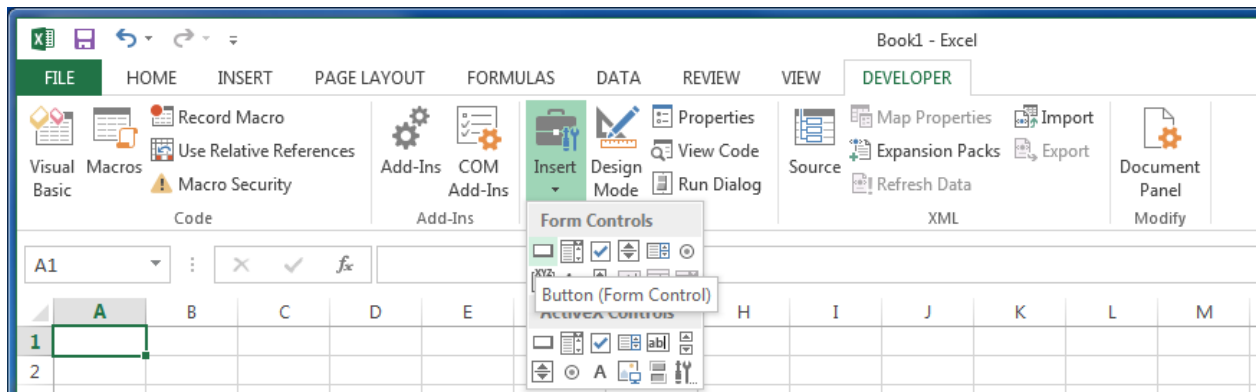


Introduction to VBA for Excel-Tutorial 4

In this tutorial, we will learn more ways to execute (run) the macro, learn about the most used mathematical functions, how to build an array (something you have been waiting for) and finally learn about some debugging tools. It is hard to believe you will write codes with errors (so called bugs), but let's be honest with ourselves it is more common than we can imagine.

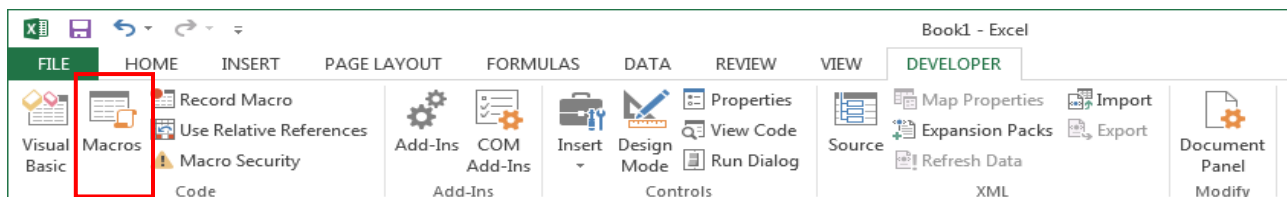
We learned three ways thus far to execute the macro:

- 1) Press **F5** while in VBE environment,
- 2) Press the Run button (red box in figure below) on the standard tool bar while in VBE environment, and
- 3) Adding a Control Button from the Developer tab on the Ribbon while in the Excel environment.

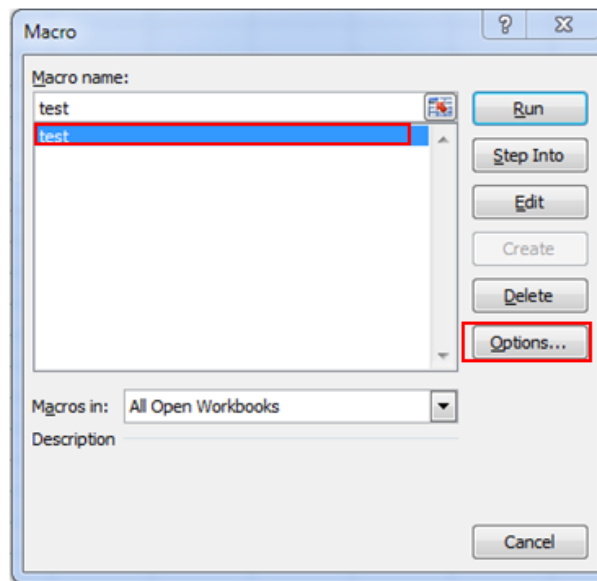


The following method is suitable for those of you that like keyboard shortcuts. To assign a shortcut key to run a macro:

