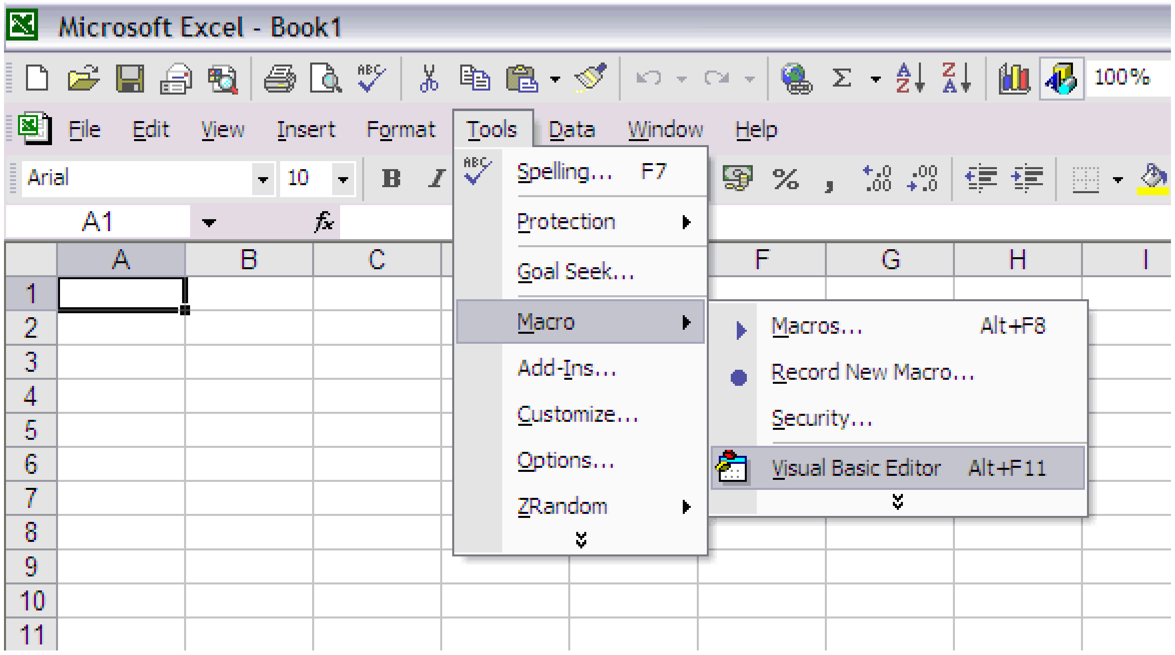
# Excel VBA Basic Tutorial 1

This page contains the 1st lesson on the Excel VBA Basic Tutorial series. It covers topics in creating and managing array and understanding the VBA decision and loop structures. Beginners in VBA programming are encouraged to go through the prior lessons in this series if they had not already done so. This document contains information about the following topics.

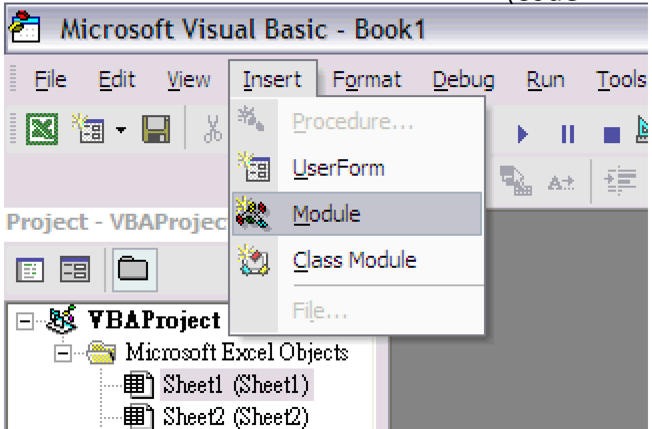
## Creating Your First Macro Microsoft Support

In this sub section, we will show you how to create your first macro (VBA program). We will use the world classic "Hello World!" example. To create the example, please follow the following steps:

1. Open Visual Basic Editor by go to **Tools**...**Macro**...**Visual Basic Editor** or just simply press the [Alt] and [F11] keys at the same time.

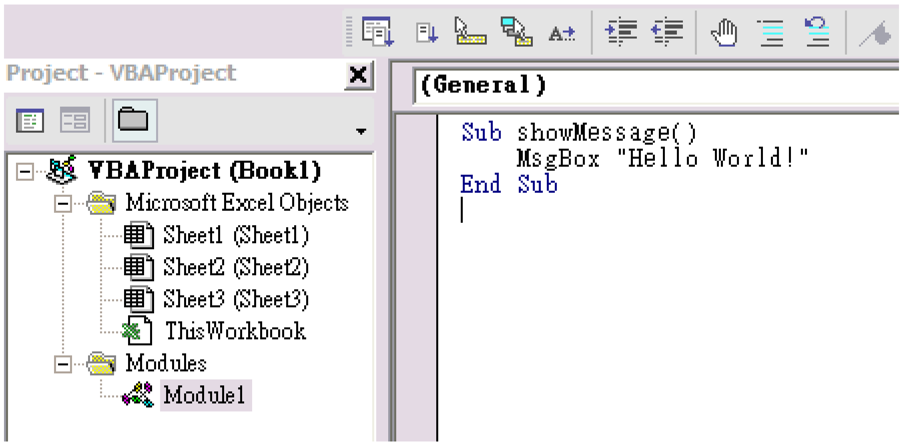


2. In the **Insert** menu on top of the Visual Basic Editor, select Module to open the **Module window** (code window).



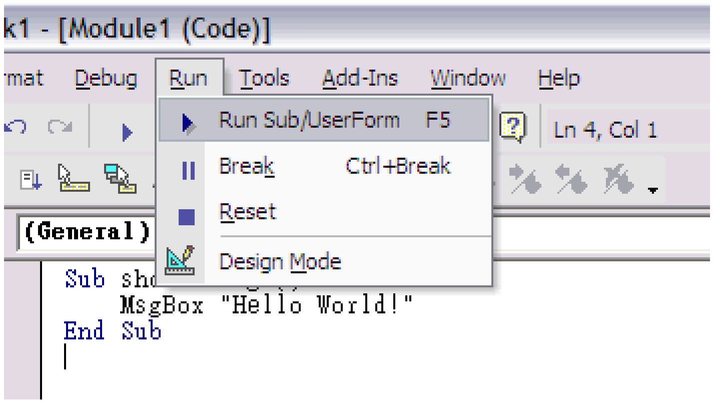
3. In the Module window, type the following:

```
Sub showMessage()
   MsgBox "Hello World!"
End Sub
```



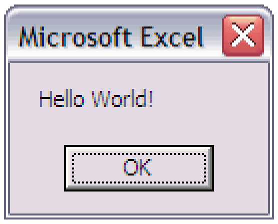
```
Sub showMessage()
    MsgBox "Hello World!"
End Sub
```

4. Click the **Run** button, , press [F5], or go to **Run**..**Run Sub/UserForm** to run the program



```
Sub sho
    MsgBox "Hello World!"
End Sub
```

5. The message box pops up with the "Hello World!" greeting.



This is your first VBA programmer.

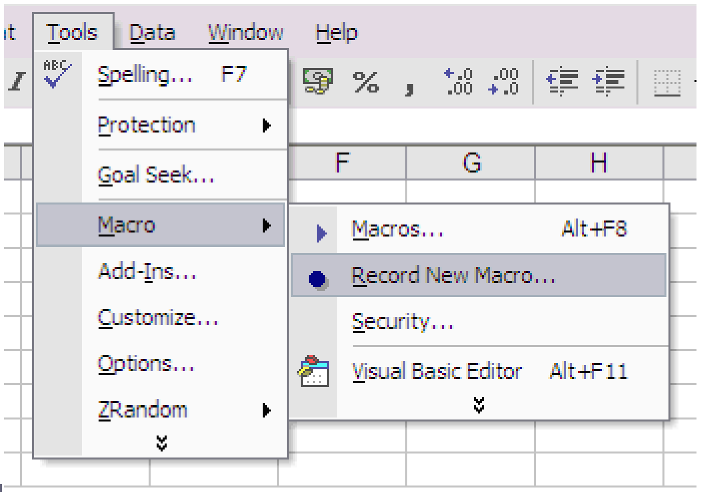
**Recording Your First Macro**

**Recording a Macro**

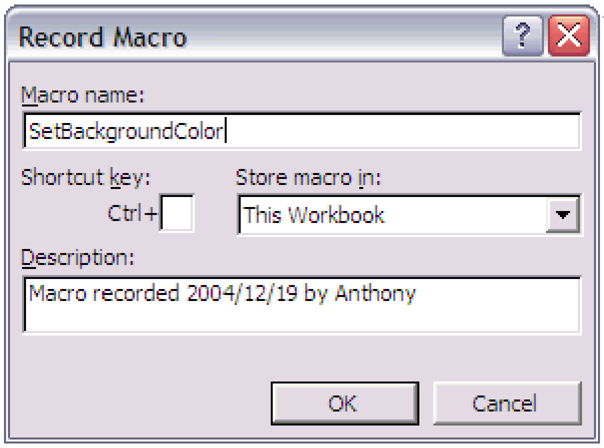
Macrosoft Excel has a build-in macro recorder that translates your actions into VBA macro commands. After you recorded the macro, you will be able to see the layout and syntax. Before you record or write a macro, plan the steps and commands you want the macro to perform. Every actions that you take during the recording of the macro will be recorded - including the correction that you made.

In this example, we will record a macro that sets the cell background color to light yellow. To record the macro, follow the steps below:

1. Select **Record New Macro**... under **Tools...Macro**



2. In the Record Macro dailog box, type "SetBackgroundColor" in the Macro Name textbox to set the macro name. Leave all other option by default then click the Ok button. This will start the macro recording.


