The purpose of a Custom Function is to save you time. Just as Excel's built-in functions (sum, average, etc.) save you time by allowing you to plug in values while they do the math behind the scenes, the same is true with custom functions. When you find yourself typing the same long formula over and over again and Excel doesn't provide a built-in function, you might want to consider creating a custom function. Typically, users create custom functions when the task at hand is unique to their industry or company. For example, tiered sales commissions based on hire date, sales, or employee; grades based on a class curve; the amount of concrete needed for a block wall of a specific size, etc.

- Introduction to the VBA Editor
- Introduction Custom Functions
- Using Variables
- Commenting Your Code
- If Then Else Statements
- Select Case Statements
- For Each Next Loops
- Using VBA Functions
- Using Worksheet Functions in VBA
- Accessing Specific Cells (Ranges)
- Accessing Function in Other Workbooks

For a complete list of topics, see the Table of Contents below.

Table of Contents

About Visual Basic for Applications (VBA) ................................................................. 3
Getting Started: Downloading the Workshop Files ....................................................... 3
  Getting Started Exercise A: Download the Excel File ............................................... 3
  Getting Started Exercise B: Macro Enabling the Excel File ....................................... 3
Accessing the VBA Editor ............................................................................................... 4
  Editor Navigation ........................................................................................................... 4
  Getting Started Exercise C: Inserting a Module ......................................................... 4
Custom Functions vs. Sub Procedures ............................................................................ 5
  Structure of a Custom Function .................................................................................. 5
Exercise 1: Simple Custom Function – Display the Area of a Rectangle ......................... 6
Debugging Your Code ..................................................................................................... 7
Commenting Your Code: ' ............................................................................................ 8
  Exercise 1 (Part 2): Adding Comments ................................................................. 8
  Exercise 2: Ratio Formula ......................................................................................... 9
Getting Assistance with Typing Custom Functions ....................................................... 10
  Solutions Available for Providing Assistance with Custom Functions ...................... 10
Introduction to Variables .............................................................................................. 11
  Exercise 3: Ratio2() Function using Variable Names ............................................ 11
More on Variables......................................................................................................... 12
  Exercise 4: Number of Blocks Needed in a Concrete Block Wall ............................ 13
Declaring Variables and Using Data Types: DIM ......................................................... 14
  Exercise 5: Declaring Variables in our Blocks() Function ...................................... 14
  How Long Do Variables Last? (Scope) ...................................................................... 15
Another Example of Declaring Variables ..................................................................... 16
Range Object Properties – Display Information about a Cell ....................................... 17
  Exercise 7: Display the Formula Used in a Cell in another Cell ............................... 17
Creating Conditional Code Execution with "IF THEN” STATEMENTS ......................... 18
  VBA IF Statement Conditions Available ............................................................... 18
Practice Exercises: IF Structures ................................................................................. 19
MAKING VBA CASE INSSENSITIVE ......................................................................... 21
  IF Exercise 1: Specifying a Grade based on a Numeric Score ................................ 23
Understanding How Excel Stores Dates and Time ..................................................... 25
  Dates ......................................................................................................................... 25
Time & Time Durations ............................................................................................... 26
IF Structure Exercise 2: Determine Time Difference ................................................................. 28
Built-in VBA Functions for Excel .............................................................................................. 30
VBA Functions Exercise 1: Return the Number of Days in a Month ........................................ 31
  DateSerial(year,month,day) ................................................................................................. 31
  Year(Serial Date) .................................................................................................................. 31
  Month(Serial Date) ............................................................................................................... 31
  Day(Serial Date) ................................................................................................................... 31
Creating the Function to Return the Date of the Last Day of the Month ................................. 32
VBA Functions Exercise 2: Remove the Leading “The" from Movie Names .............................. 33
Select Case (An Alternative to “If Then”) ................................................................................ 35
  Select Case Exercise 1: Or Condition with a Text String ...................................................... 36
  Select Case Exercise 2: Using "Case Else" ......................................................................... 37
  Select Case Exercise 3: Number Ranges (Tiered Commission Based on Sales) .................. 38
  Select Case Exercise 4: Using Constants (Commission Based on Employee ID) ............... 39
  Select Case Exercise 5: Handling Dates (Pension Rate Based on Hire Date) ..................... 40

USING WORKSHEET FUNCTIONS IN VBA ................................................................. 41

Using Worksheet Functions in VBA Exercise 1: Weighted Average Price .............................. 42
Worksheet Function Exercise 2: Employee Name with the Highest Sales .............................. 43
Process Each Item in a Range: For Each…Next Loop ............................................................... 45
  For Each…Next Loop Exercise 1: Summations .................................................................. 45
  For Each…Next Loop Exercise 2: Number of Items Above Average ................................. 46
Percent of Movie Profits Above Average .................................................................................. 46
Prompting the User for Values with Input Boxes ..................................................................... 47
Informing the User with a Message Box .................................................................................. 48
Working with a Specific Cell Address ...................................................................................... 49
  Range() Function Exercise 1: Determine the Interest for the Period ................................. 49
Updating Non-Volatile Custom Functions: CONTROL + ALT + F9 ........................................... 50
Range Exercise 2: Working with a Table of Data ..................................................................... 51
Creating the Custom Function ................................................................................................. 52
Using Macros and Custom Functions in Other Files ............................................................... 54
  Specifying the Workbook Name Method (Functions Only) .................................................. 54
  Exporting / Importing a Module Method .......................................................................... 54
  Exercise: Saving Your Custom Functions & Macros as an Add-in ...................................... 55
Take Home Exercise 1: Tiered Tax Structure using Select Case & If Then ............................... 56
Take Home Exercise 2: Select Case & If (Vacation Rate Based on Hire Date) ....................... 62
VaBal() Function with Variables Declared .............................................................................. 65
About Visual Basic for Applications (VBA)

As mentioned above, custom functions are written in the VBA editor introduced in 1993 as a means of customizing and automating MS Office applications. It is accessible through Excel and most other MS Office applications for Windows but Mac availability has been sporadic.

- **Windows Computers** - VBA is available in all versions of Excel for Windows.
- **Mac Computers** - VBA was removed from Excel 2008 for the Mac but was restored in Excel 2011 for the Mac.

**Learning VBA**

Note that VBA is a programming language and like learning any language, can take months or years to learn; however, you can create many custom functions relatively quickly without having to delve too deeply into VBA. Note that VBA is not the only or the most powerful language you can use to customize MS Office applications but is still the easiest, most relevant, and widely used.

**The Scope of Custom Functions**

Custom functions are created in the Visual Basic for Applications editor (VBA) which comes with most versions of Excel and is accessed through Excel. The functions you create in the editor are saved with the Excel file you created them in. The functions can be used on any sheet within the workbook they are associated with and can be used in other workbooks too if you open the file containing the custom function and precede the name of the function with the name of the workbook or save your Excel file as an Add-in. More on both of these topics is covered later.

---

### Getting Started: Downloading the Workshop Files

Some of the examples within this guide utilize Excel worksheets that contain data.

**Getting Started Exercise A: Download the Excel File**

To download the Excel example file:

1. Go to: [http://info.marshall.usc.edu/dept/training/Pages/Applications-Guides.aspx#Excel](http://info.marshall.usc.edu/dept/training/Pages/Applications-Guides.aspx#Excel)
2. Download and open the "Custom Functions.xlsx" file.

**Getting Started Exercise B: Macro Enabling the Excel File**

If you create custom functions, macros, or sub procedures, to save them in Excel you must save your Excel file as a *Macro Enabled Workbook* (XLSM). You do not lose any Excel functionality when saving as this type.

1. Open the **Custom_Functions.xlsx** file you downloaded earlier.
2. Click "File – Save As".
3. At *Save as Type* select: "Excel Macro-Enabled Workbook (*.xlsm)"
4. Give it a name and click "Save".
Accessing the VBA Editor

While the code for your custom functions, sub procedures and macros are stored in your macro enabled workbook, the code is viewed, created, and edited from within the VBA editor. To access the VBA editor:

1. While in Excel press:  **ALT + F11** (Toggles between Excel and the Editor.)
   (You can also access the VBA Editor "View – Macros" and the "Developer" tab.)

Editor Navigation

The illustration below shows the components beginners might use within the VBA Editor. Components can be closed by clicking their X's and reopened either under "View" or pressing their shortcut keys. For this workshop, all the functions we create will be stored within modules.

Getting Started Exercise C: Inserting a Module

Code is stored within modules. You can store all of your custom functions in a single module or use multiple modules. To insert a module:

1. While in the VBA editor, click "Insert – Module".
**Custom Functions vs. Sub Procedures**

While in a VBA module, you might see two types of code: *Custom Functions* and *Sub Procedures*. Both automate your tasks but have two different purposes. This guide addresses Custom Functions only.

---

### Structure of a Custom Function

The basic structure of a custom function is shown below.

- **Function** `FunctionName`  
  Lines of Code  
  **End Function**

Custom functions are always enclosed in “Function” and “End Function”. They are typically used to take input from the user and return an answer within a cell. Unlike sub procedures, they cannot be used to manipulate the environment. For example, they can't format cells, insert columns/row, execute menu commands, etc. Their purpose is to return a solution to a cell or pop-up box.

- **Sub** `ProcedureName`  
  Lines of Code  
  **End Sub**

Sub procedures are always enclosed in "Sub" and "End Sub". They can be used to automate just about anything the user can do manually and then some - just much more quickly. For example, format cells, type formulas, type text, insert columns/rows, print, etc. Note that you can execute a custom function from a sub procedure.

---

### Structure of a Custom Function

The basic structure of a custom function is shown below.

- **Function** `FunctionName( argument1, argument2, etc. )`  
  Lines of Code  
  **End Function**

Custom functions must begin with "Function" and end with "End Function".

- **FunctionName** = expression to return

You must give your function a name followed by ( ) which may contain multiple arguments that typically represent cell addresses, variables, or constants.

Any code used to calculate your function goes here.

The name of your function acts as a variable and is what the function ultimately returns.
**Exercise 1: Simple Custom Function – Display the Area of a Rectangle**

Just to get the hang of the structure, this first custom function will be very simple. This function will give you the area of a rectangle. The user supplies the length and width.

1. Access the VBA Editor (Alt + F11).
2. If you have not done so already, insert a module ("Insert – Module" from within the VBA Editor).
3. Within the module you opened, type the code shown below.

```vba
Function Area(L,W)
    Area = L * W
End Function
```

Note that you may have several macro enabled files open. Be sure to use the module on the branch for the Custom_Functions.xlsm file.

**Testing the Function in a Worksheet**

You should be able to test your function within any sheet of the file.

1. Press ALT + F11 to access Excel.
2. In a blank sheet (Sheet1 for example), create the spreadsheet shown.

Note that this would also work: =Area(20,50)

You can also test some functions within the Immediate Window (Control + G) by tying a question mark in front of them. For example,

1. In the Immediate Window, type: ?Area(10,30)
2. Press Enter.
Debugging Your Code

When learning VBA, chances are you won't get your custom functions right the first time you use them. In general, when you type your custom function one of four things will happen:

- It will work (congratulations, you are gifted).
- It will return the wrong answer or a 0.
- It will return an error message in the cell (i.e. #Value!, #Name!, etc.)
- It will take you back to the editor and place you in debugger mode.

Returned 0 or the Wrong Answer

There are numerous reasons for this but the most common are:

- You misspelled a variable name so VBA created it for you on the fly with a value of zero.
- Your mathematical equation does not take everything into account.
- Your conditional logic is incorrect.

Returned an Error Message (i.e. #Value!, #Name!)

#Value! could mean that you supplied letters when you should have specified numbers or one of your variable types is wrong. #Name! typically means you misspelled the name of a function or your code specifies a range name that doesn't exist. There are other error messages too of course. To be honest, I typically Google my error messages and get pretty good advice.

Debugger Mode (Control + Shift + F8 to Step Out)

With some errors, Excel will take you back to the VBA editor and launch debugger mode. As you can see to the right, the debugger automatically highlights the name of the functions with issues in yellow as well as the problem. (In this case, I spelled the name of the Ucase() function wrong.) When it takes you into debugger mode, you will have to do two things to get up and running again:

a. Either fix the problematic code or comment it out (see next page.).
b. Press CONTROL + SHIFT + F8 to step out of debugger mode (or "Debug – Step Out" from the menu.)

Debugger Exercise

1. Misspell the words "End Function" in the Area() function.
2. Go to a sheet and try to use the function.
3. Fix the spelling and then step out of debugger mode (Control + Shift + F8)

Updating your Function in Excel

Note that after fixing your code in the editor, when you go back to Excel your function will not update its answer until you do one of the following:

- Press F9 to recalculate the spreadsheet (works when the function refers to cells in the sheet).
- Take the function into edit mode (F2) and then press Enter.
- Press CONTROL + ALT + F9 from any cell (always works).

Updating Exercise

1. Edit the Area() function above to: Area = L + W
2. Return to Excel and note that the answer did not update.
3. Use one of the methods above to update your spreadsheet.
4. Change the Area() function back to: Area = L * W
Commenting Your Code: ' 

Comments are an excellent means of describing to yourself or someone else what your code is doing. What you did may have made sense at the time but six months later when you need to edit it, comments can help you remember. Note that in the VBA editor, text recognized as a comment appears green.

To enter a comment, type an apostrophe at the beginning of the comment. In the example below, the first line of text is a comment and we have also added a comment to the right of some executable code.

**Exercise 1 (Part 2): Adding Comments**

- This function returns the area of a rectangle.
  ```vba
  Function Area(L,W)
  Area = L * W ' L is the length and W is the width.
  End Function
  ```

- Note that anything to the left of the apostrophe on the same line is considered a comment. The code below would not have worked because the definition of Area is now a comment:
  ```vba
  ' (L is the Width and W is the Length) Area = L * W
  ```

- Note that apostrophe's within quotes are not treated as comments:
  ```vba
  Msg = "You haven't entered a number"
  ```

**Wrapping Comments**
The VBA Editor will not wrap long lines of text. If you have a long comment to type and wish to wrap it you have two choices:

- Press ENTER after typing short lines and then begin the next line(s) with an apostrophe.
  ```vba
  ' This code is designed to take
  ' a given Income and filing status
  ' and return the Taxes.
  Function Taxes13(N,F)
  ```

- Type a space then an underscore at the end of each line then press ENTER to continue it to the next line.
  ```vba
  ' This code is designed to take _
  ' a given Income and filing status _
  ' and return the Taxes.
  Function Taxes13(N,F)
  ```

**Using Comments to Debug Code**
Comments are also useful when debugging your code. If you place an apostrophe to the left of the line in question, it will no longer be read by Excel. Once you have tested or fixed it, simply remove the apostrophe.

*Note that if you display the "EDIT" toolbar, there is a button that will let you comment out multiple lines of code with a single click!*
**Exercise 2: Ratio Formula**  
Take a look at the "Adj_Scores" sheet. This function is designed to adjust the scale of a test score. For example, a student got 19 out of 20 questions right and the instructor would like to adjust the score to fit a 10 point scale. Mathematically, this would be:

\[
\text{Points Earned / Points Possible} \times \text{Scaled Points Possible} \quad \text{or} \quad \frac{19}{20} \times 10 = 9.5
\]

1. Press Alt + F11 to return to the VBA editor.  
2. Below your previous function, type the custom function shown below.

```
' This function returns the points earned based on an adjusted scale.
' E is the Actual Points earned.
' P is the actual points possible.
' N is the points possible on the adjusted scale.

Function Ratio(E, P, N)
    Ratio = E / P * N
End Function
```

**Testing the Function**  
1. Press ALT + F11 to return to Excel  
2. Go to the "Adj_Scores" worksheet.  
3. In cell C2 type: =Ratio(B2,20,10) and press enter.  
4. Copy the formula down.

![Ratio Table](Image)

Note you can also test this in the **Immediate Window**.

```
?Ratio(7.3,20,10)
3.65
```
Getting Assistance with Typing Custom Functions

No Smart Tags for Custom Functions
You may have noticed that when you were typing your custom function no Smart Tags appeared to assist you in understanding the function's arguments as Excel does with its built-in functions. Unfortunately, VBA does not provide a method of creating Smart Tags.

Solutions Available for Providing Assistance with Custom Functions
You have a few methods of providing users assistance in typing custom functions.

Pressing CONTROL + SHIFT + A
If you type the name of any function and then press CONTROL + SHIFT + A Excel will list the function's arguments. (This works with built-in functions too.)

1. In a blank cell type: =Ratio
2. Press CONTROL + SHIFT + A
Excel should display the function and its arguments.

Accessing the Insert Function Dialogue Box (SHIFT + F3)
Another method is to pull up the Functions Arguments dialogue box after typing the name of the function followed by the opening (Again, this works for built-in functions too.

