the line of the code that finally highlights the double-clicked selection.

Macro 7: Deleting Blank Rows

Work with Excel long enough, and you'll find out that blank rows can often cause havoc on many levels. They can cause problems with formulas, introduce risk when copying and pasting,

and sometimes cause strange behaviors in PivotTables. If you find that you are manually searching out and deleting blank rows in your data sets, this macro can help automate that task.
In this macro, we are using the UsedRange property of the Activesheet object to define the range we are working with. The UsedRange property gives us a range that encompasses the cells that have been used to enter data. We then establish a counter that starts at the last row of the used range to check if the entire row is empty. If the entire row is indeed empty, we remove the row. We keep doing that same delete for every loop, each time incrementing the counter to the previous row.

```
Sub Macro7()
        ‘Step1: Declare your variables.
                Dim MyRange As Range
                Dim iCounter As Long
        ‘Step 2: Define the target Range.
                Set MyRange = ActiveSheet.UsedRange
        ‘Step 3: Start reverse looping through the range.
                For iCounter = MyRange.Rows.Count To 1 Step -1
        ‘Step 4: If entire row is empty then delete it.
        If Application.CountA(Rows(iCounter).EntireRow) = 0 Then
        Rows(iCounter).Delete
        End If
        ‘Step 5: Increment the counter down
        Next iCounter
End Sub
```

1. The macro first declares two variables. The first variable is an Object variable called MyRange. This is an object variable that defines our target range. The other variable is a Long Integer variable called iCounter. This variable serves as an incremental counter.
2. In Step 2, the macro fills the MyRange variable with the UsedRange property of the ActiveSheet object. The UsedRange property gives us a range that encompasses the cells that have been used to enter data. Note that if we wanted to specify an actual range or a named range, we could simply enter its name — Range(“MyNamedRange”).
3. In this step, the macro sets the parameters for the incremental counter to start at the max count for the range (MyRange.Rows.Count) and end at 1 (the first row of the chosen range). Note that we are using the Step-1 qualifier. Because we specify Step -1, Excel knows we are going to increment the counter backwards, moving back one increment on each iteration. In all, Step 3 tells Excel to start at the last row of the chosen range, moving backward until it gets to the first row of the range.
4. When working with a range, you can explicitly call out a specific row in the range by passing a row index number to the Rows collection of the range. For instance, Range(“D6:D17”).Rows(5) points to the fifth row in the range D6:D17.
5. In Step 4, the macro uses the iCounter variable as an index number for the Rows collection of MyRange. This helps pinpoint which exact row we are working with in the current loop. The macro checks to see whether the cells in that row are empty. If they are, the macro deletes the entire row.
6. In Step 5, the macro loops back to increment the counter down.

Macro 8: Deleting Blank Columns

Just as with blank rows, blank columns also have the potential of causing unforeseen errors. If you find that you are manually searching out and deleting blank columns in your data sets, this macro can automate that task.

In this macro, we are using the UsedRange property of the ActiveSheet object to define the range we are working with. The UsedRange property gives us a range that encompasses the cells that have been used to enter data. We then establish a counter that starts at the last column of the used range, checking if the entire column is empty. If the entire column is indeed empty, we remove the column. We keep doing that same delete for every loop, each time incrementing the counter to the previous column.

```
Sub Macro8()
        ‘Step1: Declare your variables.
                Dim MyRange As Range
                Dim iCounter As Long
        ‘Step 2: Define the target Range.
                Set MyRange = ActiveSheet.UsedRange
        ‘Step 3: Start reverse looping through the range.
                For iCounter = MyRange.Columns.Count To 1 Step -1
        ‘Step 4: If entire column is empty then delete it.
                 If Application.CountA(Columns(iCounter).EntireColumn) = 0 Then
                Columns(iCounter).Delete
                End If
        ‘Step 5: Increment the counter down
                Next iCounter
End Sub
```

