MICROSOFT EXCEL:

RECORDING, USING, AND EDITING MACROS

neil@knacktraining.com

http://youtube.com/neilmalek

http://facebook.com/knacktraining

http://instagram.com/neilmalek

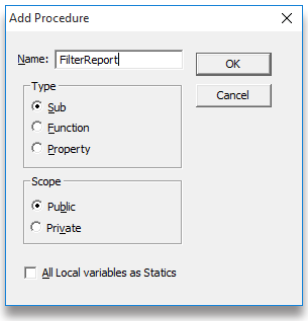
http://twitter.com/neilmalek

http://linkedin.com/in/neilmalek

TERMINOLOGY 2

VBA: The implementation of Microsoft's *programming language* that is used in Word, Excel, and the rest of the Office Suite. The language is called *Visual Basic*, and the full acronym is *Visual Basic for Applications*.

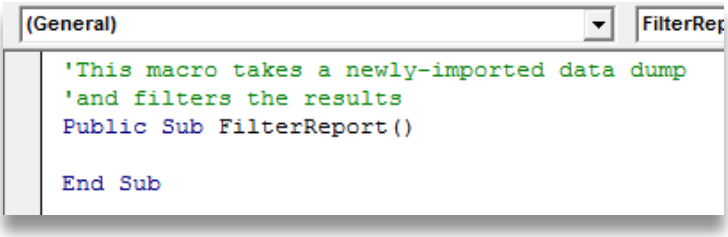
Subroutine: Listed as a *sub* in the language, a subroutine is a chunk of code that can be executed. This is the container for your macro.



Module: The code container for subroutines. When looking at the VBA window, it is the page that opens to be typed into.

Keywords: The words that are protected in the Visual Basic language. These terms are used by the language to accomplish standard tasks, and should not be used for user-created elements.

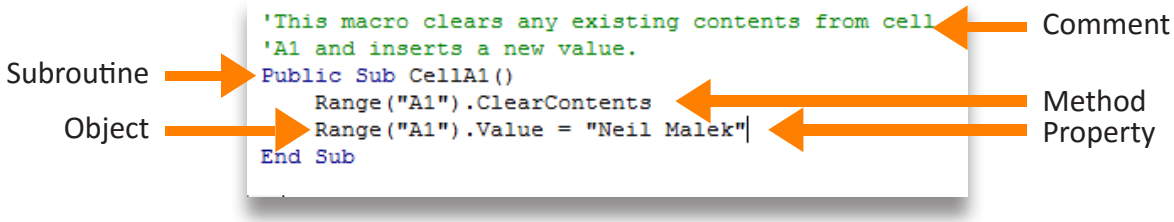
Comments: The plain-English portions of your macro. Since you'll be revisiting your macro after long periods away, you should create comments to read what you were thinking when you originally created the macro.



Objects: In a programming language, in order to affect a real element (like a workbook, worksheet, or cell), that language must have an entry that reflects that element. An *object* in a programming language is effectively the *noun*.

Properties: In the language, these *objects* have attributes (the *color* of a cell, for example), which are called *properties*. If the *object* is the *noun*, the *property* is the *adjective*.

Methods: In the language, these *objects* have actions they can take (*copy*, for example), which are called *methods*. If the *object* is the *noun*, the *method* is the *verb*.



Programming for the Layperson

The entirety of the OFfice suite - Word, Excel, etc. - was written in the programming language *Visual Basic*. As described on the previous page, the various tools that are given to us as users are composed of *objects*, and code that accesses those objects' *methods* and *properties*. For example, if you select cell C3 and press the button to make the contents bold, the command Excel executes would look like this:

```
Range("C3").Select
Selection.Font.Bold
```

A *macro*, then, is a set of these commands that we piece together for our own use. The commands can be as straightforward as the above example (where a single command is executed on a single cell), or they can become vastly more complicated (where the macro *decides* whether to implement a series of commands on various cells *based on conditions you set*).

In this document, we will be *recording* a new macro by telling Excel to watch our sequence of button presses and typing. Afterward, we'll be exploring the code that this recording creates for us, and *modifying* it. Finally, we'll discuss how to *write* a macro without any recording involved.

Recording vs. Writing Macros

Remember that Excel is not an intelligent program; if it *records* the steps you perform, it has no way of understanding *why* you performed those steps, in that order, on those cells. For example, if you *record* a macro that has you clicking in cell C3, later the macro will click into cell C3. This happens even if you selected C3 because it was the first cell with data, and now the first cell with data is C4.

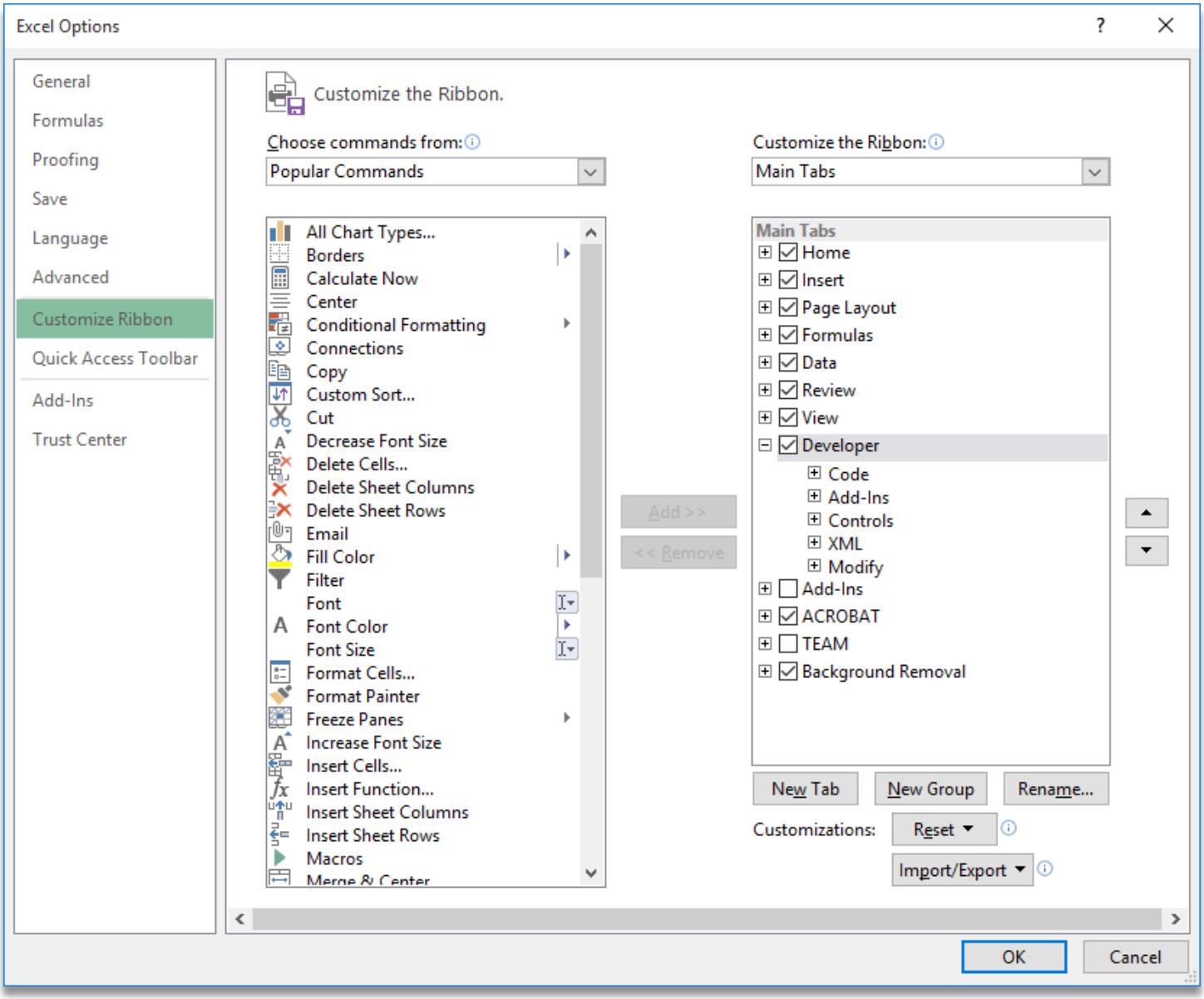
When we write our own macro, the coding language can include decisions and tests - things like *look for the last cell with data* or *continue until you reach a cell greater than 50,000*. Recorded macros cannot include these tests.