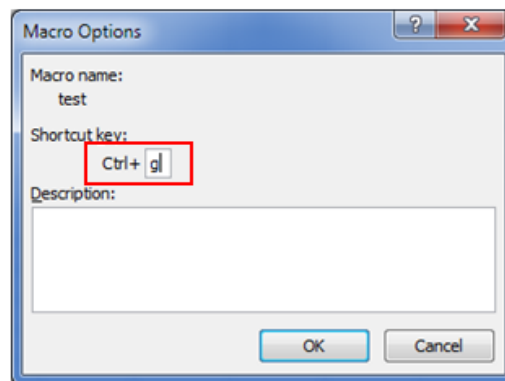
- 1- Click Developer/Code sub menu/ Macros



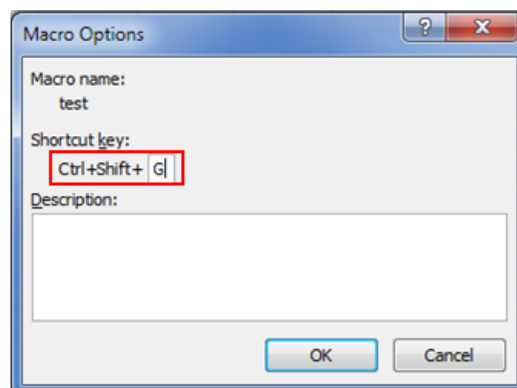
- 2- Excel will prompt you with Macro dialog box, Select the macro name you like to assign shortcut to. If you have multiple macros in the workbook, make sure you select the correct one.
- 3- Click Option button



- 4- Excel will prompt you with Macros Option dialog box, Insert the letter in the box beside **Ctrl+**, thus to execute the macro, you press **Ctrl+g**



If you entered an uppercase letter, for example "G" instead of "g", note the shortcut combination is **Ctrl+Shift+letter_you_entered**. In our example, you would press **Ctrl+Shift+G**



Tidbit: sometimes it is desired to execute a macro from another macro, I admit this is for more advanced users but it won't hurt to share this with you, to do so use **Call statement**, such as:

```
Sub test()
MsgBox "Hello Dr. George"
End Sub
```

```
Sub newsub()
Call test
End Sub
```

If you execute newsub procedure, the message in test procedure will be displayed.

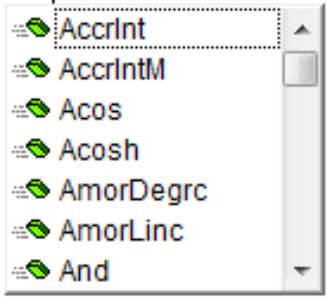
However, VBA has limited number of math functions there is a way to go around this limitation. First, here a list of the few useful functions in VBA, some of which mentioned in Tutorial #2:

Function	Description	syntax
Abs	Returns the absolute value of the same type that is passed to it	Abs (number)
Atn	Returns the arctangent of a number (in radians)	Atn (number)
Cos	Returns the cosine of a number (in radians)	Cos (number)
Sin	Returns the sine of a number (in radians)	Sin (number)
Exp	Returns the base of the natural logarithms e raised to a power	Exp (number)
Log	Returns the natural logarithm of a number	Log (number)
Sqr	Returns the square root of a number	Sqr (number)

We have practiced these functions in homework and in-class exercises, but we all know that Excel has a lot more functions (~ 340 functions) than these mentioned above. Thus, the question can we integrate these functions into our macros? The answer is simply **Yes!** This is in part due to the objectivity of the VBA. Remember, we said before that everything in MS Office is treated as an object, therefore we can simply call any function from Excel environment to VBA code such that:

`g = Application.WorksheetFunction.`

Worksheet Function
call syntax



Access to all
Excel Functions

Or simply use **WorksheetFunction.function_name** (i.e. WorksheetFunction.Sinh to call the hyperbolic sine function)

