Built in Functions in VBA

Objectives: At the end of this lesson you shall be able to
- **describe the math functions in VBA**
- **describe the logical functions in VBA**
- **describe the date/time functions in VBA**
- **describe the conversion functions in VBA.**

Introduction

VBA has a rich collection of built in functions that perform a variety of tasks and calculations for you. There are functions to convert data types, perform calculations on dates, perform simple to complex mathematics, make financial calculations, manage text strings, format values, and retrieve data from tables, among others. Using the VBA Built in Functions will help coding much easier for the user. We have already used many built in functions in our earlier lessons like the msgbox function and many string manipulation functions just to name a few.

MS Excel: VBA Functions (VBA Formulae) - Category wise

The commonly used VBA functions in Excel, sorted by Category are shown here.

String Functions: The string functions were already discussed in the related theory for Ex. 2.2.09

Math Functions

Table 1 : Lists some of the common Built in Functions in the Mathematical category.

Table 1

Function Name	Description
Abs	Returns the absolute value of a number.
Cos	Returns the cosine of the specified angle.
Cosh	Returns the hyperbolic cosine of the specified angle.
Exp	Returns e (the base of natural logarithms) raised to the specified power.
Fix	Returns the integer portion of a number.
Format	Takes a numeric expression and returns it as a formatted string.
Int	Returns the integer portion of a number.
Log	Returns the natural (base e) logarithm of a specified number or the logarithm of a specified number in a specified base.
Rnd	Generates a random number (integer value)
Round	Returns a Decimal or Double value rounded to the nearest integral value or to a specified number of fractional digits.
Sign	Returns an Integer value indicating the sign of a number.
Sin	Returns the sine of the specified angle.
Sqr	Returns the square root of a specified number.
Tan	Returns the tangent of the specified angle.
Val	Accepts a string as input and returns the numbers found in that string.

1 Dim a, b as integer

```
a=81

debug.print sqr(a)
```

This will display the square root of 81 ie. 9

Logical Functions

Table 2. lists some of the common Built in Functions in the Logical category.

Table 2

Function Name	Description
ISDATE	Returns TRUE if the expression is a valid date. Otherwise, it returns FALSE.
ISERROR	Checks for error values.
ISNULL	Returns TRUE if the expression is a null value. Otherwise, it returns FALSE.
ISNUMERIC	Returns TRUE if the expression is a valid number. Otherwise, it returns FALSE.

Examples:

1
```
Sub Button1_Click()
N = TextBox1.Text
If IsNumeric(N) = True Then
MsgBox "correct"
Else
MsgBox "Insert only numbers"
End If
```

This checks if the data entered in the textbox is a number or not.

2
```
Sub Button1_Click()
N = TextBox1.Text
If IsDate(N) = True Then
MsgBox "correct"
Else
MsgBox "Insert only dates"
End If
End Sub
```

This checks if the data entered in the textbox is a valid date or not.

Date / Time Functions

Table 3. lists some of the common Built in Functions in the Date / Time category.

Table 3

Function	Return Value
DATE	Returns the current system date.
DATEADD	Returns a date after which a certain time/date interval has been added.
DATEDIFF	Returns the difference between two date values, based on the interval specified.
DATEPART	Returns a specified part of a given date.
DATESERIAL	Returns a date given a year, month, and day value.
DATEVALUE	Returns the serial number of a date.
DAY	Returns the day of the month (a number from 1 to 31) given a date value.
FORMAT Dates	Takes a date expression and returns it as a formatted string.
HOUR	Returns the hour of a time value (from 0 to 23).
MINUTE	Returns the minute of a time value (from 0 to 59).
MONTH	Returns the month (a number from 1 to 12) given a date value.
MONTHNAME	Returns a string representing the month given a number from 1 to 12.
NOW	Returns the current system date and time.
TIMESERIAL	Returns a time given an hour, minute, and second value.
TIMEVALUE	Returns the serial number of a time.
WEEKDAY	Returns a number representing the day of the week, given a date value.
WEEKDAYNAME	Returns a string representing the day of the week given a number from 1 to 7.
YEAR	Returns the year portion of the date argument.

