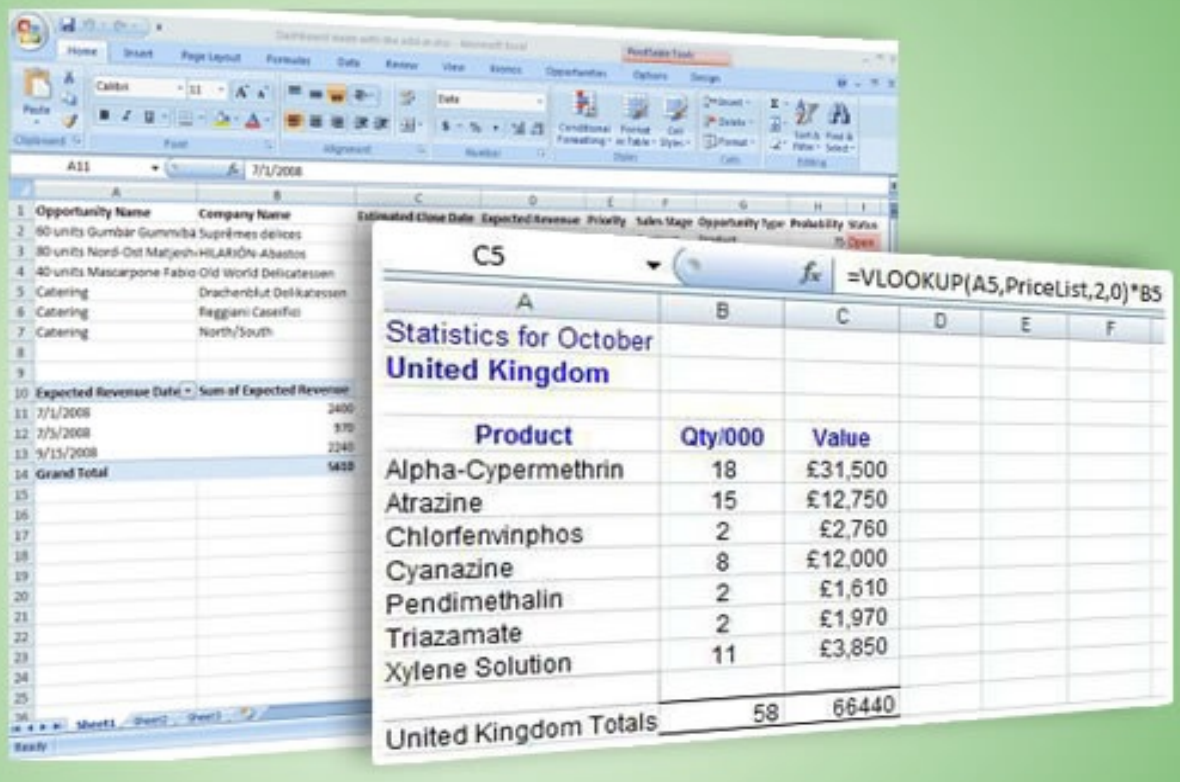
Mastering Excel Macros

Function Procedures - Book 11



Mark Moore

Mastering Excel Macros

Book 11 – Function Procedures

Mark Moore

Introduction

Welcome to another Mastering Excel lesson. If you have previous lessons, thanks for sticking around. If you are new, I hope you enjoy the lesson. The lessons are easy going, relaxed, no-nonsense and easy to understand. I try my best to explain complex topics in a simple and entertaining way. My goal is that you will finish reading each lesson and have immediately applicable skills you can use at work or home.

If you want to work along the exercises in this lesson (I strongly recommend this) please go to my website and download the follow-along workbook.

My website is: http://markmoorebooks.com/mastering-excel-macros-function-procedures/

A bit of clarification on how to get the follow-along workbooks. You will input your name and email address. You will receive a confirmation email. Once you confirm, you will receive a second email with the follow-along workbook.

Why do I do this?

I can’t package an Excel file with an eBook. Amazon will not allow it. Also, the only thing I do with your email is send you the workbook and periodically send you updates about new lessons that I am working on.

What Are Function Procedures?

Function procedures are similar to subroutines. In case you forgot (because I explained it way back in Lesson 1), when you create a macro, you always start it with Sub ( ). Sub is short for subroutine. All macros are subroutines. Anyway, why are they ‘sub’routines? Because they all run inside Excel which is the ’main’ routine.

Ok, back to function procedures (functions) for short. Subroutines run and do stuff; they can resize columns, change fonts, perform calculations, etc. Functions can perform calculations and can return values. You can have extremely complex calculations in a function and then call the function by its name. The function will return the result of all the calculations.

Think of the VLOOKUP function. That function searches through a data set and returns a match from a specified row and column. Can you imagine the tediousness of programming that lookup logic every single time? Instead, you can encapsulate the logic in the VLOOKUP function and just use that.