Record Macro

Macro name: SetBackgroundColor

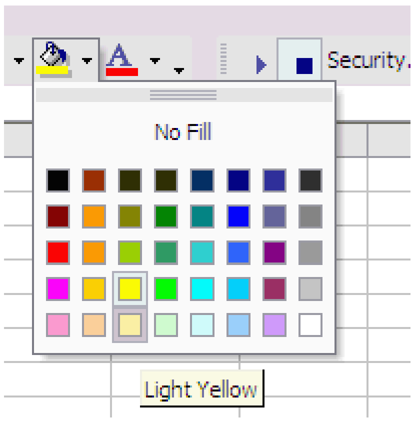
Shortcut key: Ctrl+

Store macro in: This Workbook

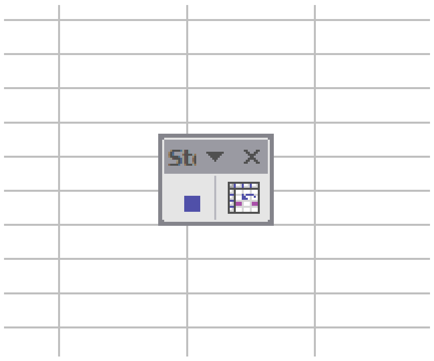
Description: Macro recorded 2004/12/19 by Anthony

OK Cancel

3. In the Background Color Panel, select the Ligth Yellow color box. This action will set the background of the current cell (A1) in light yellow color.



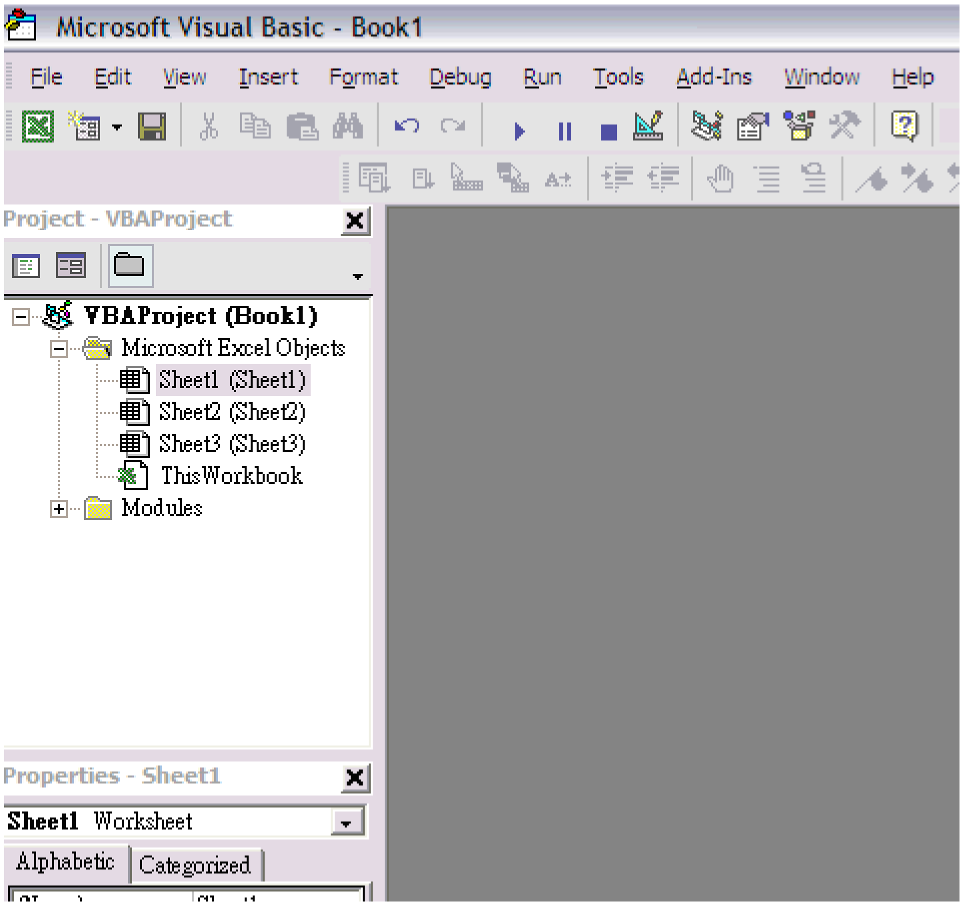
4. To stop the macro recording, click the Stop button (the navy blue rectangle) on the Macro Recorder toolbar.



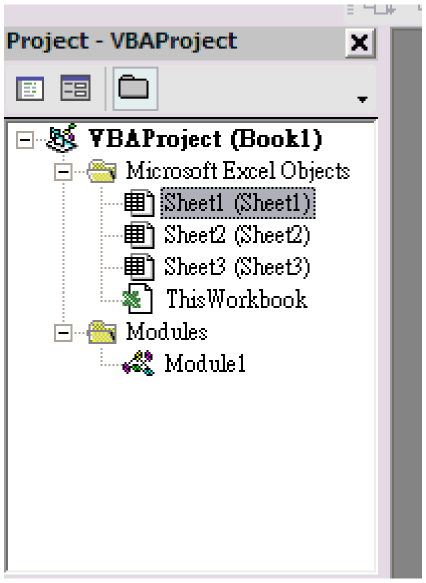
Now you have recorded a macro that set cell background to light yellow.

## See the Recorded Syntax

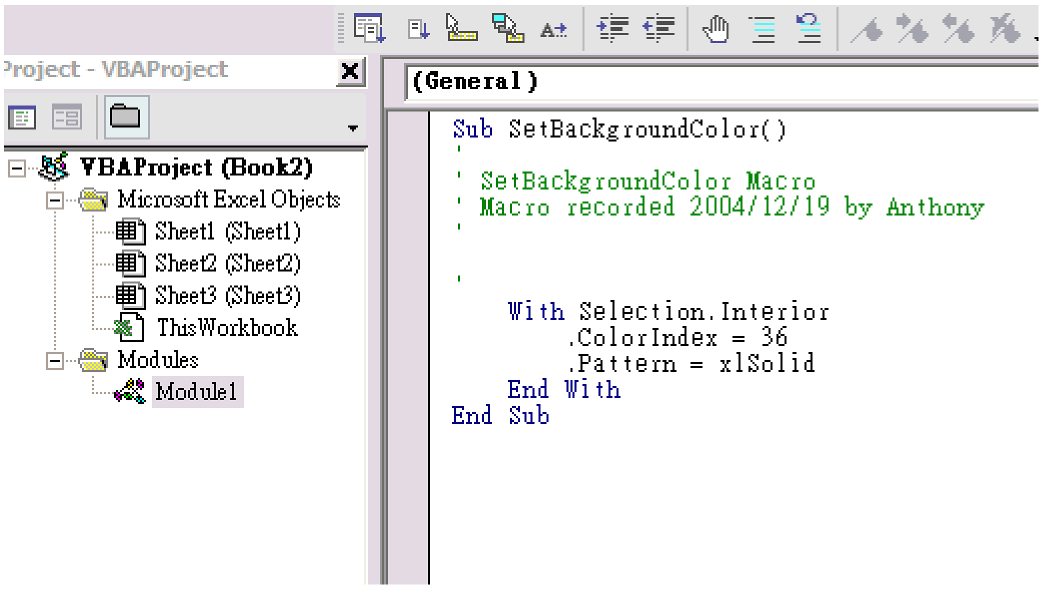
The recorded macro is ready for use.  Before we run the marco, let's look into the syntax.
1. To load the Visual Basic Editor, press [Alt] and [F11] at the same time.  (Remember from our prior lesson?)  The Visual Basic Editor comes up.



2. Expand the **Modules** folder in the **Project Explorer** by clicking on the plus (+) sign.



3. Double click the **Module1** folder to see the sub routine (marco).



```
Sub SetBackgroundColor()
'
' SetBackgroundColor Macro
' Macro recorded 2004/12/19 by Anthony
'

'
    With Selection.Interior
        .ColorIndex = 36
        .Pattern = xlSolid
    End With
End Sub
```

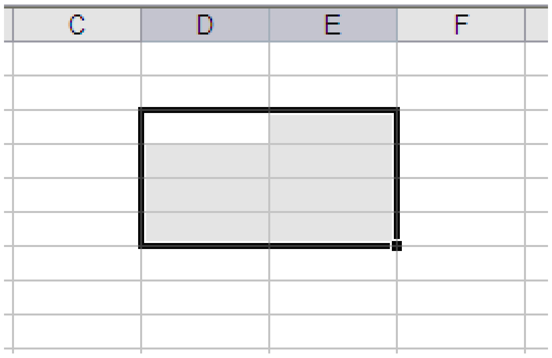
As the figure shows, the name of the sub routine is "SetBackgroundColor".   The color index for the light yellow is 36.  The background pattern is soild.

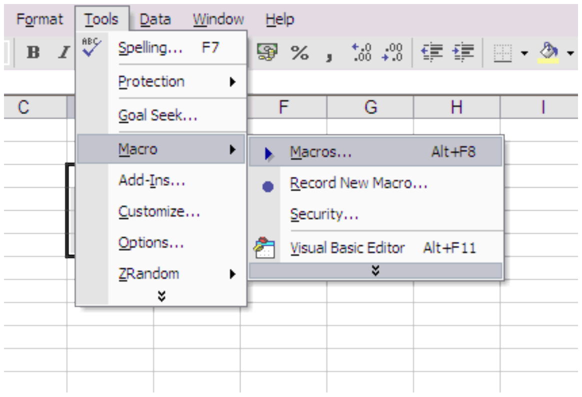
---

**Run the Recorded Macro**

In our prior example, we created the "Hello World!" marco.  We ran the macro within the Visual Basic Editor.  This time we will run the recorded macro in the worksheet.