Examples

1 DateDiff() Function

Syntax for the DateDiff function is :

DateDiff (interval, date1, date2, [firstdayofweek], [firstweekofyear])

Parameters or Arguments

Interval is the interval of time to use to calculate the difference between date1 and date2. Below is a list of valid interval values as in Table 4

Table 4

Interval	Explanation
yyyy	Year
q	Quarter
m	Month
y	Day of year
d	Day
w	Weekday
ww	Week
h	Hour
n	Minute
s	Second

Date1 and Date2 are the two dates to calculate the difference between.

first day of week is optional. It is a constant that specifies the first day of the week. If this parameter is omitted, Excel assumes that Sunday is the first day of the week.

first week of year is optional. It is a constant that specifies the first week of the year. If this parameter is omitted, Excel assumes that the week containing Jan 1st is the first week of the year.

```
Sub test()

Debug.PrintDateDiff("yyyy", "1/12/1999", "31/1/2000")

Debug.PrintDateDiff("q", "1/12/1999", "31/1/2000")

Debug.PrintDateDiff("m", "1/12/1999", "31/1/2000")

End Sub
```

The result will be

```
1

4

12
```

2 Format Date

Syntax

The syntax for the Microsoft Excel FORMAT function is:

Format (expression, [format, [firstdayofweek, [firstweekofyear]]])

Parameters or Arguments

Expression is the value to format.

Format is optional. It is the format to apply to the expression. You can either define your own format or use one of the named formats that Excel has predefined such as shown in Table 5.

Table 5

Format	Explanation
General Date	Displays date based on your system settings
Long Date	Displays date based on your system's long date setting
Medium Date	Displays date based on your system's medium date setting
Short Date	Displays date based on your system's short date setting
Long Time	Displays time based on your system's long time setting
Medium Time	Displays time based on your system's medium time setting
Short Time	Displays time based on your system's short time setting

First day of week is optional. It is a value that specifies the first day of the week. If this parameter is omitted, the FORMAT function assumes that Sunday is the first day of the week. This parameter can be one of the following values as shown in Table 6.

First week of year is optional. It is a value that specifies the first week of the year. If this parameter is omitted, the FORMAT function assumes that the week that contains January 1 is the first week of the year. This parameter can be one of the following values as shown in Table 7.

```
Sub test()

Debug.Print Format(#1/1/1990#, "Short Date")

Debug.Print Format(#1/1/1990#, "Long Date")

Debug.Print Format(#1/1/1990#, "yyyy/mm/dd")

End Sub
```

Table 6

Constant	Value	Explanation
vbUseSystem	0	Uses the NLS API setting
VbSunday	1	Sunday (default, if parameter is omitted)
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

The result will be

1/1/1990

Table 7

Constant	Value	Explanation
vbUseSystem	0	Uses the NLS API setting
vbFirstJan1	1	The week that contains January 1
vbFirstFourDays	2	The first week that has at least 4 days in the year
vbFirstFullWeek	3	The first full week of the year

Monday, January 01, 1990

1990/01/01

Data Type Conversion Functions

Table 8. below lists some of the common Built in Functions in the Data Type Conversion category.

Table 8

Function	Return Type	Range for expression argument
CBool	Boolean	Any valid string or numeric expression.
CByte	Byte	0 to 255.
CCur	Currency	-922,337,203,685,477.5808 to 922,337,203,685,477.5807.
CDate	Date	Any valid date expression.
CDbl	Double	-1.79769313486231E308 to -4.94065645841247E-324 for negative values; 4.94065645841247E- 324 to 1.79769313486232E308 for positive values.
CDec	Decimal	+/-79,228,162,514,264,337,593,543,950,335 for zero-scaled numbers, that is, numbers with no decimal places. For numbers with 28 decimal places, the range is +/-7.9228162514264337593543950335. The smallest possible non-zero number is 0.0000000000000000000000000001.
CInt	Integer	-32,768 to 32,767; fractions are rounded.
CLng	Long	-2,147,483,648 to 2,147,483,647; fractions are rounded.
CSng	Single	-3.402823E38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values.
CStr	String	Returns for CStr depend on the expression argument.
CVar	Variant	Same range as Double for numerics. Same range as String for non-numerics.