1. In a blank cell type: =Ratio(
2. Press SHIFT + F3
(Or click the Insert Function button ¿)

You can then type the cell addresses in the different boxes provided or click the collapse buttons and click the cells to specify them. When done, click "OK".

Use More Descriptive Function Argument Names
Using descriptive argument names also helps. Below is the Ratio() function but with more descriptive argument names. Note that we have used underscores rather than spaces. Argument names can't have spaces.

Function Ratio(Pts_Earned, Points_Possible, New_Points_Possible)
Ratio = Pts_Earned / Points_Possible * New_Points_Possible
End Function
Introduction to Variables

On the bottom of the previous page, I recommended using more descriptive argument names for your custom functions as a means of providing usage assistance to your end users. However, the downside of this is that you have to type longer names when using those arguments in the code that runs your function. For more complex custom functions where arguments are referenced frequently, this can become a pain and lead to typos which lead to errors. (Sometimes when you misspell an argument's name, VBA assumes you are creating a new variable and assigns a value of 0 to it.)

You can rename an argument by assigning a "Variable" to it as shown in the example below.

Exercise 3: Ratio2() Function using Variable Names
In this exercise, we are using longer names as our function's arguments and then redefine them below with shorter names. This will allow us to provide more descriptive help for our end users without having to type the lengthy name later in the function.

1. Double click a module from the Project Explorer.
2. Click within the "Code Window" and type the custom function shown below.
   Note that it may go within the same module as our previous function.

   ```vba
   Function Ratio2(Points_Earned, Old_Points_Possible, New_Points_Possible)
       E = Points_Earned
       P = Old_Points_Possible
       N = New_Points_Possible
       Ratio2 = E / P * N
   End Function
   ```

Testing the Ratio2() Function
Test the new function in the same manner we tested Ratio().
Note that the Insert Function box now has more descriptive names.

1. Press "Alt + F11" to return to Excel.
2. In a blank cell type: =Ratio(Points_Earned, Old_Points_Possible, New_Points_Possible)
3. Press SHIFT + F3

<table>
<thead>
<tr>
<th>Function Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ratio2</td>
</tr>
<tr>
<td>Points_Earned</td>
</tr>
<tr>
<td>Old_Points_Possible</td>
</tr>
<tr>
<td>New_Points_Possible</td>
</tr>
</tbody>
</table>

"Points_Earned becomes "E", "Old_Points_Possible" becomes "P", "New_Points_Possible" becomes "N". We can now use E, P, and N throughout the rest of our function."
More on Variables

A variable is a sort of placeholder that can stand in for a mathematical expression whose value can change or "vary" as it is processed. Although not required, variables are essential for writing code that is short, flexible, and easy to read. In the example above, we created three variables: E, P, and S to represent user input.

**Naming Variables**

When you name a variable, there are a few rules:
- Variable names must start with a letter. (A-Z)
- Variable names can contain letters, numbers, and underscores (A-Z, 0-0, _).
- Variable names cannot contain spaces or other special characters. (Use an underscore if you want a space.)
- Variable names are not case sensitive although the VBA Editor will help you keep the same case of a specific variable throughout your use of it in your code.
- VBA has reserve words (e.g. next, for, then) that will cause errors if you use them as variable names.

**Using Variables is Usually Shorter because they can be used Multiple Times**

Variables also save you the trouble of typing the same equation multiple times. The example below computes the value of X. Note that the same expression is used throughout the equation.

\[
Y = ((100 - 50) / (25 - 5))^2 + ((100 - 50) / (25 - 5))^{1/3} - 3 \times ((100 - 50) / (25 - 5))
\]

We can shorten it using the variable X to represent the expression \((100 - 50)/(25-5)\) as shown:

\[
X = (100 - 50) / (25 - 5)
Y = X^2 + X^{1/3} - 3X
\]

Again, we don't have to use variables but as you can see below, the same equation without variables is longer. We can now use X over and over again anywhere in the function it was created within.

**The Value of a Variable can Change (hence the "vary" in "variable")**

The custom function example below is fairly useless but it does illustrate how a variable (X) can change as it is processed within a loop.

```vba
Function Counting()
    X = 0
    For L = 1 To 5 Step 1
        X = X + 3
    Next L
    Counting = X
End Function
```

The **For Next** loop specifies that the loop should occur five times. \(L\) starts at 1 and ends when \(L = 5\). **Step** tells the counter what to count by. In the first loop, the variable \(X\) is 0 and then 3 is added to make \(X = 3\). In the second loop, \(X\) becomes 6, the third \(X\) becomes 9, in the fourth \(X\) becomes 12, and in the fifth and final loop \(X\) becomes 15. **Counting** = \(X\) is outside of the loop and isn't executed until the loop is finished. It returns \(X\)'s final value of 15.

Note that the equal sign is not taken to indicate equality; it is used to assign variables. Obviously, \(X\) is not equal to \(X + 3\) Note also that if you were multiplying \(X\) by 3 you will need to give \(X\) an initial value of something other than 0 because \(0 \times 3\) is equal to 0.
Variables Usually Makes Code Easier to Decipher
As your functions get more complicated, breaking them down into different parts by using variables usually makes them easier to read or decipher if you come back six months from now and need to edit them.

Exercise 4: Number of Blocks Needed in a Concrete Block Wall
This function calculates the number of blocks needed in a block wall given the wall's length and height (in feet).

' This function calculates the number of blocks needed in a wall given the length and height.  
' Assumes a Standard Block is 15-1/2" Long x 7-1/2" High.

Function Blocks(FootLength, FootHeight)

\[
L = \text{FootLength} \times 0.75 \\
H = \text{FootHeight} \times 1.5 \\
\text{Blocks} = L \times H
\]

End Function

Testing the Function
1. Go to the "BlockWall" sheet in Excel.
2. In cell D3, type: =Blocks(B3,C3)

Alternate Version without Variables
Again, this one is still pretty easy to understand if you wrote it as shown below without using variables but the point is, eventually your functions will be more complicated than this and it will be easier to follow if you use variables to solve it in units. Incidentally, the parentheses around our calculations are not necessary but they don't hurt anything and they do make it easier to read.

Function Blocks(FootLength, FootHeight)

\[
\text{Blocks} = (\text{FootLength} \times 0.75) \times (\text{FootHeight} \times 1.5)
\]

End Function
Declaring Variables and Using Data Types: DIM

Up until now we have assigned variables without specifying their data type as so:
- \( X = \frac{(100 - 50)}{(25 - 5)} \)
- Function Ratio(Points_Earned, Old_Points_Possible, New_Points_Possible)

When you assign variables in this manner, Excel has to decide how much space to set aside in its memory to store the variable. It does this by setting the variable's data type to "variant". While convenient to the coder, this can dramatically slow down how fast your code processes. The recommended method is to specify the data type yourself using the DIM statement or AS. When specifying a data type, you should select the type that allows you to get the job done but uses the least amount of memory.

Note however that declaring the wrong variable type can cause a function to work incorrectly (or not at all). If you are new to VBA, you may wish to initially write your function without declaring its variables, get the function working, and then declare its variables. (On the other hand, some functions won't work until you declare the variable's data type.) The table below lists the different data types available.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Bytes Used</th>
<th>Default Value</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>2 bytes</td>
<td>FALSE</td>
<td>True or False</td>
</tr>
<tr>
<td>Byte</td>
<td>1 byte</td>
<td>0</td>
<td>0 to 255</td>
</tr>
<tr>
<td>Currency</td>
<td>8 bytes</td>
<td>0</td>
<td>-922,337,203,477.5808 to 922,337,203,685,477.5807</td>
</tr>
<tr>
<td>Date</td>
<td>8 bytes</td>
<td>0</td>
<td>January 1, 100 to December 31, 9999</td>
</tr>
<tr>
<td>Decimal</td>
<td>14 bytes</td>
<td>0</td>
<td>-79,228,162,514,264,337,593,543,950,335 to 79,228,162,514,264,337,593,543,950,335 or -7.2998162514264337593543950335 to 7.9228162514264337593543950335</td>
</tr>
<tr>
<td>Double</td>
<td>8 bytes</td>
<td>0</td>
<td>-1.79769313486232E308 to -4.94065645841247E-324 or 1.79769313486232E308 to 4.94065645841247E-324</td>
</tr>
<tr>
<td>Fixed String</td>
<td>String's length</td>
<td>String Length</td>
<td>1 to 65,400 characters</td>
</tr>
<tr>
<td>Integer</td>
<td>2 bytes</td>
<td>0</td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td>Long Integer</td>
<td>4 bytes</td>
<td>0</td>
<td>-2,147,483,648 to 2,147,486,647</td>
</tr>
<tr>
<td>Object</td>
<td>4 bytes</td>
<td>Nothing</td>
<td>Any Access object, ActiveX component or Class object</td>
</tr>
<tr>
<td>Single</td>
<td>4 bytes</td>
<td>0</td>
<td>-3402823E38 to -1.401298E-45 or 1.401298E-45 to 3.402823E38</td>
</tr>
<tr>
<td>Variable String</td>
<td>10 bytes plus the number of characters</td>
<td>Zero-length</td>
<td>0 to 2 billion characters</td>
</tr>
<tr>
<td>Variant</td>
<td>16 bytes</td>
<td>Empty</td>
<td>Same as Double</td>
</tr>
</tbody>
</table>

**Exercise 5: Declaring Variables in our Blocks() Function**

To declare variables in our Blocks() function, add the items shown below.

```
Function Blocks(FootLength As Single, FootHeight As Single)
    Dim NumberBlocksLong As Single
    Dim NumberBlocksHigh As Single
    NumberBlocksLong = FootLength * 0.75
    NumberBlocksHigh = FootHeight * 1.5
    Blocks = NumberBlocksLong * NumberBlocksHigh
End Function
```

**Note on Rounding:**
Should you use a data type that does not allow numbers to the right of the decimal but the number you are referring to does contain a fraction, VBA will round, not truncate.
How Long Do Variables Last? (Scope)
Can you use the same variable more than once in the same module? Are variables remembered from one function to the next? The answer is it depends on how you declare them.

Function / Sub Procedure Only Variables
When you declare a variable within a function or sub procedure, as soon as the function or sub procedure is done, the value of the variable evaporates. This means that variables names defined within a function can be used over and over again in different functions and procedures and one will have no effect on the other.

Function BillingDate()

    Dim NDays As Byte
    NDays = 30
    BillingDate = Now() + NDays

End Function

Function EscrowDate()

    EscrowDate = Now() + NDays

End Function

The function to the left creates a variable called "NDays" within the function and sets NDays equal to 30. It then adds NDays to the current date to determine what the date would be 30 days from today. (Adding 1 to a date returns the next day.)

Because NDays was defined within the previous function, its value is not used by this function. VBA creates a variable called NDays on the fly and sets it to 0.

Module Wide Function / Sub Procedure Variables
If you want the variable NDays to be available to all functions within the module, you must declare NDays at the top of the module before the first function or procedure as shown here:

Dim NDays As Byte

Function BillingDate()

    NDays = 30
    BillingDate = Now() + NDays

End Function

Function EscrowDate()

    EscrowDate = Now() + NDays

End Function

Because we have declared our variable above all of the other functions in our module, its current value is available to other functions in the module.

Note that we declare it at the top but we cannot set its value at the top (i.e. NDays = 30 doesn't work up there).

Also note that NDays only takes on a value if the top function is run before the bottom function. If the bottom function is run first then NDays will be zero.

Make Variables Available to all Modules
To make a variable available to all modules within a workbook, you must declare it at the top of one of your modules using the "Public" keyword. For example:

Public Ndays as Byte
Another Example of Declaring Variables

**Practice Exercise: Find Miles Per Gallon Function** *(End Miles – Begin Miles) / Gallons*

This function returns *Miles per Gallon* by subtracting the miles of the current fill-up from the miles of the previous fill-up and then dividing by the number of gallons purchased in the current fill-up.

1. Open the VBA Editor (ALT + F11).
2. Below our previous function, create the function shown below.

```vba
Function MPG(BeginMiles As Integer, EndMiles As Integer, Gallons As Single)
    MPG = (EndMiles - BeginMiles) / Gallons
End Function
```

**Testing the Function**

1. Press ALT + F11 to return to Excel
2. Go to the "MPG" worksheet.
3. In cell D3 type: =MPG(B2,B3,C3) and press enter.
4. Copy the formula down.

```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Date</td>
<td>Mileage</td>
<td>Gallons to Fill Tank</td>
<td>Mile Per Gallon</td>
</tr>
<tr>
<td>2</td>
<td>1/12/2013</td>
<td>15,230</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1/22/2013</td>
<td>15,800</td>
<td>12</td>
<td>=MPG(B2,B3,C3)</td>
</tr>
<tr>
<td>4</td>
<td>1/24/2013</td>
<td>16,400</td>
<td>11.5</td>
<td>52.2</td>
</tr>
<tr>
<td>5</td>
<td>2/2/2013</td>
<td>16,600</td>
<td>4</td>
<td>50.0</td>
</tr>
<tr>
<td>6</td>
<td>2/15/2013</td>
<td>17,000</td>
<td>8.5</td>
<td>47.1</td>
</tr>
<tr>
<td>7</td>
<td>2/23/2013</td>
<td>17,450</td>
<td>10</td>
<td>45.0</td>
</tr>
<tr>
<td>8</td>
<td>3/4/2013</td>
<td>17,900</td>
<td>9.8</td>
<td>45.9</td>
</tr>
</tbody>
</table>
```

To test it in the Immediate Window:

```
?MPG(15230,15800,12)  // 47.5
```
Range Object Properties – Display Information about a Cell

VBA is an object orientated programming language. Cells are referred to as "Range Objects" and they have properties that you can use VBA to return. In the example below, we will return the formula used in a specific cell. Some other Range Object Properties you can return include:

- **.Formula** Returns the formula used in the specified range.
- **.Count** Returns the number of cells in a range (empty or not).
- **.Value** Returns the value of the first cell in a range.
- **.ColumnWidth** Returns the column width of the cell referred to.
- **.RowHeight** Returns the row height of the cell referred to.
- **.Height** Returns the height of the row.
- **.Row** Returns the row number that a cell address refers to.
- **.Column** Returns the column number that a cell address refers to.
- **.Worksheet.Name** Returns the name of the sheet the cell address referred to is on.

**A Peek at the VBA Object Model**

VBA can take months or years to learn but if you would like to see where the properties above are coming from, you might want to try the following:

- You can also click the ? in VBA and do a search on "Microsoft Excel Object Model" or "Excel Object Model Reference".

**Exercise 7: Display the Formula Used in a Cell in another Cell**

Because I write documentation on using Excel, I found this custom function very useful. It allows you to type a function that will display the formula used in a cell. (Apparently Microsoft thought it was useful too because starting with Excel 2013, they included a function called FormulaText(celladdress) that does the same thing.)

1. Within the module you opened on the previous page, type the code shown below.

```vba
Function FormulaIs(X)
    FormulaIs = X.Formula
End Function
```

**Testing the Function**

1. Press ALT + F11 to access Excel.
2. In a blank sheet, type a 1 in A1 and a 2 in A2.
4. In B3, type: =Formulas(A3) and press enter.

Excel should display the formula used in A3 (i.e. =Sum(A1:A2)
Creating Conditional Code Execution with "IF THEN" STATEMENTS

If you have used Excel's IF() function then you are familiar executing code only when a specific condition is true. If the condition evaluates to true, the code is executed; otherwise, it is not. For example, if cell A1 has a number in it, execute the following code; otherwise, do not execute the code. VBA also has its own version of an IF Statement and can take several forms as shown in the examples below.

**If Condition Then**
- Lines of code to execute

**Else (optional)**
- Lines of code to execute

**ElseIf Condition Then** (optional)
- Lines of code to execute

**Else (optional)**
- Lines of code to execute

**End If (sometimes optional)**

Most of the structural options available in a VBA If statement are shown to the right. Note that you can only combine these in certain ways:
- An IF Then can have only one Else but multiple Elsifs.
- An ElseIf can contain only one Else.
- An Else cannot contain an ElseIf or a Condition but can contain another IF.

The examples below illustrate different structural possibilities.