You can use functions in a few ways:

1. You can call a function from a VBA subroutine. This is good practice when you have to do the same calculation over and over again. Separate out the logic into a function and have the main subroutine call the function when needed.
2. You can call functions directly from Excel. Yes, you can type in =MyFunction into a cell and your custom calculations will be performed and the answer will appear in the cell.
3. In conditional formatting formula. As long as your function returns TRUE or FALSE, you can use it to drive your conditional formats.

Here’s how I differentiate normal macros and functions (and how I decide when to use one or the other).

Macros (Subroutines)

- Use when I need to change something in a worksheet/workbook
- Macros generally need user interaction to execute. A user must click a button or otherwise ‘do’ something to execute the macro

Functions

- Use when I have complex math that I need to simplify
- Use when I want the calculations to execute without user intervention (functions are executed whenever Excel calculates)

Important note: A defining feature of function procedures is that a value **is always assigned** to the function. This is how it returns a value.

- Don't name functions with a name that is also a cell address. In other words, don't create a function called abc444 since that is also a cell address.
- Don't name functions that match existing Excel functions. In these cases, Excel will use the native Excel function, not yours.

Limitations

One thing that functions cannot do is change anything on a worksheet. They can't change the column width, change a cell color or anything like that. The only thing a function does is perform a calculation and return a value.

Sharing functions is a bit tedious. If you send a worksheet to a coworker that does not have the custom code, Excel will not know how to use the function. I will show you several ways to share your functions later in this lesson.

Speed. Compared to native Excel functions, custom functions are very slow. I'm talking slow in computer terms, nanoseconds vs. seconds. Realistically speaking, you might not ever notice a difference in your workbooks unless you have thousands of custom functions.

Enough theory, let's get started with the hands on part of the lesson, shall we?

Functions with No Arguments

An argument is data that you pass to a function. Consider this formula, =SUM(A1:A10). This SUM function has the argument A1:A10), which is a range.

Most functions need arguments, but Excel has several built in functions that do not need arguments. For example you can type in =NOW() or =TODAY() into a cell and those functions will return a value. No argument there! (hahahah…I thought it was funny…)

You are going to write a function that does not need any argument. This function will return the active printer Excel is using.

1. Open the follow-along workbook, FunctionProcedures.xlsm.
2. Go to the Visual Basic macro editor and create a new module named Module1.
3. Type in Function MyPrinter As String.
4. Press Enter.

You should have this in your module:

```
Function MyPrinter() As String

End Function
```

- All functions begin with the Function keyword.
- After Function, you type in the name of the function. In this case, it is MyPrinter.
- 'As String' determines the data type returned by the function. This is optional but it is a good idea to use this. If you don't use the data type returned, Excel will assign the Variant data type. In most cases, this is ok.
- All functions end with an 'End Function' statement.

5. Between the first and last line, type in this line:

```
MyPrinter = Application.ActivePrinter
```

This is the completed function.

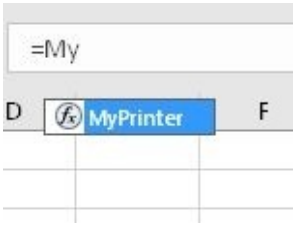
```
Function MyPrinter() As String

    MyPrinter = Application.ActivePrinter

End Function
```

Returning Values

How does a function return a value? Notice in the function above you set the function name (MyPrinter) equal to the result of Application.ActivePrinter. That's how a function returns a value to a cell (or another macro). You set the answer equal to the name of the function.

6. Save the workbook.
7. Go to Sheet1 in Excel.
8. In any cell you like, type in =MyPrinter ().

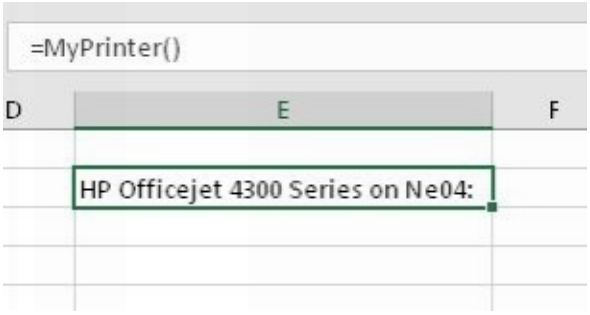
Whoa. Did you notice that? As soon as you type in =My, this appears:



Excel knows that there is a new function and it's trying to help you out by showing it in formula pop up.

9. Press Enter.

You should have the name of the active printer appear in the cell. This is my printer (it's old but still kicking!).



You just extended Excel's capabilities by creating a new function. Granted, this function is very simple, but it might be useful in certain environments. For example, if you need to format your worksheet to print in black and white for certain printers, you can use this

function to drive other macros. Alternatively, you can use this function to document where the workbook was printed.

Functions with One Argument

Like regular Excel functions, arguments are passed to a function inside the parentheses after the function name. You can then use the arguments in the calculations inside the function.

Let's build a function that calculates the letter grade based on a numerical score. If you were going to do this without a custom function, you would either build a long and complicated nested IF statement or use a VLOOKUP table. The third option is to use a custom function.

1. Go to Module1 in the follow-along workbook.
2. Insert a new function called GetGrade. The return value should be string (since the letter grade is text).