Example

CDate function

```
Sub test()
Dim INum As Long
Dim a As String
a = 12345
Debug.PrintCDate(a)
b = "January 1, 1990"
Debug.PrintCDate(b)
c = "1:23:45 PM"
Debug.PrintCDate(c)
End Sub
```

This will display

```
10/18/1933
1/1/1990
1:23:45 PM
```

User defined functions in VBA

Objectives: At the end of this lesson you shall be able to
- **create user defined functions**
- **describe passing values to functions byval and byref**
- **describe using arrays with functions**
- **describe the scope of variables**
- **describe the access specifiers public and private.**

Introduction

In Excel Visual Basic too, like in most programming languages, a set of commands to perform a specific task is placed into a procedure, which can be a function or a subroutine. The main difference between a VBA function and a VBA subroutine is that a function (generally) returns a result, whereas a subroutine does not.

Therefore, if you wish to perform a task that returns a result (ex. summing of a group of numbers), you will generally use a function, but if you just need a set of actions to be carried out (ex. formatting a set of cells), you might choose to use a subroutine.

User Defined Functions

One of the most power features of Excel VBA is that you can create your own functions or UDFs. A UDF (User Defined Function) is simply a function that you create yourself with VBA for your own defined tasks. UDFs are often called "Custom Functions". A UDF can remain in a code module attached to a workbook, in which case it will always be available when that workbook is open. Alternatively you can create your own add-in containing one or more functions that you can install into Excel. Here the user-defined functions can be entered into any cell or on the formula bar of the spreadsheet just like entering the built-in formulas of the MS Excel spreadsheet.

Custom functions, like macros, use the Visual Basic for Applications (VBA) programming language. They differ from macros in two significant ways. First, they use function procedures instead of sub procedures. They start with a Function statement instead of a Sub statement and end with End Function instead of End Sub. Second, they perform calculations instead of taking actions. Certain kinds of statements (such as statements that select and format ranges) are generally excluded from custom functions.

A simple function may look like this:

```
Function area()

Dim l, b

l = 10

b = 20

Debug.Print "area Is " & l * b

End Function
```

When executed from the immediate window this function displays the area.

Alternately this function can be called by another subroutine, for ex.

```
Sub test_fn()

Call area

End Sub
```

Returning a value from the procedures

In the example given below, the area() function calculates l*b.

The subroutine that calls this function is returned this value.

```
Sub test_fn()

Debug.Print "The function has returned the value " & area

End Sub

Function area()

Dim l, b, A

l = 10

b = 20

area = l * b

End Function
```

The result will be:The function has returned the value 200

Passing Arguments to functions

We can pass the arguments in two different ways:

1 By Value (ByVal): We pass the copy of the actual value to the arguments

2 By Reference (ByRef): We pass the reference to the arguments

By Ref is the default method of passing argument type in VBA. This means, if you are not specifying any type of the argument it will consider it as ByRef type. However, it is always a good practice to specify the ByRef even if it is not mandatory.

The following example shows the method of passing variables to a function byVal.

```
Sub test_fn()

Dim a, b As Integer

a = 4

b = multiply(a)

Debug.Print "a is " & a

Debug.Print "The function has returned the value " & b

End Sub

Function multiply(ByVal a As Integer)

a = a * 10

multiply = a

End Function
```

The result of this program will be :

```
a is 4

The function has returned the value 40

a is 4
```

This means that the value of the variable that was passed is not disturbed by the function.

The following example shows the method of passing variables to a function byRef.

```
Sub Test()

Dim A As Integer

A = 10

Debug.Print "The function has returned the value " & Modify(A)

Debug.Print "A is now  " & A

End Sub

Function Modify(ByRef A As Integer)

A = A * 2

  Modify = A

End Function
```

The result will be:

```
The function has returned the value 20

A is now 20
```

Calling a User Defined Function from Worksheet:

You call the user defined functions as similar to the built-in excel function. To do this type the arguments in the cells and type the name of the function as is done with normal functions in Excel.

Passing Arrays to User Defined functions

A Function can accept an array as an input parameter. Arrays are always passed by reference (ByRef). You will receive a compiler error if you attempt to pass an array ByVal. This means that any modification that the called procedure does to the array parameter is done on the actual array declared in the calling procedure.

(If you need to pass an array ByVal then you would need to use the Variant data type.)