Preparation

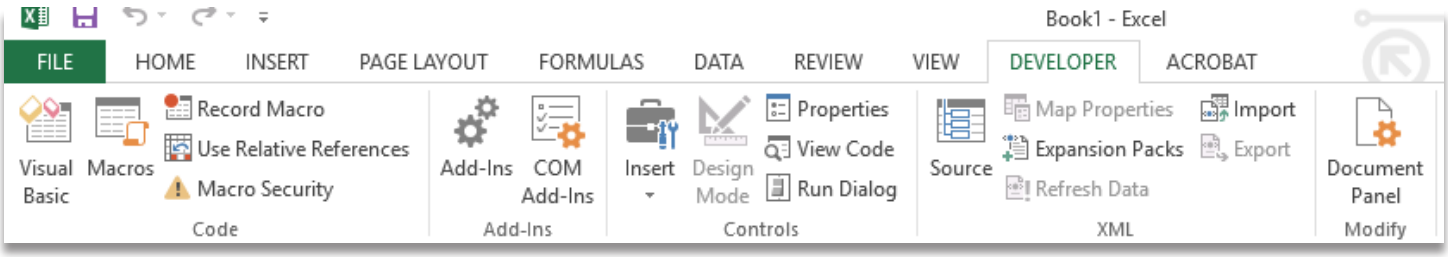
This cannot be stressed enough - the most important step for recording a macro is the *preparation* stage. Once you press the button to begin recording, Excel is watching and recording every click and entry. Begin by determining what should be included in your macro (navigation, selection, and other commands), and what shouldn't. Also, consider that you'll want to understand your macro later - moving back-and-forth from one area to another will mean that the recorded steps are difficult to follow. Organize your thoughts on paper, then start the recording process.

The Developer Tab

In order to record and edit macros, you'll need an additional tab available at the top of the screen - *Developer*. If you don't see it yet, click **File > Options > Customize Ribbon**. Select the checkbox for **Developer**.

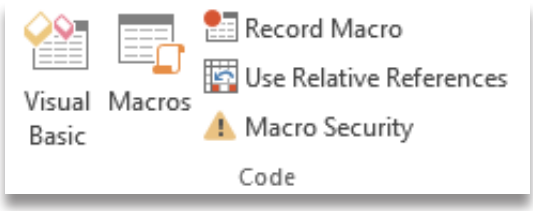


Now your Ribbon should look like this:



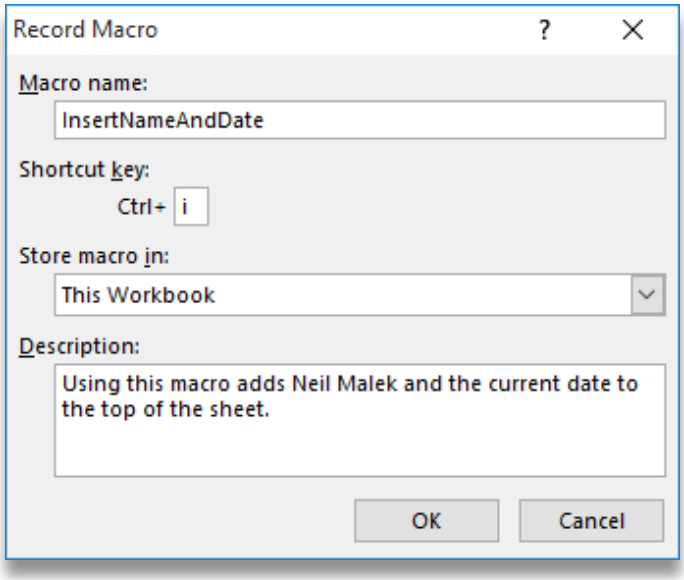
Press Record

On the **Developer Tab**, in the **Code Group**, select **Record Macro**. A new dialog box will appear. Into this dialog box:



Name the macro. It should be self-explanatory, and it cannot have spaces in it. I like to use *camel case* (every first letter of a new word is capitalized). I'll name mine **InsertNameAndDate**.

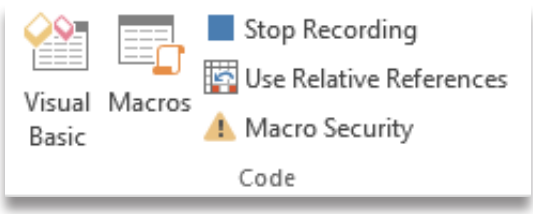
A **Shortcut key** is not essential (you can use your macro without a keyboard shortcut), and it's essential to remember that whatever shortcut you choose will replace the existing shortcut tool. In this example, understand that **Ctrl + i** is the shortcut for *italics*, and you wouldn't be able to use that shortcut in this spreadsheet for italicizing.