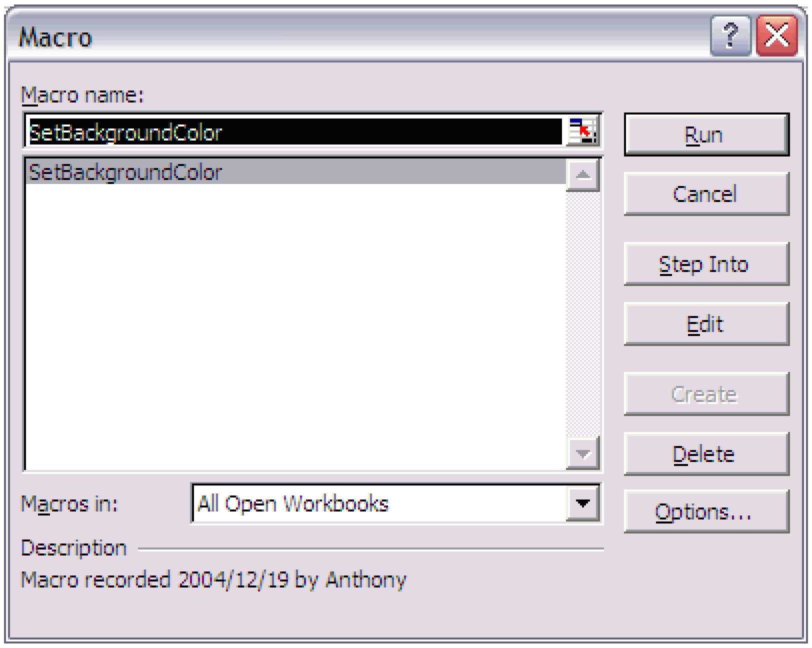
1. On any worksheet, select from D3 to E6.





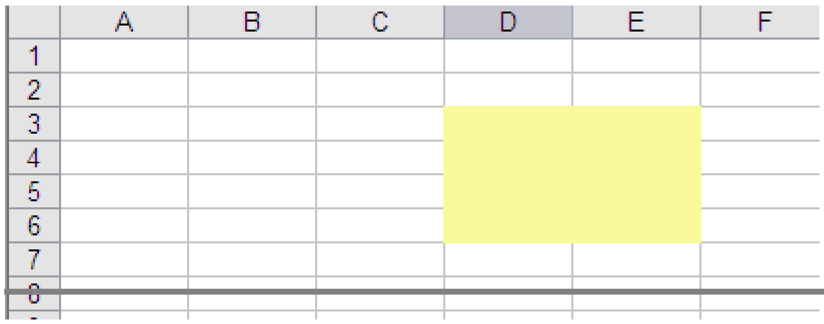
2. Run the recorded macro by select Tools...Macro...Macros... or press [Alt] and [F8] at the same time.

3. The Macro dialog box displayed. Since there is only one macro in the module, by default the only macro, SetBackgroundColor is selected. Click the Run button to run the macro.



Macro recorded 2004/12/19 by Anthony

4. Cells D3 to E6 now have light yellow background color.



## Modules and Procedures

### Modules and Procedures and Their Scope

A **module** is a container for procedures as shown in our prior examples.  A **procedure** is a unit of code enclosed either between the **Sub** and **End Sub** statement or between the **Function** and **End Function** statements.

The following sub procedure (or sub routine) print the current date and time on cell C1:

```
Sub ShowTime()
   Range("C1") = Now()
End Sub
```

The following function sum up two numbers:

```
Function sumNo(x, y)
    sumNo = x + y
End Function
```

Procedures in Visual Basic can have either private or public scope.  A procedure with private scope is only accessible to the other procedures in the same module; a procedure with public scope is accessible to all procedures in in every module in the workbook in which the procedure is declared, and in all workbooks that contain a reference to that workbook.  By default, procedures has public scope.
Here are examples of defining the scope for procedure.

```
Public Sub ShowTime()
   Range("C1") = Now()
End Sub

Private Sub ShowTime()
   Range("C1") = Now()
End Sub
```

---

### Calling Sub Procedures and Function Procedures

There are two ways to call a sub procedure.  The following example shows how a sub procedure can be called by other sub procedures.

```
Sub z(a)
   MsgBox a
End Sub

Sub x()
   Call z("ABC")
End Sub

Sub y()
   z "ABC"
End Sub
```

Sub z procedure takes an argument (a) and display the argument value ("ABC") in a message box.  Running either Sub x or Sub y will yield the same result.

The following example calls a function procedure from a sub procedure.

```
Sub ShowSum()
    msgbox sumNo(3,5)
End Sub

Function sumNo(x, y)
    sumNo = x + y
End Function
```

The ShowSum sub procedure calls the sumNo function and returns an "8" in a message box.

If there are procedures with duplicate names in different modules, you must need to include a module qualifier before the procedure name when calling the procedure.

For example:

```
Module1.ShowSum
```

---

**Passing Argument by Reference or by Value**
If you pass an argument **by reference** when calling a procedure, the procedure access to the actual variable in memory. As a result, the variable's value can be changed by the procedure. Passing by reference is the default in VBA. If you do not explicitly specify to pass an argurment **by value**, VBA will pass it by reference. The following two statements yield the same outcome.

```
Sub AddNo(ByRef x as integer)
Sub AddNo(x as integer)
```

Here is an example to show the by reference behavior. The sub procedure, TestPassing 1 calls AddNo1 by reference and display "60" (50 + 10) on the message box.

```
Sub TestPassing1()
   Dim y As Integer
   y = 50
   AddNo1 y
   MsgBox y
End Sub

Sub AddNo1(ByRef x As Integer)
   x = x + 10
End Sub
```

The following example shows the by value behavior. The sub procedure, TestPassing 2 calls AddNo2 by value and display "50" on the message box.

```
Sub TestPassing2()
   Dim y As Integer
   y = 50
   AddNo2 y
   MsgBox y
End Sub

Sub AddNo2(ByVal x As Integer)
   x = x + 10
End Sub
```

---

## Excel VBA Basic Tutorial 2

This page contains the 2nd lesson on the Excel VBA Basic Tutorial series. It covers topics in the most used Excel objects and collections. Beginners in VBA programming are encouraged to go through the 1st lessons in this series if they had not already done so. This document contains information about the following topics.

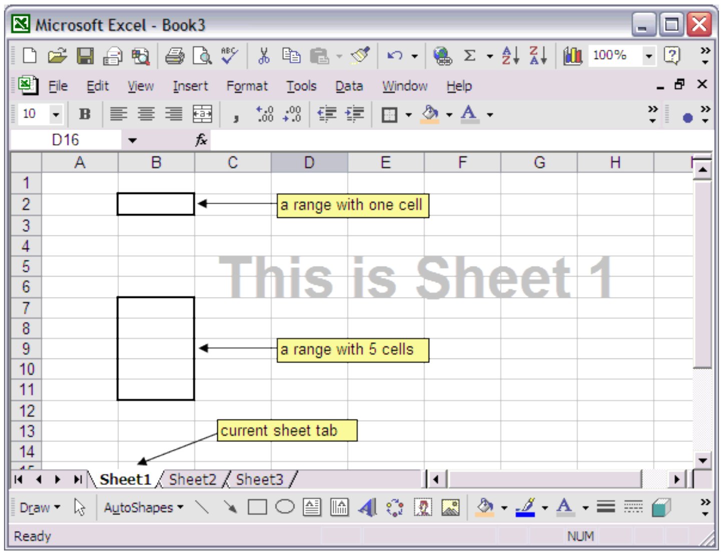
*Microsoft Support site or the Excel VBA Help section on your computer contains comprehensive examples on most the issues covered on this page. For more information, please refer to them.*

## Objects and Collections    Microsoft Support

Objects are the fundamental building blocks of Visual Basic. An **object** is a special type of variable that contains both data and codes. A **collection** is a group of objects of the same class. The most used Excel objects in VBA programming are Workbook, Worksheet, Sheet, and Range.

Workbooks is a collection of all Workbook objects. Worksheets is a collection of Worksheet objects.
The Workbook object represents a workbook, the Worksheet object represents a worksheet, the Sheet object represents a worksheet or chartsheet, and the Range object represents a range of cells.

The following figure shows all the objects mentioned. The workbook (Excel file) is currently Book3.xls. The current worksheet is Sheet1 as the Sheet Tab indicated. Two ranges are selected, range B2 and B7:B11.



## Workbook and Worksheet Object

A **workbook** is the same as an Excel file. The Workbook collection contains all the workbooks that are currently opened. Inside of a workbook contains at least one **worksheet**. In VBA, a worksheet can be referenced as followed:

```
Worksheets("Sheet1")
```

Worksheets("Sheet1") is the worksheet that named "Sheet1."
Another way to refer to a worksheet is to use number index like the following:

```
Worksheets(1)
```

The above refers to the first worksheet in the collection.

* Note that Worksheets(1) is not necessary the same sheet as Worksheets("Sheet1").

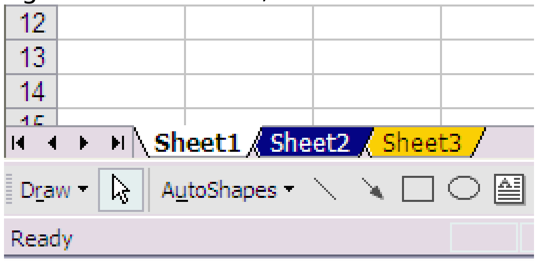
Sheets is a collection of worksheets and chart sheets (if present).  A sheet can be indexed just like a worksheet.  Sheets(1) is the first sheet in the workbook.

To refer sheets (or other objects) with the same name, you have to qualify the object.  For example:

```
Workbooks("Book1").Worksheets("Sheet1")
Workbooks("Book2").Worksheets("Sheet1")
```

*If the object is not qualified, the active or the current object (for example workbook or worksheet) is used.*

The sheet tab on the buttom the spreadsheet (worksheet) shows which sheet is active.  As the figure below shows, the active sheet is "Sheet1" (show in bold font and white background).



* You can change the color of the sheet tabs by right click the tab, choose Tab Color, then select the color for the tab.

The sub routine below shows the name of each sheet in the current opened workbook.  You can use **For Each...Next loop** to loop throgh the Worksheets collection.

```
Sub ShowWorkSheets()
   Dim mySheet As Worksheet

   For Each mySheet In Worksheets
      MsgBox mySheet.Name
   Next mySheet

End Sub
```

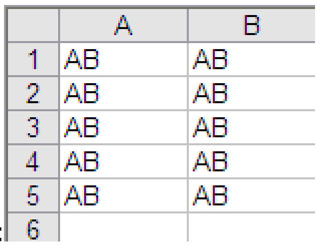
---

**Range Object and Cells Property**

**Range** represents a cell, a row, a column, a selection of cells containing one or more contiguous blocks of cells, or a 3-D range.  We will show you some examples on how Range object can be used.