An example of passing an array to a function is as follows:

```
Sub test()

Dim arr(1 To 10) As Integer

Dim i As Integer

'populates the array with the values 1 to 10

For i = 1 To 10

arr(i) = i

Next i

'call the function example 1 with arrIntegers as an input parameter

Call fn1(arr)

For i = 1 To 10

Debug.Print arr(i); Spc(2);

Next i

End Sub
```

```
'prints the values in arrIntegers to column A

Sub fn1(ByRef arr() As Integer)

Dim i As Integer

For i = LBound(arr) To UBound(arr)

  arr (i) = arr (i) * 2

  Cells (i,1) = arr (i)

Next i

End Sub
```

Scope of variables

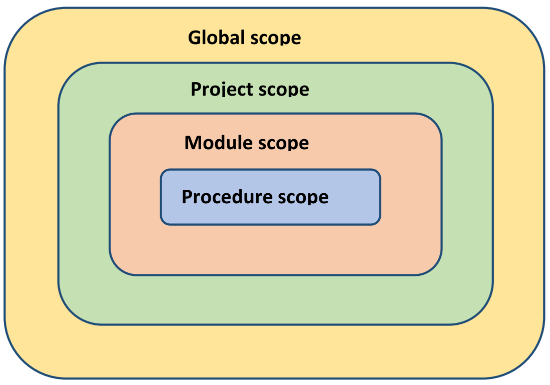
The term Scope is used to describe how a variable may be accessed. Depending on where and how a variable is declared, it may be accessible only to a single procedure, to all procedures within a module, and so on up the hierarchy of a project or group of related projects. The term visibilty is also is sometimes used to describe scope.

There are four levels of Scope:

- Procedure-Level Scope
- Module-Level Scope
- Project-Level Scope
- Global-Level Scope

Fig 1 shows the various scopes and their levels.

Fig 1



Procedure (local) scope

A local variable with procedure scope is recognized only within the procedure in which it is declared. A local variable can be declared with a Dim or Static statement.

When a local variable is declared with the Dim statement, the variable remains in existence only as long as the procedure in which it is declared is running. Usually, when the procedure is finished running, the values of the procedure's local variables are not preserved, and the memory allocated to those variables is released. The next time the procedure is executed, all of its local variables are reinitialized.

For Example the following subroutine has been created in Module1 code.

```
Sub disp()

Dim s As string

s="hello"

MsgBox s

End Sub
```

Run the subroutine "disp" in Module1 and it will display the message "Hello" in the message box.

Now the following subroutine has been created in Sheet1 code to call the disp() subroutine from Module1.

```
Sub Button1_Click()

disp

End Sub
```

This will generate an error since the subroutine disp() and the variable s are local to Module1 and cannot be accessed from elsewhere.

Static:

A local variable declared with the Static statement remains in existence the entire time Visual Basic is running. The variable is reset when any of the following occur:

- The macro generates an untrapped run-time error.
- Visual Basic is halted.
- You quit Microsoft Excel.
- You change the module.

For example, in the FindTotal example, the Accumulate variable retains its value every time it is executed. The first time the module is run, if you enter the number 2, the message box will display the value "2." The next time the module is run, if the value 3 is entered, the message box will display the running total value to be 5.

```
Sub FindTotal()

Static Total

Dim n as integer

n =InputBox("Enter a number: ")

Total = Total + n

MsgBox "The total is " &n

End Sub
```

Module scope

A variable that is recognized among all of the procedures on a module sheet is called a "module-level" variable. A module-level variable is available to all of the procedures in that module, but it is not available to procedures in other modules. A module-level variable remains in existence while Visual Basic is running until the module in which it is declared is edited. Module-level variables can be declared with a Dim or Private statement at the top of the module above the first procedure definition.

At the module level, there is no difference between Dim and Private. Note that module-level variables cannot be declared within a procedure.

Note If you use Private instead of Dim for module-level variables, your code may be easier to read (that is, if you use Dim for local variables only, and Private for module-level variables, the scope of a particular variable will be more clear).

In the following example, two variables, A and B, are declared at the module level. These two variables are available to any of the procedures on the module sheet. The third variable, C, which is declared in the Example3 macro, is a local variable and is only available to that procedure.

