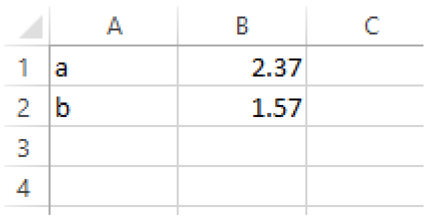
VBA in Excel 1

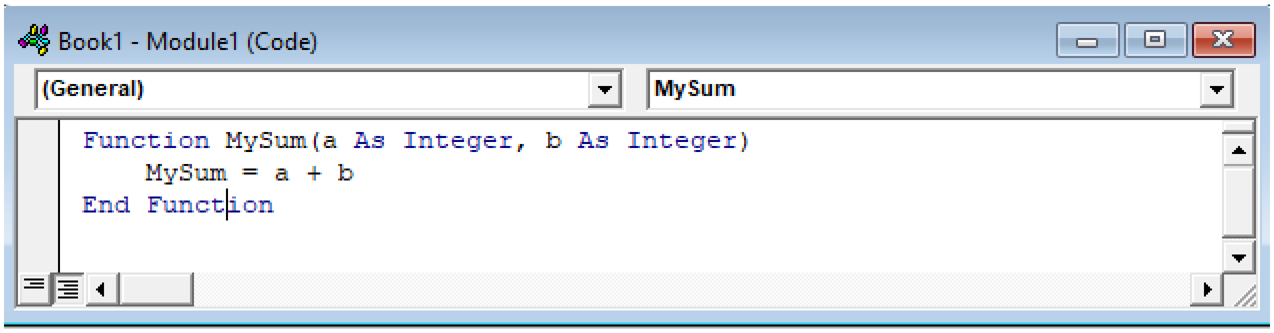
Creating Simple Function in VBA in Excel

1. Start Microsoft Excel 2013
2. Add the values form the figure below



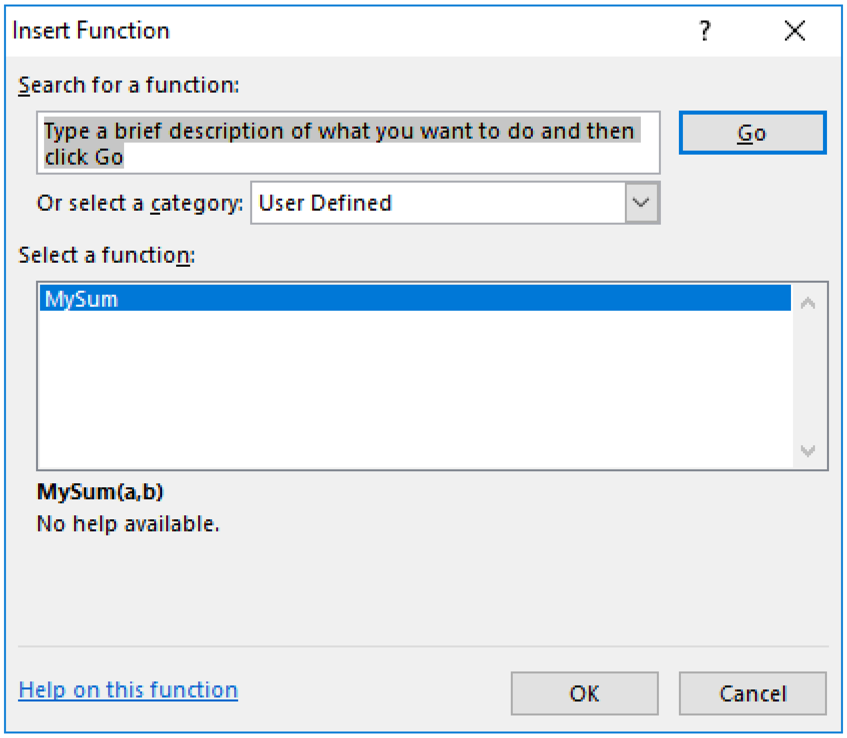
	A	B	C
1	a	2.37	
2	b	1.57	
3			
4			

1. Launch Visual Basic from the DEVELOPER tab.
2. In Visual Basic Editor (VBE) insert a module (menu **Insert → Module**)
3. Type the following code:



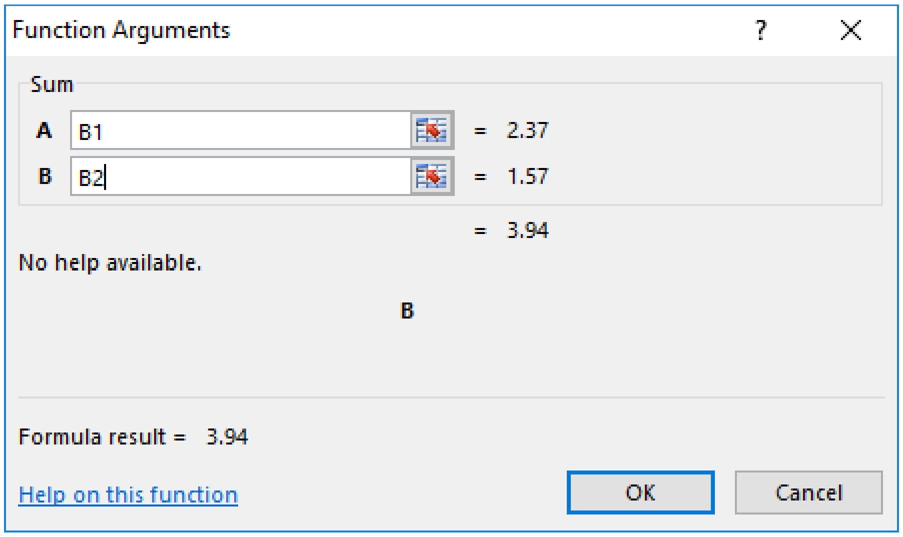
```
Function MySum(a As Integer, b As Integer)
    MySum = a + b
End Function
```

4. Close VBE and return to Microsoft Excel 2013
5. Move the cursor to cell C2
6. Launch Insert Function command



MySum(a,b)
No help available.

7. In the opened dialog box select the category User Defined from the dropdown list and the function MySum from the list of available functions and press OK.
8. In the Function Arguments dialog box select the two parameters of the function by clicking in the B1 and B2 cell respectively.


Function Arguments

Sum

A B1 = 2.37

B B2 = 1.57

= 3.94

No help available.

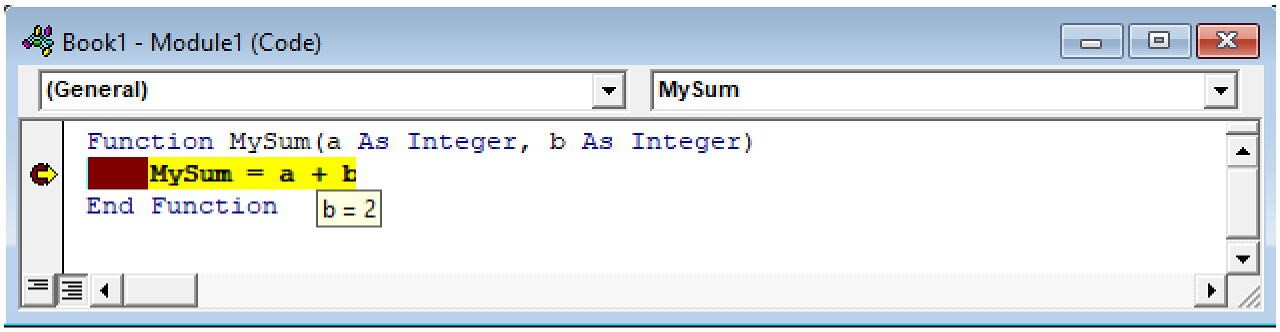
B

Formula result = 3.94

9. Press OK

Debugging and inspecting variables

1. Reopen VBE
2. In the statement MySum = a + b toggle a breakpoint (**Debug →Toggle Breakpoint**)
3. Activate the Excel workbook
4. Change the content of cell B2 to 1.53 and press **Enter**


Book1 - Module1 (Code)

(General) MySum

```
Function MySum(a As Integer, b As Integer)
MySum = a + b
End Function
```

b = 2

Notice that Excel will recalculate the sheet and call again the Visual Basic function MySum. Since we've toggled a breakpoint in line MySum = a + b, the execution stops in this point. When you move the mouse pointer over a variable, a tooltip window will pop up with the actual value of the variable.

You can advance the execution step by step by pressing **F8** (**Debug → Step Into**) or Continue (**F5**).