1. Step 1 first declares two variables. The first variable is an object variable called MyRange. This is an Object variable that defines the target range. The other variable is a Long Integer variable called iCounter. This variable serves as an incremental counter.
2. Step 2 fills the MyRange variable with the UsedRange property of the ActiveSheet object. The UsedRange property gives us a range that encompasses the cells that have been used to enter data. Note that if we wanted to specify an actual range or a named range, we could simply enter its name — Range(“MyNamedRange”).
3. In this step, the macro sets the parameters for our incremental counter to start at the max count for the range (MyRange.Columns.Count) and end at 1 (the first row of the chosen range). Note that we are using the Step-1 qualifier. Because we specify Step -1, Excel knows we are going to increment the counter backwards; moving back one increment on each iteration. In all, Step 3 tells Excel that we want to start at the last column of the chosen range, moving backward until we get to the first column of the range.
4. When working with a range, you can explicitly call out a specific column in the range by passing a column index number to the Columns collection of the range. For instance, Range(“A1:D17”).Columns(2) points to the second column in the range (column B). In Step 4, the macro uses the iCounter variable as an index number for the Columns collection of MyRange. This helps pinpoint exactly which column we are working with

in the current loop. The macro checks to see whether all the cells in that column are empty. If they are, the macro deletes the entire column.

5. In Step 5, the macro loops back to increment the counter down.

Macro 9: Trim Spaces from All Cells in a Range

A frequent problem when you import dates from other sources is leading or trailing spaces. That is, the values that are imported have spaces at the beginning or end of the cell. This obviously makes it difficult to do things like VLOOKUP or sorting. Here is a macro that makes it easy to search for and remove extra spaces in your cells. In this macro, we enumerate through a target range, passing each cell in that range through the Trim function.

```
Sub Macro9()
        ‘Step 1:Declare your variables
                Dim MyRange As Range
                Dim MyCell As Range
        ‘Step 2: Save the Workbook before changing cells?
                Select Case MsgBox(“Can’t Undo this action. “ & _
                “Save Workbook First?”, vbYesNoCancel)
                Case Is = vbYes
                ThisWorkbook.Save
                Case Is = vbCancel
                Exit Sub
                End Select
        ‘Step 3: Define the target Range.
                Set MyRange = Selection
        ‘Step 4: Start looping through the range.
                For Each MyCell In MyRange
        ‘Step 5: Trim the Spaces.
                If Not IsEmpty(MyCell) Then
                MyCell = Trim(MyCell)
                End If
        ‘Step 6: Get the next cell in the range
                Next MyCell
End Sub
```

1. Step 1 declares two Range object variables, one called MyRange to hold the entire target range, and the other called MyCell to hold each cell in the range as the macro enumerates through them one by one.
2. When you run a macro, it destroys the undo stack. You can’t undo the changes a macro makes. Because we are actually changing data, we need to give ourselves the option of saving the workbook before running the macro. Step 2 does this. Here, we call up a message box that asks if we want to save the workbook first. It then gives us three choices: Yes, No, and Cancel. Clicking Yes saves the workbook and continues with the macro. Clicking Cancel exits the procedure without running the macro. Clicking No runs the macro without saving the workbook.
3. Step 3 fills the MyRange variable with the target range. In this example, we are using the selected range — the range that was selected on the spreadsheet. You can easily

set the MyRange variable to a specific range such as Range(“A1:Z100”). Also, if your target range is a named range, you can simply enter its name — Range(“MyNamedRange”).
4. Step 4 starts looping through each cell in the target range, activating each cell as we go through.
5. After a cell is activated, the macro uses the IsEmpty function to make sure the cell is not empty. We do this to save a little on performance by skipping the cell if there is nothing in it. We then pass the value of that cell to the Trim function. The Trim function is a native Excel function that removes leading and trailing spaces.
6. Step 6 loops back to get the next cell. After all cells in the target range are activated, the macro ends.