The following example places text "AB" in range A1:B5, on Sheet2.

```
Worksheets("Sheet2").Range("A1:B5") = "AB"
```

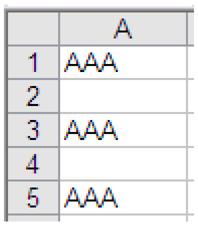
:

| | A | B |
|---|---|---|
| 1 | AB | AB |
| 2 | AB | AB |
| 3 | AB | AB |
| 4 | AB | AB |
| 5 | AB | AB |
| 6 | | |

Note that, Worksheets.Range("A1", "B5") = "AB" will yield the same result as the above example.

The following place "AAA" on cell A1, A3, and A5 on Sheet2.

Worksheets("Sheet2").Range("A1, A3, A5") = "AAA"



| | A |
|---|---|
| 1 | AAA |
| 2 | |
| 3 | AAA |
| 4 | |
| 5 | AAA |

Range object has a **Cells** property. This property is used in every VBA projects on this website (**very important**). The Cells property takes one or two indexes as its parameters.

For example,
Cells(*index*) or Cells(*row*, *column*)

where *row* is the row index and *column* is the column index.
The following three statements are interchangable:

ActiveSheet.Range.Cells(1,1)
Range.Cells(1,1)
Cells(1,1)

The following returns the same outcome:

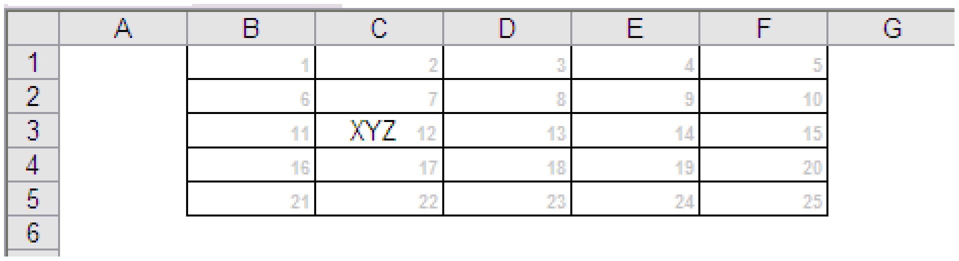
Range("A1") = 123    and    Cells(1,1) = 123

The following puts "XYZ" on Cells(1,12) or Range("L1") assume cell A1 is the current cell:

Cells(12) = "XYZ"

The following puts "XYZ" on cell C3:

Range("B1:F5").cells(12) = "ZYZ"

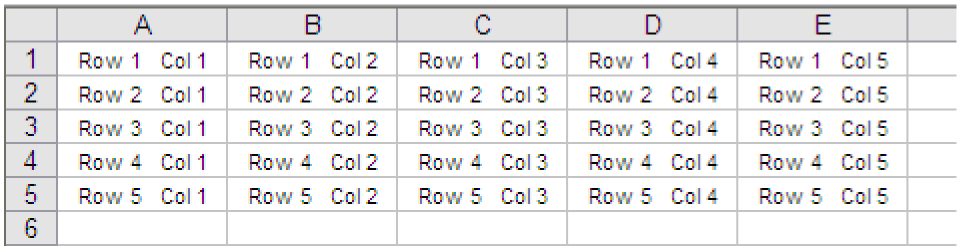


|  | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 1 |  | 1 | 2 | 3 | 4 | 5 |  |
| 2 |  | 6 | 7 | 8 | 9 | 10 |  |
| 3 |  | 11 | XYZ 12 | 13 | 14 | 15 |  |
| 4 |  | 16 | 17 | 18 | 19 | 20 |  |
| 5 |  | 21 | 22 | 23 | 24 | 25 |  |
| 6 |  |  |  |  |  |  |  |

* The small gray number on each of the cells is just for reference purpose only. They are used to show how the cells are indexed within the range.

Here is a sub routine that prints the corresponding row and column index from A1 to E5.

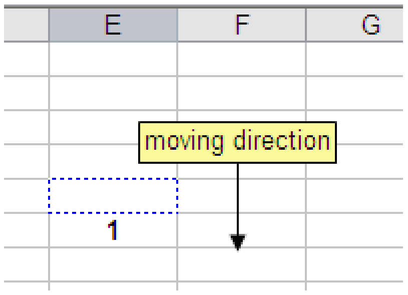
```
Sub CellsExample()
   For i = 1 To 5
        For j = 1 To 5
            Cells(i, j) = "Row " & i & "   Col " & j
        Next j
   Next i
End Sub
```



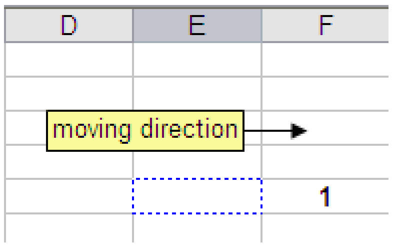
|  | A | B | C | D | E |  |
|---|---|---|---|---|---|---|
| 1 | Row 1 Col 1 | Row 1 Col 2 | Row 1 Col 3 | Row 1 Col 4 | Row 1 Col 5 |  |
| 2 | Row 2 Col 1 | Row 2 Col 2 | Row 2 Col 3 | Row 2 Col 4 | Row 2 Col 5 |  |
| 3 | Row 3 Col 1 | Row 3 Col 2 | Row 3 Col 3 | Row 3 Col 4 | Row 3 Col 5 |  |
| 4 | Row 4 Col 1 | Row 4 Col 2 | Row 4 Col 3 | Row 4 Col 4 | Row 4 Col 5 |  |
| 5 | Row 5 Col 1 | Row 5 Col 2 | Row 5 Col 3 | Row 5 Col 4 | Row 5 Col 5 |  |
| 6 |  |  |  |  |  |  |

Range object has an **Offset** property that can be very handy when one wants to move the active cell around. The following examples demostrate how the Offset property can be implemented (assume the current cell before the move is E5):

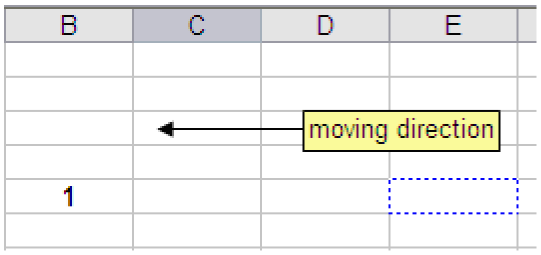
`ActiveCell.Offset(1,0) = 1` Place a "1" one row under E5 (on E6)



`ActiveCell.Offset(0,1) = 1` Place a "1" one column to the right of E5 (on F5)



ActiveCell.Offset(0,-3) = 1 Place a "1" three columns to the left of E5 (on B5)



---

**Methods and Properties**

Each object contains its own methods and properties.

A **Property** represents a built-in or user-defined characteristic of the object. A **method** is an action that you perform with an object. Below are examples of a method and a property for the Workbook Object:

Workbooks.Close
**Close** method close the active workbook

Workbooks.Count
**Count** property returns the number of workbooks that are currently opened

Some objects have default properties. For example, Range's default property is Value.
The following yields the same outcome.

Range("A1") = 1 and Range("A1").Value = 1

Here are examples on how to set and to get a Range property value:
The following sets the value of range A1 or Cells(1,1) as "2005". It actually prints "2005" on A1.

Range("A1").Value = 2005

The following gets the value from range A1 or Cells(1,1).

X = Range("A1").Value

Method can be used with or without argument(s). The following two examples demostrate this behavior.

Methods That Take No Arguments:

Worksheets("Sheet").Column("A:B").AutoFit

Methods That Take Arguments:

```
Worksheets("Sheet1").Range("A1:A10").Sort _
Worksheets("Sheet1").Range("A1")
```

Worksheets("Sheet1").Range("A1") is the **Key** (or column) to sort by.

---

**Assigning Object Variables and Using Named Argument**

Sometime a method takes more than one argument. For example, the Open method for the Workbook
object, takes 12 arguments. To open a workbook with password protection, you would need to write the following code:

```
Workbooks.Open "Book1.xls", , , ,"pswd"
```

Since this method takes so many arguments, it is easy to misplace the password argument. To overcome this potential problem, one can use named arguments like the following example:

```
Workbook.Open fileName:="Book1.xls", password:="pswd"
```

You can also assign an object to an object variable using the **Set** Statement.

For example:

```
Dim myRange as Range
Set myRange = Range("A1:A10")
```

---

**Excel VBA Basic Tutorial 3**

This page contains the 3rd lesson on the Excel VBA Basic Tutorial series. It covers topics in creating and managing array and understanding the VBA decision and loop structures. Beginners in VBA programming are encouraged to go through the prior lessons in this series if they had not already done so. This document contains information about the following topics.

**Do ... Loop Until**

*Microsoft Support site or the Excel VBA Help section on your computer contains comprehensive examples on most the issues covered on this page.  For more information, please refer to them.*

---

**Creating and Managing Array**  **Microsoft Support**

**Declaring an Array With Dim Statement**

An array is a set of sequentially indexed elements having the same intrinsic data type. Each element of an array has a unique identifying index number. Changes made to one element of an array don't affect the other elements.

Before signing values to an array, the array needs to be created.  You can declare the array by using the **Dim** statement.

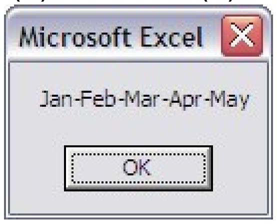
For example, to declare a one-dimensional array with 5 elements, type the following:

```
Dim Arr(4)
```

The element’s index of the array starts from 0 unless **Option Base 1** is specified in the public area (area outside of the sub procedure).  If Option Base 1 is specified, the index will start from 1.

The following example assigns values to the array and displays all values in a message box :

```
Option Base 1
Sub assignArray( )
   Dim Arr(5)

      Arr(1) = “Jan”
      Arr(2) = “Feb”
          Arr(3) = “Mar”
          Arr(4) = “Apr”
      Arr(5) = “May”

   Msgbox Arr(1) & "-" & Arr(2) & "-" & Arr(3) & "-" & Arr(4) & "-" & Arr(5)
```



Jan-Feb-Mar-Apr-May

```
End Sub
```

* The number inside the array, i.e. Arr(1), is the index.  One (1) is the index of the first element in the
  array.

---

**Resize an Array With Redim Statement**

The **ReDim** statement is used to size or resize a dynamic array that has already been formally declared.