**VBA IF Statement Conditions Available**
The following Boolean operators are available within IF statements.

<table>
<thead>
<tr>
<th>Task</th>
<th>Operator(s)</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Boolean Operators</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Equal to</td>
<td>=</td>
<td>If X = 10 Then</td>
</tr>
<tr>
<td>Not Equal to</td>
<td>&lt;&gt;</td>
<td>If X &lt;&gt; 10 Then</td>
</tr>
<tr>
<td>Greater Than</td>
<td>&gt;</td>
<td>If X &gt; 10 Then</td>
</tr>
<tr>
<td>Less Than</td>
<td>&lt;</td>
<td>If X &lt; 10 Then</td>
</tr>
<tr>
<td>Greater Than or Equal to</td>
<td>&gt;=</td>
<td>If X &gt;= 10 Then</td>
</tr>
<tr>
<td>Less Than or Equal to</td>
<td>&lt;=</td>
<td>If X &lt;= 10 Then</td>
</tr>
<tr>
<td><strong>AND Conditions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Between Two Points</td>
<td>And</td>
<td>If X &gt;= 10 and X &lt;=20 then</td>
</tr>
<tr>
<td>Multiple AND Conditions</td>
<td>And</td>
<td>If X = &quot;USC&quot; and Y = &quot;Marshall&quot; Then</td>
</tr>
<tr>
<td><strong>OR Conditions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiple OR Conditions</td>
<td>OR</td>
<td>If X = 1 or X =8 or Y =&quot;USC&quot; then</td>
</tr>
<tr>
<td>If X contains a 1, 2 or a 3</td>
<td>Like &quot;[numbers]&quot;</td>
<td>If X Like &quot;[123]&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This can only find the individual numbers 1, 2, or 3. You cannot use this technique to find numbers larger than 10 and if in cells, the 1,2, or 3 must be the only thing in the cell.</td>
</tr>
<tr>
<td><strong>XOR Conditions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>One and only one of two possibilities.</td>
<td>XOR</td>
<td>If a = 10 XOR b = 15 Then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Only returns true when a=10 or b=15 but not both. In other words, one must be true and one must be false for XOR to return true.</td>
</tr>
<tr>
<td><strong>Combine an &quot;AND&quot; Condition with an &quot;OR&quot; Condition</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X must either be between two points or equal to a specific value or string.</td>
<td>( around AND )</td>
<td>If (X &gt; 1 And X &lt; 5) Or X = &quot;USC&quot; Then</td>
</tr>
<tr>
<td>X can be many things AND Y must be equal to a specific string or value.</td>
<td>( around OR )</td>
<td>If (X = 0 Or X = 9 or X = 100) and Y = &quot;USC&quot;</td>
</tr>
</tbody>
</table>
### Wild Cards

<table>
<thead>
<tr>
<th>Description</th>
<th>LIKE Syntax</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Starts with USC and ends in anything.</td>
<td>* LIKE *</td>
<td>If X Like &quot;USC*&quot; then</td>
</tr>
<tr>
<td>Ends with USC and starts with anything.</td>
<td>* Like</td>
<td>If X Like &quot;*USC&quot; then</td>
</tr>
<tr>
<td>Contains the text USC anywhere in the string.</td>
<td>* Like *</td>
<td>If X Like &quot;<em>USC</em>&quot; then</td>
</tr>
<tr>
<td>Contains an A, B, or C anywhere within the string.</td>
<td>* Like <em>[ABC]</em></td>
<td>If X Like &quot;<em>[ABC]</em>&quot; then</td>
</tr>
<tr>
<td>Contains a 1, 2 or 3 anywhere within the string.</td>
<td>Like &quot;*[123]&quot;</td>
<td>Careful, 15, 26, and 38 would also be found because they also have a 1, 2 or 3 in them.</td>
</tr>
</tbody>
</table>

### NOT Conditions

<table>
<thead>
<tr>
<th>Description</th>
<th>Operator</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Does NOT contain &quot;Dog&quot;.</td>
<td>&lt;&gt;</td>
<td>IF X &lt;&gt; &quot;Dog&quot; then</td>
</tr>
<tr>
<td>With multiple NOT Conditions, be sure to use &quot;AND&quot; not &quot;or&quot;.</td>
<td>&lt;&gt; AND</td>
<td>IF X &lt;&gt; &quot;Dog&quot; AND X &lt;&gt; &quot;Cat&quot; AND X = &quot;Rat&quot; Then</td>
</tr>
</tbody>
</table>

### Checking Cell Content Type

<table>
<thead>
<tr>
<th>Description</th>
<th>Function</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checking for Numbers</td>
<td>IsNumeric(expression)</td>
<td>If IsNumeric(A1) then</td>
</tr>
<tr>
<td>Checking for Blank Cell</td>
<td>IsEmpty(expression)</td>
<td>If IsEmpty(A1) then</td>
</tr>
<tr>
<td>Check for a Formula</td>
<td>expression.HasFormula (This is a Property)</td>
<td>If X.HasFormula then</td>
</tr>
</tbody>
</table>

### Practice Exercises: IF Structures

In a new module, try the IF Structures shown below

**Function Animal(X As String)**

```vba
Function Animal(X As String)
    If X = "Dog" Then Animal = "Canine"
End Function
```

**IF on a Single Line – Don’t use End IF**

When you put the result on the same line as the IF Then then you don’t need End IF. This custom function checks to see if the contents of a cell contains "Dog". If it does, the function returns "Canine". If it does not, the function returns 0. **Note** "Dog" is case sensitive.

**Immediate**

```
Animal("Dog")
```

**Multiple Lines of Executable Code**

These lines of code all execute when X > 10. If X is 10 or less, the function returns a zero.

Interestingly enough, a cell containing text also results in the condition being true but if you change the condition to =10 or <10, the result if false for text. You can check for numbers using the IsNumber function.

```vba
Function Status(X)
    If X > 10 Then
        A = 100
        B = 200
        Status = A + B
    End If
End Function
```

**IF on Multiple Lines – Use End IF**

Here we need End If because we pressed Enter after "Then". **Note** "Dog" is case sensitive.

**Immediate**

```
Animal2("Dog")
```

```vba
Function Animal2(X As String)
    If X = "Dog" Then
        Animal2 = "Canine"
    End If
End Function
```

```vba
Function Status(X)
    If X > 10 Then
        A = 100
        B = 200
        Status = A + B
    End If
End Function
```
Animal3() is similar to the Animal() example above except we are using multiple IF statements. The custom function below checks the contents of a cell. If the cell contains "Cat", the function returns "Feline"; if it has "Dog", "Canine" is returned; and if it contains "Rat", "Rodent" is returned. If the cell contains none of these, 0 is returned.

```
Function Animal3(X As String)
  If X = "Cat" Then Animal3 = "Feline"
  If X = "Dog" Then Animal3 = "Canine"
  If X = "Rat" Then Animal3 = "Rodent"
End Function
```

**Multiple IF Statements**

In this structure, each IF statement is read but only the ones whose condition is true will return an answer. Though not possible with this type of condition, if multiple IF statements are true, then the result of the last true IF read is returned. Also, because the result is on the same line as the condition, End IF was not needed. Note this "Cat", "Dog", and "Rat" are case sensitive.

### IF Statement with Multiple Conditions: "AND"

```
Function Level(X)
  If IsNumeric(X) And X > 10 Then
    Level = "VIP"
  End If
End Function
```

**AND Condition**

When execution of your code depends upon multiple conditions being true, use "AND". This function will only return VIP if the cell IsNumeric() is referring to contains a number and that number is greater than 10. IsNumeric() is a VBA function that only returns true when what it is evaluating is a value. Note you can also use multiple ANDs:

(e.g. X =10 And Y =7 And Z = 15)

### IF Statement with Multiple Conditions: "OR"

```
Function Species(X)
  If X="Dog" Or X = "Wolf" Then
    Species = "Canine"
  End If
End Function
```

**OR Condition**

When a condition can be true based on more than one possible outcome then use the OR operator. In this example, if a cell contains either "Dog" or "Wolf", the function returns "Canine". If the cell contains neither, then the function returns 0. You can also use "OR" multiple times:

If X = “Dog" OR X = "Wolf" OR X = "Coyote" Then

Note that if no matches are found, in this example nothing is returned.
MAKING VBA CASE INSENSITIVE

You may have noticed the warning "Note this is case sensitive" in some of the previous examples. This means that typing "Cat" is not the same as typing "cat" or "CAT". This is because by default, VBA text comparisons are binary. To make VBA case insensitive, you have a couple of options.

**Case-by-Case Option: VBA's UCASE() Function**

The UCASE() function will convert a text string to all upper case letters. (Alternately, you can also use the Lcase() function to make them lower case.) You can use this to change the variable returned.

```vba
Function Animal(X)
    If X = "Dog" Then Animal = "Canine"
End Function

Function Animal(X)
    If UCase(X) = "DOG" Then Animal = "Canine"
End Function
```

**Case Sensitive**

This is the original case sensitive function. "Dog" ≠ DOG".

**Case Insensitive**

Here we are using UCASE() to make the value of X all upper case. We also have to change the comparison to upper case.

**Affect the Entire Module: Option Compare Text**

If you will be doing a lot of text comparison, there is a way to make an entire module case insensitive by placing the following code at the top of the module before any subs or functions. Note you cannot place it within a function. It affects all functions and subs within the module it is used in.

```
Option Compare Text
```
**Number Ranges Using SEPARATE Multiple IF Statements**

The example below uses multiple IF statements to determine the range a variable lies within. Using this structure, each IF statement is always read; therefore, unless using AND to assign a value range, you must be very careful that all of the conditions are mutually exclusive. For example, if you had one IF statement with this condition: \( S < 10 \) and another IF statement with this condition: \( S < 25 \), you may not get the results you intended. 5 for example is both less than 10 and less than 25 so both statements would be true. When there are multiple true results, the ultimate value of \( S \) becomes the last value assigned to it.

```vbnet
If S < 60 Then
    Line of code to execute
    Line of code to execute
End if

If S >= 60 And S <70 Then
    Line of code to execute
    Line of code to execute
End if

If S >= 70 And S <80 Then
    Line of code to execute
    Line of code to execute
End If

If S >= 80 And S <90 Then
    Line of code to execute
    Line of code to execute
End if

If S >= 90 Then
    Line of code to execute
    Line of code to execute
End if
```

These lines of code only execute when \( S \) is less than 60.

These lines of code only execute when \( S \) is between greater or equal to 60 and less than 70.

These lines of code only execute when \( S \) is between greater or equal to 70 and less than 80.

These lines of code only execute when \( S \) is greater or equal to 80 and less than 90.

These lines of code only execute when \( S \) is greater or equal to 90.

**Number Ranges Using a SINGLE IF Statement with the Optional "Else If" and "Else"**

The example below shows the IF structure using both the optional "Else" and "ElseIf" clauses to search within a range. It is important to realize that IF Else and Else are only read when the conditions above them are false. Because of this, you do not have to specify both an upper and lower range as we did in the previous example but they must be in the correct order and you must use the correct operator. For example, starting with \( S < 80 \) would not have worked but starting with \( S > 80 \) would.

```vbnet
If S < 60 Then
    Line of code to execute
    Line of code to execute
End if

ElseIf S < 70 Then
    Line of code to execute
    Line of code to execute
End ElseIf

ElseIf S <80 Then
    Line of code to execute
    Line of code to execute
End ElseIf

ElseIf S <90 Then
    Line of code to execute
    Line of code to execute
End ElseIf

Else
    Line of code to execute
    Line of code to execute
End Else

End If
```

These lines of code only execute when \( S \) is less than 60.

These lines of code only execute when the condition above is False and \( S \) is less than 70.

These lines of code only execute when all the condition above are False and \( S \) is less than 80.

These lines of code only execute when all the condition above are False and \( S \) is less than 90.

These lines of code only execute when all of the conditions above are false. Note that no conditions are used when using Else as they are with ElseIf.
IF Exercise 1: Specifying a Grade based on a Numeric Score

In this exercise, we will use an If Then structure with the optional IfElse and Else to assign letter grades to numeric scores.

1. In a module, type the function shown below.

```
Function Grade(S) 
    Grade is the name of our function. Its argument "S" represents a numeric score.
    If S < 60 Then
        Letter = "F"
        This condition is always read but Letter is only set to "F" when the condition is true.
    ElseIf S < 70 Then
        Letter = "D"
        This condition is only read when all conditions above it are false. If the condition is true, Letter is set to "D".
    ElseIf S < 80 Then
        Letter = "C"
        This condition is only read when all conditions above it are false. If the condition is true, Letter is set to "C".
    ElseIf S < 90 Then
        Letter = "B"
        This condition is only read when all conditions above it are false. If the condition is true, Letter is set to "B".
    Else
        Letter = "A"
        The Else section is only read when all of the conditions above it are false in which case Letter is set to "A".
    End If

    Grade = Letter
    This sets the name of the function equal to the value the variable "Letter" turned out to be.
End Function
```

**Testing the Function**

You can test this function either in the Immediate Window (Control + G) or on the "Scores" sheet in Excel.

```
?Grade(100)  A
?Grade (59)  F
```

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Student</td>
<td>Grade(B2)</td>
</tr>
<tr>
<td>2</td>
<td>Rocky</td>
<td>56</td>
</tr>
<tr>
<td>3</td>
<td>Susan</td>
<td>95</td>
</tr>
<tr>
<td>4</td>
<td>Kraig</td>
<td>81</td>
</tr>
<tr>
<td>5</td>
<td>Lettie</td>
<td>76</td>
</tr>
<tr>
<td>6</td>
<td>Karel</td>
<td>68</td>
</tr>
</tbody>
</table>

**Alternate Method without using Variables**

Here is the same function but here we didn't create the "Letter" variable.

```
Function Grade(S) 
    If S < 60 Then
        Grade = "F"
    ElseIf S < 70 Then
        Grade = "D"
    ElseIf S < 80 Then
        Grade = "C"
    ElseIf S < 90 Then
        Grade = "B"
    Else
        Grade = "A"
    End If
End Function
```

The variable method is useful when more work is needed before returning the solution but in this case, the "Letter" variable isn't really necessary.
Not Recommended – Nesting Multiple If Statements within "Else"
These can be hard to follow but you might encounter them. Another way to structure multiple conditions within a single IF statement is to nest them within "Else". Rather than using ElseIf, these use multiple Else statements which is allowable because we have multiple IF Then statements. Note you will need to have multiple "End IF"’s to match each opening IF.

Function Grade(S)

    If S < 60 Then
        Grade = "D"
    Else
        If S < 70 Then
            Grade = "C"
        Else
            If S < 80 Then
                Grade = "B"
            Else
                Grade = "A"
            End If
        End If
    End If

End Function
Understanding How Excel Stores Dates and Time

If you haven't worked with dates before in Excel, you should read through this section before proceeding to the next IF THEN exercise.

Dates
When you type a date in Excel, Excel stores it as a number, not a date. The numbering starts with 1 by assigning a 1 to January 1, 1900 and then simply increases the number with each successive date. For example, the number representing 12/21/2012 is 41,264 because it has been 41,264 days since 1/1/1900.

<table>
<thead>
<tr>
<th>Date</th>
<th>Serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/1/1900</td>
<td>1</td>
</tr>
<tr>
<td>5/8/1930</td>
<td>11,086</td>
</tr>
<tr>
<td>2/4/1968</td>
<td>24,872</td>
</tr>
<tr>
<td>1/1/2000</td>
<td>36,526</td>
</tr>
<tr>
<td>12/21/2012</td>
<td>41,264</td>
</tr>
</tbody>
</table>

You can type date in Excel using several different formats including:

- 12-15-2010
- 12/15/2010
- December 15, 2010
- Dec 15, 2010
- December 15, 10
- Dec 15, 10
- 15-December-2010
- 15-Dec-2010

**Date Exercise: Clearing Formats & Subtracting**
1. Go to the "Understanding_Time" sheet.

2. Type **12/31/2013** in A2.
3. Also type **12/31/2013** in B2.
4. Clear the formatting in B3.
5. In C2 type: =A2-Today()
6. Clear the formats if necessary.

Note that when subtracting dates the answer is in days.
Time & Time Durations
This section covers how Excel works with time and time durations.

The 24-Hour Clock
For time, Excel utilizes the 24 hour clock: 1am=1, 2am=2, etc. and once you reach 1pm you keep counting: 1pm is 13, 2pm is 14, until you reach midnight which is 24.
The image to the right shows the relationship between normal time and the 24 hour clock. Note that for clarity not every hour is shown.