Note that you can have multiple functions in one module.

```
Function MyPrinter() As String

    MyPrinter = Application.ActivePrinter

End Function
Function GetGrade() As String

End Function
```

The GetGrade function as currently written does not accept any arguments. You are going to change that in the next step.

3. Input Score in the parenthesis following the function name

```
Function GetGrade(Score) As String

End Function
```

That's it. Now a user can pass a cell value to the GetGrade function. Now you have to input the logic to calculate the letter grade based on a score.

Let’s use this table to see the letter grade to score relationships.

Grade	Score
A	90-100
B	80-89
C	70-79
D	60-69
F	0-59

I am going to use the Select Case statement in the function to calculate the score.

4. Input the select case logic in the function:

Select Case Score

Case 90 To 100

GetGrade = “A”

Case 80 To 89

GetGrade = “B”

Case 70 To 79

GetGrade = “C”

Case 60 To 69

GetGrade = “D”

Case Else

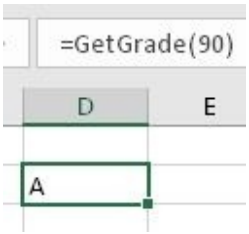
GetGrade = “F”

End Select

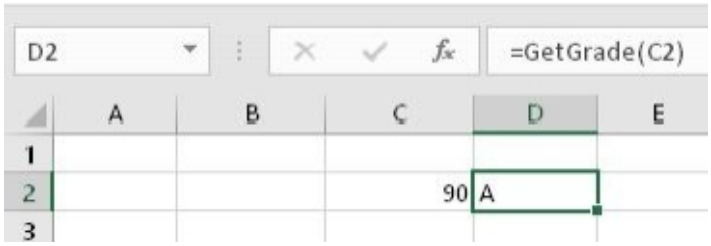
The macro should look like this:

```
Function GetGrade(Score) As String

    Select Case Score
        Case 90 To 100
            GetGrade = "A"
        Case 80 To 89
            GetGrade = "B"
        Case 70 To 79
            GetGrade = "C"
        Case 60 To 69
            GetGrade = "D"
        Case Else
            GetGrade = "F"
    End Select

End Function
```

5. In Excel, type in =GetGrade(90). The letter grade A should be returned.



In the example above, I typed in the 90 directly into the function; however, I can just as easily use a cell reference.



Functions with Two Arguments

Let's add some functionality to the GetGrade function. Let's suppose that we also want to include a grading curve. The data for the curve is shown below.

Grade	Score	Curve
A	90-100	80-100
B	80-89	70-89
C	70-79	60-79
D	60-69	50-69
F	0-59	0-49

Let's do **two** cool things with this function:

1. Let's add a second argument that will let us choose if we want to apply a curve or not.
2. Let's make the second argument optional. If we don't put in the second argument, the function will not apply the curve.

This is the function you are using as a starting point:

```
Function GetGrade(Score) As String

    Select Case Score
        Case 90 To 100
            GetGrade = "A"
        Case 80 To 89
            GetGrade = "B"
        Case 70 To 79
            GetGrade = "C"
        Case 60 To 69
            GetGrade = "D"
        Case Else
            GetGrade = "F"
    End Select

End Function
```

Adding a Second Argument

To add another argument, you need to include it after Score, and separated by a comma.

Curve will be either a Y or a 1. In order to handle both numeric and string data types, you will need to declare the argument as a Variant data type. Also, there is a good reason to use Variant data type, you can use the IsMissing command in your code. I will explain this command below.

3. Change the GetGrade function so it reads GetGrade(Score, Curve as Variant)

```
Function GetGrade(Score, Curve As Variant) As String

    Select Case Score
        Case 90 To 100
            GetGrade = "A"
        Case 80 To 89
            GetGrade = "B"
        Case 70 To 79
            GetGrade = "C"
        Case 60 To 69
            GetGrade = "D"
        Case Else
            GetGrade = "F"
    End Select

End Function
```

Ok, now GetGrade is dealing with the Score and the Curve argument is not being used. You need to alter the code so that the calculations are switched based on the value of Curve. We want to make this function flexible. We need to account for the fact that the value for Curve can be 1, 0, ’Y’ or ‘N’.

There are a lot of ways to use two grading scales. I’m going to keep it simple. I mean, the entire point of writing a custom function is to keep the calculations simple. What good does it do to craft some complicated logic and then hide it in the function? I say, simple is better. Not to mention that three, six or twelve months later when I have to return and change this function, I don’t want to have to figure out what I did.

What I am going to do is use an If statement with two Select Case statements inside them. It will make sense when you see the code. And without further ado, here it is!