The choices for **Store macro in** are **This Workbook** (available only on this file), **New Workbook** (saved into the template for blank spreadsheets), and **Personal Macro Workbook** (a universal source for all files you open on your computer). For our purposes, I'll choose **This Workbook**.

Finally, add any useful **Description** in plain English, so future users of the macro (and you) can understand the point of this code.

Click OK. The recording process will begin.



You'll know that you're actively recording, because the button that used to say *Record Macro* is now a blue square that says **Stop Recording**.

Perform Your Steps

For the purposes of this macro, we'll focus on *absolute reference* steps, which means that if you click cell **A1**, the macro will always go to **A1**. More on the other option soon.

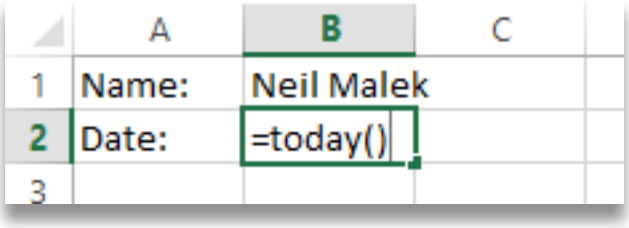
Click in cell **A1**. Type **Name:**

Click in cell **B1**. Type *<your name>*

Click in cell **A2**. Type **Date:**

Click in cell **B2**. Type **=TODAY()**

Press **[Enter]**.

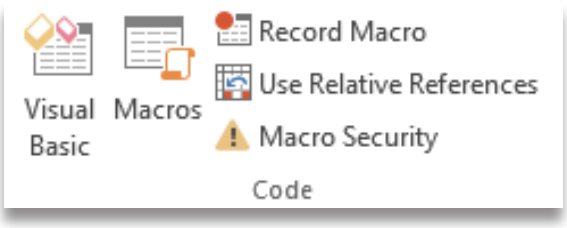


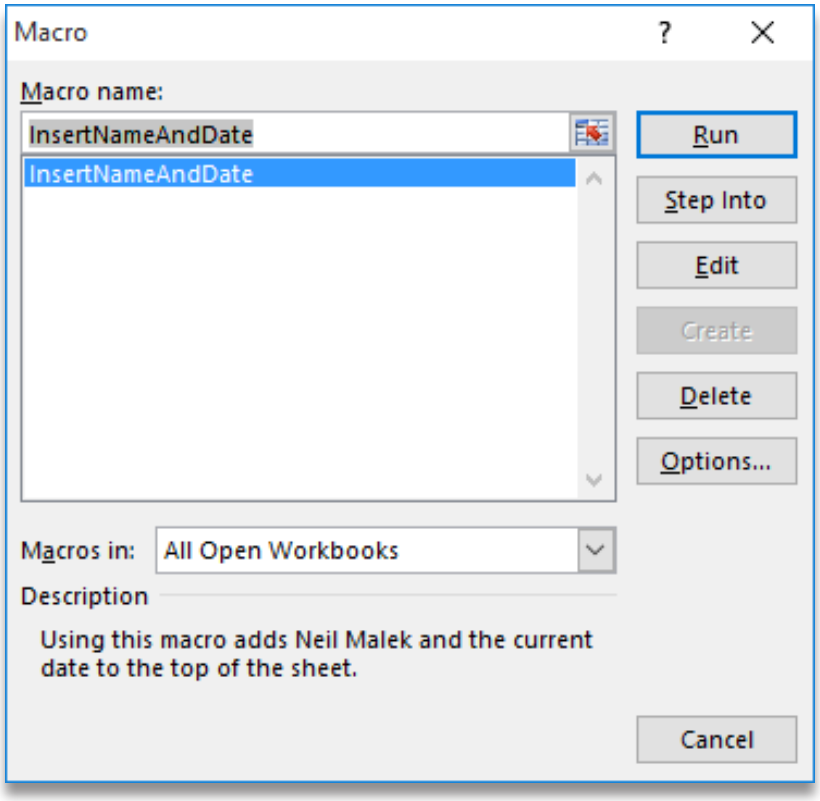
At this point, we've done all we're going to record.

Click **Developer > Code Group > Stop Recording**.

Inspect Your Recording

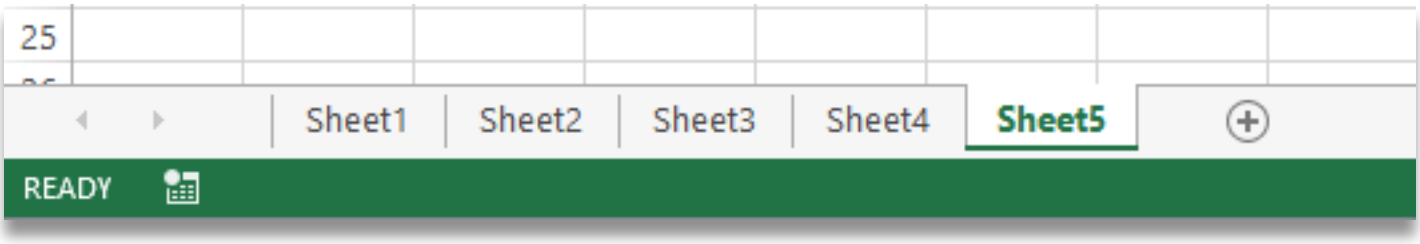
To see the macro you've created, click **Developer > Macros**. This will open the **Macro** dialog box. Any macros you have available will be listed on this dialog.





Use Your Recording

We'll use this macro four times, through the four execution methods you might find useful. To do this, let's create four new, blank sheets. Click the **[+]** button in the tabbed section at the bottom of the screen four times, to create these sheets.

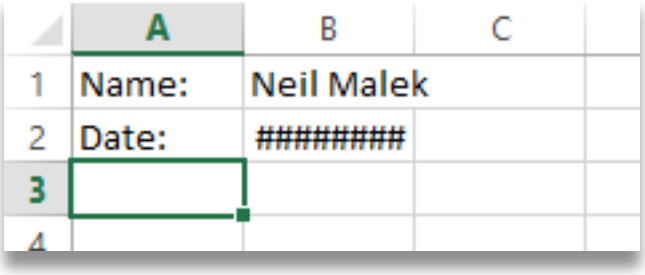


Click **Sheet2**.