Hours are Stored as a Serial Fraction of 24
Further, Excel stores time as a serial fraction of the 24 hour clock by dividing hours by 24. For example:
- The serial fraction for 9am would be: \( \frac{9}{24} = 0.38 \)
- The serial fraction for 3:00pm (15 in 24hr time) would be: \( \frac{15}{24} = 0.63 \)

Minutes are Divided by 60 and stored as a Serial Fraction of 24
Because hours are stored as a fraction of 24, minutes must be expressed as a fraction of an hour (60 minutes) and then that must be divided by 24. For example, 3:15 AM is stored as follows:
- \( \frac{(3 + 15/60)}{24} \)

Time and Time Durations are typed in this format:

<table>
<thead>
<tr>
<th>Type</th>
<th>Syntax</th>
<th>Example</th>
<th>Stored As</th>
<th>The math</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time of Day PM</td>
<td>HH:MM AM or PM</td>
<td>12:30 pm (space required)</td>
<td>0.52083333333</td>
<td>( \frac{12+30}{60} \div 24 )</td>
</tr>
<tr>
<td>Time of Day AM</td>
<td>HH:MM AM</td>
<td>12:30 am (space required)</td>
<td>0.02083333333</td>
<td>( \frac{0+30}{60} \div 24 )</td>
</tr>
<tr>
<td>Time Duration</td>
<td>HH:MM:SS</td>
<td>12:30 (12 hours and 30 minutes)</td>
<td>0.52083333333</td>
<td>( \frac{12+30}{60} \div 24 )</td>
</tr>
</tbody>
</table>

Exercise: Subtracting Time Occurring in Same Day
When the Time in and Time out are in the same day, finding the total time is easy: Find the duration of a person who started at 12pm and ended at 6:30pm.

1. Go to the "Understanding_Time" sheet.

2. Type 12:00 pm in A6.

3. Type 6:30 pm in B6.

4. In C6 type: \( = (B6 - A6) \)

5. Clear its formats.

6. Edit C6 and type: \( = (B6 - A6) \times 24 \)

Now you get 6.5

Note that dividing hours by 24 and minutes by 60 then 24 is just to get it into serial time. To express a serial duration as a decimal duration, just multiple the serial duration by 24.
Exercise: Time In & Time Out Spans Midnight
When a shift starts and ends on different days, we must use: 1-Time In + Time Out

For example, this shifts starts at 9pm (.88) and ends at 3am (.13) the next day for a total duration of 6 hours (.25 in serial time).

If you do Time Out – Time out: .13 -.88 = -.75
If you do Time In – Time Out: .88-.13 = .75

However, if you do 1 – Time In you get the half of the shift before midnight and if you then add this to the Time Out you get the second half of the shift. So, the final formula when shifts span midnight is:

1-Time In + Time Out 1 - .88 + .13 = .25 or 6hrs (.25*24)

1. Go to the "Understanding_Time" sheet.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Time In</td>
<td>Time Out</td>
</tr>
<tr>
<td>6</td>
<td>12:00 PM</td>
<td>6:30 PM</td>
</tr>
<tr>
<td>7</td>
<td>9:00 PM</td>
<td>3:00 AM</td>
</tr>
<tr>
<td>8</td>
<td>1/1/2013 9:00 PM</td>
<td>1/2/2013 3:00 AM</td>
</tr>
</tbody>
</table>

2. Type 9:00 pm in A7.
3. Type 3:00 am in B7.
4. In C7 type: =(1-A7+B7)*24
5. Clear the formats.

Exercise: Duration when using Date and Time is Used
When the date and the time is typed into the same cell, the mathematics of getting the duration is much less complicated. Later Date – Earlier Date

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Time In</td>
<td>Time Out</td>
</tr>
<tr>
<td>6</td>
<td>12:00 PM</td>
<td>6:30 PM</td>
</tr>
<tr>
<td>7</td>
<td>9:00 PM</td>
<td>3:00 AM</td>
</tr>
<tr>
<td>8</td>
<td>1/1/2013 9:00 PM</td>
<td>1/2/2013 3:00 AM</td>
</tr>
</tbody>
</table>

2. Type 1/1/2013 9 pm in A8.
3. Type 1/2/2013 3 am in B8.
4. In C8 type: =(B8-A8)*24
5. Clear the formats.
**IF Structure Exercise 2: Determine Time Difference**

Take a look at the "Timesheets" sheet. As you can see, shifts have been entered in time only format. We would like to create a custom function that returns the duration. As we learned on the previous pages, there are two different formulas we must use to return time only durations:

- Time Out > Time In: \( \text{Time Out} - \text{Time In} \)
- Time Out < Time In: \( 1 - \text{Time In} + \text{Time Out} \)

1. If necessary, press **ALT + F11** to toggle to the VBA editor.
2. In a module, type the function shown below.

```vba
Function TimeDiff(TimeIn As Double, TimeOut As Double)
    If TimeOut >= TimeIn Then
        TimeDiff = (TimeOut - TimeIn) * 24
    Else
        TimeDiff = (1 - TimeIn + TimeOut) * 24
    End If
End Function
```

*Because the times are really just decimal numbers, we will use the "As Double" data type.*

**If Then**
This handles the possibility of the start and end times being in the same day by first checking to see if the TimeOut is greater or equal to the TimeIn. If it is then the \( \text{TimeOut} - \text{TimeIn} \) expression is used.

**Else**
Else is only read if the condition is false. It handles what to do when the start and end times span midnight by using this formula: \( 1 - \text{TimeIn} + \text{Timeout} \)

---

**Testing the Function**

1. If necessary, press **ALT + F11** to toggle to Excel.
2. Go to the "Timesheets" sheet.

3. In D4 type: \( \text{=TimeDiff(B4,C4)} \)

5. In G4 type: \( \text{=TimeDiff(E4,F4)} \)
Alternative Approach to Exercise 2

Note that we didn’t have to include the multiplication of 24 in our custom function. We could have done it later and formatted columns D and G to display as durations. For example:

However you need to realize that C2 does not contain a 9, it actually contains 0.375. This means that when you wish to calculate your GROSS, you will still have to multiple by 24. In J24 you would have to type: **=H4**24**I4**
### Built-in VBA Functions for Excel

Just as there are worksheet functions such as `Sum()` and `Average()`, VBA has functions as well. Below is a partial list of some of the more useful VBA functions you can use in Excel. Note that when Excel has both a worksheet function and a VBA function that do the same thing (i.e. "Len()", "Right()", etc., you must use the VBA function in VBA code. To learn more about any of the functions below, type the name of the function in a module, press your spacebar, then press F1 for online help.

<table>
<thead>
<tr>
<th>Builtin VBA Function</th>
<th>Example Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abs</td>
<td>Abs(number)</td>
<td>Returns the absolute value of a number.</td>
</tr>
<tr>
<td>Array</td>
<td>CreateArray()</td>
<td>Creates a new array.</td>
</tr>
<tr>
<td>Asc</td>
<td>Asc(string)</td>
<td>Returns the ASCII value of a character.</td>
</tr>
<tr>
<td>Atn</td>
<td>Atn(number)</td>
<td>Returns the arctangent of a number.</td>
</tr>
<tr>
<td>CBool</td>
<td>CBool(boolean)</td>
<td>Converts a value to a Boolean.</td>
</tr>
<tr>
<td>CByte</td>
<td>CByte(number)</td>
<td>Returns the byte representation of a number.</td>
</tr>
<tr>
<td>CCur</td>
<td>CCur(value)</td>
<td>Returns the currency representation of a number.</td>
</tr>
<tr>
<td>CDate</td>
<td>CDate(date)</td>
<td>Returns the date representation of a number.</td>
</tr>
<tr>
<td>CDbl</td>
<td>CDbl(number)</td>
<td>Returns the double precision floating-point representation of a number.</td>
</tr>
<tr>
<td>CDec</td>
<td>CDec(number)</td>
<td>Returns the decimal representation of a number.</td>
</tr>
<tr>
<td>Choose</td>
<td>Choose(index)</td>
<td>Chooses an element from a list.</td>
</tr>
<tr>
<td>Chr</td>
<td>Chr(character)</td>
<td>Returns the character associated with a code.</td>
</tr>
<tr>
<td>CInt</td>
<td>CInt(number)</td>
<td>Returns the integer representation of a number.</td>
</tr>
<tr>
<td>CLng</td>
<td>CLng(number)</td>
<td>Returns the long precision floating-point representation of a number.</td>
</tr>
<tr>
<td>CreateObject</td>
<td>CreateObject()</td>
<td>Creates an object.</td>
</tr>
<tr>
<td>CSng</td>
<td>CSng(number)</td>
<td>Returns the single precision floating-point representation of a number.</td>
</tr>
<tr>
<td>CStr</td>
<td>CStr(string)</td>
<td>Returns the string representation of a number.</td>
</tr>
<tr>
<td>CurDir</td>
<td>CurDir(path)</td>
<td>Returns the current directory path.</td>
</tr>
<tr>
<td>CVar</td>
<td>CVar()</td>
<td>Creates a variable.</td>
</tr>
<tr>
<td>CVDate</td>
<td>CVDate(date)</td>
<td>Returns the date representation of a number.</td>
</tr>
<tr>
<td>CVErr</td>
<td>CVErr()</td>
<td>Creates an error.</td>
</tr>
<tr>
<td>Date</td>
<td>Date()</td>
<td>Creates a date.</td>
</tr>
<tr>
<td>DateAdd</td>
<td>DateAdd(days)</td>
<td>Adds days to a date.</td>
</tr>
<tr>
<td>DateDiff</td>
<td>DateDiff()</td>
<td>Returns the difference between two dates.</td>
</tr>
<tr>
<td>DatePart</td>
<td>DatePart()</td>
<td>Returns a part of a date.</td>
</tr>
<tr>
<td>DateSerial</td>
<td>DateSerial()</td>
<td>Returns the serial number of a date.</td>
</tr>
<tr>
<td>DateValue</td>
<td>DateValue()</td>
<td>Creates a date value.</td>
</tr>
<tr>
<td>Day</td>
<td>Day()</td>
<td>Returns the day of the month.</td>
</tr>
<tr>
<td>Dir</td>
<td>Dir()</td>
<td>Creates a directory.</td>
</tr>
<tr>
<td>DoEvents</td>
<td>DoEvents()</td>
<td>Creates an event.</td>
</tr>
<tr>
<td>EOF</td>
<td>EOF()</td>
<td>Returns the end of file indicator.</td>
</tr>
<tr>
<td>Error</td>
<td>Error()</td>
<td>Returns an error.</td>
</tr>
<tr>
<td>Exp</td>
<td>Exp(number)</td>
<td>Returns the exponential of a number.</td>
</tr>
<tr>
<td>FileAttr</td>
<td>FileAttr(path)</td>
<td>Returns the file attributes.</td>
</tr>
<tr>
<td>FileDate</td>
<td>FileDate(date)</td>
<td>Returns the date representation of a number.</td>
</tr>
<tr>
<td>FileDateTime</td>
<td>FileDateTime()</td>
<td>Returns the file date time.</td>
</tr>
<tr>
<td>FileLen</td>
<td>FileLen()</td>
<td>Returns the file length.</td>
</tr>
<tr>
<td>Fix</td>
<td>Fix(number)</td>
<td>Returns the fixed number.</td>
</tr>
<tr>
<td>Format</td>
<td>Format()</td>
<td>Formats a number.</td>
</tr>
<tr>
<td>FormatCurrency</td>
<td>FormatCurrency()</td>
<td>Formats a currency.</td>
</tr>
<tr>
<td>FormatDateTime</td>
<td>FormatDateTime()</td>
<td>Formats a date time.</td>
</tr>
<tr>
<td>FormatNumber</td>
<td>FormatNumber()</td>
<td>Formats a number.</td>
</tr>
<tr>
<td>FormatPercent</td>
<td>FormatPercent()</td>
<td>Formats a percent.</td>
</tr>
<tr>
<td>FreeFile</td>
<td>FreeFile()</td>
<td>Returns the free file pointer.</td>
</tr>
<tr>
<td>GetAll</td>
<td>GetAll()</td>
<td>Returns all objects.</td>
</tr>
<tr>
<td>GetAttr</td>
<td>GetAttr()</td>
<td>Returns the file attributes.</td>
</tr>
<tr>
<td>GetObject</td>
<td>GetObject()</td>
<td>Returns an object.</td>
</tr>
<tr>
<td>GetSetting</td>
<td>GetSetting()</td>
<td>Returns a setting.</td>
</tr>
<tr>
<td>Hex</td>
<td>Hex()</td>
<td>Returns the hexadecimal value of a number.</td>
</tr>
<tr>
<td>Hour</td>
<td>Hour()</td>
<td>Returns the hour of the day.</td>
</tr>
<tr>
<td>If</td>
<td>If(condition)</td>
<td>Evaluates a condition.</td>
</tr>
<tr>
<td>IfIs</td>
<td>IfIs()</td>
<td>Returns a value.</td>
</tr>
<tr>
<td>Input</td>
<td>Input()</td>
<td>Returns input from the user.</td>
</tr>
<tr>
<td>InputBox</td>
<td>InputBox()</td>
<td>Returns input from the user.</td>
</tr>
<tr>
<td>Join</td>
<td>Join()</td>
<td>Joins two strings.</td>
</tr>
<tr>
<td>LBound</td>
<td>LBound()</td>
<td>Returns the lower bound of an array.</td>
</tr>
<tr>
<td>Len</td>
<td>Len()</td>
<td>Returns the length of a string.</td>
</tr>
<tr>
<td>LCase</td>
<td>LCase(string)</td>
<td>Converts a string to lowercase.</td>
</tr>
<tr>
<td>Left</td>
<td>Left()</td>
<td>Returns the leftmost characters of a string.</td>
</tr>
<tr>
<td>LeftChar</td>
<td>LeftChar()</td>
<td>Returns the leftmost character of a string.</td>
</tr>
<tr>
<td>LenB</td>
<td>LenB()</td>
<td>Returns the length of a byte array.</td>
</tr>
<tr>
<td>LTrim</td>
<td>LTrim()</td>
<td>Trims leading spaces from a string.</td>
</tr>
<tr>
<td>Mid</td>
<td>Mid()</td>
<td>Returns the middle characters of a string.</td>
</tr>
<tr>
<td>MidB</td>
<td>MidB()</td>
<td>Returns the middle byte of a byte array.</td>
</tr>
<tr>
<td>Min</td>
<td>Min()</td>
<td>Returns the minimum value.</td>
</tr>
<tr>
<td>Minute</td>
<td>Minute()</td>
<td>Returns the minute of the hour.</td>
</tr>
<tr>
<td>Month</td>
<td>Month()</td>
<td>Returns the month of the year.</td>
</tr>
<tr>
<td>MonthName</td>
<td>MonthName()</td>
<td>Returns the month name.</td>
</tr>
<tr>
<td>MonthSerial</td>
<td>MonthSerial()</td>
<td>Returns the month serial number.</td>
</tr>
<tr>
<td>MsgBox</td>
<td>MsgBox()</td>
<td>Displays a message box.</td>
</tr>
<tr>
<td>Now</td>
<td>Now()</td>
<td>Returns the current date and time.</td>
</tr>
<tr>
<td>Oct</td>
<td>Oct()</td>
<td>Returns the octal number.</td>
</tr>
<tr>
<td>Opt</td>
<td>Opt()</td>
<td>Returns an optional value.</td>
</tr>
<tr>
<td>Panic</td>
<td>Panic()</td>
<td>Raises an error.</td>
</tr>
<tr>
<td>Replace</td>
<td>Replace()</td>
<td>Replaces text in a string.</td>
</tr>
<tr>
<td>RGB</td>
<td>RGB()</td>
<td>Returns the RGB color value.</td>
</tr>
<tr>
<td>Right</td>
<td>Right()</td>
<td>Returns the rightmost characters of a string.</td>
</tr>
<tr>
<td>RightChar</td>
<td>RightChar()</td>
<td>Returns the rightmost character of a string.</td>
</tr>
<tr>
<td>Round</td>
<td>Round()</td>
<td>Rounds a number.</td>
</tr>
<tr>
<td>RTrim</td>
<td>RTrim()</td>
<td>Trims trailing spaces from a string.</td>
</tr>
<tr>
<td>Serie</td>
<td>Serie()</td>
<td>Returns a serial number.</td>
</tr>
<tr>
<td>Short</td>
<td>Short()</td>
<td>Returns a short integer.</td>
</tr>
<tr>
<td>Shell</td>
<td>Shell()</td>
<td>Launches a command.</td>
</tr>
<tr>
<td>Space</td>
<td>Space()</td>
<td>Returns a space character.</td>
</tr>
<tr>
<td>Split</td>
<td>Split()</td>
<td>Splits a string.</td>
</tr>
<tr>
<td>Sqr</td>
<td>Sqr()</td>
<td>Returns the square root of a number.</td>
</tr>
<tr>
<td>Str</td>
<td>Str()</td>
<td>Returns a string.</td>
</tr>
<tr>
<td>StrComp</td>
<td>StrComp()</td>
<td>Converts a string to uppercase.</td>
</tr>
<tr>
<td>StrConv</td>
<td>StrConv()</td>
<td>Converts a string to a number.</td>
</tr>
<tr>
<td>String</td>
<td>String()</td>
<td>Returns a string.</td>
</tr>
<tr>
<td>StrReverse</td>
<td>StrReverse()</td>
<td>Reverses a string.</td>
</tr>
<tr>
<td>Tab</td>
<td>Tab()</td>
<td>Returns a tab character.</td>
</tr>
<tr>
<td>Tan</td>
<td>Tan()</td>
<td>Returns the tangent of a number.</td>
</tr>
<tr>
<td>Time</td>
<td>Time()</td>
<td>Returns the current time.</td>
</tr>
<tr>
<td>TimeSerial</td>
<td>TimeSerial()</td>
<td>Returns the time serial number.</td>
</tr>
<tr>
<td>Trim</td>
<td>Trim()</td>
<td>Removes extra spaces from a string.</td>
</tr>
<tr>
<td>TypeName</td>
<td>TypeName()</td>
<td>Returns a type name.</td>
</tr>
<tr>
<td>UBounded</td>
<td>UBounded()</td>
<td>Returns a bounded value.</td>
</tr>
<tr>
<td>UCased</td>
<td>UCased()</td>
<td>Converts a string to uppercase.</td>
</tr>
<tr>
<td>Val</td>
<td>Val()</td>
<td>Returns a value.</td>
</tr>
<tr>
<td>VarType</td>
<td>VarType()</td>
<td>Returns a variable type.</td>
</tr>
<tr>
<td>Weak</td>
<td>Weak()</td>
<td>Creates a weak reference.</td>
</tr>
<tr>
<td>Weekday</td>
<td>Weekday()</td>
<td>Returns the weekday of the month.</td>
</tr>
<tr>
<td>WeekdayName</td>
<td>WeekdayName()</td>
<td>Returns the weekday name.</td>
</tr>
<tr>
<td>Year</td>
<td>Year()</td>
<td>Returns the year of the month.</td>
</tr>
</tbody>
</table>
VBA Functions Exercise 1: Return the Number of Days in a Month

In this exercise, we will create a function to return the number of days in a month for any given date or in other words, return the last day of a given month. We will need to use the following VBA Functions to accomplish this:

**DateSerial(year,month,day)**

This VBA function returns a serial date value given a year, month, and day in that order. Below are some examples:

- `DateSerial(1983,4,3)` returns 4/3/1983 when formatted as a date. (30409 without formatting.)
- `DateSerial(1983,4,0)` returns 3/31/1983 when formatted as a date. (30406 without formatting.)