4. Change the GetGrade function to match what appears below:

```
Function GetGrade(Score, Optional Curve As Variant) As String

    If IsMissing(Curve) = True Then 'curve is missing
            Select Case Score
                Case 90 To 100
                    GetGrade = "A"
                Case 80 To 89
                    GetGrade = "B"
                Case 70 To 79
                    GetGrade = "C"
                Case 60 To 69
                    GetGrade = "D"
                Case Else
                    GetGrade = "F"
            End Select
    ElseIf Curve = 1 Or UCase(Curve) = "Y" Then 'apply curve
            Select Case Score
                Case 80 To 100
                    GetGrade = "A"
                Case 70 To 89
                    GetGrade = "B"
                Case 60 To 79
                    GetGrade = "C"
                Case 50 To 69
                    GetGrade = "D"
                Case Else
                    GetGrade = "F"
            End Select
    Else ' do not apply curve
            Select Case Score
                Case 90 To 100
                    GetGrade = "A"
                Case 80 To 89
                    GetGrade = "B"
                Case 70 To 79
                    GetGrade = "C"
                Case 60 To 69
                    GetGrade = "D"
                Case Else
                    GetGrade = "F"
            End Select
    End If

End Function
```

Yeah, I know there's a lot going on there but each step is simple. Let's go through it. I'm going to focus on the *why*. If you have worked through the previous lessons, you already know the *how* of If statements and Select Case statements.

Let's re-group.

You have a function that will return a letter grade based on a score. Users have the option of using a curved scale or a standard scale. Users can indicate that they want to use a scale by inputting a 1, "y", or "Y". If users forget to put a curve indicator, the function will use the standard scale.

Logical Step 1

```
If IsMissing(Curve) = True Then 'curve is missing
        Select Case Score
            Case 90 To 100
                GetGrade = "A"
            Case 80 To 89
                GetGrade = "B"
            Case 70 To 79
                GetGrade = "C"
            Case 60 To 69
                GetGrade = "D"
            Case Else
                GetGrade = "F"
        End Select
```

You need to check if the Curve argument is missing. If it is, then you need to apply the standard scale. Note that IsMissing can only be used with Variant data types.

Logical Step 2

```
ElseIf Curve = 1 Or UCase(Curve) = "Y" Then 'apply curve
        Select Case Score
            Case 80 To 100
                GetGrade = "A"
            Case 70 To 89
                GetGrade = "B"
            Case 60 To 79
                GetGrade = "C"
            Case 50 To 69
                GetGrade = "D"
            Case Else
                GetGrade = "F"
        End Select
```

Now you need to see if the user wants to apply the curve. The valid values are 1, “y”, or “Y”. You will not be sure if the user decides to input a lower case “y” or an upper case “Y”. (Yes, they are different.) The If statement checks if Curve = 1 and then it does something clever, it uses UCase to convert the value in Curve to uppercase. Now regardless if the user inputs “y” or “Y,” it has been converted to a “Y”. If any of these conditions are True, the curved scale will be used.

Logical Step 3

```
Else ' do not apply curve
        Select Case Score
            Case 90 To 100
                GetGrade = "A"
            Case 80 To 89
                GetGrade = "B"
            Case 70 To 79
                GetGrade = "C"
            Case 60 To 69
                GetGrade = "D"
            Case Else
                GetGrade = "F"
        End Select
```

If the Curve variable is not missing and the user does not want to apply the curve, then the function uses the standard grading scale.

One important note, you cannot mix IsMissing with other Or clauses. That is why I had to put IsMissing separate in Logical Step 1. If it is possible to mix them and you know how, send me an email and let me know.

The new functionality of the GetGrade function makes calculating letter grades much easier. Notice how you had to take into account user inputs. Users could input multiple values for the Curve argument. You had to account for every possibility.

And you know, you haven't covered every possibility. Suppose a user inputs ".90" instead of 90? Do you want to add code to account for that possibility or do you want to just return an error or return an F? There's no right or wrong answer, it all depends on how flexible you want the function to be.

Does this exercise give you a bit of a glimpse into how much programming effort went into creating over 300 built-in Excel formulas?

Testing

You aren't done with GetGrade yet. You need to test it. Granted, you could just put in a number in Excel but what if you get an error or the value is incorrect? How do you test functions? That's what I am going to cover in this section.

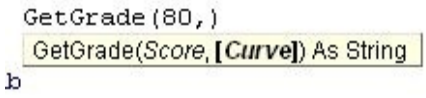
Testing Using a Subroutine

You can use a function in a subroutine just like any other command. To test GetGrade, you

can create a temporary subroutine and call Get Grade.

1. Create a new subroutine. I called mine Test.
2. Type in temp = GetGrade(80, 1)

Did you see that? When you type in GetGrade, Excel treats it as if it was a built-in function and it shows you the arguments.



The testing macro looks like this:

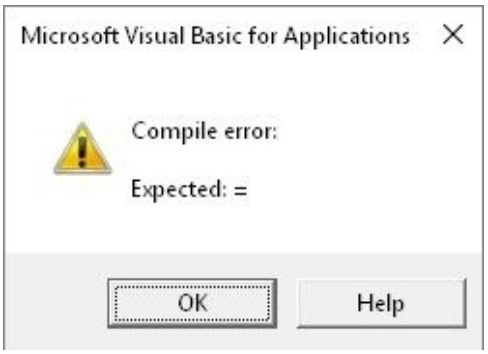
```
Sub Test()

temp = GetGrade(80, 1)

End Sub
```

3. Click inside the Test macro and press F8 to move the execution line by line.
4. Change the parameters to make sure the correct letter grade is returned.