Run Macro

Click **Developer Tab > Code Group > Macros**. From the **Macro** dialog box, select your macro (*InsertNameAndDate*), and choose **[Run]**.

What will happen is that the content described in our previous exercise is inserted. Try to **Undo** the operation, and you'll notice that *macros can't be undone*.



Keyboard Shortcut

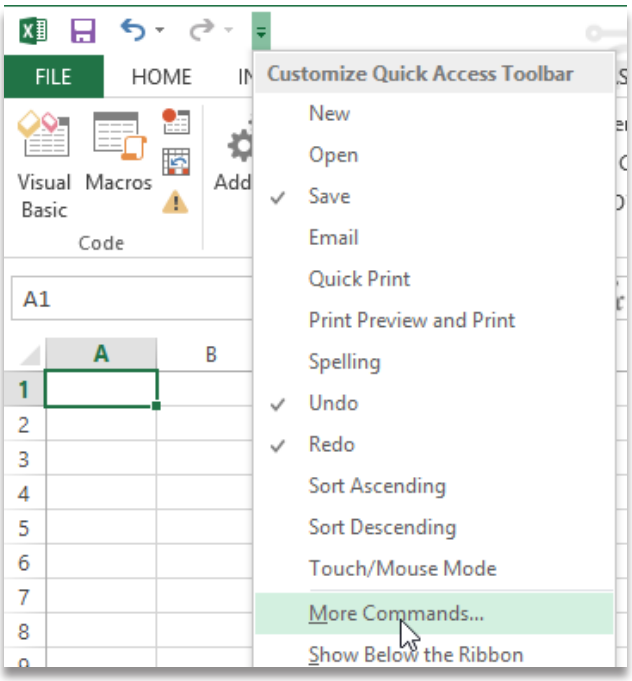
In the previous exercise, we set **Ctrl + i** as the keyboard shortcut for this macro. Select a new, blank worksheet, and use the shortcut you created. Again, this *can't be undone*.

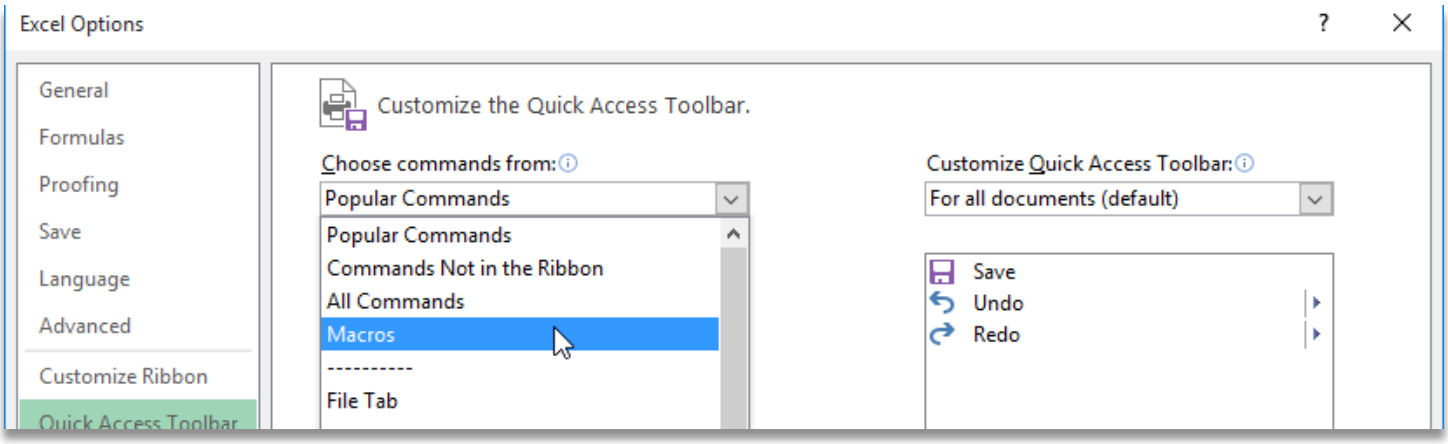
Quick Access Toolbar

So far, we've used a multiple-button-press procedure, and a keyboard shortcut that needs to be remembered to be leveraged. One of my preferred methods is to add a custom button to the **Quick Access Toolbar** (the short toolbar just above the **Home Tab** at the top of the screen.

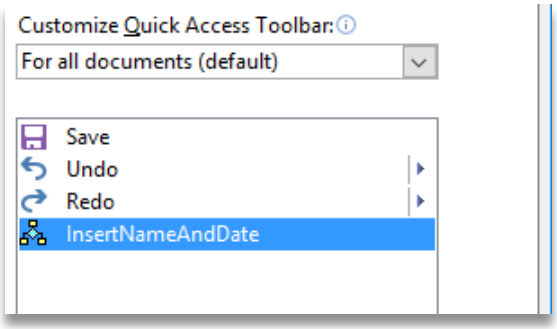
Click the **drop-down menu** on the **Quick Access Toolbar**, and select **More Commands...**

Now, you'll have the **Options dialog** open on the screen. On the left panel, you'll see the drop-down menu says **Popular Commands**. Click the drop-down menu and choose **Macros**, to see the available macros to add to the menu.

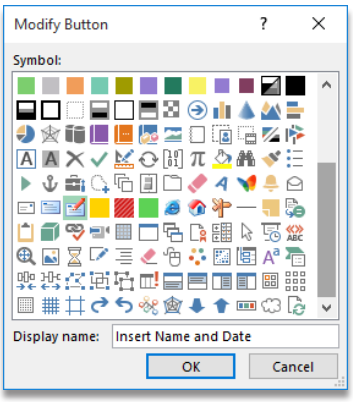




Select the macro you've created (*InsertNameAndDate*), and click the **[Add>>]** button to push the macro to the right panel.



Tweak the button for this by selecting *InsertNameAndDate*, and choosing the **[Modify...]** button at the bottom of the screen. This will give you more available icons to choose from. Also, rewrite the title of the macro to be more user-friendly (as you can see, I've added spaces, and turned the word *And* lowercase).

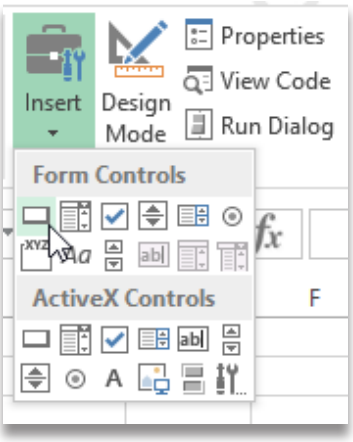


Finally, let's use this work. Click **[OK]** to exit the **Modify Button** dialog, and **[OK]** to exit the **Options** dialog.