For example, if you have already declared an array with an element value of 5 and decided to change the number of the element to 6, you can do the following to resize the array:

```
Redim Arr(6)
```

We incorporate it into our last example:

```
Option Base 1
Sub assignArray( )
    'Dim Arr(5)
    Redim Arr(6)

    Arr(1) = “Jan”
    Arr(2) = “Feb”
    Arr(3) = “Mar”
    Arr(4) = “Apr”
    Arr(5) = “May”
    Arr(6) = “Jun”

    Msgbox Arr(1) & "-" & Arr(2) & "-" & Arr(3) & "-" & Arr(4) & "-" & Arr(5)
  End Sub
```

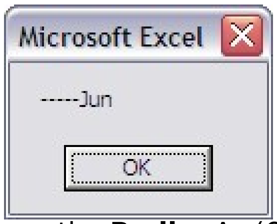
Note that the **Dim** Arr(5) statement is commoned out, because leaving this original statement in the sub will causing a compile error.

---

**Manage Dynamic Array**

A word of caution in using the **Redim** Statement to resize an array - resize the array can erase the elements in it. In the following example, all the values assigned prior to resize the array are erased. Only the value assigned to the array after resize remains.

```
Option Base 1
 Sub assignArray( )
    Redim Arr(5)

    Arr(1) = “Jan”
    Arr(2) = “Feb”
    Arr(3) = “Mar”
    Arr(4) = “Apr”
    Arr(5) = “May”

    Redim Arr(6)

    Arr(6) = “Jun”

    Msgbox Arr(1) & "-" & Arr(2) & "-" & Arr(3) & "-" & Arr(4) & "-" & Arr(5) & "-" & Arr(6)
 End Sub
```

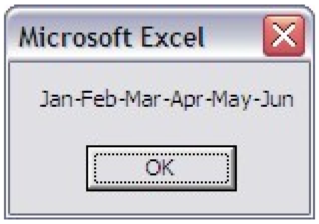


```
-----Jun
```

By replace the **Redim** Arr(6) with **Redim Preserve** Arr(6), all values will remain. For example:

```
Option Base 1
Sub assignArray( )
    Redim Arr(5)
```

```
        Arr(1) = “Jan”
        Arr(2) = “Feb”
        Arr(3) = “Mar”
        Arr(4) = “Apr”
        Arr(5) = “May”

        Redim Preserve Arr(6)

        Arr(6) = “Jun”

        Msgbox Arr(1) & "-" & Arr(2) & "-" & Arr(3) & "-" & Arr(4) & "-" & Arr(5) & "-" & Arr(6)
    End Sub
```



```
Jan-Feb-Mar-Apr-May-Jun
```

---

**Create Multi-Dimensional Array**

An array can also store multiple dimensional data.  To simplify our tutorial, example on a two-dimensional array is used.  Assume you have data of a local store's yearly sale in the following table and you want to store the data in a two-dimensional array:

| | Year 2003 | Year 2004 |
|---|---|---|
| CD Sale | *1,000* | *1,500* |
| DVD Sale | *1,200* | *2,000* |

First we create the array as follow:

```
    Dim Arr(2,2)
```

Then we assign the values into the array.  We treat the first dimension as the year and the second dimension as the product sale:

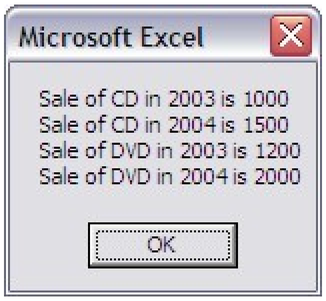
```
    arr(1,1) = 1000
    arr(1,2) = 1200
    arr(2,1) = 1500
    arr(2,2) = 2000
```

We now display the values of the array with a message box:

```
   Msgbox "Sale of CD in 2003 is " & arr(1,1) & vbCrLf  & "Sale of CD in 2004 is " _
                  & arr(2,1)  & vbCrLf  & "Sale of DVD in 2003 is " & arr(1,2) & vbCrLf  _
             & "Sale of DVD in 2004 is " & arr(2,2)
```

The complete procedure is as followed:

```
    Option Base 1
    Sub multDimArray( )
        Dim Arr(2,2)
        arr(1,1) = 1000
        arr(1,2) = 1200
        arr(2,1) = 1500
        arr(2,2) = 2000

          Msgbox "Sale of CD in 2003 is " & arr(1,1) & vbCrLf  & "Sale of CD in 2004 is " _
               & arr(2,1) & vbCrLf  & "Sale of DVD in 2003 is " & arr(1,2) & vbCrLf  _
               & "Sale of DVD in 2004 is " & arr(2,2)
    End Sub
```



```
Microsoft Excel

Sale of CD in 2003 is 1000
Sale of CD in 2004 is 1500
Sale of DVD in 2003 is 1200
Sale of DVD in 2004 is 2000

OK
```

* vbCrLf stands for VB Carriage Return Line Feed.  It puts a return and a new line as shown in the message box above.  The underscore "_" on the back of the first line of the message box means "continue to the next line"

---

**Find The Size of an Array**

The largest available subscript for the indicated dimension of an array can be obtained by using the **Ubound** function.  In our one-dimensional array example, Ubound(arr) is 5.

In our two-dimensional array example above, there are two upper bound figures - both are 2. **UBound** returns the following values for an array with these dimensions*:

```
Dim A(1 To 100, 0 To 3, -3 To 4)
```

| Statement | Return Value |
|---|---|
| UBound(A, 1) | 100 |
| UBound(A, 2) | 3 |
| UBound(A, 3) | 4 |

* Example taken from Excel VBA Help section.

The **UBound** function is used with the LBound function to determine the size of an array. Use the **LBound** function to find the lower limit of an array dimension.

| Statement | Return Value |
|---|---|
| LBound(A, 1) | 1 |
| LBound(A, 2) | 0 |
| LBound(A, 3) | -3 |

To get the size of an array, use the following formula:

**UBound(Arr) - LBound(Arr) + 1**

For example:

```
Ubound(A,1) - LBound(A,1) + 1
 = 100 - 1 + 1
 = 100

Ubound(A,2) - LBound(A,2) + 1
 = 3 - 0 + 1
 = 4

Ubound(A,3) - LBound(A,3) + 1
 = 4 - (-3) + 1
 = 8
```

For more information on arrays check Microsoft Support

---

## Decision Structures - IF and Select Case

### IF ... Then Statement

The **IF ... Then** is a single condition and run a single statement or a block of statement.

Example, the following statement set variable Status to "Adult" if the statement is true:

```
If Age >= 18 Then Status = "Adult"
```

You can also use multiple-line block in the If statement as followed:

```
If Ago >= 18 Then
    Status = "Adult"
    Vote = "Yes"
End If
```

Note that in the multiple-line block case, End If statement is needed, where the single-line case does not.

---

### IF ... Then ... Else

The **If ... Then ... Else** statement is used to define two blocks of conditions - true and false.

Example:

```
 If Age >=22 Then
    Drink = "Yes"
Else
    Drink = "No"
End If
```

Again, note that End If statement is needed in this case as well since there is more than one block of statements.

---

### IF ... Then ... ElseIf

The **IF ... Then ... ElseIf** is used to test additional conditions without using new If ... Then statements.

For Example:

```
 If Age >= 18 and Age < 22 Then
     Msgbox "You can vote"
 ElseIf Age >=22 and Age < 62 Then
     Msgbox "You can drink and vote"
 ElseIf Age >=62 Then
     Msgbox "You are eligible to apply for Social Security Benefit"
 Else
     Msgbox "You cannot drink or vote"
 End If
```

Note that the last condition under Else is, implicitly, Age < 18.

---

**Select Case**

**Select Case** statement is an alternative to the ElseIf statement.  This method is more efficient and readable in coding the the **If ... Then ... ElseIf** statment.

Example:

```
 Select Case Grade
     Case Is >= 90
         LetterGrade = "A"
     Case Is >= 80
         LetterGrade = "B"
     Case Is >= 70
         LetterGrade = "C"
     Case Is >= 60
         LetterGrade = "D"
     Case Else
         LetterGrade = "Sorry"
 End Select
```

---

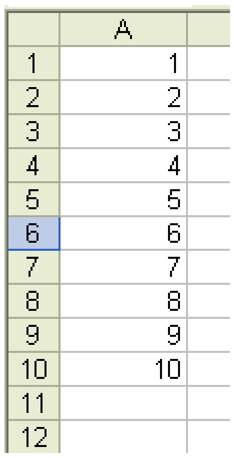
**Loop Structures**

**For ... Next**

Use **For ... Next** loop if the number of loops is already defined and known.  A **For ... Next** loop uses a counter variable that increases or decreases in value during each iteration of the loop.  This loop structure is being used the most for our examples on this site.

Here is an example of the **For ... Next** loop:

```
 For i = 1 to 10
     Cells(i, 1) = i
 Next i
```



| | A |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
| 5 | 5 |
| 6 | 6 |
| 7 | 7 |
| 8 | 8 |
| 9 | 9 |
| 10 | 10 |
| 11 | |
| 12 | |

In this example, i is the counter variable from 1 to 10.  The looping process will send value to the first column of the active sheet and print i (which is 1 to 10) to row 1 to 10 of that column.

Note that the counter variable, by default, increases by an increment of 1.

---

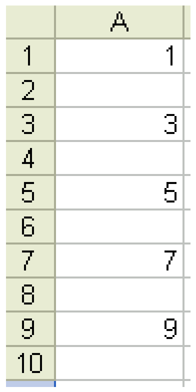
**For ... Next Loop With Step**

You can use the **Step** Keyword to sepcify a different increment for the counter variable.

For example:

```
For i = 1 to 10 Step 2
   Cells(i, 1) = i
Next i
```

This looping process will print values with an increment of 2 on row 1, 3, 5, 7 and 9 on column one.



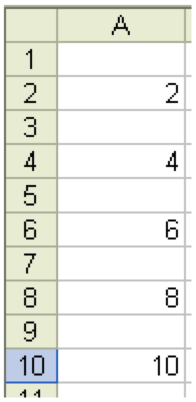
| | A |
|---|---|
| 1 | 1 |
| 2 | |
| 3 | 3 |
| 4 | |
| 5 | 5 |
| 6 | |
| 7 | 7 |
| 8 | |
| 9 | 9 |
| 10 | |

You can also have decrement in the loop by assign a negative value afte the **Step** keyword.