When you use a zero for the day, Excel returns a date for the last day of the previous month.

- `DateSerial(1983,4+1,0)` returns 4/30/1983 when formatted as a date.

To return the last day of the current month, add 1 to the month and use 0 as the day.

You can try all of these in the Immediate Window although it will format them as dates rather than displaying serial dates. (Control + G) as shown to the right. For example:

In the Immediate Window, type:

- `?DateSerial(1983,4,3)` and press enter.
- `?DateSerial(1983,4,0)` and press enter.
- `?DateSerial(1983,4+1,0)` and press enter.

**Year(Serial Date)**

This extracts and returns the year from a given date. We will use it to pull the year out of a given date and place it in the year section of the DateSerial() function.

- `Year(30409)` returns 1983

**Month(Serial Date)**

This extracts and returns the month from a given date. We will use it to pull the month out of a given date and place it in the month section of the DateSerial() function.

- `Month(30409)` returns 4 (i.e. April)

**Day(Serial Date)**

This extracts and returns the day of the month from a given date. We will use it to pull the last day of the month out of a date by using 0 as the day and adding 1 to the month. (See above.)

- `Day(30409)` returns 3 (i.e. 3rd day of the month)
Our final formula where "D" is a variable representing a date in a cell is:

\[ \text{Day(DateSerial(Year(D), \text{Month(D)}+1,0))} \]

Creating the Function to Return the Date of the Last Day of the Month

1. In a module, type the following custom function:

```
Function LastDay(D As Date)
    LastDay = Day(DateSerial(Year(D), Month(D) + 1, 0))
End Function
```

Testing the Function

To test it out, go to the `LastDay_of_Month` sheet and type the formulas shown. Our goal is to determine the rate paid per day and the average rate paid for the month.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Month</td>
<td>Date Charged</td>
<td>Monthly Total</td>
<td>Per Day Amount</td>
</tr>
<tr>
<td>2</td>
<td>January</td>
<td>1/15/2013</td>
<td>$500</td>
<td>$16.1 =C2/LastDay(B2)</td>
</tr>
<tr>
<td>3</td>
<td>February</td>
<td>2/18/2013</td>
<td>$505</td>
<td>$18.0</td>
</tr>
<tr>
<td>4</td>
<td>March</td>
<td>3/14/2013</td>
<td>$650</td>
<td>$21.0</td>
</tr>
<tr>
<td>5</td>
<td>April</td>
<td>4/7/2013</td>
<td>$450</td>
<td>$15.0</td>
</tr>
<tr>
<td>6</td>
<td>May</td>
<td>5/25/2013</td>
<td>$750</td>
<td>$24.2</td>
</tr>
<tr>
<td>7</td>
<td>June</td>
<td>6/18/2013</td>
<td>$635</td>
<td>$21.2</td>
</tr>
<tr>
<td>8</td>
<td>July</td>
<td>7/12/2013</td>
<td>$400</td>
<td>$12.9</td>
</tr>
<tr>
<td>9</td>
<td>August</td>
<td>8/29/2013</td>
<td>$450</td>
<td>$14.5</td>
</tr>
<tr>
<td>10</td>
<td>September</td>
<td>9/22/2013</td>
<td>$650</td>
<td>$21.7</td>
</tr>
<tr>
<td>11</td>
<td>October</td>
<td>10/16/2013</td>
<td>$440</td>
<td>$14.2</td>
</tr>
<tr>
<td>12</td>
<td>November</td>
<td>11/9/2013</td>
<td>$530</td>
<td>$17.7</td>
</tr>
<tr>
<td>13</td>
<td>December</td>
<td>12/3/2013</td>
<td>$625</td>
<td>$20.2</td>
</tr>
<tr>
<td>14</td>
<td>Average:</td>
<td></td>
<td></td>
<td>$548.75 =AVERAGE(C2:C13)</td>
</tr>
<tr>
<td>16</td>
<td>Avg Daily Rate:</td>
<td>$18.05</td>
<td>=AVERAGE(D2:D13)</td>
<td></td>
</tr>
</tbody>
</table>
VBA Functions Exercise 2: Remove the Leading “The” from Movie Names

We have a list of movie names that we would like to sort; however, we wish to ignore the leading “The”. For example, ”The Bride” should be with the ”Bs” not the ”Ts”. To do this we will need to see if the first 4 characters of a movie's name is ”The ” and if it is, remove that part. To accomplish this, we will use the following VBA functions:

- Len()
- Left()
- Right()
- As well as the IF Then Else structure

Before we begin, let's see the VBA functions work in the Immediate Window.

**Len(expression)**

Len returns the number of characters in a character string. The expression can be the name of a variable or the actual character string. If the actual string is used, it must be in quotes. For example:

```vba
?Len("The Bride") returns 9 because there are 9 characters including the space in the string.
```

**Left(string being evaluated, Number of charters to display)**

Left returns the left side of a string. The number of characters to display is given by the second part of the function. For example:

```vba
Left("The Bride",4) returns "The " because they are the first 4 characters in the string.
```

**Right(string being evaluated, Number of charters to display)**

Right returns the right side of a string. The number of characters to display is given by the second part of the function. For example:

```vba
?Right("The Bride",5) returns "Bride" because the last five characters are Bride.
```

**Putting it All Together**

We can examine the first four characters of a move's name to see if it contains ”The ” by using the Left() VBA function. Where ”M” represents the name of the move:

Left(M, 4) = "The 

To return the remaining characters excluding ”The ”, we need to find the total number of characters in the movie's name (Len), subtract 4 from that to know how many characters starting from the Right() to count back and display its name:

Right(M, Len(M) - 4)
Creating the Function
Now that we understand its parts, we will put it together.

1. In a module, type the following custom function to remove "The " from movie names starting with "The ".

```vba
Function NoThe(M)
    NoThe = Right(M, Len(M) - 4)
End Function
```

Testing the Function
1. Press ALT + F11 to return to Excel.
2. Go to the "Remove_The" sheet.
3. To test it out, type the formula shown in E2 in cell D2 and copy down:  =NoThe(A2)

You can now sort by the new column.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Movie</td>
<td>Rating</td>
<td>Profit</td>
<td>Name with The Removed</td>
<td>NoThe(A2)</td>
</tr>
<tr>
<td>2</td>
<td>Alien 1</td>
<td>R</td>
<td>$ 87,665,701</td>
<td>Alien 1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Alien 2</td>
<td>R</td>
<td>$ 76,574,870</td>
<td>Alien 2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Alien 3</td>
<td>R</td>
<td>$ 78,680,580</td>
<td>Alien 3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Basic Instinct</td>
<td>R</td>
<td>$ 91,837,580</td>
<td>Basic Instinct</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Home Alone</td>
<td>G</td>
<td>$ 140,099,000</td>
<td>Home Alone</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Home Alone 2</td>
<td>PG</td>
<td>$ 102,000,000</td>
<td>Home Alone 2</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Return of the Jedi</td>
<td>G</td>
<td>$ 169,193,000</td>
<td>Return of the Jedi</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Star Wars</td>
<td>G</td>
<td>$ 193,777,000</td>
<td>Star Wars</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>The Commitments</td>
<td>R</td>
<td>$ 49,876,300</td>
<td>Commitments</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>The Empire Strikes Back</td>
<td>G</td>
<td>$ 141,672,000</td>
<td>Empire Strikes Back</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>The Princess Bride</td>
<td>G</td>
<td>$ 67,574,560</td>
<td>Princess Bride</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Aladdin</td>
<td>G</td>
<td>$ 89,767,800</td>
<td>Aladdin</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Honey, I Blew Up the Kids</td>
<td>G</td>
<td>$ 58,767,800</td>
<td>Honey, I Blew Up the Kids</td>
<td></td>
</tr>
</tbody>
</table>

Note that if you wanted to place ", The" after the names of movies that started with "The ", you would use the concatenation character at the end of our code like this:

```
NoThe = Right(M, Len(M) - 4) & ", The"
```
Select Case (An Alternative to "If Then")

An alternative to using IF THEN is to use SELECT CASE. Although IF and SELECT CASE can usually be used interchangeably, many people find SELECT CASE easier to follow. In general,

- Use SELECT CASE when you are testing a single variable (e.g. X) with more than two conditions.
- Use IF THEN when you are testing more than one variable (e.g. X and Y). Note that the "AND" keyword does not work with "Select Case" but you can make "OR" conditions using commas.

Select Case Structure

- The Select Case statement must begin with Select Case and end with End Select.
- Select Case reads from top down and stops when it finds a match.
- You can use "Exit Case" in any of the code lines to stop processing the Select Case statement.

The Condition can take several forms:

- OR Conditions: Case 1, 3, 5 (i.e. 1 or 3 or 5)
  Case "Wolf", "Dog", "Coyote" (i.e. Wolf or Dog or Coyote)
- Combo OR Conditions: Case 1 to 3, 7, 11, Is > X (i.e. 1 through 3 or 7 or 11 or greater than X)
- Within Range: Case 5 to 10 (i.e. between 5 and 10 inclusively)
- Over/Under Range: Case Is >40 (i.e. greater than 40)
Select Case Exercise 1: Or Condition with a Text String
In this exercise, we wish to return "Canine" if a cell contains Dog, Wolf, or Coyote. Note there are several structures you can use as shown in the two examples below.

1. In a module, type either of the functions shown below.

---

Function KNine(X As String)
Select Case X
  Case "Wolf", "Dog", "Coyote"
    KNine = "Canine"
End Select
End Function

---

or

---

Function KNine (X As String)
Select Case X
  Case "Wolf", "Dog", "Coyote"
    Sp = "Canine"
End Select
KNine = Sp
End Function

---

Function Result Inside of Select Statement
Here we have placed KNine = "Canine" in the answer part of the Select statement. This structure can be used when no further analysis is needed before returning an answer.

X - This acts as a variable representing a cell address.
Select Case X – This is testing what X could be.
Case – This line tests to see if X contains "Wolf", or "Dog", or "Coyote". If it does, then the function returns "Canine".

Function Result Outside of Select Statement
This uses the same logic as the function above but instead of returning the end result from within the Select statement, we are returning it after by creating the variable "Sp". Sp has the correct value to return and KNine is set to equal Species. This structure is useful when further processing of X is needed before returning a solution.
Note we didn't declare our variable Species but we really should have. VBA will allow you to create variable on the fly like this.

Testing the Function
1. Press ALT + F11 to return to Excel.
2. On the "Canines" sheet, type the formula shown: =KNine(A1)
3. Copy the function down.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dog</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Wolf</td>
<td>Canine</td>
</tr>
<tr>
<td>3</td>
<td>Coyote</td>
<td>Canine</td>
</tr>
<tr>
<td>4</td>
<td>Fox</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Jackal</td>
<td>0</td>
</tr>
</tbody>
</table>
Select Case Exercise 2: Using "Case Else"

This is the same example as on the previous page except we are using the optional "Case Else" to return "Not on List" if Wolf, Dog, or Coyote are not found.

Here we are using the optional Case Else to return "Not on List" if X is not a Wolf, Dog, or Coyote. Note that we are returning the functions result from within the Select Case statement.

Or

Here we are using the optional Case Else to return "Not on List" if X is not a Wolf, Dog, or Coyote. Note when the case is true, we are setting the variable "Sp" equal to "Not on List" and then setting that to our function name outside of the select case statement. This structure is useful when you wish to do further processing with the result of the select statement prior to setting it to what your function returns.

Testing the Function

1. Press ALT + F11 to return to Excel.
2. On the "Canines" sheet, type the formula shown: =KNine2(A1)
3. Copy the function down.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dog</td>
<td>Canine</td>
</tr>
<tr>
<td>2</td>
<td>Wolf</td>
<td>Canine</td>
</tr>
<tr>
<td>3</td>
<td>Coyote</td>
<td>Canine</td>
</tr>
<tr>
<td>4</td>
<td>Fox</td>
<td>Not on List</td>
</tr>
<tr>
<td>5</td>
<td>Jackal</td>
<td>Not on List</td>
</tr>
</tbody>
</table>
Select Case Exercise 3: Number Ranges (Tiered Commission Based on Sales)

For this exercise, take a look at the "Commissions" sheet. In this scenario, a company pays its salespeople a commission based upon their sales:

\[ \text{Commission} = \text{Sales} \times \text{Commission Rate} \]

However, the commission rate is not flat but is tiered based upon sales amount as follows:

<table>
<thead>
<tr>
<th>Amount</th>
<th>Commission Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than $1,000</td>
<td>8%</td>
</tr>
<tr>
<td>Between $1,000 and $4,999.00</td>
<td>9%</td>
</tr>
<tr>
<td>Between $5,000 and $9,999.99</td>
<td>10%</td>
</tr>
<tr>
<td>Greater than or equal to $10,000</td>
<td>11%</td>
</tr>
</tbody>
</table>

1. Within a module, create the custom function shown below.

```
Function Comm(Sales As Double)
    Dim R As Double

    Select Case Sales
        Case Is < 1000
            R = 0.08
        Case 1000 To 4999.99
            R = 0.09
        Case 5000 To 9999.99
            R = 0.1
        Case Is >= 10000
            R = 0.11
    End Select

    Comm = Sales * R
End Function
```

**Testing the Function**

To test the Comm() function:

1. Press `ALT + F11` to toggle to Excel.
2. Go to the "Commissions" sheet.
3. In E2, type: `=Comm(D2)` and press enter.
4. Copy the formula down.