Select a new, blank sheet, and click the new button on your Quick Access Toolbar. The content should be inserted.

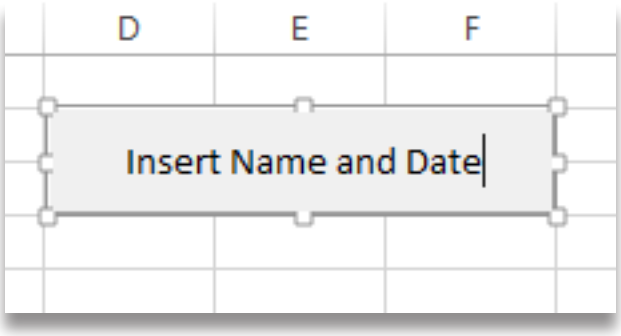
Creating a Form Control Button

To create a button *on the page*, click **Developer Tab > Controls Group > Insert Drop-Down Menu > Form Controls Group > Button**. This will give you a crosshairs cursor.

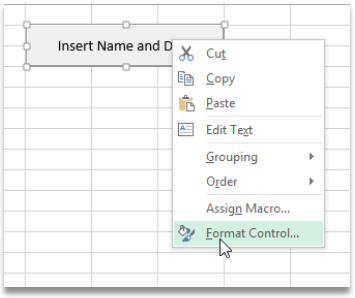


With the crosshairs cursor, **click-and-hold** the top-left corner of the button you want to create, then **drag** to the bottom-right corner of the button, then **release**.

You're asked at this point what macro you'd like to associate with this button. Select our macro, *InsertNameAndDate*.



Now, customize the button's appearance by selecting the text within it and changing the text, or by **right-clicking** the button and choosing **Format Control**.



Defaulting to Absolute References

An *absolute reference* is one in which referring to **A1** *always* points to A1. This seems completely intuitive, but many times in Excel, the way you specify location is by the *distance and direction* from the referencing cell. By default, all macros in Excel use **absolute references**.

Absolute Reference Example

In the example we just finished, clicking into cell **A1** created a reference which told Excel, every time the macro is executed, return to **A1**. This means that the user of the macro could have their selection in cell A1, or C19, or ZZ23. Each execution of the macro returns to A1.

Relative Reference Example

Let's say that I want a macro to highlight three cells in a row - if the user is in cell A14, and the macro is executed, cells A14, B14, and C14 should be turned red. The reality of this macro is that *the actual cell doesn't matter*. In this scenario, if the user was selected in cell A97, you'd want A97, B97, and C97 all turned red.

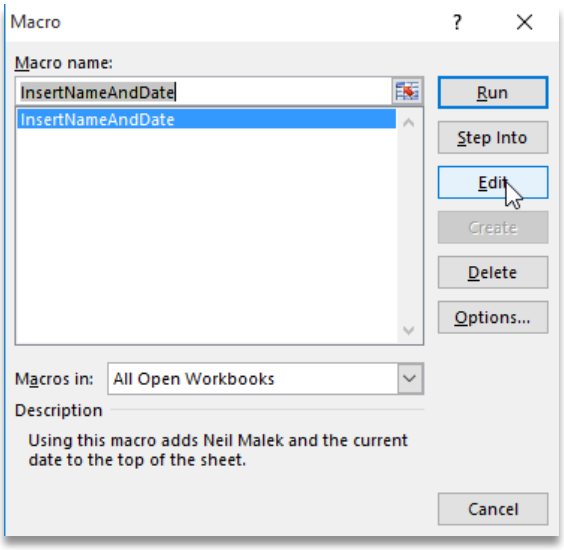
	A	B	C	D	E
1					
2					
3	Ordered Item	Item Description	Product Group	Bus Admin	
4	QOG-8468	Greenlam	ABA-LMN-023443	Reynolds	
5	LVV-6397	Med Quadstring	ABA-LMN-565221	Smith	
6	LOM-6717	Lotcom	ABA-UMA-557631	Michaels	
7	NKW-6186	Subron	ABA-LMN-578843	Jones	
8	LED-9270	Sanla	ABA-UMA-557631	Michaels	
9	RLF-1867	Stattrax	ABA-LMN-023443	Reynolds	
10	VYX-9059	Holdlex	ABA-LMN-023443	Reynolds	
11	NHW-4839	Lamla	ABA-UMA-654475	Sanchez	
12	QVD-3197	Hottone	ABA-LMN-578843	Jones	
13	QGQ-9406	Unait	ABA-LMN-565221	Smith	

Choosing Relative References

At any point, you can click **Developer Tab > Code Group > Use Relative References**, and the macro will concern itself with locations *relative to the selected cell*.

Dig Into the Code

To see the code that was recorded for your macro, click **Developer Tab > Code Group > Macros**. On the **Macro** dialog box, select our macro *InsertNameAndDate*, and click **[Edit]**.