Note that in Test4, when the macro tries to use the variable C, the message box is empty. The message box is empty because C is a local variable and is not available to Test4, whereas variables A and B are.

```
Dim A As Integer       ' Module-level variable.

Private B As Integer   ' Module-level variable.

Sub Test1()

A = 10

B = A * 10

End Sub

Sub Test2()

MsgBox "The value of A is " & A

MsgBox "The value of B is " & B

End Sub

Sub Test3()

Dim C As Integer    ' Local variable.

C = A + B

MsgBox "The value of C is " & C

End Sub

Sub Test4()

MsgBox A

MsgBox B

MsgBox C

 ' The message box is blank since C is a local variable.

End Sub
```

Project Scope

Project scope variables are those declared using the Public keyword. These variables are accessible from any procedure in any module in the project. In Excel, a Project is all of the code modules, userforms, class modules, and object modules (e.g. ThisWorkbook and Sheet1) that are contained within a workbook.

In order to make a variable accessible from anywhere in the project, you must use the Public keyword in the declaration of the variable. However, this makes the variable accessible to any other project that reference the project containing the variable. If you want a variable to be accessible from anywhere within the project, but not accessible from another project, you need to use Option Private Module as the first line in the module (above and outside of any variable declaration or procedure). This option makes everything in the module accessible only from within the project. The project variables that should not be accessible to other projects should be declared in a module that has the Option Private Module directive. Variables that should be accessible to other project should be declared in a different module that does not use the Option Private Module directive. In both cases, however, you need to use the Public keyword.

Global Scope

Global scope variables are those that are accessible from anywhere in the project that declares them as well as any other project that references the first project. To declare a variable with global scope, you need to declare it using the Public keyword in a module that does not use the Option Private Module directive. In order to access variables in another project, you can simply use the variable's name. If, however, it is possible that the calling project also has a variable by the same name, you need to prefix the variable name with the project name. For example, if Project1 declares a global variable named x, and Project2 references Project1, code that is in Project2 can access x with either of the following lines of code:

```
x = 78

Project1.x = 78
```

If both Project1 and Project2 have variables with at least project scope, you need to include the project name with the variable. For clarity and maintainability, you should always include the project name when accessing a variable that is declared in another project. Even if this is not necessary, it makes the code more readable and maintainable.

There is no way to give some variables project, but not global, scope and give others in the same module global scope. Project versus global scope is handled only at the module level, not at the variable level.

The Access Specifiers

One of the techniques in object-oriented programming is encapsulation. It concerns the hiding of data in a class and making them available only through its methods. Most programming languages implementing OOPS allow you to control access to classes, methods, and fields via so-called access modifiers. The access to classes, constructors, methods and fields are regulated using access modifiers i.e. a class can control what information or data can be accessible by other classes. The VBA access specifiers are:

1 Private

2 Public

A Public procedure is accessible to all code inside the module and all code outside the module, essentially making it global. A VBA Private Sub can only be called from anywhere in the Module in which it resides. A Public Sub in the objects, ThisWorkbook, ThisDocument, Sheet1, etc. cannot be called from anywhere in a project. However, if you declare a Module level variable with the Public Keyword it can be used anywhere in the project and retains its value.

If you exclude the key word private in your declaration then by default the procedure is public. So Sub MySub() and Public Sub MySub() are exactly the same thing.

Public [variable] means that the variable can be accessed or used by subroutines in outside modules. These variables must be declared outside of a subroutine (usually at the very top of your module). You can use this type of variable when you have one subroutine generating a value and you want to pass that value on to another subroutine stored in a separate module.

A Private procedure is only available to the current module. It cannot be accessed from any other modules, or from the Excel workbook.Private Sub sets the scope so that subroutines from outside modules cannot call that particular subroutine. This means that a sub in Module 1 could not use the Call method to initiate a Private Sub in Module 2.

Private [variable] means that the variable cannot be accessed or used by subroutines in other modules. In order to be used, these variables must be declared outside of a subroutine (usually at the very top of your module). You can use this type of variable when you have one subroutine generating a value and you want to pass that value on to another subroutine in the same module.

Dim[variable] is used to state the scope inside of a subroutine (you cannot use Private in its place). Dim can be used either inside a subroutine or outside a subroutine (using it outside a subroutine would be the same as using Private).

Example of Public, Private Variables and Procedures.