Note:

This will drive you crazy, so just let me tell you about it now to save you untold amounts of frustration.

Sometimes, when testing functions or other VBA code, you will see this error:



What does this mean? It’s not a very helpful error.

I got this error when I used this code:

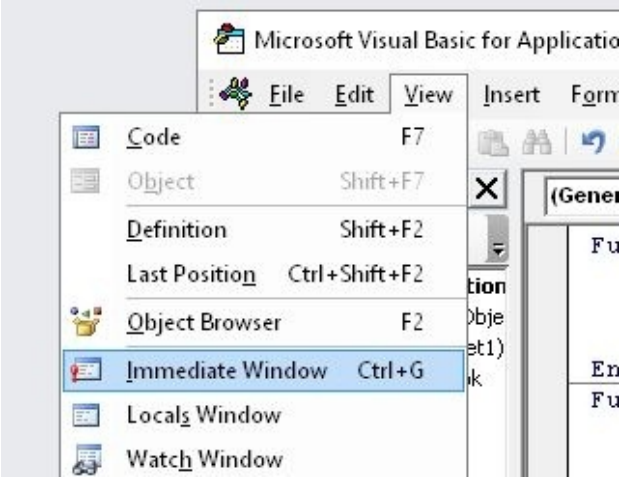
```
Sub Test()
GetGrade(80, 1)
End Sub
```

GetGrade is causing the error. Why? Because GetGrade is a function. Functions always return values. You can’t call a function and not assign it to a variable. In other words, Excel won’t know where to put the answer (i.e. the letter grade). You need to set GetGrade equal to a variable so the answer can be placed there. That’s why you used temp = GetGrade(80,1) so that the letter grade gets loaded into the temp variable.

Testing Using the Immediate Window

This test isn’t intended to let you move line-by-line through your function. For that, test using a subroutine. However, you don’t always have to input your function into Excel to see the answer. You might be switching back and forth a lot between the macro editor and Excel. Instead, you can test the function in the Immediate window.

5. In the macro editor, go to View > Immediate Window.



A blank window will appear at the bottom of the macro editor.

6. Type in ?GetGrade(90,1).
7. Press Enter.

The corresponding letter grade will be displayed.

Yeah, it's kind of strange but if you want the immediate window to show you a result, you have to start the line with a ?.

Functions with Range Arguments

So far you have only written functions that accept single cell values or single numbers. You can also create functions that accept ranges of values. The way to do that is to set up the function to accept an array of values. Inside the function you need to loop through the array and perform the calculation on each individual value.

Let's create a function that subtracts 1 from each value then sums them all up.

1. Create a new function called MinusOneSum that returns a Long data type.

```
Function MinusOneSum() As Long

End Function
```

2. Enter the parameter argument (in between the parentheses).

```
Function MinusOneSum(ParamArray list() As Variant) As Long

End Function
```

Note:

- The array must be the only, or the last argument in the argument list
- The array must be defined with the ParamArray keyword
- Even though it doesn't say Optional, Excel sets the array as optional

Now you have to process each item in the list. You need two loops, one to process each argument and another to process each cell in each argument.

```
Function MinusOneSum(ParamArray list() As Variant) As Long
Dim c As Range
    For Each arg In list
        For Each c In arg
            temp = temp + (c.Value - 1)
        Next c
    Next arg
    MinusOneSum = temp

End Function
```

The first loop (i.e. For Each arg in list) is a bit confusing in this example. But bear with me for a second. Ignore that for now. You declared a variable “c” as a range. You then loop through each variable, subtract 1 and add it to the temp variable.

At the end of the function, you set the function equal to the temp variable so the function can return the aggregate value.

Now, let’s review the first loop.

The first loop (For Each arg in list) processes all the ranges you chose in the function. For this function =MinusOneSum(“A1:A2”), the loop only processes one arg (A1:A2). List only has one arg: A1:A2.

Now consider this function: =MinusOneSum(A9:A10,B9:B10).

This new function has two ranges. This means that list now has two args: A9:A10 and B9:B10. So the outer loop says ‘I have two arg to process’. *For Each* arg, the inner loop says ‘I need to process each cell’.

Using the two-loop method, you can add as many ranges as you like to this function. The follow-along workbook has a MinusOneSum worksheet that has additional examples.

Controlling Calculation

Custom functions are only recalculated when any of its arguments change. If you are using Automatic calculation mode, you can force the function to recalculate more frequently.

Type in:

```
Application.Volatile True
```

to make the function recalculate whenever any cell is changed in the worksheet.

Use:

```
Application.Volatile False
```

to cause the function to be recalculated only when any of its arguments are changed.

Returning an Error Value

Sometimes you need your function to return an error value. You don't want to return the string "#N/A" because although it looks like an error, it isn't one. Other Excel formulas and error-related features will not detect that as an error, they will think it is a string… because it is.

Let's work through a new function and include returning an error.