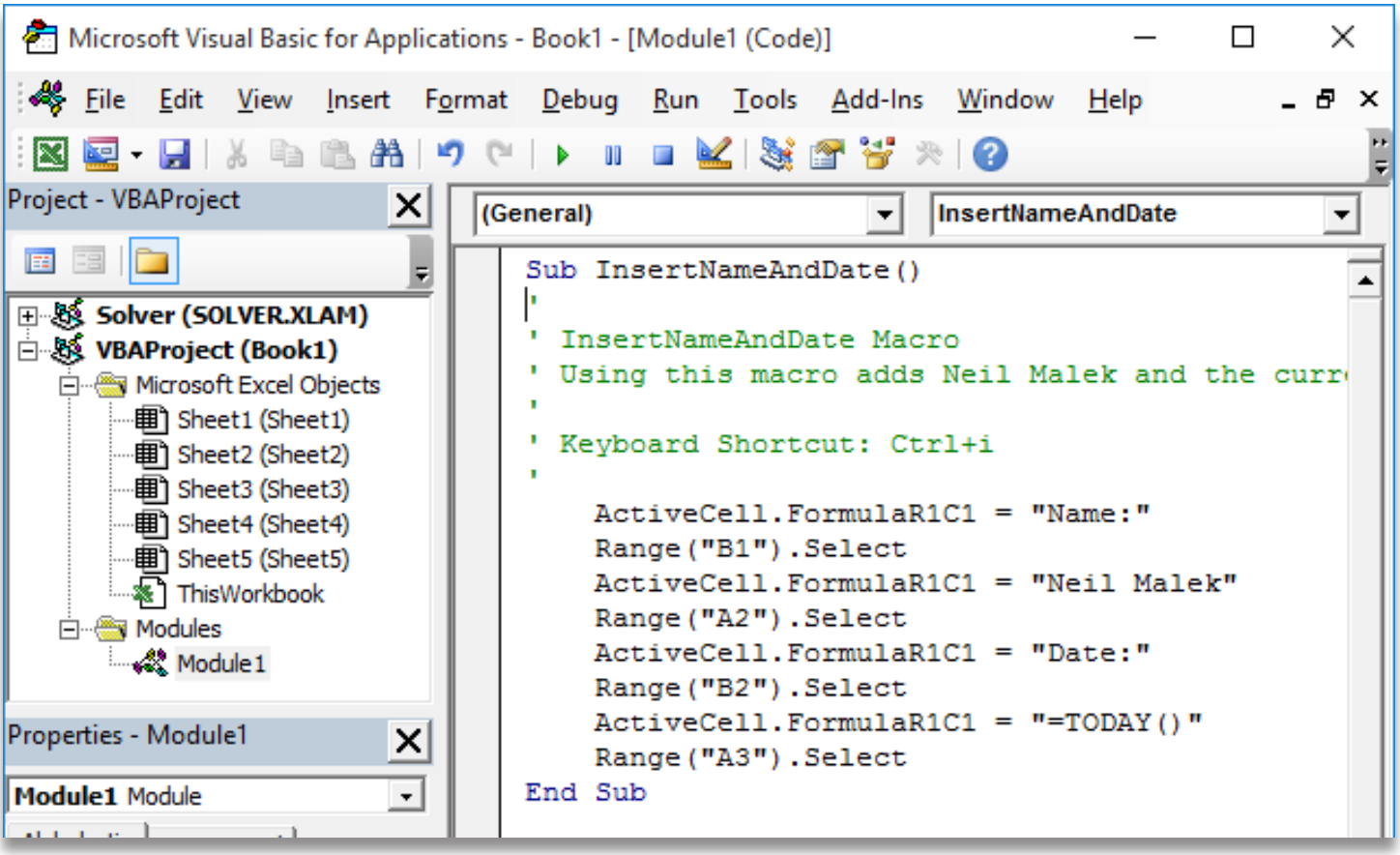


On the new window that opens, you'll see that you're editing **Microsoft Visual Basic for Applications**, for the open workbook (*Book1*).

You'll also see that your code has been inserted (on the top-left panel) into a **Module** called **Module 1**.

Finally, you'll see an open coding window, where your **sub** called **InsertNameAndDate** is entered, including the **comments** in green that you added when we began recording.

This is our macro.



The part of the code we're interested in editing is the indented section - ActiveCell, Range, Select, and all the rest of it. This section includes the commands we want to tweak, so let's take a look at the plain-English version of what we've done:

```
ActiveCell.FormulaR1C1 = "Name:"
```

In this line, we state that we want to affect the **ActiveCell** - aka, the cell you have selected. This was cell A1 in our example. The effect we want to have is to change the **FormulaR1C1**. This means that we're changing the contents of that cell (its *formula*). As you can see, we enter the text **Name:**, which is surrounded by quotation marks.

Note: In Visual Basic, quotation marks around text means that the text is entered *as text*. Without this, Excel assumes everything you type is code to be executed.

This line of code is known as affecting a **property** - you are changing the value entered into the cell, one of its *properties*.

The next line is:

```
Range("B1").Select
```

As you can see, there is a **Range** object in Visual Basic that needs to be set - by telling it that the text **B1** is its value, you can move on to perform an action.

This line of code is known as a **method** - you are performing an action that is already made available in Visual Basic. Whereas properties are the attributes of an object (the value saved into the active cell above), methods are the verbs that can be executed (you can select something).

Modify Something

Let's perform two quick operations here. First, find the line that has your name in it. Mine says:

```
ActiveCell.FormulaR1C1 = "Neil Malek"
```

Simply replace the text inside the quotation marks to a different person.

```
ActiveCell.FormulaR1C1 = "Steven Rogers"
```

Next, change where the date information is entered. The line

```
Range("A2").Select
```

will be changed to

```
Range("C1").Select
```

and the line

```
Range("B2").Select
```

Will be changed to

```
Range("D1").Select
```

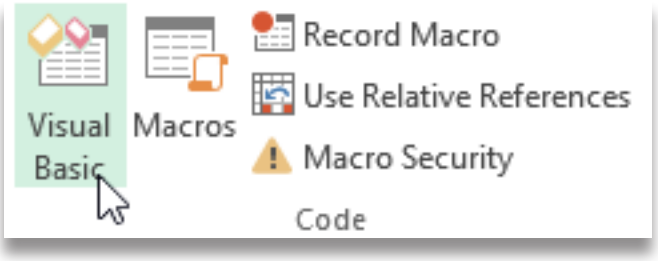
The finished result is:

```
ActiveCell.FormulaR1C1 = "Name:"
Range("B1").Select
ActiveCell.FormulaR1C1 = "Steven Rogers"
Range("C1").Select
ActiveCell.FormulaR1C1 = "Date:"
Range("D1").Select
ActiveCell.FormulaR1C1 = "=TODAY()"
```

Return to your spreadsheet, find a new, blank worksheet, and execute the newly-changed code. You'll see the new name, and the date will be to the *right* of the name information, instead of *below it*.

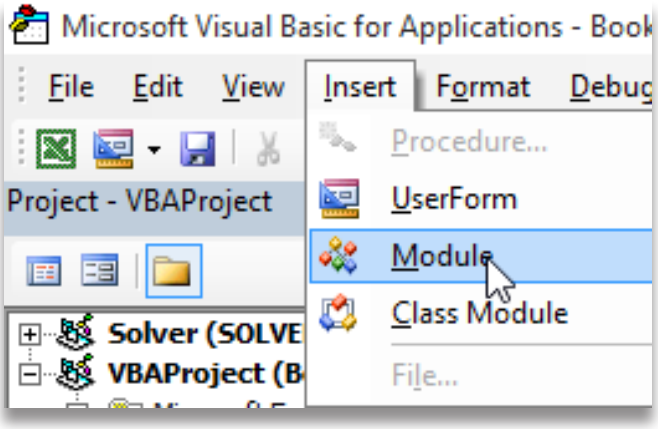
Start a New, Blank Macro

Let's say you wanted to try to create the same macro without using the **Record Macro** button. To start, you'd need the **Visual Basic window** open.