- **Arrays**

Arrays are the most used structure in engineering because variables take on multiple values and it is of interest to study the behavior of such change. Therefore, array is a group of variables, which could be string variables (i.e. text) or numbers (i.e. integer, double etc.). You can declare arrays the same way you declare any variable using Dim statement. If you know the size of the array a priori, then the Dim statement would be as follow for 1D array:

Dim array_name(start to end) as datatype

For example: if you need to define a time array that has 100 elements, the syntax should look like:

1	<i>Dim time(1 to 100) as single</i>	or
2	<i>Dim time(0 to 100) as single</i>	or
3	<i>Dim time(100) as single</i>	

In the first statement, you defined an array with 100 elements, where the start index (sometimes referred to as the lower index) is 1 and the end index (sometimes referred to as higher index) is 100.

In the second statement, you defined an array with 101 elements, where the start index is 0 and the end index is 100.

In the third statement, you defined an array with 101 elements, where the start index is 0 and the end index is 100.

Word of advice, starting an array with 0 index is very confusing because it's hard to keep track of the number of elements. If you noticed Dim statements 2 and 3 show the size to be 100, but the actual size is 101. To elevate the confusion, we will force VBA to start all arrays with 1 as the lower index. Add the following statement following the Option Explicit statement on the top of your code:

Option Explicit

Option Base 1

In this case, all statements above to declare an array are similar, the lower index = 1 and the higher index =100 and all arrays will have only 100 elements.

A very simple example using arrays: In this example, you declare a 1x3 array, assign values from the worksheet to each element and then output the successive sum. Note the Option Base 1 statement.

```

Option Explicit
Option Base 1
Sub array_demo()
'delclare
Dim input_array(1 To 3) As Single
'inputs
input_array(1) = Range("A1").Value
input_array(2) = Range("A2").Value
input_array(3) = Range("A3").Value
'outputs
Cells(1, 2).Value = input_array(1)
Cells(2, 2).Value = input_array(2) + input_array(1)
Cells(3, 2).Value = input_array(3) + input_array(2) + input_array(1)
End Sub

```

Consider another example. In this example, declare the variable “a” as variant, then assign the values from a range on the UI to that variable, such as shown below. I would like you to note two points, first how you assign the values using the Range Statement and second, how to access the item after casting.

```

Sub test()
Dim a As Variant
a = Range(Cells(1, 1), Cells(1, 10)).Value
MsgBox a(1, 1)
End Sub

```

Note: whether you declare the variable as variant (i.e. Dim a As Variant) or just declare the variable (i.e. Dim a) the end result is the same.

- **Complex numbers:**

In the homework, we solved the quadratic equation when the roots expected to be real. However, we know that it not always true. Sometimes, the roots are complex conjugates based on the value of the discriminant. This is considered a big problem; VBA does not have Complex data-type! We need to define our complex number data-type before we can perform any complex number based algebra. Here is the code to define a new complex data-type:

```

Option Explicit

Type complex
real As Double
imag As Double
End Type

```

An example to add two complex numbers is presented next. However, it is a trivial and brief example, it illustrate the structure and usage of complex number data-type. We will revisit this example again, once we learn conditional statements.

```

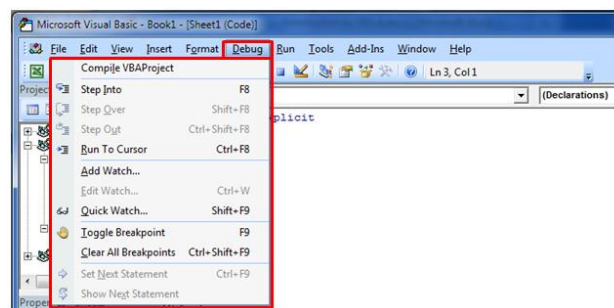
A.real = -2
A.imag = 2
B.real = 2
B.imag = 2

C.real = A.real + B.real
C.imag = A.imag + B.imag

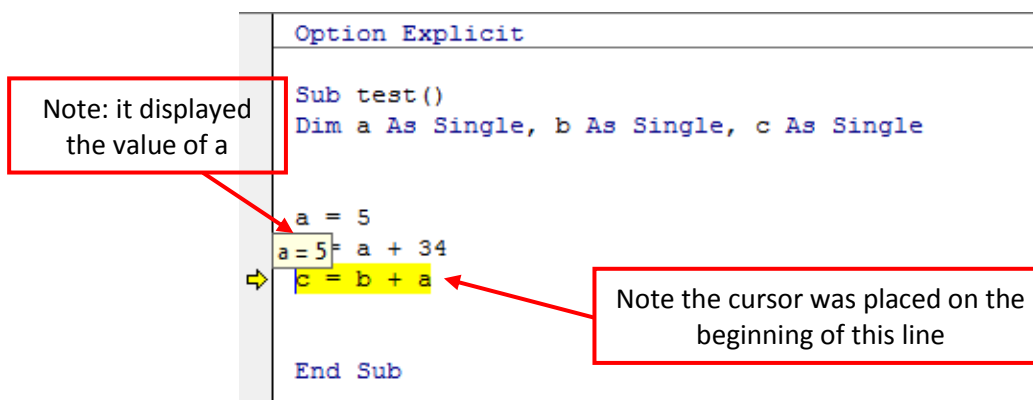
```