	A
1	Part Numbers
2	876-90-ICL
3	987-987-OLT
4	9-9999-TTI
5	900-42-OOL
6	564-4212
7	

Suppose you routinely get lists of part numbers that are like the image above. You need to get the middle part of the part number, (the part between the dashes) and use it in another part of the worksheet. Notice that the middle part varies in length and that someone messed up that last part number, there is only one dash. Let's write a function that extracts the middle portion of the part number.

1. In the macro editor, create a new function called ExtractPN. The function will accept one argument and return a variant data type.

```
Function ExtractPN(PN) As Variant

End Function
```

You could use the InStr VBA function to find the position of the first dash, then use it again to find the position of the second dash and finally return the text between both dashes. That's a perfectly acceptable solution. However, I'm not going to do that, I'm going to show you another way to do it. I'm going to use the Split function.

The Split function is only available in VBA. It accepts a text argument and a delimiter argument. The function then splits out the text at each delimiter into an array. Basically, it works like the text-to-columns feature in Excel but instead of splitting into columns, it

splits into an array.

If this still confuses you, this image should help you visualize the effect:

Array Position --->	0	1	2
	876	90	ICL

After using Split, the function needs to retrieve the second item from the array.

2. Add the Split function that splits out the PN argument by dashes. Remember that Split returns an array not a single value.

SegmentArray = Split(PN, “-“)

3. Write the line of code that retrieves the second item of the array:

Segment = SegmentArray(1)

You want the second item, why are you using the number 1? Because arrays start counting at 0. Look at the image above and you’ll see that the 90 is in position 1.

The function should look like this now:

```
Function ExtractPN(PN) As Variant

    SegmentArray = Split(PN, "-")
    Segment = SegmentArray(1)

End Function
```

At this point, the function will split the part number at each dash and populate the Segment variable with the second segment. Looking back at the data, you can see that the last part number will not be right.

Well, let me be **very** precise. The function will return the second segment, 4212 without error (there is no **functional** error). However, there is a **logical** error. Part numbers need three sections; this last one has two sections. This is probably a typo, you are going to add logic so that the function catches this error and returns an error.

	A
1	Part Numbers
2	876-90-ICL
3	987-987-OLT
4	9-9999-TTI
5	900-42-OOL
6	564-4212
7	

An easy way to catch these typos is to count the number of items in the array after it is split. If there are not three items, return an error.

To return an error, you need to use the CVError command. You also need to specify which type of error you want the function to return.

Here is a list of possible error types:

- xlErrDiv0 for #DIV/0!
- xlErrNA for #N/A
- xlErrName for #NAME?
- xlErrNull for #NULL!
- xlErrNum for #NUM!
- xlErrRef for #REF!
- xlErrValue for #VALUE!

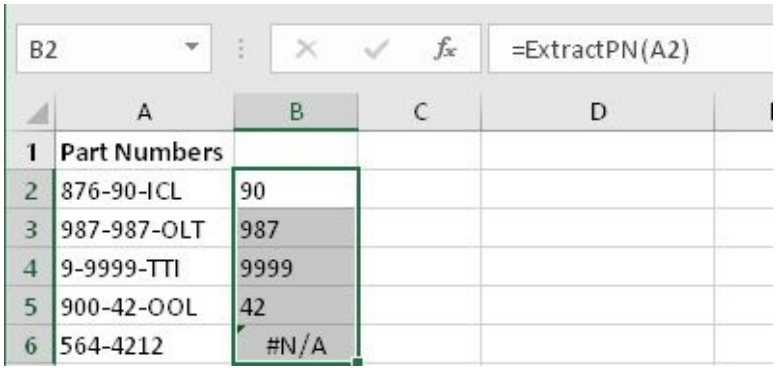
4. Type in the following code to count the items in the array and if there are not three, return an N/A error.
5. Add the last line where you set the function equal to the segment variable.

```
Function ExtractPN(PN) As Variant

    SegmentArray = Split(PN, "-")
    If UBound(SegmentArray) <> 2 Then
        Segment = CVErr(xlErrNA)
    Else
        Segment = SegmentArray(1)
    End If
    ExtractPN = Segment

End Function
```

Yes, I know I put <>2. You have to. Remember the array starts at 0 so the third segment is in position two. Aren't computers fun???

Test it out.

6. Save the workbook.
7. Enter the function ExtractPN function in the worksheet and see if it works for every part number. It should return an error for the last part number.



Sharing Functions

In a workbook: If you created functions in a workbook (like the follow-along workbook for this lesson), you can use the functions anywhere in the workbook. You need to save the workbook as xlsm and users will need to enable macros to use the functions.

If you want to use the functions so you can use them in several workbooks, you need to use a different option.

As an Add-in: An add-in is an Excel workbook that can be loaded and its macros used in any workbook. You need to save the workbook with the functions as xlsm. Users will need to go to File, Options, Add-ins and browse to the workbook to add it to their Excel.

Every time they open Excel, the add-in will load. They will need to enable macros and all the functions will be available for them to use.