Click **Developer Tab > Code Group > Visual Basic** button. The Visual Basic window will open.

Now, you'll need a container for your code. These containers are called **Modules**. To create a new, blank module, click **Insert Menu > Module**. (not *Class Module*)



A new, blank page will open in the middle of your screen. This is where you'll be typing.

Click in the main body of the window, and type

```
Sub AddDisclaimer()
```

Press **[Enter]**, and this will be added below:

```
End Sub
```

Notice that **Sub** and **End** are both blue. This means that these terms are *keywords* - they are default words used in Visual Basic code, and shouldn't be used by a programmer for other purposes. Your macro's name - *AddDisclaimer* - is black, meaning it is something you made up, and unrelated to any system terms.

```
Sub AddDisclaimer()

End Sub
```

Note: Every sub that is created must have an *end* in order to segment it off from other subroutines.

All code that is to be executed in this macro must be typed between the first line and the last. Use the **[Enter]** key a few times to give yourself some space.

Items to be Worked Upon

Think of everything you want to do - *select cells*, or *create new worksheets*, or *filter data*. Each of these actions has an *object* - the thing that will be worked upon. In the world of programming, we must either know the objects that are given to us, or create our own. Luckily, when we're creating macros, it's almost always using objects you are given.

The most common objects we'll use in our code include:

Application: Excel itself.

Workbook: A spreadsheet that is created in Excel.

Worksheet: A single sheet within a workbook.

Range: A contiguous set of cells.

Additionally, we'll regularly refer to a specific entry - these are the properties:

ActiveWorkbook: The open workbook in Excel.

ActiveSheet: The sheet that is currently being worked upon.

Selection: The cells that are currently selected.

Affecting Properties in Your Code

There are two common uses for objects in code; you either access the *properties* of that object, or you access its *methods*. When you work with the properties, you can get information from the property (e.g. use the *value* that was typed into a cell), or change the property (e.g. change the *cell color*).

Let's look at a few examples:

```
Range("C5").Font.Name = "Cambria"
Range("C5").Font.Size = 24
Range("C5").Font.Bold = True
```

In this example, notice the following:

Range("C5") is used repeatedly. This is our *object*, and the thing that we need to use to get our work done. We could have selected a larger range, but we're only changing a single cell here.

The Font object is used repeatedly. Now that we have gotten to the cell that's important to us, that cell has an entire object called *Font* with many properties.

Every property is set with an equal sign (=). When you are changing the property, you set it equal to something - in this case, the font, size, and bolding.

Cambria is in quotation marks - nothing else is. This is an interesting point. The *Size* property is set to a *number*, and the *Bold* property is set to a *boolean*. These are *data types* - the types of values that are permitted for a given property. Boolean is the term for *true/false* values. Our entry of Cambria is the only one that is purely done in *text format*.

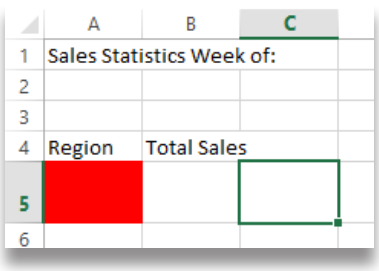
Here's another:

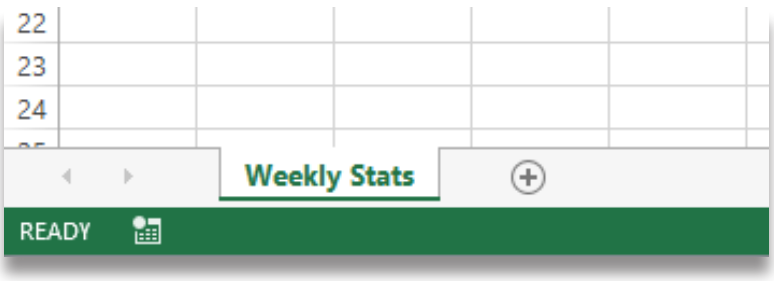
```
ActiveSheet.Name = "Weekly Stats"
Range("A1").Value = "Sales Statistics Week of:"
Range("A4").Value = "Region"
Range("B4").Value = "Total Sales"
Range("A5").Interior.Color = RGB(255, 0, 0)
```

Here, we're using the **ActiveSheet object** - the sheet we have open - to reference the name on the tab at the bottom of the screen. By changing the **property**, you can get a new name.

Additionally, we're accessing the **Value** property of multiple **Range objects** to type in standard information.

Finally, the **Range object** includes an object called its **Interior**. There are many Interior **properties**, but we'll change the **Color** to affect the background color. As you can see, we've set it to red - **RGB(255, 0, 0)**.





Executing Existing Methods

All of the objects in the Excel world have methods associated that can be executed. On the previous page, we used a *property* by typing this structure:

```
ObjectName.PropertyName = Value
```

With methods, we use the same *dot notation* (the object we're referencing, followed by a period, followed by part of that object), but we are simply *executing* the method, instead of setting a value. This means that after the dot, we simply use the method.

```
Range("A1").Select
Range("C4:C1045").ClearContents
Worksheets.Add
```

You'll notice that each of these 'verbs' is simply named, and their command is executed.

Understanding Macro Writing

Now that we've seen all the pieces, we can build a full macro without recording. Open the Visual Basic window, and create a new **Module**. Type the following content, and close the window. Choose your favorite execution method to run the macro to test.

```
Sub MyMacro()
    'Create a title for the spreadsheet in the
    'merged area of A1:D1

    Range("A1:D1").Merge
    Range("A1").FormulaR1C1 = "Weekly KPI Report"
    Range("A1").Font.Size = 24
    Range("A1").Font.Bold = True

    'Column labels

    Range("A3").FormulaR1C1 = "KPI"
    Range("B3").FormulaR1C1 = "Description"
    Range("C3").FormulaR1C1 = "Expected Level"
    Range("D3").FormulaR1C1 = "Week Level"

    'Format cells as table

    ActiveSheet.ListObjects.Add(xlSrcRange, Range("A3:D3"), , xlYes).Name = "KPITable"

End Sub
```

The Concept of Variables

A variable is a container for information. If you need your code to remember something, like the row you're working on, or the name of the user, you'll need to create a variable to remember that information. To name a variable, follow these rules:

- Name must be less than 255 characters
- No spaces
- Must begin with a letter
- No periods

Create a Variable

To create a variable, invent a simple, understandable name, and determine what type of data that variable will use. This could be a *number* data type like an **integer** (whole number) or **double** (decimal number), a **string** (text), or a **boolean** (true/false). Then, you *declare* the variable:

```
Dim password As String
Dim totalSales As Integer
```

Don't worry about the term 'Dim' - it used to be short for the word *dimension*, but programming has evolved to the point where it just means *variable*.