For example:

```
For i = 10 to 1 Step -2
   Cells(i, 1) = i
Next i
```

This looping process will print values with an increment of -2 starts from 10 on row  10, 8, 6, 4 and 2 on column one.



| | A |
|---|---|
| 1 | |
| 2 | 2 |
| 3 | |
| 4 | 4 |
| 5 | |
| 6 | 6 |
| 7 | |
| 8 | 8 |
| 9 | |
| 10 | 10 |

**Do While ... Loop**

You can use the **Do While ... Loop** to test a condition at the start of the loop.  It will run the loop as long as the condition is ture and stops when the condition becomes false.  For Example:

```
i = 1
 Do While i =< 10
     Cells(i, 1) = i
     i = i + 1
 Loop
```

This looping process yields the same result as in the **For ... Next** structures example.

One thing to be caution is that sometimes the loop might be a infinite loop.  And it happens when the condition never becomes false.  In such case, you can stop the loop by press **[ESC]** or **[CTRL]** + **[BREAK]**.

**Do Until ... Loop**

You can test the condition at the beginning of the loop and then run the loop until the test condition becomes true.

Example:

```
 i = 1
 Do Until i = 11
     Cells(i, 1) = i
     i = i + 1
 Loop
```

This looping process yields the same result as in the **For ... Next** structures example.

**Do ... Loop While**

When you want to make sure that the loop will run at least once, you can put the test at the end of loop.  The loop will stop when the condition becomes false.   (compare this loop structure to the Do ... While Loop.)

For Example:

```
i = 1
 Do
```

```
      Cells(i, 1) = i
      i  = i + 1
   Loop While i < 11
```

This looping process yields the same result as in the **For ... Next** structures example.

---

### Do ... Loop Until

This loop structure, like the **Do ... Loop While**, makes sure that the loop will run at least once, you can put the test at the end of loop.  The loop will stop when the condition becomes true. (compare this loop structure to the **Do ... Until** Loop.)

For Example:

```
  i = 1
   Do
      Cells(i, 1) = i
      i  = i + 1
   Loop Until i = 11
```

This looping process yields the same result as in the **For ... Next** structures example.

---

## Excel VBA Simulation Basic Tutorial 101

This page contains basic Excel VBA skills needed for creating simulations.  Beginners who wish to learn simulation programming using Excel VBA are encouraged to go through the entire document if he or she had not already done so.  This tutorial is the prerequisite of Excel VBA Simulation-Based Tutorial 201.  This document contains information about the following topics.

- **Find Maximum and Minimum Values in an Array**
- **Double Sorting - The Secret of Resampling Without Replacement**

Microsoft Support site or the Excel VBA Help section on your computer contains comprehensive examples on most the issues covered on this page.  For more information, please refer to them.

---

**Creating and Managing Array**    **Microsoft Support**

**Declaring an Array With Dim Statement**

An array is a set of sequentially indexed elements having the same intrinsic data type. Each element of an array has a unique identifying index number. Changes made to one element of an array don't affect the other elements.

Before signing values to an array, the array needs to be created.  You can declare the array by using the **Dim** statement.

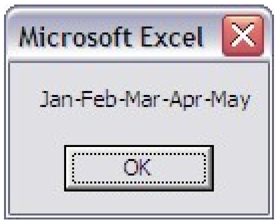
For example, to declare a one-dimensional array with 5 elements, type the following:

```
Dim Arr(4)
```

The element’s index of the array starts from 0 unless **Option Base 1** is specified in the public area (area outside of the sub procedure).  If Option Base 1 is specified, the index will start from 1.

The following example assigns values to the array and displays all values in a message box :

```
Option Base 1
Sub assignArray( )
   Dim Arr(5)

      Arr(5) = “Jan”
      Arr(2) = “Feb”
          Arr(3) = “Mar”
          Arr(4) = “Apr”
      Arr(5) = “May”

   Msgbox Arr(1) & "-" & Arr(2) & "-" & Arr(3) & "-" & Arr(4) & "-" & Arr(5)
End Sub
```



```
Jan-Feb-Mar-Apr-May
```

* The number inside the array, i.e. Arr(1), is the index.  One (1) is the index of the first element in the
 array.

---

**Resize an Array With Redim Statement**

The **ReDim** statement is used to size or resize a dynamic array that has already been formally declared.

For example, if you have already declared an array with an element value of 5 and decided to change the number of the element to 6, you can do the following to resize the array:

```
        Redim Arr(6)
```

We incorporate it into our last example:

```
        Option Base 1
        Sub assignArray( )
                'Dim Arr(5)
                Redim Arr(6)

                Arr(1) = "Jan"
                Arr(2) = "Feb"
                Arr(3) = "Mar"
                Arr(4) = "Apr"
                Arr(5) = "May"
                Arr(6) = "Jun"

                Msgbox Arr(1) & "-" & Arr(2) & "-" & Arr(3) & "-" & Arr(4) & "-" & Arr(5)
          End Sub
```

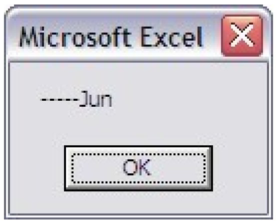
Note that the **Dim** Arr(5) statement is commoned out, because leaving this original statement in the sub will causing a compile error.

---

**Manage Dynamic Array**

A word of caution in using the **Redim** Statement to resize an array - resize the array can erase the elements in it.  In the following example, all the values assigned prior to resize the array are erased.  Only the value assigned to the array after resize remains.

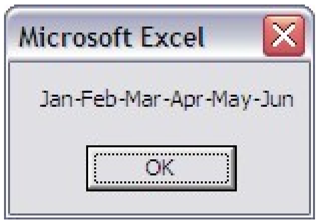
```
         Option Base 1
          Sub assignArray( )
                Redim Arr(5)

                Arr(1) = "Jan"
                Arr(2) = "Feb"
                Arr(3) = "Mar"
                Arr(4) = "Apr"
                Arr(5) = "May"

                Redim Arr(6)

                Arr(6) = "Jun"

                Msgbox Arr(1) & "-" & Arr(2) & "-" & Arr(3) & "-" & Arr(4) & "-" & Arr(5) & "-" & Arr(6)
          End Sub
```



```
-----Jun
```

By replace the **Redim** Arr(6) with **Redim Preserve** Arr(6), all values will remain. For example:

```
Option Base 1
Sub assignArray( )
     Redim Arr(5)

     Arr(1) = “Jan”
     Arr(2) = “Feb”
     Arr(3) = “Mar”
     Arr(4) = “Apr”
     Arr(5) = “May”

     Redim Preserve Arr(6)

     Arr(6) = “Jun”

     Msgbox Arr(1) & "-" & Arr(2) & "-" & Arr(3) & "-" & Arr(4) & "-" & Arr(5) & "-" & Arr(6)
 End Sub
```



```
Jan-Feb-Mar-Apr-May-Jun
```

**Create Multi-Dimensional Array**

An array can also store multiple dimensional data. To simplify our tutorial, example on a two-dimensional array is used. Assume you have data of a local store's yearly sale in the following table and you want to store the data in a two-dimensional array:

| | Year 2003 | Year 2004 |
|---|---|---|
| CD Sale | *1,000* | *1,500* |
| DVD Sale | *1,200* | *2,000* |

First we create the array as follow:

```
Dim Arr(2,2)
```

Then we assign the values into the array. We treat the first dimension as the year and the second dimension as the product sale:

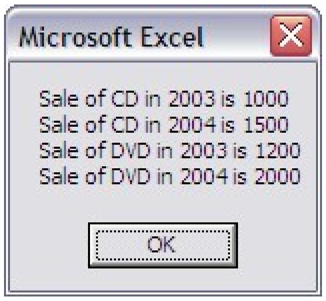
```
arr(1,1) = 1000
arr(1,2) = 1200
arr(2,1) = 1500
arr(2,2) = 2000
```

We now display the values of the array with a message box:

```
Msgbox "Sale of CD in 2003 is " & arr(1,1) & vbCrLf  & "Sale of CD in 2004 is " _
                & arr(2,1)  & vbCrLf  & "Sale of DVD in 2003 is " & arr(1,2) & vbCrLf  _
          & "Sale of DVD in 2004 is " & arr(2,2)
```

The complete precedure is as followed:

```
Option Base 1
  Sub multDimArray( )
      Dim Arr(2,2)

      arr(1,1) = 1000
      arr(1,2) = 1200
      arr(2,1) = 1500
      arr(2,2) = 2000

          Msgbox "Sale of CD in 2003 is " & arr(1,1) & vbCrLf  & "Sale of CD in 2004 is " _
                  & arr(2,1) & vbCrLf  & "Sale of DVD in 2003 is " & arr(1,2) & vbCrLf  _
                  & "Sale of DVD in 2004 is " & arr(2,2)
  End Sub
```



```
Sale of CD in 2003 is 1000
Sale of CD in 2004 is 1500
Sale of DVD in 2003 is 1200
Sale of DVD in 2004 is 2000
```

* vbCrLf stands for VB Carriage Return Line Feed.  It puts a return and a new line as shown in the message box above.  The underscore "_" on the back of the first line of the message box means "continue to the next line"

---

**Find The Size of  an Array**

The largest available subscript for the indicated dimension of an array can be obtained by using the **Ubound** function.  In our one-dimensional array example, Ubound(arr) is 5.

In our two-dimensional array example above, there are two upper bound figures - both are 2. **UBound** returns the following values for an array with these dimensions*:

Dim A(1 To 100, 0 To 3, -3 To 4)

| Statement | Return Value |
|---|---|
| UBound(A, 1) | 100 |
| UBound(A, 2) | 3 |
| UBound(A, 3) | 4 |

* Example taken from Excel VBA Help section.