1. Change the MySum function as follows:

```
Function MySum(a As Double, b As Double) As Double
    MySum = a + b
End Function
```

2. Activate the current workbook in Excel and do a recalculate by pressing **CTRL+ALT+F9**
3. Explain the differences between the previous and current example

Decision Algorithms

We will write a VB function that evaluates the mathematical function:

$$f(x) = \begin{cases} x^2 + 2x & x \leq 0 \\ x + 3 & 0 < x < 1 \\ 2x & x \geq 1 \end{cases}$$

1. Activate **VBE** and in module **Module1** add the following code:

```
Function MyFunction(x As Double) As Double
    If (x <= 0) Then
        MyFunction = x ^ 2 + 2 * x
    Else
        If (x >= 1) Then
            MyFunction = 2 * x
        Else
            MyFunction = x + 3
        End If
    End If
End Function
```

2. Add a breakpoint in the first line of the function and execute the function for three diferent value : -1, 2 and 0.5. Follow the flow of the function by pressing repeatedly **F8** to see how the function is executed based on the entered value

Loops with known number of steps

We will write a VB function to calculate the factorial of a number.

1. Activate **VBE** and in module **Module1** add the following code

```
Function Factorial(n As Long) As Long
    Dim i As Integer

    Factorial = 1
    For i = 1 To n
        Factorial = Factorial * i
    Next i
End Function
```

2. Add a breakpoint in the line **Factorial = Factorial * i** and run the function for different values of the parameter **n**.

REM. You'll notice that the function is able to calculate the factorial of a number up to 12. If the parameter is greater than 12 an overflow of the **Long** data type occurs.

1. Change the function to deal with numbers greater than 12. In case of a number greater than 12, the function will return an error message "Overflow".

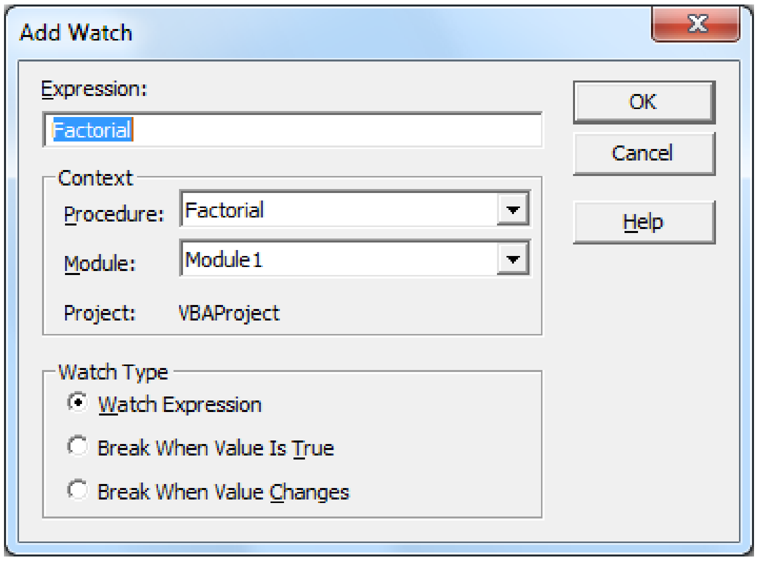
```
Function Factorial(n As Long) As Variant
    Dim i As Integer
    If (n > 12) Then
        Factorial = "Overflow"
     Else
        Factorial = CLng(1)
        For i = 1 To n
            Factorial = Factorial * i
        Next i
    End If
End Function
```

REM. You'll notice that we changed the data type of the function from Long to Variant. The Variant data type can hold any of the other data types: numerical (**Integer**, **Long**, **Single** or **Double**) or string (**String**).

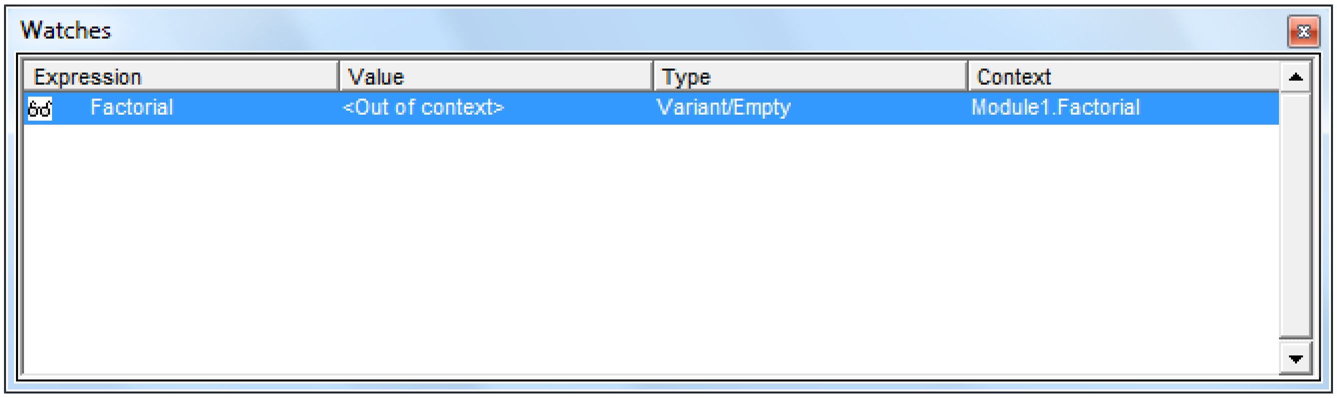
Also in the **Else** branch we have added a conversion from 1 (which defaults to Integer) to **CLng**(1) which will force the Factorial function to hold a **Long** type.