As you can see in the above code, you state that you have a new variable (**Dim**) with a name (**password** or **totalSales**), that is set to a data type (**String** or **Integer**). Now you can use these variables like this:

```
Sub GetInput()
    'Create a variable to remember the user's name
    'and total sales
    Dim userName As String
    Dim totalSales As Double

    'Gather the user name and total sales from
    'the user with input boxes

    userName = Application.InputBox("What is your name?")
    totalSales = Application.InputBox("Total sales this week:", Type:=1)

    'Input those values into the table

    Range("A1").FormulaR1C1 = userName
    Range("A2").FormulaR1C1 = totalSales

End Sub
```

Data Types

As you saw on the previous page, we set the variable **totalSales** to a *data type* called **Double**, and the line of code

```
totalSales = Application.InputBox("Total sales this week:",
Type:=1)
```

...includes a section that says **Type:=1**. The InputBox method we've called for the Application object takes in a textbox of information. If, as in the previous line, we don't put anything after the **prompt**, the *data type* of this information is text, or a **String**. By typing the added element **Type:=1**, we inform the InputBox to accept a **number**. We must, then, also match this data type with our variable - **Dim totalSales As Double**.

It is important for any programming language to successfully handle these data types, because trying to multiply by a word, or handle **T**, **true**, **TRUE**, and **YES** as a *true* statement are much too complicated. As programmers, we must know the data type that is relevant for the situation, and stay consistent.

Common Data Type List

Integer: *whole numbers* between -32,768 and 32,767

Long: *whole numbers* between -2,147,483,648 and 2,147,486,647

Single: *decimal numbers* taking 4 bytes of memory

Double: *decimal numbers* taking 8 bytes of memory

Date: 8 bytes of date information (until the year 9999)

String: Text information

Boolean: **True / False information**

Decision Trees

One of the most essential reasons for writing our macros instead of recording them is the ability to build *decision trees* into our code through the use of **If**, **Then**, and **Else**. Before we can use these tools, we must understand both **conditional operators** and **logical operators**

Conditional Operators

A *conditional operator* is the key to using an **If** statement. If asks whether a test executes as *true* or *false*. The conditional operator is the *greater than*, *less than*, or other test.

List of Conditional Operators

=	Equal to (valid for text or number values)
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to

Logical Operators

Many times, a single conditional test will not be adequate for what we're attempting to do. A *logical operator* strings multiple *conditional operators* together.

List of Conditional Operators

And	Both conditions must be true
Or	One or both conditions must be true
Xor	One condition must be true, but not both
Not	Negates true statement

Creating an If Statement

First, we gather any necessary information to make our judgment on the conditions. Then, we place the conditions into an If statement, and test whether they are true. Here's an example:

```
Sub ManagerCheck()
    'Create variable to gather information from the user
    Dim userType As String

    'Get user type from the user
    userType = Application.InputBox("Are you a manager? Y/N")

    'Test the input
    If (userType = "Y") Then
        Range("A1").FormulaR1C1 = "Welcome manager"

    ElseIf (userType = "N") Then
        Range("A1").FormulaR1C1 = "Welcome user"

    Else
        MsgBox ("I didn't recognize your input")

    End If

End Sub
```

Notice in this example: there is both an **If** and an **ElseIf** - If is only used for the first check, and every other check in the same **If block** is done with an **ElseIf** statement. Also notice that, just as there is **End Sub** at the end of the macro, there is **End If** at the end of the block. This *must be* in place to finish the test properly.

Programming Loops

A programming *loop* is a section of your program that can be repeated until some condition is met. They are used very frequently to perform the same section of code on multiple rows of content - you perform the section of code once, then go to the next row and *loop* through the code a second time. This is the other deciding factor in *recording* vs. *writing* macros, as it is impossible to record a loop.

There are two fundamental loop types in programming - **For loops** and **While loops**. Let's look at the differences and similarities:

For Loops

If you **know the number of executions you need**, you'll be using a **for loop**. The first thing you'll do is create a **counter**, and the for loop will watch the counter to see whether it should execute the loop again. Here's an example:

```
Sub FormatFirstXRows()
    'Create a counter for the number of rows you want
    Dim counter As Integer
    Dim numberOfRows As Integer
    Dim rowToFormat As String

    'Get user input
    numberOfRows = Application.InputBox("How many rows are we formatting?", Type:=1)

    'Run the loop
    For counter = 1 To numberOfRows
        rowToFormat = "A" & counter & ":G" & counter
        Range(rowToFormat).Interior.Color = RGB(230, 230, 255)
    Next

End Sub
```

In this example, we're using *variables* to determine *how many times we have performed the operation* and *how many times it was requested*. If the user types in **5**, the counter will increase until it has been performed 5 times. The **rowToFormat** variable will keep changing from *A1:G1* until *A5:G5*.

While Loops

The **While loop** idea performs a given set of code, but only *until it matches a criteria*. In the previous example, we used a counter as our criteria. In the while loop, we often have a condition that is calculated along the way. Here's an example:

```
Sub FindPointOfBreakeven()
    'Create variable for the desired result, and one
    'for the current value
    Dim breakevenSales As Double
    Dim currentMonthSales As Double
    Dim currentRow As Integer

    'Set the breakeven value
    breakevenSales = Application.InputBox("What is our breakeven point for sales this month?", Type:=1)
    currentMonthSales = 0
    currentRow = 2

    'Run the loop
    Do While currentMonthSales < breakevenSales
        currentMonthSales = WorksheetFunction.Sum(ActiveSheet.Range("A2:A" & currentRow))
        currentRow = currentRow + 1
    Loop

    'Mark the point we crossed over
    Range("A" & currentRow).Interior.Color = RGB(255, 0, 0)

End Sub
```

In this example, we use *variables* to compare the *running sales total* against our *breakeven value*. In the InputBox, we have the user tell us what our breakeven number is. Then, starting at row 2, we keep adding up the values in that column until we find our breakeven total.

The loop then stops, and the last line of code is executed - take the current cell and mark it in red.

In a **while loop**, the code within the loop executes over and over *until the condition is met*. This compares with our **for loop** earlier, which executed over and over *until the counter expired*. They are very similar tools, and can often be used interchangeably.