- **Debugging**

As you have seen thus far, most of the time when you execute a macro, VBA gets upset and bombard you with errors. These errors called bugs and the task of removing these bugs is not an easy one. It requires patience, skills, vigilance, and of course luck. VBA equipped with some good debugging tools, which can be accessed from Debug menu in VBE environment (pretty relevant name).



First debug tool is “**Run To Cursor**” or **Ctrl+F8**, the code will run to the location of the cursor and stop executing the rest including the line where it is point at. This is very handy tool, you can review the code section by section and execute up the section where you finished reviewing. A variable that has been defined or calculated will display its value if you hover the cursor over the variable. If the variable has not yet been calculated, the value will be labeled *Empty* or *Null* or *0*.



```


Sub test()
Dim a As Single, b As Single, c As Single

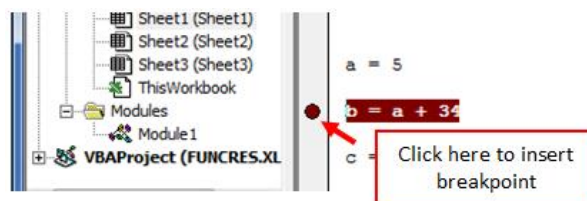
a = 5
b = a + 34
c = b + a
c = 0

End Sub

```

Note: c=0 because this line is not evaluated yet.

Second debugging tool is “**Toggle Breakpoint**”, which work similarly to the first tool. The advantage is you can set multiple breakpoints in the code. Think of breakpoint as Traffic Light, the code will run till the breakpoint and wait for you to give it the green to continue. To add a breakpoint, click on the gray bar beside the line you would like to insert the breakpoint. To continue execution after the breakpoint (1) press Run  on the tool bar, (2) select “**Continue**” from Run Menu or (3) press F5.



```

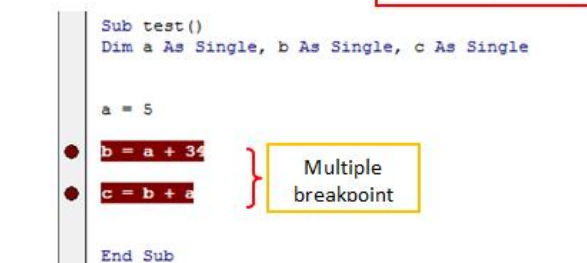
Sub test()
Dim a As Single, b As Single, c As Single

a = 5
b = a + 34
c =

End Sub

```

Click here to insert breakpoint



```

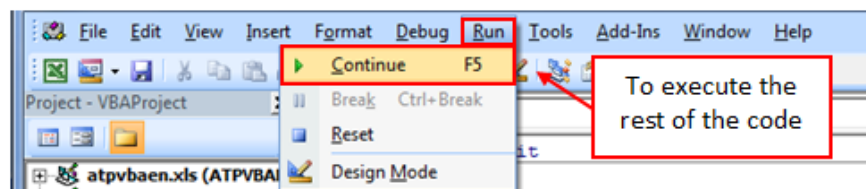
Sub test()
Dim a As Single, b As Single, c As Single

a = 5
b = a + 34
c = b + a

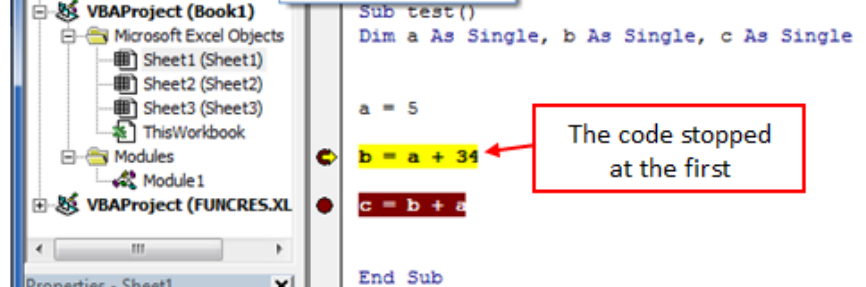
End Sub