Macro 10: Copy Filtered Rows to a New Workbook
Often, when you’re working with a set of data that is AutoFiltered, you want to extract the filtered rows to a new workbook. Of course, you can manually copy the filtered rows, open a new workbook, paste the rows, and then format the newly pasted data so that all the columns fit. But if you are doing this frequently enough, you may want to have a macro to speed up the process. This macro captures the AutoFilter range, opens a new workbook, then pastes the data.

```
Sub Macro10 ()
        ‘Step 1: Check for AutoFilter - Exit if none exists
                If ActiveSheet.AutoFilterMode = False Then
                Exit Sub
                End If
        ‘Step 2: Copy the Autofiltered Range to new workbook
                ActiveSheet.AutoFilter.Range.Copy
                Workbooks.Add.Worksheets(1).Paste
        ‘Step 3: Size the columns to fit
                Cells.EntireColumn.AutoFit
End Sub
```

1. Step 1 uses the AutoFilterMode property to check whether the sheet even has AutoFilters applied. If not, we exit the procedure.
2. Each AutoFilter object has a Range property. This Range property obligingly returns the rows to which the AutoFilter applies, meaning it returns only the rows that are shown in the filtered data set. In Step 2, we use the Copy method to capture those rows, and then we paste the rows to a new workbook. Note that we use Workbooks.Add.Worksheets(1). This tells Excel to paste the data into the first sheet of the newly created workbook.
3. Step 3 simply tells Excel to size the column widths to autofit the data we just pasted.

Macro 11: Create a PivotTable Inventory Summary
When your workbook contains multiple PivotTables, it’s often helpful to have an inventory summary that outlines basic details about the PivotTables. With this type of summary, you can quickly see important information like the location of each PivotTable, the location of each PivotTable’s source data, and the pivot cache index each PivotTable is using.

When you create a PivotTable object variable, you expose all of a PivotTable's properties — properties like its name, location, cache index, and so on. In this macro, we loop through each PivotTable in the workbook and extract specific properties into a new worksheet. Because each PivotTable object is a child of the worksheet it sits in, we have to first loop through the worksheets in a workbook first, and then loop through the PivotTables in each worksheet. Take a moment to walk through the steps of this macro in detail.

```
Sub Macro11()
        'Step 1: Declare your Variables
                Dim ws As Worksheet
                Dim pt As PivotTable
                Dim MyCell As Range
        'Step 2: Add a new sheet with column headers
                Worksheets.Add
                Range("A1:F1") = Array("Pivot Name", "Worksheet", _
                "Location", "Cache Index", _
                "Source Data Location", _
                "Row Count")
        'Step 3: Start Cursor at Cell A2 setting the anchor here
                Set MyCell = ActiveSheet.Range("A2")
        'Step 4: Loop through each sheet in workbook
                For Each ws In Worksheets
        'Step 5: Loop through each PivotTable
                For Each pt In ws.PivotTables
                MyCell.Offset(0, 0) = pt.Name
                MyCell.Offset(0, 1) = pt.Parent.Name
                MyRange.Offset(0, 2) = pt.TableRange2.Address
                MyRange.Offset(0, 3) = pt.CacheIndex
                MyRange.Offset(0, 4) = Application.ConvertFormula _
                (pt.PivotCache.SourceData, xlR1C1, xlA1)
                MyRange.Offset(0, 5) = pt.PivotCache.RecordCount
        'Step 6: Move Cursor down one row and set a new anchor
                Set MyRange = MyRange.Offset(1, 0)
        'Step 7: Work through all PivotTables and worksheets
                Next pt
                Next ws
        'Step 8: Size columns to fit
                ActiveSheet.Cells.EntireColumn.AutoFit
End Sub
```

1. Step 1 declares an object called ws. This creates a memory container for each worksheet we loop through. We then declare an object called pt, which holds each PivotTable we loop through. Finally, we create a range variable called MyCell. This variable acts as our cursor as we fill in the inventory summary.
2. Step 2 creates a new worksheet and adds column headings that range from A1 to F1. Note that we can add column headings using a simple array that contains our header labels. This new worksheet remains our active sheet from here on out.