Beware! Now you are getting to be a programmer! If you improve/update your functions, you need to create a new add-in file and send it out to your users. Now you have to keep

track of versions and will be helping users update their function files.

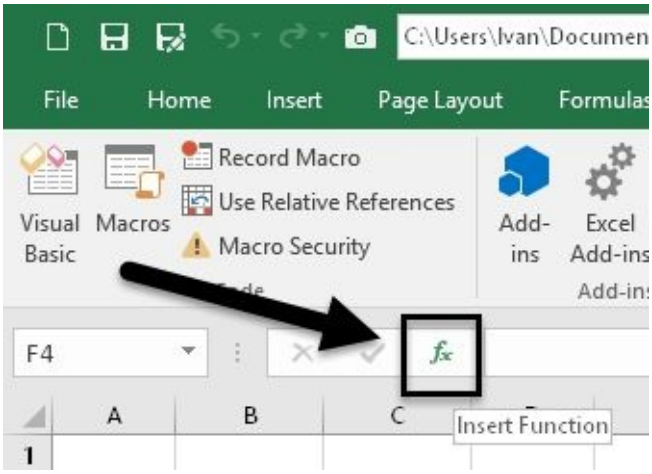
Welcome to the world of programming!! Muahahahaha (evil laugh).

Note: Outlook does not like xlsm files, it usually blocks them. You will need to zip them and then distribute them.

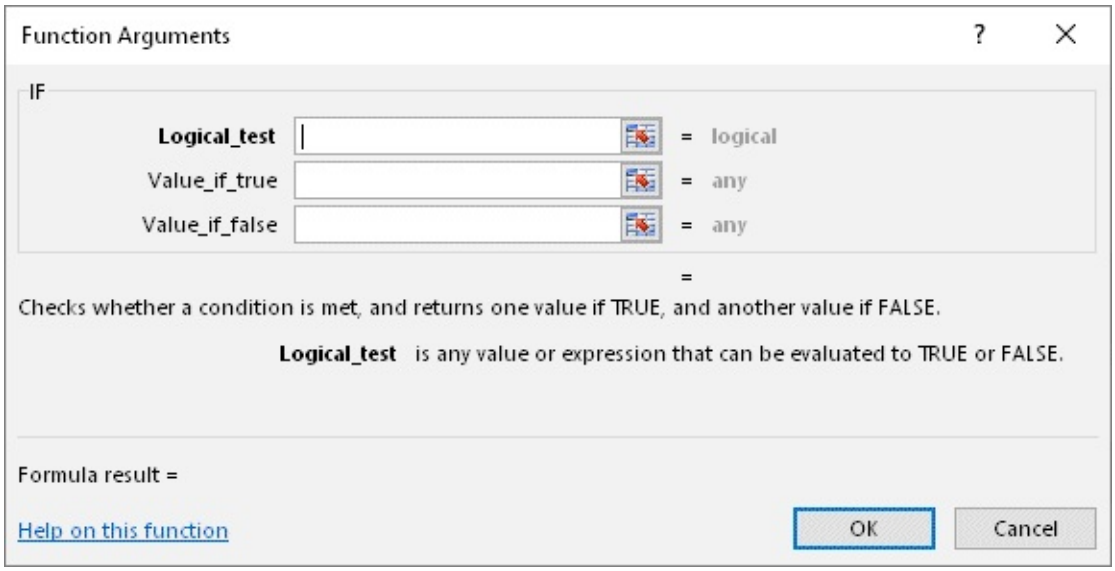
Adding Help

You can add explanations about your functions so that when users use the function wizard, they will see helpful hints.

If you haven't seen the function wizard, click this button to launch it:



This is how the help appears for the IF function.



You are going to add descriptions to the GetGrade function that will appear in the Insert Function window.

To add help, you need to create a subroutine (not a function), set the descriptions and run it once. After you run it the descriptions are stored in the workbook. You do not have to run the macro again unless you need to change/update the descriptions.

Insert a new **subroutine** called InsertDescription ()

```
Sub InsertDescription()

End Sub
```

Type in the following code into the macro

```
Sub InsertDescription()

Argument1 = "Score as a number from 1 to 100"
Argument2 = "[Optional] Input a Y to apply a curve"

Application.MacroOptions Macro:="GetGrade", _
Description:="Returns letter grade based on score.", _
Category:=14, _
ArgumentDescriptions:=Array(Argument1, Argument2)


End Sub
```