```

Multiple breakpoint



To execute the rest of the code



```

Sub test()
Dim a As Single, b As Single, c As Single

a = 5
b = a + 34
c = b + a

End Sub

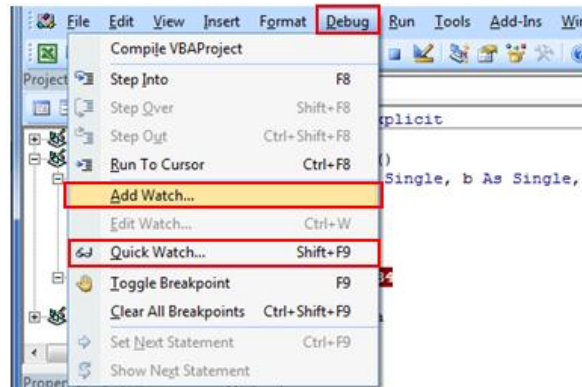
```

The code stopped at the first

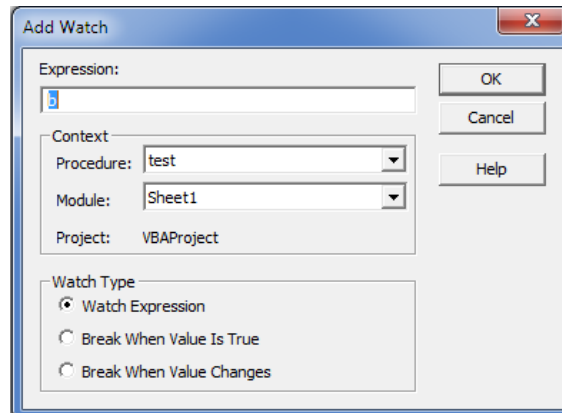
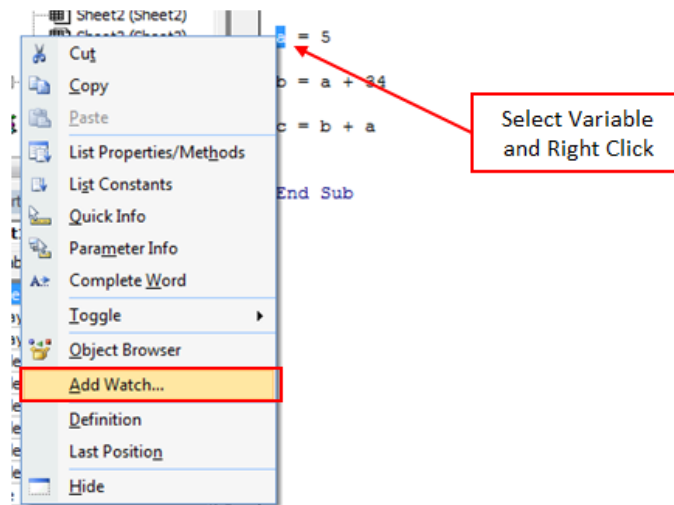
The last tool that we want to learn is “**Add Watch**,” it lets you select variables in advance whose current value at a point in the program will be displayed. You can access the Add Watch from Debug/Add Watch. Or select the variable you want to observe, right click and select Add Watch.

A quick way to add a variable to Add Watch list is **Quick Watch** from Debug menu. Excel will then prompt you with a dialog box.

Method 1:



Method 2:



Once you click ok, notice that Watches window is opened and a list of variables are added.

Expression	Value	Type	Context
a	<Out of context>	Empty	Sheet1.test