If you have issues, try the following:
- Fix any obvious issues.
- Comment out code.
- Press `CONTROL + SHIFT + F8` to step out of the debugger.
**Select Case Exercise 4: Using Constants (Commission Based on Employee ID)**

This uses the same spreadsheet as the previous except that here, each salesperson has their own commission rate:

\[
\text{Commission} = \text{Employee's Commission Rate} \times \text{Sales for the day.}
\]

Note that this example uses the optional colon (:) to place the Case Is and the result on the same line.

```vba
Function EmpComm(EmpID As Integer, Sales As Double)

    Dim ER As Double 'Employee rate

    Select Case EmpID
        Case Is = 556: ER = Rocky_556
        Case Is = 636: ER = Susan_636
        Case Is = 552: ER = Kraig_552
        Case Is = 551: ER = Lettie_551
        Case Is = 365: ER = Karel_365
    End Select

    EmpComm = Sales * ER
End Function
```

**Testing the Function**

To test the EmpComm() function:

1. Press **Alt + F11** to toggle to Excel.
2. Go to the "Commissions" sheet.
3. In F2, type: =EmpComm(A2,D2) and press enter.
4. Copy the formula down.

```vba
Function EmpComm(EmpID As Integer, Sales As Double)

    Dim ER As Double 'Employee rate

    Select Case EmpID
        Case Is = 556: ER = Rocky_556
        Case Is = 636: ER = Susan_636
        Case Is = 552: ER = Kraig_552
        Case Is = 551: ER = Lettie_551
        Case Is = 365: ER = Karel_365
    End Select

    EmpComm = Sales * ER
End Function
```

**ALTERNATIVE CODING**

Note that we didn't have to use constants to specify the employee rates or use the colon. Here is the same function as above without either. The constants are nice though because we have associated names with employee IDs.
Select Case Exercise 5: Handling Dates (Pension Rate Based on Hire Date)

When specifying specific dates, they must be entered in M/D/YEAR format and then must be enclosed in # signs. If you would like to dynamically specify the current date, use the VBA function Now(). Also, use the “As Date” variable type when declaring dates.

In this example, our company has scaled back the amount it contributes to its employee's pensions over the years as follows:

- Hired before 1/1/1990 → 25% Contribution
- Hired in the 1990's → 20% Contribution
- Hired in the 2000's → 15% Contribution
- Hired after 12/31/2009 → 10% Contribution

The function below returns contribution rate provided by the employer based on the employee's hire date.

```vba
Function TribRate(HireDate As Date)
    Dim TR As Double

    Select Case HireDate
        Case Is < #1/1/1990#
            TR = 0.25
        Case #1/1/1990# To #12/31/2000#
            TR = 0.2
        Case #1/1/2000# To #12/31/2010#
            TR = 0.15
        Case Is > #12/31/2010#
            TR = 0.1
    End Select

    TribRate = TR
End Function
```

TribRate is the name of our function. HireDate will hold the employee's Hire Date. TR is a variable representing the contribution rate. Select Case determines which rate to assign the variable TR by comparing the Hire Date the user specifies with the cases listed. Whatever case turns out to be true is the final value of TR. Note the use of “Case Is” when using Boolean operators and “To” when checking (inclusively) between two points. This sets what the function (TribRate) returns to the final value of TR (the rate).

Testing the Function

1. Press ALT + F11 to return to Excel.
2. On the "HireDate" sheet, type the following in D2: =TribRate(C2)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Emp ID</td>
<td>Name</td>
<td>Hire Date</td>
</tr>
<tr>
<td>2</td>
<td>556</td>
<td>Rocky</td>
<td>5/8/1998</td>
</tr>
<tr>
<td>3</td>
<td>636</td>
<td>Susan</td>
<td>12/15/1985</td>
</tr>
<tr>
<td>4</td>
<td>552</td>
<td>Kraig</td>
<td>6/10/2006</td>
</tr>
<tr>
<td>5</td>
<td>551</td>
<td>Lettie</td>
<td>5/4/2011</td>
</tr>
<tr>
<td>6</td>
<td>365</td>
<td>Karel</td>
<td>4/3/1989</td>
</tr>
</tbody>
</table>
USC Marshall School of Business  wilmeth@usc.edu  Custom_Functions.docx  10/23/2013  Page 41 of 65

# USING WORKSHEET FUNCTIONS IN VBA

If an Excel worksheet function does not have a corresponding function in VBA, you can use the worksheet function in a custom function. If it does have a corresponding VBA function then you must use the VBA version of it. To use a worksheet function in VBA, you must precede it with the object: `WorksheetFunction`.

Syntax:  `WorksheetFunction.FunctionName()`  
*(Note that "Application.FunctionName()" works too.)*

Example:  `WorksheetFunction.Average(arguments)`

Below is a list (from 2003) of worksheet functions you can use in a custom function.

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acos</td>
<td>Dollar</td>
<td>IsText</td>
<td>Proper</td>
<td>Trend</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Acosh</td>
<td>DProduct</td>
<td>Kurt</td>
<td>Pv</td>
<td>Trim</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>And</td>
<td>DSTdev</td>
<td>Large</td>
<td>Quartile</td>
<td>TrimMean</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asc</td>
<td>DSTdevP</td>
<td>LinEst</td>
<td>Radians</td>
<td>TTest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asin</td>
<td>DSum</td>
<td>Ln</td>
<td>Rank</td>
<td>USDollar</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asinh</td>
<td>DVar</td>
<td>Log</td>
<td>Rate</td>
<td>Var</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atan2</td>
<td>DVarP</td>
<td>Log10</td>
<td>Replace</td>
<td>VarP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atanh</td>
<td>Even</td>
<td>LogEst</td>
<td>ReplaceB</td>
<td>Vdb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AveDev</td>
<td>ExponDist</td>
<td>LogInv</td>
<td>Rept</td>
<td>Vlookup</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>Fact</td>
<td>LogNormDist</td>
<td>Roman</td>
<td>Weekday</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BahaText</td>
<td>FDist</td>
<td>Lookup</td>
<td>Round</td>
<td>Weibull</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BetaDist</td>
<td>Find</td>
<td>Match</td>
<td>RoundDown</td>
<td>ZTest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BetaInv</td>
<td>FindB</td>
<td>Max</td>
<td>RoundUp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BinomDist</td>
<td>Flv</td>
<td>MDeterm</td>
<td>RSq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ceiling</td>
<td>Fisher</td>
<td>Median</td>
<td>RTD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ChiDist</td>
<td>FisherInv</td>
<td>Min</td>
<td>Search</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ChiInv</td>
<td>Fixed</td>
<td>MInverse</td>
<td>SearchB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ChiTest</td>
<td>Floor</td>
<td>M irr</td>
<td>Sinh</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Choose</td>
<td>Forecast</td>
<td>MMult</td>
<td>Skew</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clean</td>
<td>Frequency</td>
<td>Mode</td>
<td>Sln</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combin</td>
<td>FTtest</td>
<td>NegBinomDist</td>
<td>Slope</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Confidence</td>
<td>Fv</td>
<td>NormDist</td>
<td>Small</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Correl</td>
<td>GammaDist</td>
<td>NormInv</td>
<td>Standardize</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cosh</td>
<td>GammaInv</td>
<td>NormSDist</td>
<td>StDev</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Count</td>
<td>GammaLn</td>
<td>NormSInv</td>
<td>StDevP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CountA</td>
<td>GeoMean</td>
<td>NPer</td>
<td>StEyX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CountBlank</td>
<td>Growth</td>
<td>Npv</td>
<td>Substitute</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CountIf</td>
<td>HarMean</td>
<td>Odd</td>
<td>Subtotal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Covar</td>
<td>Hlookup</td>
<td>Or</td>
<td>Sum</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CritBinom</td>
<td>HypGeomDist</td>
<td>Pearson</td>
<td>SumIf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DAverage</td>
<td>Index</td>
<td>Percentile</td>
<td>SumProduct</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Days360</td>
<td>Intercept</td>
<td>PercentRank</td>
<td>SumSq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Db</td>
<td>Ipmt</td>
<td>Permut</td>
<td>Sum2MY2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCOUNT</td>
<td>Irr</td>
<td>Phonetic</td>
<td>Sum2PY2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCOUNTA</td>
<td>IsErr</td>
<td>Pi</td>
<td>SumXMY2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DDb</td>
<td>IsError</td>
<td>Pmt</td>
<td>Syd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Degrees</td>
<td>IsLogical</td>
<td>Poisson</td>
<td>Tanh</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEVsq</td>
<td>IsNA</td>
<td>Power</td>
<td>TDist</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DGET</td>
<td>IsNonText</td>
<td>Ppmt</td>
<td>Text</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DMAX</td>
<td>IsNumber</td>
<td>Prob</td>
<td>TInv</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DMIN</td>
<td>Ispt</td>
<td>Product</td>
<td>Transpose</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Using Worksheet Functions in VBA Exercise 1: Weighted Average Price

In a normal average, each data point contributes equally to the average. With a weighted average, a data point point can be given more importance than the others by weighting it.

For example, the image to the right shows the different prices we have paid for coffee and the number of units we purchased at that price. When getting the average price, we would like to take into account the number of units purchased at each given price.

Mathematically, you would get this by:
- a.) Multiplying each price by the number of units sold at that price (Price Per Unit * Units Sold).
- b.) Summing the results from step a.
- c.) Summing the Units Sold
- d.) Dividing step b by step c. (i.e. Sum of the Product of Price Per Unit * Units Sold / Sum of Units Sold).

One way to do this in Excel is to type:

\[ \text{=SumProduct(C2:C5, D2:D5) / Sum(D2:D5)} \]

We will make this a little shorter by incorporating SumProduct and Sum into a custom function.

Creating a Weighted Average Custom Function

1. Press ALT + F11 to toggle to the VBA Editor.
2. Within a module, type the code shown below.

```vba
Function WtAvg(Item As Range, Weight As Range)
    WtAvg = WorksheetFunction.SumProduct(Item, Weight) / WorksheetFunction.Sum(Weight)
End Function
```

Testing the Function

1. Press ALT + F11 to toggle to Excel.
2. Go to the "Weighted_Average" sheet.
3. In C7 type: =WtAvg(C2:C5, D2:D5)

You should get around 25.6
Worksheet Function Exercise 2: Employee Name with the Highest Sales

In this example, we wish to return the name of the employee with the highest sales. You could use Vlookup() for this but it would require that our sales be to the right of the employee names and that is not how our table is structured. This example shows how to look for something that could be on either side of what you wish to return. This will work as illustrated below.

Finding the Maximum Sales: Max()
Finding the highest sales figure is easy enough using the Max() function. (Try this in the Sales sheet.)

= Max(D2:D12)  returns 28,000.

Returning Maximum Sales Row Number: Match()
For reasons that are explained below, we also need to know which row of our block contains the maximum sales. The Match() function returns an item's position within a block.

=Match(What to find, Block to Search, 0 for exact match)

To return the vertical position of 28000 in the block D2:D12, in the Sales sheet type:

=Match(28000,D2:D12,0)  returns 8

And to return the position dynamically, we must nest Max() within Match():

=Match(Max(D2:D12),D2:D12,0)  returns 8

Returning the Name: Index()
Now we need to return the name. For this we can use the Index() function. If you give Index() a block and specify an X and Y coordinate within that block, it will return the contents at the intersection of the two coordinates. Its syntax is:

=Index(Block of Data, Row#, Column#)

We know the names are in the second column and 28000 is in the 8th row so, to locate Audrey you would type:

=Index(A2:D12,8,2)  returns Audrey

Finally, to make this dynamic, we must type the formula below.

=INDEX(A2:D12,MATCH(MAX(D2:D12),D2:D12,0),2)
Creating the Function

Now that we know how this works in Excel, we can shorten it using a custom function. However, we will make one more change. We want the function to give the user the choice of returning the Name, Employee ID, or the date so we will add a third argument to our custom function to all for that.

1. Press **ALT + F11** to access the VBA Editor.
2. Within a module, type the function below.
   
   Note that I like to work in a more modular way so I'm breaking the different functions into variables.

---

**Function**  
*MostSales*(Block As Range, SalesCol As Range, ReturnCol As Single)

'This returns the largest sales figure.

Dim MS As Double  
MS = worksheetfunction.Max(SalesCol)

'This returns the vertical position of the largest sales figure.

Dim MSR As Single  
MSR = worksheetfunction.Match(MS, SalesCol, 0)

'This returns the name of the person with the largest sales figure.

MostSales = worksheetfunction.Index(Block, MSR, ReturnCol)

End Function

---

Testing the Function

1. Press **ALT + F8** to access Excel.
2. Click the "Sales" sheet.
3. Type the following custom function:

   =MostSales(A2:D6,D2:D6,2)

Note that if you would like to return the Emp ID, change the 2 to a 1. If you would like the date, change the 2 to a 3.
Process Each Item in a Range:  For Each…Next Loop

This structure is very useful when you don’t know the number of cells in a range and you wish to loop through every cell in a range just once. It basically states: For each item in the selected items, do something. Once each item in the selected items are evaluated, the loop terminates automatically. The selected items can be cells, worksheets, etc.

<table>
<thead>
<tr>
<th>For Each element In selection</th>
<th>Line of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exit For [optional]</td>
<td>Line of code</td>
</tr>
<tr>
<td>Next element</td>
<td>Line of code</td>
</tr>
</tbody>
</table>

**For Each ... Next Loop Exercise 1: Summations**

This function will raise each value in any range to a given power (or root) and then sum the results.

For example: \(6^2 + 24^2 + 13^2 = 781\)

1. Within a module, create the code shown below.

```vba
Function PowerSum(Rng As Range, Power As Double)
    Dim X As Double
    X = 0
    For Each Item In Rng
        X = X + Item ^ Power
    Next Item
    PowerSum = X
End Function
```

**Testing the Function**

1. Input a range of numbers (horizontal, vertical, or in a block).
2. Use our custom PowerSum(Rng,Power) function to raise each number in the range to a given power.

- The example shown raises each number in the range A1:A8 to the second power and then sums up those results:
  - \(\text{=PowerSum(A1:A8,2)}\)
- To sum raised to the 3rd power: \(\text{=PowerSum(A1:A8,3)}\)
- To sum the cube roots: \(\text{=PowerSum(A1:A8,1/4)}\)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
For Each...Next Loop Exercise 2:  Number of Items Above Average

We wish to count the number of values in a range that are greater than the average of the values in the range. We will do this by using the following tools:

- The Average() function to determine the average of the range.
- The For Each...Next loop to test the value of every cell in the range against the average.
- An IF Then statement to only count a cell if its value is above the average.

Function AboveAvgCt(Rng As Range)

```vba
Dim Avg As Double
Dim X As Double
Dim Item As Object

Avg = worksheetfunction.Average(Rng)
X = 0

For Each Item In Rng
    If Item > Avg Then
        X = X + 1
    End If
Next Item

AboveAvgCt = X
```

Testing the Function

1. Input a range of numbers (horizontal, vertical, or in a block).
2. Use our custom AboveAvgCt() function, click in a cell and type it. For example:

   =AboveAvgCt(A1:A8)

   Should return 4

Percent of Movie Profits Above Average

If you want the percent of items above average, alter the second to the last line of the function as shown. .Count counts the number of cell in a range. In English, this reads: divide the number of cells above average by the total number of cells in the range.

```
AboveAvgCt = X / Rng.Count
```
Prompting the User for Values with Input Boxes

In previous examples, we have collected data needed by a function by asking the user's input through the function's arguments. For example, in the Ratio function the user had to supply the value of E, P, & N: \( \text{Ratio}(E, P, N) \)

In this section we will cover how to get these arguments by prompting the user with a pop-up window.

The syntax for Excel's VBA Inbox function is shown below. Items in [ ] are optional

\[
\text{InputBox}(\text{prompt}[, \text{title}] [, \text{default}] [, \text{xpos}] [, \text{ypos}] [, \text{helpfile}, \text{context}])
\]

Typically, users only supply the first two items so the here it is again with more descriptive arguments:

\[
\text{InputBox}("\text{Question to ask the user"},"\text{Text you want to appear on the titlebar of pop-up window")}
\]

1. Within a module, create the code shown below.

```
Function AdjScore()
    Dim E As Integer
    Dim P As Integer
    Dim N As Integer

    E = InputBox("Type the unadjusted points the student received", "Points Earned")
    P = InputBox("How many points were possible on the original test?", "Original Points Possible")
    N = InputBox("How many points will be possible on the new scale?", "New Points Possible")

    AdjScore = E / P * N
End Function
```

**Testing the Function**

In a cell on any sheet, type the following: \( =\text{AdjScore}() \) and press enter.

a. Type score received and press enter.

b. Type the number of points that were possible on the original test.

c. Type the number of points possible for the new scale.

Excel should return the adjusted points.
Informing the User with a Message Box

In the previous example, we used a pop-up box to ask the user for input. In this example, we will cover how to use VBA's MessageBox function to tell the user something.

The Syntax is shown below. Items in [ ] are optional.

\[
\text{MsgBox(prompt[, buttons] [, title] [, helpfile, context])}
\]

1. Add the code in bold below to the function we did on the previous page.

```
Function AdjScore()

    Dim E As Integer
    Dim P As Integer
    Dim N As Integer

    E = InputBox("Type the unadjusted points the student received", "Points Earned")
    P = InputBox("How many points were possible on the original test?", "Original Points Possible")
    N = InputBox("How many points will be possible on the new scale?", "New Points Possible")

    AdjScore = E / P * N

    MsgBox("The adjusted score is: " & AdjScore)

End Function
```
Working with a Specific Cell Address

The purpose of this section is to address how to make a function work with spreadsheet data that exists in a set location. In other words, rather than creating a function with an argument that allows the end user to specify which cell they would like to work with, these functions will always work specific cells. This is useful when the structure of the data you are working with is very static such as with a database.