Macro: You use this to determine which macro to add descriptions to

Description: The description

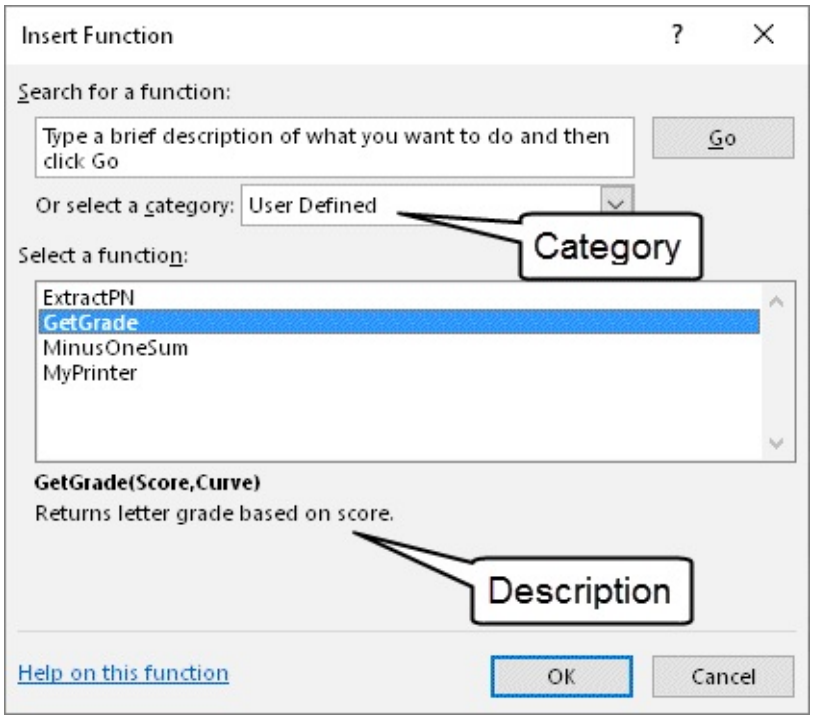
Category: You can choose which category the function will appear in. (I'll talk more about this below)

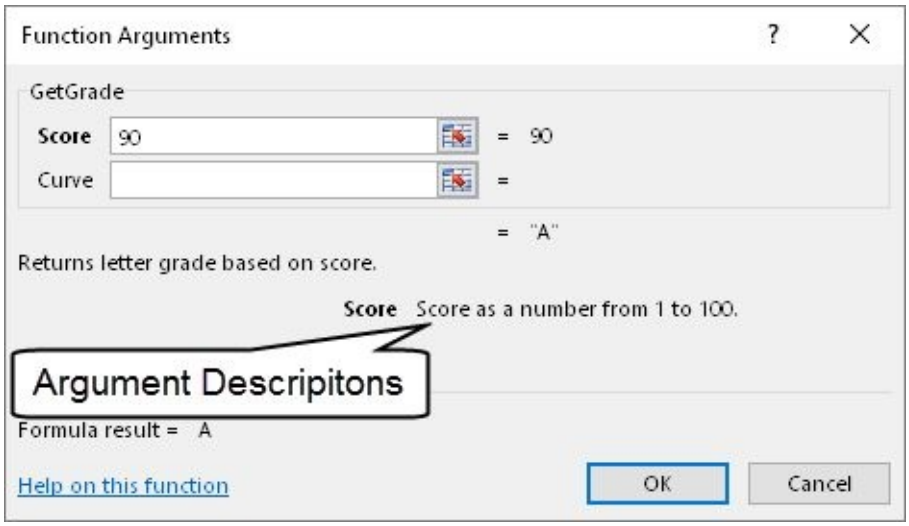
Argument Descriptions: The descriptions for each function argument.

Important: The last parameter, the ArgumentDescriptions, must be input as an array. Even if your function only has 1 argument, you must input it as an array.

And just to be clear: I used the named arguments (e.g. Macro:=) to make the parameters easier to read. I also used the [space] underscore to break up the command into several lines also to make it easier to read.

When you run this macro, this is what appears in the Insert Function window(s):





Pretty cool huh?

Ok, one more bit about categories. You can choose which category your function appears in. I choose 14, User Defined. Below is a list of categories and their respective numerical values.

Integer	Category
1	**Financial**
2	**Date & Time**
3	**Math & Trig**
4	**Statistical**
5	**Lookup & Reference**
6	**Database**
7	**Text**
8	**Logical**
9	**Information**
10	**Commands**
11	**Customizing**
12	**Macro Control**
13	**DDE/External**
14	**User Defined**
15	First custom category
16	Second custom category
17	Third custom category
18	Fourth custom category
19	Fifth custom category
20	Sixth custom category
21	Seventh custom category
22	Eighth custom category

23	Ninth custom category
24	Tenth custom category
25	Eleventh custom category
26	Twelfth custom category
27	Thirteenth custom category
28	Fourteenth custom category
29	Fifteenth custom category
30	Sixteenth custom category
31	Seventeenth custom category
32	Eighteenth custom category

You can even create new function categories.

Use this command to assign the TestMacro function to the ‘My Category’ category.

```
Application.MacroOptions Macro:=“TestMacro”, Category:=“My Category”
```

Summary

Ok, that it, that's really all you need to know to start crafting your own custom functions in Excel. Now you can take routine, custom calculations and encapsulate them in a function that you (and other less experienced Excel users) can use in your workbooks.

Other Lessons

I have many other lessons covering various Excel topics.

You can find all of them on my website at: http://markmoorebooks.com/excel-lessons/

If this lesson has helped you, please take a few minutes and leave a review on Amazon. The more reviews the lesson gets, the easier other students will be able to find it.

Thank you!