Module 1 code

```
Dim x As Integer ' This is a Private Variable since it is declared using Dim.

Public y As Integer

Sub First_Sub()

  x = 10

  y = 20

  Call Third_Sub()

End Sub

Private Sub Second_Sub()

MsgBox "Gone through First, Second and Third Subroutines !"

End Sub
```

Module 2 code

```
Sub Third_Sub()

Debug.Print x

Debug.Print y

Call Second_Sub()

End Sub
```

The two variables x and y that are declared outside a subroutine. This means that their values can carry over into other macros. The variable x has a private scope so only subroutines in the same module can access its value. The variable y has a public scope, meaning that subroutines inside and outside its module can access its value.

The First_Sub() assigns values to x and y and then initiates the Third_Sub().

Third_Sub() can be called even though it is not in the same module because it is a Public Sub.

The Third_Sub() has been designed to display the values of x and y in the immediate window. When you try to print variable x it outputs nothing. This is because x does not exist in Module 2. Therefore, a new variable x is created in Module 2 and since we did not give this new x a value, nothing is printed for the statement Debug.Print x

When we print the value of the variable y, 12 is displayed in the Immediate Window. This is because Module 2 subroutines have access to the public variables declared in Module 1.But the statement "Call Second_Sub()" in the Third_Sub() will result in an error. This is because we are trying to call a private subroutine " Second_Sub" from here. The following changes can be done to avoid this:

1 We could remove the word "Private" from Display_Message

2 We could replace "Private" with "Public" in Second_Sub()

3 We can use the Application level and instead of using Call we could write Application.Run "Second_Sub " (this method serves as an override in case we wanted to keep Second_Sub private for subroutines outside the module.)

Create and Edit Macros

Objective: At the end of this lesson you shall be able t0
• **explain about Macros in VBA.**

Macros offer a powerful and flexible way to extend the features of Excel. They allow the automation of repetitive tasks such as printing, formatting, configuring, or otherwise manipulating data in Excel. In its' simplest form, a macro is a recording of your keystrokes. While macros represent one of the stronger features found in Excel, they are rather easy to create and use. There are six major points that I like to make about macros as follows.

1 **Record, Use Excel, Stop Recording**

To create a macro, simply turn on the macro recorder, use Excel as you normally do, then turn off the recorder. Presto – you have created a macro. While the process is simple from the user's point of view, underneath the covers Excel creates a Visual Basic subroutine using sophisticated Visual Basic programming commands.

2 **Macro Location**

Macros can be stored in either of two locations, as follows:

The workbook you are using, or the Personal Macro Workbook (which by default is hidden from view) If the macro applies to all workbooks, then store it in the Personal Macro Workbook so it will always be available in all of the Excel workbooks; otherwise store it in the current workbook. A macro stored in the current workbook will embedded and included in the workbook, even if you email the workbook to another user.

3 **Assign the Macro to an Icon, Text or a Button**

To make it easy to run your macro, you should assign it to a toolbar icon so it will always be available no matter which workbooks you have open. If the macro applies only to the current workbook, then assign it to Text or a macro Button so it will be quickly available in the current workbook.

4 **Absolute versus Relative Macros**

An "Absolute" macro will always affect the same cells each time whereas a "Relative" macro will affect those cells relative to where the cursor is positioned when invoke the macro. It is crucial that understand the difference.

5 **Editing Macros**

Once created, you can view and/or edit your macro using the View Macros option. This will open the macro subroutine in a Visual basic programming window and provide you with a plethora of VB tools.

6 **Advanced Visual Basic Programming**

For the truly ambitious CPA, in the Visual Basic Programming window, you have the necessary tools you need to build very sophisticated macros with dialog boxes, drop down menu options, check boxes, radio buttons – the whole works. To see all of this power, turn on the "Developer Tab" in "Excel Options". Presented below are more detailed comments and stepbystep instructions for creating and invoking macros, followed by some example macros.

Page Setup Macro Start recording a new macro called page setup. Select all of the worksheets and then choose Page Setup and customize the header and footers to include page numbers, date and time stamps, file locations, tab names, etc. Assign the macro to an Icon onthe toolbar or Quick Access Bar and insetting headers and footers will be a breeze for the rest of your life.

Print Macros Do you have a template that print frequently from? If so, insert several macro buttons to print each report, a group of reports, and even multiple reports and reporting will be snap in the future.

Delete Data Macro create a macro that visits each cell and erases that data, resetting the worksheet for use in a new set of criteria. Assign the macro to a macro button and will never again have old assumptions mixed in with your newer template