The **UBound** function is used with the LBound function to determine the size of an array. Use the **LBound** function to find the lower limit of an array dimension.

| Statement | Return Value |
|---|---|
| LBound(A, 1) | 1 |
| LBound(A, 2) | 0 |
| LBound(A, 3) | -3 |

To get the size of an array, use the following formula:

```
UBound(Arr) - LBound(Arr) + 1
```

For example:

```
Ubound(A,1) - LBound(A,1) + 1
 = 100 - 1 + 1
 = 100

Ubound(A,2) - LBound(A,2) + 1
 = 3 - 0 + 1
 = 4

Ubound(A,3) - LBound(A,3) + 1
 = 4 - (-3) + 1
 = 8
```

For more information on arrays check Microsoft Support

## Decision Structures - IF and Select Case

### IF ... Then Statement

The **IF ... Then** is a single condition and run a single statement or a block of statement.

Example, the following statement set variable Status to "Adult" if the statement is true:

```
If Age >= 18 Then Status = "Adult"
```

You can also use multiple-line block in the If statement as followed:

```
If Ago >= 18 Then
    Status = "Adult"
    Vote = "Yes"
End If
```

Note that in the multiple-line block case, End If statement is needed, where the single-line case does not.

### IF ... Then ... Else

The **If ... Then ... Else** statement is used to define two blocks of conditions - true and false.

Example:

```
 If Age >=22 Then
    Drink = "Yes"
Else
    Drink = "No"
End If
```

Again, note that End If statement is needed in this case as well since there is more than one block of statements.

### IF ... Then ... ElseIf

The **IF ... Then ... ElseIf** is used to test additional conditions without using new If ... Then statements.

For Example:

```
 If Age >= 18 and Age < 22 Then
     Msgbox "You can vote"
  ElseIf Age >=22 and Age < 62 Then
     Msgbox "You can drink and vote"
  ElseIf Age >=62 Then
     Msgbox "You are eligible to apply for Social Security Benefit"
  Else
     Msgbox "You cannot drink or vote"
  End If
```

Note that the last condition under Else is, implicitly, Age < 18.

---

**Select Case**

**Select Case** statement is an alternative to the ElseIf statement.  This method is more efficient and readable in coding the the **If ... Then ... ElseIf** statment.

Example:

```
 Select Case Grade
     Case Is >= 90
        LetterGrade = "A"
     Case Is >= 80
        LetterGrade = "B"
     Case Is >= 70
        LetterGrade = "C"
     Case Is >= 60
        LetterGrade = "D"
     Case Else
        LetterGrade = "Sorry"
  End Select
```

---

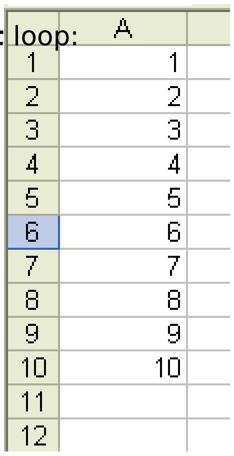
**Loop Structures**

**For ... Next**

Use **For ... Next** loop if the number of loops is already defined and known.  A **For ... Next** loop uses a counter variable that increases or decreases in value during each iteration of the loop.  This loop structure is being used the most for our examples on this site.

Here is an example of the **For ... Next** loop:

```
 For i = 1 to 10
     Cells(i, 1) = i
  Next i
```



|  | A |  |
|---|---|---|
| 1 | 1 |  |
| 2 | 2 |  |
| 3 | 3 |  |
| 4 | 4 |  |
| 5 | 5 |  |
| 6 | 6 |  |
| 7 | 7 |  |
| 8 | 8 |  |
| 9 | 9 |  |
| 10 | 10 |  |
| 11 |  |  |
| 12 |  |  |

In this example, i is the counter variable from 1 to 10.  The looping process will send value to the first column of the active sheet and print i (which is 1 to 10) to row 1 to 10 of that column.

Note that the counter variable, by default, increases by an increment of 1.

---

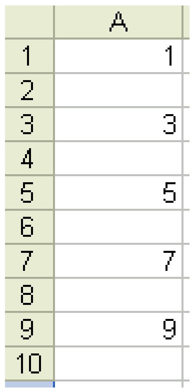
**For ... Next Loop With Step**

You can use the **Step** Keyword to sepcify a different increment for the counter variable.

For example:

```
For i = 1 to 10 Step 2
    Cells(i, 1) = i
Next i
```

This looping process will print values with an increment of 2 on row 1, 3, 5, 7 and 9 on column one.



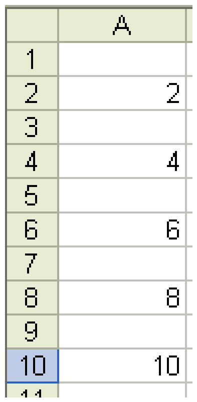
|  | A |
|---|---|
| 1 | 1 |
| 2 |  |
| 3 | 3 |
| 4 |  |
| 5 | 5 |
| 6 |  |
| 7 | 7 |
| 8 |  |
| 9 | 9 |
| 10 |  |

You can also have decrement in the loop by assign a negative value afte the **Step** keyword.

For example:

```
For i = 10 to 1 Step -2
    Cells(i, 1) = i
Next i
```

This looping process will print values with an increment of -2 starts from 10 on row  10, 8, 6, 4 and 2 on column one.



|  | A |
|---|---|
| 1 |  |
| 2 | 2 |
| 3 |  |
| 4 | 4 |
| 5 |  |
| 6 | 6 |
| 7 |  |
| 8 | 8 |
| 9 |  |
| 10 | 10 |

**Do While ... Loop**

You can use the **Do While ... Loop** to test a condition at the start of the loop.  It will run the loop as long as the condition is ture and stops when the condition becomes false.  For Example:

```
i = 1
 Do While i =< 10
    Cells(i, 1) = i
    i = i + 1
 Loop
```

This looping process yields the same result as in the **For ... Next** structures example.

One thing to be caution is that sometimes the loop might be a infinite loop.  And it happens when the condition never beomes false.  In such case, you can stop the loop by press **[ESC]** or **[CTRL]** + **[BREAK]**.

**Do Until ... Loop**

You can test the condition at the beginning of the loop and then run the loop until the test condition becomes true.

Example:

```
i = 1
Do Until i = 11
   Cells(i, 1) = i
   i = i + 1
Loop
```

This looping process yields the same result as in the **For ... Next** structures example.

**Do ... Loop While**

When you want to make sure that the loop will run at least once, you can put the test at the end of loop.  The loop will stop when the condition becomes false.   (compare this loop structure to the Do ... While Loop.)

For Example:

```
i = 1
```

```
 Do
    Cells(i, 1) = i
    i  = i + 1
 Loop While i < 11
```

This looping process yields the same result as in the **For ... Next** structures example.

---

### Do ... Loop Until

This loop structure, like the **Do ... Loop While**, makes sure that the loop will run at least once, you can put the test at the end of loop.  The loop will stop when the condition becomes true. (compare this loop structure to the **Do ... Until** Loop.)

For Example:

```
i = 1
 Do
    Cells(i, 1) = i
    i  = i + 1
 Loop Until i = 11
```

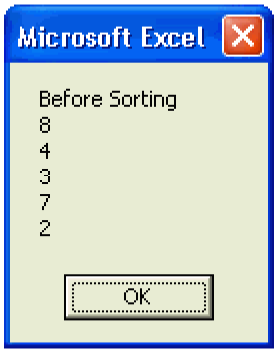
This looping process yields the same result as in the **For ... Next** structures example.

---

### Sorting Numbers In an Array

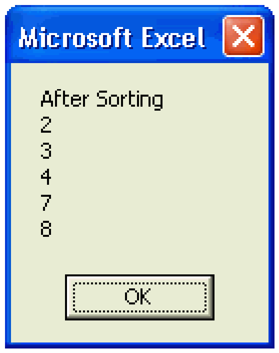
Sorting plays a very importance role in simulation.  The sorting procedure in this example is used in many ot the tutorial on this site.  The following provides an example on how to call the Sorting sub procedure, passes the array to it, and returns the array with sorted elements.

The sub getSort procedure calls the Sort sub procedure, pass arr( ) to it, and then get a sorted array back.  The two message boxes are used to display the array before and after sorting.



```
Before Sorting
8
4
3
7
2
```

This message box shows the array before sorting


```
After Sorting
2
3
4
7
8
```

This message box shows the array after sorting

```
Sub getSort( )
    Dim arr(5) As Integer
    Dim str As String

    arr(1) = 8
    arr(2) = 4
    arr(3) = 3
    arr(4) = 7
    arr(5) = 2
    str = ""

    For i = 1 To 5
        str = str & arr(i) & vbCrLf
    Next i

    MsgBox "Before Sorting" & vbCrLf & str

    Call Sort(arr)

    str = ""
    For i = 1 To 5
        str = str & arr(i) & vbCrLf
    Next i
    MsgBox "After Sorting" & vbCrLf & str

 End Sub

Sub Sort(arr( ) As Integer)

    Dim Temp As Double
    Dim i As Long
    Dim j As Long

    For j = 2 To UBound(arr)
        Temp = arr(j)
        For i = j - 1 To 1 Step -1
            If (arr(i) <= Temp) Then GoTo 10
            arr(i + 1) = arr(i)
        Next i
        i = 0
10      arr(i + 1) = Temp
    Next j
```

```
End Sub
```

## Find Maximum and Minimum Values in an Array

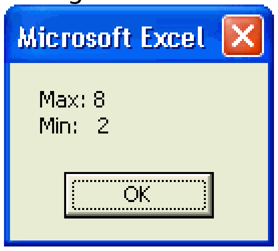
In order to find the maximum and the minimum values in an array, the array needs to be sorted. Once it is sorted, finding the maximum and minumum is very simple. Using the prior example to get the maminum and the minimun, you can simplely assign the upper bound index and 1, respectively to the sorted array following:

```
    arr(UBound(arr))
    arr(1)
```

Note that UBound(arr) will be 5 since there is 5 elements (start from index 1) in the array. We use 1 as the lowest index since we did not assign any value to index 0.