**Range("range") - Current Sheet**

One method of naming a specific location in a custom function or sub is to use the "Range()" object. For example, to set the variable X equal to the contents of cell A1 on the current sheet you would use:

```
X = Range("A1")
```

**Worksheets("name").Range("range") - Specific Sheet**

To set a variable equal to the contents of a cell on a specific sheet, you must use the "Worksheets()" object. For example, to set variable X equal to what is in cell A1 on Sheet1, you would type:

```
X = Worksheets("Sheet1").Range("A1")
```

Try this in the Immediate Window: `?Msgbox(Worksheets("Canines").Range("A1"))`

---

**Range() Function Exercise 1: Determine the Interest for the Period**

This example is probably not the most useful but I chose it because it illustrates two techniques that I want to cover in a very simple way:

- How to Reference Specific Cells in a Custom Function (or sub procedure).
- How to Update Non-Volatile Functions.

1. Click the "Int" sheet to see the sheet to the right.
   Cell B1 contains a loan amount and cell B2 contains the yearly interest rate for a loan that compounds monthly. We would like to compute the interest for a single period.

   The formula is: **Balance * Interest Rate / 12**

**Creating the Function**

1. In a module, create the function shown below.

   ```vba
   Function PeriodInt()
   PeriodInt = Worksheets("Int").Range("B1") * Worksheets("Int").Range("B2") / 12
   End Function
   ```

**Testing the Function**

Although you could type the function on any sheet in the workbook, let’s type it in cell B3 on the "Int" sheet.

1. In a blank cell, type: `=PeriodInt()` and press enter.
2. Now, change the Loan Amount to 50,000 and press enter. **Note that the Period interest did not update!**
3. Press **Control + ALT + F9** to update it and see the next page.
Updating Non-Volatile Custom Functions: CONTROL + ALT + F9

In the example on the previous page, when we change the Loan Amount or Nominal Rate, the Period Interest does not update!

This is because the formula in cell B3 PeriodInt() does not contain direct references to B1 or B2. Those references are buried within the PeriodInt() function itself (see previous page).

When you change the contents of a cell, Excel normally only checks the formulas in the workbook to see if they contain references to the cells you change. If they do, Excel updates the formula but if they don't Excel does not update the formula.

To update our Period Interest we have two choices:

- To make Excel recalculate everything, you must press CONTROL + ALT + F9
  You will need to remember to press this every time you change B1 or B2.
- Within the function, type the Code: Application.volatile True
  This will force a recalculation of the function every time any cell in the workbook changes.

---

**Function PeriodInt()**

Application.Volatile True

PeriodInt = Worksheets("Int").Range("B1") * Worksheets("Int").Range("B2") / 12

End Function

---

1. Return to the Int sheet and change either the loan amount or the rate.
   Excel should now update the period interest.
Range Exercise 2: Working with a Table of Data

In a previous example we created a function that calculated an employee's commission based on their sales and Employee ID similar to the function shown to the right. This is fine if you have just a few employees but managing this function could get cumbersome if you had hundreds of employees, wanted to track other information, or use that other information for more calculations. A better method would be to store the information in a database and use Vlookup().

Vlookup() and Saving as an Excel Add-in

Our goal is to be able to provide an employee's EmpID and Sales and have the custom function return their Commission:

\[ =\text{EmpCom}(\text{Sales},\text{EmpID}) \]

To accomplish this, we will need to do the following:

- Provide all of the employee information in a table on a worksheet.
- Create a custom function that utilizes Vlookup() to pluck information from the table.
- Save our VBA as an Excel Add-in

How Vlookup() Works

Vlookup() is a worksheet function that we will use in our VBA.

1. Click the "EmployeesTbl" sheet.
2. Try the function below in a blank cell on the EmployeesTbl sheet.

You should get 15%.

The Vlookup() function below returns the commission rate of Kraig Moore. If you haven’t used Vlookup() before, it works as follows:

\[ =\text{Vlookup}(22,\text{A2:F5},5,\text{False}) \]

This is what to look for. In this case, Kraig’s EmpID is 22. What you are looking for must be the first column of the range given in the second function argument.

This is the range that specifies where Vlookup searches. What you are looking for (EmpID) and what you want to return (CommRate) must be within the range and what you are looking for must be to the left of what you want to return.

This is the column number of what you want to return. (CommRate is in the 5th column of the given range.)

False means you are looking for an exact match.
Creating the Custom Function

1. Press ALT + F11 to go to the VBA editor.
2. Create the code shown below.

**Function EmpCom(Sales As Double, EmpID As Double)**

**Dim EmpRate As Double**

EmpRate = WorksheetFunction.VLookup(EmpID, Worksheets("EmployeesTbl"), Range("A2:F5"), 5, False)

EmpCom = Sales * EmpRate

End Function

*EmpCom* is the name of our function and is used like a variable to return an answer in the worksheet. *Sales* is one of the function’s arguments and is supplied by the user when they type the function. *EmpRate* is a variable representing the rate Vlookup() returns.

**EmpRate** = WorksheetFunction.VLookup(EmpID, Worksheets("EmployeesTbl"), Range("A2:F5"), 5, False)

To use a worksheet function within VBA, you must precede it with the WorksheetFunction. Object.

This specifies which sheet the range A2:F5 should be taken from. If absent, Excel assumes it is the same sheet you are typing your custom function on.

This is our Vlookup() worksheet function. Its explanation is on the previous page. Its four parts are:

Vlookup(What to find, Where to Look, Column # of what to return, False for exact match)

Note that we could have done most of this on one line and not use the EmpRate variable but it is easier to follow using variables. Here it is on one line:

EmpCom = Sales * WorksheetFunction.VLookup(EmpID, Worksheets("EmployeesTbl"), Range("A2:F5"), 5, False)
Testing the Function

To test the function,

1. Go to the "Sales" sheet.
2. In D2 type:
   =EmpCom(C2,A2)
3. Copy the formula down.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Emp ID</td>
<td>Date</td>
<td>Sales</td>
<td>Commission</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>5/8/2013</td>
<td>$1,000</td>
<td>=EmpCom(C2,A2)</td>
</tr>
<tr>
<td>3</td>
<td>22</td>
<td>5/8/2013</td>
<td>$100</td>
<td>$15</td>
</tr>
<tr>
<td>4</td>
<td>33</td>
<td>5/8/2013</td>
<td>$10,000</td>
<td>$3,500</td>
</tr>
<tr>
<td>5</td>
<td>44</td>
<td>5/8/2013</td>
<td>$2,000</td>
<td>$200</td>
</tr>
<tr>
<td>6</td>
<td>22</td>
<td>5/9/2013</td>
<td>$25,000</td>
<td>$3,750</td>
</tr>
</tbody>
</table>

Note that this function only works within the "Custom_Functions.xlsm" file. See the section on saving as an add-in to make it available to all Excel files.

Using the EmpCom() Function in Other Excel Files as an Add-In

The next section covers how to use a custom function within another Excel file by saving it as an Add-in. If you wish to save the EmpCom() function as an Add-In, you will first need to make a change to it. Because the EmployeesTbl sheet exists in a file other than the file we will be typing the EmpCom() function, we must specify which file the sheet is found in. Further, because we will be saving the file as an Add-In, we must also change its extension. Excel macro enabled files end in .XLSM and Excel Add-Ins end in .XLAM Therefore, right before we save our file as an add-in, we must edit its code.

To point to a specific workbook, we must use the Workbooks() object. The EmpRate line of our code from the previous page needs to be edited as follows to include the Workbooks() object:

EmpRate = Workbooks("Custom_Functions.xlsm"). WorksheetFunction.VLookup(EmpID, Worksheets("EmployeesTbl"). Range("A2:F5"), 5, False)

We just need to be sure that when we save the Excel file as an Add-in, we save it as "Custom_Functions.xlsm".
Using Macros and Custom Functions in Other Files

Macros and custom functions are saved within the module of the workbook they were created with and will work on any sheet of that workbook. They will not work in other workbooks unless you take one of the actions listed below. They are listed from the least to most widely used methods:

- **Specify Workbook Name Method (Functions Only)** – Specifying the name of the workbook containing the function before the name of the function (separated by an exclamation mark) allows you to use the function but the workbook containing the module must be open.
- **Import Module Method** – Importing the module containing your code into the workbook you wish to use it gives you access to all macros and functions in the module.
- **Add-in Method** – Saving your workbook as an Excel Add-in and then loading the Add-in allows any Excel file to use the functions and macros within the module.

**Specifying the Workbook Name Method (Functions Only)**
This is probably the least desirable method of using a custom function in a workbook it was not created within. It requires that:

- The workbook containing the custom functions to be open. If it is closed, the function will return a #name error.
- You must specify the name of the workbook before the name of the function with an exclamation mark between them: `workbookname.xlsm!FunctionName()`

**Exercise:**
To use a custom function called `Area(L,W)` that exists in a file called `MyFunctions.xlsm` in any other file:

a. Open the file called "MyFunctions.xlsm".

b. Open the file you wish to use the custom function in.

c. In a cell in the file that doesn't have the module, type: `=MyFunctions.xlsm!Area(10,20)`

If you close `MyFunctions`, the function will return an error the next time you edit it.

**Exporting / Importing a Module Method**
If you import a module containing custom functions and sub procedures into a workbook, they will be available from within the workbook as if you had created them there.

**Exporting the Module**
1. Open the Excel file containing the module you wish to use in another file.
2. Press `ALT + F11` to access the VBA Editor.
3. Within the Project Explorer (Control + R), expand the modules folder so you see the module(s) you wish to export.
4. Right click the module you wish to export and select "Export File…".
5. Go with the suggested name and click "Save".

**Importing the Module**
1. Open the Excel file you would like import the module into.
2. Press `ALT + F11` to access the VBA Editor.
3. In the VBA editor, click "File – Import File".
4. Select the file you saved in step 5 above and click "OK".

Excel should have created a Modules folder for you (if necessary) and the module you imported should be within it. You may now use any macros or custom functions in the module in the Excel file you imported it into.
Exercise: Saving Your Custom Functions & Macros as an Add-in

If you make the Excel file containing your macros, sub procedures, and custom functions an Excel Add-in, they will be available to any file you open within Excel. Utilizing add-ins requires two steps:

a. Saving the macro file as an Add-in
b. Telling Excel to load the Add-in upon Start-up

Step a) Saving the File as an Excel Add-in

The steps in this section will save your Excel file and VBA code as an Add-in.

1. Open the Excel file containing your VBA code.
2. From within Excel, click "File – Save As".

   ![Image of Save As dialog box](image)

   3. Specify a name for the Add-in.
   4. At "Save as type", select "Excel Add-in (*.xlam)".
   5. Click "Save".

Excel will specify the location where Add-ins should be saved for you when you save as an Add-in.

Step b) Loading the Add-in

The steps in this section will instruct Excel to load the Add-in into memory every time Excel starts thus making the add-in's macros and functions available to any Excel file you open.

1. Start Excel.
2. From the menu, click "File – Options".
3. Click "Add-Ins" from the left side of the window.
4. At the bottom of the Add-ins window, make sure "Manage" is set to "Excel Add-ins" then click "Go".

   ![Image of Add-Ins dialog box](image)

If you saved your Add-In in the location suggested by Excel, then it should be on your list but unchecked. (If not present, try locating it with the Browse button.)

5. Check the name of your Add-in from the list.
6. Click "OK".

The Add-Ins macros and custom functions should now be available to any file you open in Excel.

Note that you can still view the Add-In's VBA from the Editor but its worksheets will not be visible.
Take Home Exercise 1: Tiered Tax Structure using Select Case & If Then

In this exercise, we wish to create a function that returns the tax given an adjusted gross income and filing status. It will have two arguments: $Taxes2013(N, F)$

where "N" is the adjusted gross income and "F" is the filing status (1 for Single, 2 for Married Jointly, 3 for married Separately, and 4 for Head of Household). Before we jump into the function, a quick primer on how the tiered tax structure works for individuals in the US:

Rates
There are seven possible rates applied to the various tiers:
10%  15%  25%  28%  33%  35%  39.6%

Taxes Paid are Calculated in Tiers
Your adjusted gross income is not taxed at a single rate. It is a sliding scale. For example, if you are single and your adjusted gross income was 80,000, then your first $8,925 would be taxed at 10%, then the next $27,325 would be taxed at 15%, and your last $43,750 would be taxed at 28% for a total tax of $15,928.75. The table below illustrates this.

<table>
<thead>
<tr>
<th>Tier</th>
<th>Income Bracket</th>
<th>Calculations for Amount Taxed</th>
<th>Amount Taxed at this Rate</th>
<th>Rate</th>
<th>Total Taxes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tier 1</td>
<td>0 to $8,925</td>
<td>$8,925</td>
<td>$8,925</td>
<td>10%</td>
<td>$892.5</td>
</tr>
<tr>
<td>Tier 2</td>
<td>$8,926 to $36,250</td>
<td>$36,250 - $8,925 = $27,325</td>
<td>$27,325</td>
<td>15%</td>
<td>$4,098.75</td>
</tr>
<tr>
<td>Tier 3</td>
<td>$36,251 to $87,850</td>
<td>$80,000 - $36,250 = $43,750</td>
<td>$43,750</td>
<td>28%</td>
<td>$10,937.5</td>
</tr>
</tbody>
</table>

The formula for someone with an adjusted gross income of 80,000 would be:
(Cap1 * Rate for Cap1) + (Cap2 * Rate for Cap2) + (Adj Gross Income – Cap2) * Rate for Cap3

Tax Brackets Vary between the four Filing Statuses
While the tax rates don’t change for the seven different tiers, the boundaries of the tiers do depend upon your filing status: Single, Married Jointly, Married Separately, or Head of Household. The table below shows the breakdown.

<table>
<thead>
<tr>
<th>Tier</th>
<th>Tax Rate</th>
<th>Single</th>
<th>Married Filing Jointly or Qualified Widow(er)</th>
<th>Married Filing Separately</th>
<th>Head of Household</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tier 1</td>
<td>10%</td>
<td>$0 – $8,925</td>
<td>$0 – $17,850</td>
<td>$0 – $8,925</td>
<td>$0 – $12,750</td>
</tr>
<tr>
<td>Tier 2</td>
<td>15%</td>
<td>$8,926 – $36,250</td>
<td>$17,851 – $72,500</td>
<td>$8,926 – $36,250</td>
<td>$12,751 – $48,600</td>
</tr>
<tr>
<td>Tier 3</td>
<td>25%</td>
<td>$36,251 – $87,850</td>
<td>$72,501 – $146,400</td>
<td>$36,251 – $73,200</td>
<td>$48,601 – $125,450</td>
</tr>
<tr>
<td>Tier 4</td>
<td>28%</td>
<td>$87,851 – $183,250</td>
<td>$146,401 – $223,050</td>
<td>$73,201 – $111,525</td>
<td>$125,451 – $203,150</td>
</tr>
<tr>
<td>Tier 5</td>
<td>33%</td>
<td>$183,251 – $398,350</td>
<td>$223,051 – $398,350</td>
<td>$111,526 – $199,175</td>
<td>$203,151 – $398,350</td>
</tr>
<tr>
<td>Tier 6</td>
<td>35%</td>
<td>$398,351 – $400,000</td>
<td>$398,351 – $450,000</td>
<td>$199,176 – $225,000</td>
<td>$398,351 – $425,000</td>
</tr>
<tr>
<td>Tier 7</td>
<td>39.60%</td>
<td>$400,001+</td>
<td>$450,001+</td>
<td>$225,001+</td>
<td>$425,001</td>
</tr>
</tbody>
</table>

For our purposes, we are more interested in the cap for each filing status as summarized in the table below.