Inspecting the variables with the Watch window

2. Add two breakpoint in the lines: **Factorial = "Overflow"** and **Factorial = CLng(1)**.
3. Move the cursor over the variable Factorial and from the context menu select the command **Add Watch**.
4. In the opened dialog make sure that in the textbox Expression the name of the variable Factorial is displayed
5. Press **Enter.**



Expression: Factorial
Procedure: Factorial
Module: Module1
Project: VBAProject
Watch Expression [✓]
Break When Value Is True
Break When Value Changes



```
Watches
Expression    Value             Type            Context
Factorial     <Out of context>  Variant/Empty   Module1.Factorial
```

REM. You'll notice that the **Watches** window will be displayed in VBE and the name of the variable added to the list of variables to watch.

6. Rerun the function Factorial for values less or greater than 12 and inspect the values and data type of the function **Factorial** in the **Watches** window.
7. Change the statement **Factorial = CLng(1)** in **Factorial = 1** and check the data type of the **Factorial** function for different values of **n** in the Watches window.

1. Change the function to deal with negative numbers.
2. Add an **IF** block to check if n is less than zero. If this is the case, the function will return the message "Negative"

```
Function Factorial(n As Long) As Variant
    Dim i As Integer

    If (n < 0) Then
        Factorial = "Negative"
    Else
        If (n > 12) Then
            Factorial = "Overflow"
        Else
            Factorial = 1
            For i = 1 To n
                Factorial = Factorial * i
            Next i
        End If
    End If
End Function
```

Loops with unknown number of steps

We will write a function that calculates the value of e^x with a desired precision ε.

REM: To calculate e^x we will use the series approximation:

$$e^x = \sum_{i=0}^{\infty} u_k = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} \quad \text{with } u_k = \frac{x^k}{k!} = \frac{x}{k} u_{k-1}$$

1. Activate **VBE** and in module **Module1** add the following code:
2. Insert this function in Excel and calculate the value of *e*^{x} for different values of x.
3. Check the value returned from our function with the value obtained from the library function **EXP(x)**.
4. Change the value of the precision constant and rerun the code
5. Add a breakpoint at the beginning of the function and execute the code step by step for **x=1**.

```
Function ExpX(x As Double) As Double
    Const Eps = 0.0001
    Dim u As Double
    Dim k As Integer

    ExpX = 0
    u = 1
    k = 0
    Do While (u > Eps)
        ExpX = ExpX + u
        k = k + 1
        u = u * x / k
    Loop
End Function
```

REM. You'll notice that we've added a constant, declared with the keyword **Const**.

We will write a function that calculates the square root of a number with a constant precision ε.

REM: We will used the fact that the sequence (x_n) defined by $x_1 = a,\ x_n = \frac{1}{2}\left(x_{n-1} + \frac{a}{x_{n-1}}\right)$ converges to$\sqrt{a}$. The limit for $\sqrt{a}$ is the term for which $|x_n - x_{n-1}| < \varepsilon$

```
Function SquareRoot(a As Double) As Double
    Const Eps = 0.0001
    Dim xn1 As Double
    Dim xn As Double
    xn = a
    Do
        xn1 = xn
        xn = (xn1 + a / xn1) / 2
    Loop While (Abs(xn - xn1) >= Eps)
    SquareRoot = xn
End Function
```

REM. To calculate the absolute value (without a sign) of a number we use the VB function **Abs**.

1. Activate **VBE** and in module **Module1** add the previous code:
2. Insert this function in Excel and calculate the value of $\sqrt{a}$ for different values of a.
3. Check the value returned from our function against the value obtained from the library function **SQRT(a).**
4. Change the value of the precision constant and rerun the code
5. Add a breakpoint at the beginning of the function and execute the code step by step for **a=4**.