3. Just as you would manually place your cursor in a cell if you were to start typing data, Step 3 places the MyCell cursor in cell A2 of the active sheet. This is our anchor point, allowing us to navigate from here. Throughout the macro, you see the use of the Offset property. The Offset property allows us to move a cursor x number of rows and x number of columns from an anchor point. For instance, Range(A2).Offset(0,1) would move the cursor one column to the right. If we wanted to move the cursor one row down, we would enter Range(A2). Offset(1, 0). In the macro, we navigate by using Offset on MyCell. For example, MyCell. Offset(0,4) would move the cursor four columns to the right of the anchor cell. After the cursor is in place, we can enter data.
4. Step 4 starts the looping, telling Excel we want to evaluate all worksheets in this workbook.
5. Step 5 loops through all the PivotTables in each worksheet. For each PivotTable it finds, it extracts out the appropriate property and fills in the table based on the cursor position (see Step 3). We are using six PivotTable properties: Name, Parent.Range, TableRange2. Address, CacheIndex, PivotCache.SourceData, and PivotCache. Recordcount. The Name property returns the name of the PivotTable. The Parent.Range property gives us the sheet where the PivotTable resides. The TableRange2.Address property returns the range that the PivotTable object sits in. The CacheIndex property returns the index number of the pivot cache for the PivotTable. A pivot cache is a memory container that stores all the data for a PivotTable. When you create a new PivotTable, Excel takes a snapshot of the source data and creates a pivot cache. Each time you refresh a PivotTable, Excel goes back to the source data and takes another snapshot, thereby refreshing the pivot cache. Each pivot cache has a SourceData property that identifies the location of the data used to create the pivot cache. The PivotCache. SourceData property tells us which range will be called upon when we refresh the PivotTable. You can also pull out the record count of the source data by using the PivotCache.Recordcount property.
6. Each time the macro encounters a new PivotTable, it moves the MyCell cursor down a row, effectively starting a new row for each PivotTable.
7. Step 7 tells Excel to loop back around to iterate through all PivotTables and all worksheets. After all PivotTables have been evaluated, we move to the next sheet. After all sheets have been evaluated, the macro moves to the last step.
8. Step 8 finishes off with a little formatting, sizing the columns to fit the data.

Macro 12: Function Sum by color

```
Function SumByFontColor(rng As Range, fontColorCell As Range) As Double
            Dim cell As Range
            Dim sum As Double
            Dim targetColor As Long
      'Define the font color from cell G3
            targetColor = fontColorCell.Font.Color
            sum = 0
            For Each cell In rng
      'Check if the font color of the cell matches the target color
            If cell.Font.Color = targetColor Then
            If IsNumeric(cell.Value) Then ' Check if the cell contains a number
```

```
                    sum = sum + cell.Value
                    End If
                    End If
                    Next cell
                    SumByFontColor = sum
End Function

Function CountByFontColor(rng As Range, fontColorCell As Range) As Long
                    Dim cell As Range
                    Dim count As Long
                    Dim targetColor As Long
          'Define the font color from cell G3
                    targetColor = fontColorCell.Font.Color
                    count = 0
                    For Each cell In rng
          'Check if the font color of the cell matches the target color
                    If cell.Font.Color = targetColor Then
                    count = count + 1
                    End If
                    Next cell
                    CountByFontColor = count
End Function
```

Macro: Highlight difference in columns

```
Sub columnDifference ()
          Range("H7:H8,I7:I8").Select
          Selection.ColumnDifferences(ActiveCell ).Select
          Selection.Style = "Bad"
End Sub
```

Macro: highlight difference in rows

```
Sub rowDifference ()
          Range("H7:H8,I7:I8").Select
          Selection.RowDifferences(ActiveCell ).Select
          Selection.Style = "Bad"
End Sub
```