<table>
<thead>
<tr>
<th>Rate</th>
<th>Single</th>
<th>Married - Jointly</th>
<th>Married - Separately</th>
<th>Head of Household</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>$8,925</td>
<td>$17,850</td>
<td>$8,925</td>
<td>$12,750</td>
</tr>
<tr>
<td>15%</td>
<td>$36,250</td>
<td>$72,500</td>
<td>$36,250</td>
<td>$48,600</td>
</tr>
<tr>
<td>25%</td>
<td>$87,850</td>
<td>$146,400</td>
<td>$73,200</td>
<td>$125,450</td>
</tr>
<tr>
<td>28%</td>
<td>$183,250</td>
<td>$223,050</td>
<td>$111,525</td>
<td>$203,150</td>
</tr>
<tr>
<td>33%</td>
<td>$398,350</td>
<td>$298,350</td>
<td>$199,175</td>
<td>$398,350</td>
</tr>
<tr>
<td>35%</td>
<td>$400,000</td>
<td>$450,000</td>
<td>$225,000</td>
<td>$425,000</td>
</tr>
<tr>
<td>39.60%</td>
<td>No Cap</td>
<td>No Cap</td>
<td>No Cap</td>
<td>No Cap</td>
</tr>
</tbody>
</table>

Note that this exercise is for VBA learning purposes only and is not intended for use outside of the classroom.
**Formula to get Taxes**

There are several ways we could have coded this. I took the approach outlined below. Our basic logic is:

\[
\text{Taxes} = \text{Cumulative Taxes for all Tiers below Highest Tier Reached} + \text{Rate of Highest Tier Reached} \times \text{Amount Left Over within Highest Tier}
\]

**Variables & Constants Used**

- **T1Cap…T6Cap**: These variables represent each tax tier's upper limit.
- **T1CmTax…T6CmTax**: These variables represent the cumulative taxes for each tier.
- **T1Rate…T5Rate**: These constants represent the tax rate for each tier.

In this step, the user will be typing the function `Taxes2013(N,F)` in the spreadsheet. For example: `=Taxes2013(80000,1)`

**User Provides Adjusted Gross Income (N) and their Filing Status (F).**

**Determine Cap Set (SELECT CASE)**

Each Filing Status (F) has its own tiered structure. This uses SELECT CASE to determine which cap set to use.

**Calculate Cumulative Taxes**

This calculates the cumulative taxes for each tier (i.e. running total) for the cap set selected above.

**Calculate Taxes (IF THEN)**

An If Then structure looks at the user’s Gross Income (N) to decide which formula to use to calculate their taxes. The cap sets have already been decided as have the caps to use from above.
Creating the Function
The next few pages walk you through creating the function.

1. While in Excel, press \textasciitilde\textasciitilde + \textasciitilde\textasciitilde\textasciitilde\textasciitilde\textasciitilde\textasciitilde\textasciitilde\textasciitilde + F11 to access the VBA editor.
2. If necessary, insert a module (Insert – Module).
3. Create the function shown over the next few pages.

```vba
Function Taxes2013(N As Double, F As Byte)
    Code from the pages below goes here.
End Function
```

\textbf{Taxes 2013 - is the name of our function.}
\textbf{N - represents the Adjusted Gross Income.}
\textbf{F - represents the filing statuses of: 1, 2, 3, or 4.}

\textbf{All of the code from the next two pages goes here between the opening "Function" and closing "End Function".}

\begin{itemize}
    \item \textbf{Declaring the Constants for the Rates}
        Rather than using the actual rates in our formula, we will create \textbf{Constants} to represent them using the keyword "Const". This will make updating the percentages easier should the government change the rates in the future. For example, the 10\% rate for Tier1 will be represented by "T1Rate".

        \begin{verbatim}
        Const T1Rate As Double = 0.1
        Const T2Rate As Double = 0.15
        Const T3Rate As Double = 0.25
        Const T4Rate As Double = 0.28
        Const T5Rate As Double = 0.33
        Const T6Rate As Double = 0.35
        Const T7Rate As Double = 0.396
        \end{verbatim}

    \item \textbf{Declaring the Variables for the Caps (Optional)}
        Six of the seven tax tiers have caps but those caps will change depending upon the filing status supplied by the user. The variables to the right will represent the cap for whatever filing status the user provides.

        \begin{verbatim}
        Dim T1Cap As Double
        Dim T2Cap As Double
        Dim T3Cap As Double
        Dim T4Cap As Double
        Dim T5Cap As Double
        Dim T6Cap As Double
        \end{verbatim}

    \item \textbf{Declaring the Variables for the Cumulative Taxes (Optional)}
        To make our calculations shorter, we will calculate the cumulative taxes at each tax tier and use the variable to the right to represent them.

        \begin{verbatim}
        Dim T1CmTax As Double
        Dim T2CmTax As Double
        Dim T3CmTax As Double
        Dim T4CmTax As Double
        Dim T5CmTax As Double
        Dim T6CmTax As Double
        \end{verbatim}
\end{itemize}
**SPECIFYING WHICH SET OF TAX CAPS TO USE**

The Select Case statement on this page specifies which set of tax caps to use depending on which filing status the end user provided in the function. Note that it utilizes the optional Case Else clause that activates if none of the other conditions are true.

**Case where F = 1**

This specifies which set of tier caps to use if the user typed a 1 as the filing status (Single).

**Case where F = 2**

This specifies which set of tier caps to use if the user typed a 2 as the filing status (Married filing jointly). Note that it is only read if the if statement above it are false.

**Case where F = 3**

This specifies which set of tier caps to use if the user typed a 3 as the filing status (Married filing separately). Note that it is only read if the if statements above it are false.

**Case where F = 4**

This specifies which set of tier caps to use if the user typed a 4 as the filing status (Married filing separately). Note that it is only read if the if statements above it are false.

**Case Else**

If none of the other conditions are true, then this will give the end user a pop-up box telling them that they must type in a 1, 2, 3, or 4. Further, the function will stop processing because we also used the "Exit Function" command.

```
Select Case F
  'Filing Status of Single
  Case 1
  T1Cap = 8925
  T2Cap = 36250
  T3Cap = 87850
  T4Cap = 183250
  T5Cap = 398350
  T6Cap = 400000

  'Filing Status of Married Jointly
  Case 2
  T1Cap = 17850
  T2Cap = 72500
  T3Cap = 146400
  T4Cap = 223050
  T5Cap = 398350
  T6Cap = 450000

  'Filing Status of Married Separately
  Case 3
  T1Cap = 8925
  T2Cap = 36250
  T3Cap = 73200
  T4Cap = 111525
  T5Cap = 199175
  T6Cap = 225000

  'Filing Status of Head of Household
  Case 4
  T1Cap = 12750
  T2Cap = 48600
  T3Cap = 125450
  T4Cap = 203150
  T5Cap = 398350
  T6Cap = 425000

  Case Else
  MsgBox ("Your filing status must be: 1, 2, 3 or 4")
  Exit Function

End Select
```
'Cumulative Taxes for Each Bracket given filing status.

T1CmTax = (T1Cap * T1Rate)
T2CmTax = T1CmTax + (T2Cap - T1Cap) * T2Rate
T3CmTax = T2CmTax + (T3Cap - T2Cap) * T3Rate
T4CmTax = T3CmTax + (T4Cap - T3Cap) * T4Rate
T5CmTax = T4CmTax + (T5Cap - T4Cap) * T5Rate
T6CmTax = T5CmTax + (T6Cap - T5Cap) * T6Rate

Cumulative Taxes for Each Tier
This section contains a running balance for the total taxes paid at each tier. Its purpose is to make our calculations later easier. Each tier's cumulative tax is: Its own Cap * the Tier's Rate + Previous Cumulative Taxes. Note that it must go after our IF THEN statement because it needs to use the appropriate cap for the filing status provided. The Rates for each tier are the same (it's the ranges of the tiers that vary.)

If N <= T1Cap Then
    Taxes2013 = T1Rate * N

ElseIf N > T1Cap And N <= T2Cap Then
    Taxes2013 = T1CmTax + (N - T1Cap) * T2Rate

ElseIf N > T2Cap And N <= T3Cap Then
    Taxes2013 = T2CmTax + (N - T2Cap) * T3Rate

ElseIf N > T3Cap And N <= T4Cap Then
    Taxes2013 = T3CmTax + (N - T3Cap) * T4Rate

ElseIf N > T4Cap And N <= T5Cap Then
    Taxes2013 = T4CmTax + (N - T4Cap) * T5Rate

ElseIf N > T5Cap And N <= T6Cap Then
    Taxes2013 = T5CmTax + (N - T5Cap) * T6Rate

ElseIf N > T6Cap Then
    Taxes2013 = T6CmTax + (N - T6Cap) * T7Rate

End If

End Function

Returning the Taxes
This section returns the taxes paid. It first checks to see which tier a person's gross pay (N) falls within by checking the caps. (Our Select Case statement already determined which values to assign to our caps variables.). Once the proper tier is found, the math is fairly simple:

Cumulative Taxes for all tiers below the current tier + Rate for current tier times remaining gross amount in this tier.

N - This is supplied by the user and is the adjusted gross pay.

T1Cap, T2Cap, etc.
These are the upper/lower ranges for each tier. They vary depending upon the filing status supplied by the user.

T1Cumtax, T2CumTax, etc.
This is the cumulative tax at each tier. Because the tiers can shift depending upon filing status, a Select Case statement was first used to determine where the caps are prior to determining the cumulative tax for each bracket.
**Testing the Function**

1. Press **ALT + F11** to return to Excel.
2. Go to the "**Taxes**" sheet.
3. In E2, type:  
   ![Excel screenshot showing a table with columns for Emp ID, Name, Filing Status, Income, and Taxes. Data includes entries for Rocky, Susan, Kraig, Lettie, and Karel with calculated taxes.]  

   - Rocky: **Income** = 8000
   - Susan: **Income** = 20000
   - Kraig: **Income** = 80000
   - Lettie: **Income** = 100000
   - Karel: **Income** = 500000

   ![Excel screenshot showing the formula **=Taxes2013(D2,C2)** applied to E2, calculating the tax amount.]
Take Home Exercise 2: Select Case & IF (Vacation Rate Based on Hire Date)

1. Click the VacationBal sheet.

In this exercise it is your job to track your employee's vacation balance from month to month. The basic formula you will use is:

\[
\text{Previous Month's Ending Balance} - \text{Current Months' Vacation Hours Used} + \text{New Vacation Hours Accrued}
\]

Calculating New Vacation Hours Accrued

Vacation Hours Accrued for the month is based on how long the employee has been employed as the table below shows.

<table>
<thead>
<tr>
<th>Duration At Company</th>
<th>Vacation Hours Accrued Per Month</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 2 years</td>
<td>8 Hours Per Month</td>
</tr>
<tr>
<td>&gt;= 2 and &lt; 5 Years</td>
<td>16 Hours Per Month</td>
</tr>
<tr>
<td>&gt;= 5 and &lt; 10 Years</td>
<td>24 Hours Per Month</td>
</tr>
<tr>
<td>10 Years and Over</td>
<td>30 Hours Per Month</td>
</tr>
</tbody>
</table>

Calculating Duration

This table shows the hire date for each employee:

<table>
<thead>
<tr>
<th>Employee ID</th>
<th>Hire Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>556</td>
<td>5/8/1999</td>
</tr>
<tr>
<td>636</td>
<td>12/15/2007</td>
</tr>
<tr>
<td>552</td>
<td>6/10/2008</td>
</tr>
<tr>
<td>551</td>
<td>5/4/2012</td>
</tr>
<tr>
<td>365</td>
<td>4/3/2013</td>
</tr>
</tbody>
</table>

Because vacation hours are based how long an employee has been employed by our company, we need to make sure we always subtract today's date from their hire date to get their duration:

\[
\text{Duration} = \text{Today's Date} - \text{Hire date}
\]

However, to maintain historic accuracy, we can't use Now() to represent today's date; we must use a specific, hard coded date for each month.

About the Custom Function

Based on the information above, our custom function will need to address the following:

- The Hire Date of each employee by Employee ID.
- Determining the duration of each employee's tenure based on today's date and their hire date.
- Vacation hours accrued based on duration.
- A formula to find the current balance.
Creating the Custom Function

1. Press ALT + F11 to access the VBA editor.
2. Within a module, create the function below.

```vba
Function VaBal(EmpID, EndDate, PrevBal, AmtUsed)

    'This is how many years each employee has been here.

    Yrs = (EndDate - #5/8/1999#) \ 365

    ElseIf EmpID = 636 Then
        Yrs = (EndDate - #12/15/2007#) \ 365
    EndElseIf

    ElseIf EmpID = 552 Then
        Yrs = (EndDate - #6/10/2008#) \ 365
    EndElseIf

    ElseIf EmpID = 365 Then
        Yrs = (EndDate - #4/3/2013#) \ 365
    EndElseIf

    Else
        MsgBox ("Employee ID Not Valid")
        Exit Function
    End Else

    'AccrualRate Calcs (vac. hrs Accrued per month)

    Select Case Yrs
        Case Is >= 10
            Accrued = 30
        Case Is >= 5
            Accrued = 24
        Case Is >= 2
            Accrued = 16
        Case Is < 2
            Accrued = 8
    End Select

End Function
```

This IF structure returns the total number of years worked for the employee whose EmpID meets the condition. The answer is stored in the Yrs variable and EmpID is supplied by the user.

ElseIf   Each ElseIf is only read if none of the conditions above them are true.

EndDate represents the current date and is provided by the user.

# Hard coded dates must be wrapped in # signs.

\ 365 When you subtract dates, Excel expresses the answer in days. Because we want it in years we must divide by 365. If you use a backslash to divide, Excel will return only whole numbers.

Else This section is only read if none of the other conditions are met. It addresses the possibility of the user supplying an Emp ID that we have not accounted for. If so, we wish to return an error. MsgBox gives them a pop-up message telling them the Employee ID is not valid.

Exit Function stops the function.

Select Case Yrs
The Select Case structure to the left determines the number of vacation hours accrued during the month. It is based on the number of years the employee has worked (which was returned by the IF structure above and is contained in the variable "Yrs").

For example, should the IF structure determine that the specified employee has worked 6 years then "Yrs" becomes "6". Then the Select Case structure checks to see if 6 is >=10. If it is, then the variable "Accrued" is assigned a value of 30; however, it is not so it then checks to see if 6>=5 which is true so "Accrued" becomes 24 and none of the other cases are read. (Select case reads from top down and stops reading as soon as one of the cases is true.)
VaBal = PrevBal - AmtUsed + Accrued

'Vacation Balance is capped at 48 days (384hrs).
If VaBal <= 384 Then
    VaBal = VaBal
Else
    VaBal = 384
End If

End Function

Testing the Function
1. Press ALT + F11 to return to Excel.
2. Click the "VacationBal" sheet.
3. In E3 type the following: =VaBal(A3,E$2,C3,D3) and press enter.
4. Copy the formula down.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>End Balance</td>
<td></td>
<td>End Balance</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Emp ID</td>
<td>Name</td>
<td>12/31/2013</td>
<td>Vac Used</td>
<td>1/30/2013</td>
<td>Vac Used</td>
</tr>
<tr>
<td>3</td>
<td>556</td>
<td>Rocky</td>
<td>300</td>
<td>24</td>
<td>=VaBal(A3,E$2,C3,D3)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>636</td>
<td>Susan</td>
<td>200</td>
<td>16</td>
<td></td>
<td>208</td>
</tr>
<tr>
<td>5</td>
<td>552</td>
<td>Kraig</td>
<td>100</td>
<td>12</td>
<td></td>
<td>104</td>
</tr>
<tr>
<td>6</td>
<td>551</td>
<td>Lettie</td>
<td>20</td>
<td>8</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td>365</td>
<td>Karel</td>
<td>10</td>
<td>0</td>
<td></td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

You should get the figures shown in the image. Note that Rocky should have 306.

300 Hours End Balance − 24 Hours Used + 30 Hours Accrued = 306 New Balance

Note that the $ sign in E$2 is to freeze E2 so it does not shift when we copy the formula down.
**VaBal() Function with Variables Declared**

Because assigning the wrong variable type can cause ambiguous errors, I typically write my functions first without declaring my variables and then go back and put them in later. Here is the same function with variables declared:

```vbnet
Function VaBal(EmpID As Single, EndDate As Date, PrevBal As Double, AmtUsed As Double)
    'This is how many years each employee has been here.
    Dim Yrs As Double

    If EmpID = 556 Then
        Yrs = (EndDate - #5/8/1999#) \ 365
    ElseIf EmpID = 636 Then
        Yrs = (EndDate - #12/15/2007#) \ 365
    ElseIf EmpID = 552 Then
        Yrs = (EndDate - #6/10/2008#) \ 365
    ElseIf EmpID = 551 Then
        Yrs = (EndDate - #5/4/2012#) \ 365
    Else
        MsgBox ("Employee ID Not Valid")
        Exit Function
    End If

    'AccrulRate Calcs (vac. hrs Accrued per month)
    Dim Accrued As Single

    Select Case Yrs
        Case Is >= 10
            Accrued = 30
        Case Is >= 5
            Accrued = 24
        Case Is >= 2
            Accrued = 16
        Case Is < 2
            Accrued = 8
    End Select

    VaBal = PrevBal - AmtUsed + Accrued

    'You cannot accumulate more than 48 days (384hrs) of unused vacation.
    If VaBal <= 384 Then
        VaBal = VaBal
    Else
        VaBal = 384
    End If

    End Function
```