The following shows the maximum and the minimum of the array.

```
    MsgBox "Max: " & arr(UBound(arr)) & vbCrLf & "Min:   " & arr(1) & vbCrLf
```



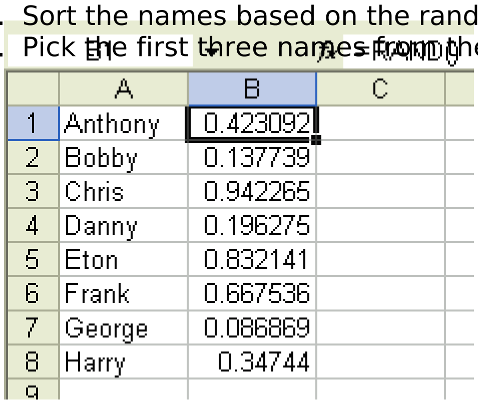
```
Max: 8
Min:   2
```

## Double Sorting - The secret of Resampling Without Replacement

Double Sorting is the word I used for sorting one array based on the values of the second array. This method is used when you want to get values from of a sample without select the same value twice (i.e. the Lotto example). The following demonstrates how this is done.

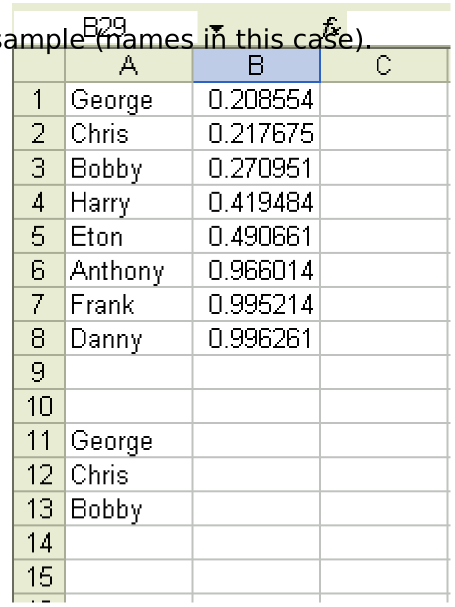
Assume you want to pick 3 people out of 8 randomly. The challenge is that if you pick them randomly, one of the names might get picked twice or even 3 times. To handle this challenge, the following steps can be taken:

1. Assign random number to each of the elements in the sample (names in this case).
2. Sort the names based on the random numbers.
3. Pick the first three names from the result.



E1 fx =RAND()

| | A | B | C |
|---|---|---|---|
| 1 | Anthony | 0.423092 | |
| 2 | Bobby | 0.137739 | |
| 3 | Chris | 0.942265 | |
| 4 | Danny | 0.196275 | |
| 5 | Eton | 0.832141 | |
| 6 | Frank | 0.667536 | |
| 7 | George | 0.086869 | |
| 8 | Harry | 0.34744 | |
| 9 | | | |



B29 fx

| | A | B | C |
|---|---|---|---|
| 1 | George | 0.208554 | |
| 2 | Chris | 0.217675 | |
| 3 | Bobby | 0.270951 | |
| 4 | Harry | 0.419484 | |
| 5 | Eton | 0.490661 | |
| 6 | Anthony | 0.966014 | |
| 7 | Frank | 0.995214 | |
| 8 | Danny | 0.996261 | |
| 9 | | | |
| 10 | | | |
| 11 | George | | |
| 12 | Chris | | |
| 13 | Bobby | | |
| 14 | | | |
| 15 | | | |

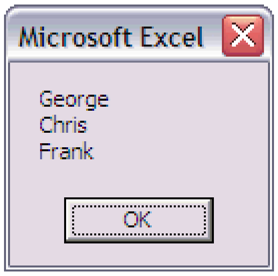
As in this case, George, Chris, and Bobby are selected since they are the first 3 names after sorting.

The following shows the example using VBA codes:

```
Sub Resample()
    Dim i As Long
    Dim Hold(8) As Single, Hold2(8) As String
    Dim str As String

    Hold2(1) = "Anthony"
    Hold2(2) = "Bobby"
    Hold2(3) = "Chris"
    Hold2(4) = "Danny"
    Hold2(5) = "Eton"
    Hold2(6) = "Frank"
    Hold2(7) = "George"
    Hold2(8) = "Harry"

    For i = 1 To UBound(Hold)
        Hold(i) = Rnd
        Cells(i, 2) = Hhold(i)
    Next i

    Call DoubleSort(Hold, Hold2)

    str = ""
    For i = 1 To 3
        str = str & Hold2(i) & vbCrLf
        Cells(i, 1) = Hold2(i)
    Next i

    MsgBox str

End Sub

Sub DoubleSort(x() As Single, y() As String)
    Dim xTemp As Double
    Dim yTemp As String
    Dim i As Long
    Dim j As Long

    For j = 2 To UBound(x)
        xTemp = x(j)
        yTemp = y(j)
            For i = j - 1 To 1 Step -1
                If (x(i) <= xTemp) Then GoTo 10
                x(i + 1) = x(i)
                y(i + 1) = y(i)
            Next i
        i = 0
10      x(i + 1) = xTemp
        y(i + 1) = yTemp
    Next j
```

```
End Sub
```

The DoubleSort sub procedure sorts array y (the names) based array x (the random numbers). The Resample sub procedure retruns three unique names from the sample in a message box.



```
George
Chris
Frank
```

**Excel VBA Simulation Basic Tutorial 102**

This page is the second part of the Excel VBA Simulation Basic Tutorial series.  It provides Excel VBA tutorials on how to create statistic estimates that are used to analyze the data from a simulation.  Many of the examples used are already available in Excel functions.  Users can use these Excel functions as tools to check against the results that come from the examples.  These examples require basic programming skills in VBA.  Users are encouraged to read Simulation Based Tutorial 101 if they have problem understanding the programming concepts and terms used on this page.

This document contains information about the following topics.

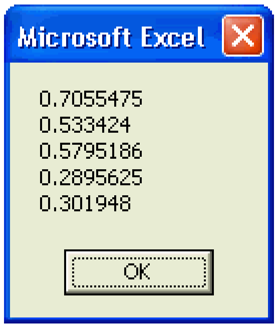
**Random Number and Randomize Statement**

To generate random number from 0 to 1 uniformly, one can use the Rand() function in Excel or the Rnd function in VBA.  These two functions are the mother of all random numbers.  You will need either one of these functions to generate random numbers from any probability distributions.

The following example generate 5 random numbers and then display them in a message box:

```
Sub rndNo()
   Dim str As String

   For i = 1 To 5
      str = str & CStr(Rnd) & vbCrLf
   Next i

   MsgBox str
End Sub
```

* CStr()
function converts the random numbers into string.



```
0.7055475
0.533424
0.5795186
0.2895625
0.301948
```

So far so good.  But when we close the file, reopen it, and run the sub routine again, the same 5 numbers come up!

The reason why this happens is that the random numbers were actually being generated from the same set of numbers (called seed).  By placing the **Randomize** statement in the sub routine, the numbers will be generated from a new seed.  (Randomize uses the return value from the Timer function as the new seed value.)

The new routine can be as followed:

```
Sub rndNo()
   Dim str As String

   Randomize
   For i = 1 To 5
      str = str & CStr(Rnd) & vbCrLf
   Next i

   MsgBox str
End Sub
```

Sometimes we might want to use the same seed over and over again by just changing the values of certain variables in our simulations to see how the change affects the outcomes.  In such case, omit the **Randomize** statement in your sub routine.

For more information, refer to Excel VBA Help in your Excel program.

---

**Standard Deviation and Mean**

Standard deviaiton and mean are the two mostly used statistic estimates of all times.  Mean is the average.  Standard deviation measures the 'spreadness' of the distribution.

$$\overline{X} = \text{Average} = \frac{\sum X_i}{n}$$

$$\sigma = \text{Standard Deviation} = \sqrt{\frac{\sum_{i=1}^{n} (X_i - \overline{X})^2}{n-1}}$$

The following are functions that compute mean and standard deviation.  These functions are similar to other  functions used in our examples; they take array as their arguments.

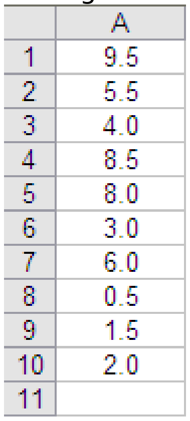
```
Function Mean(Arr() As Single)
```

```
     Dim Sum As Single
     Dim i As Integer

    Sum = 0
    For i = 1 To UBound(Arr)
        Sum = Sum + Arr(i)
    Next i

    Mean = Sum / UBound(Arr)
End Function

Function StdDev(Arr() As Single)
     Dim i As Integer
     Dim avg As Single, SumSq As Single

     avg = Mean(Arr)
     For i = 1 To UBound(Arr)
          SumSq = SumSq + (Arr(i) - avg) ^ 2
     Next i

     StdDev = Sqr(SumSq / (UBound(Arr) - 1))
End Function
```

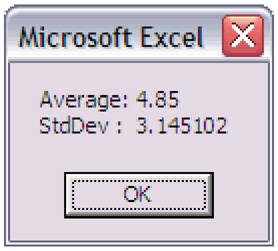
The following sub routine reads the data in column one from row 1 to 10 (of Sheet1) into the array, calls both functions by passing the arguements to them, computes the mean (average) and the standard deviation, then returns the values in a message box.

```
Sub compute()
     Dim Arr(10) As Single
     Dim Average As Single
     Dim Std_Dev As Single
     For i = 1 To UBound(Arr)
          Arr(i) = Sheets("Sheet1").Cells(i, 1)
     Next i
     Average = Mean(Arr)
     Std_Dev = StdDev(Arr)
     MsgBox "Average:" & vbTab & Average & vbCrLf & "StdDev :" & vbTab & Std_Dev
End Sub
```

The figures below show the data and the result.



|  | A |
|---|---|
| 1 | 9.5 |
| 2 | 5.5 |
| 3 | 4.0 |
| 4 | 8.5 |
| 5 | 8.0 |
| 6 | 3.0 |
| 7 | 6.0 |
| 8 | 0.5 |
| 9 | 1.5 |
| 10 | 2.0 |
| 11 |  |



```
Average: 4.85
StdDev :  3.145102
```

Similar example is also used in the Standard Deviation and Mean examples on the VBA section.

(These functions are similar to the **AVERAGE()** and the **STDEV()** functions provided by Excel.)

## Skewness and Kurtosis

**Skewness** measures the degree of asymmetry of a distribution. For example, the skewness of a normal distribution is 0 since a normal distribution is symmetric. Positive **skewness** indicates a distribution with an asymmetric tail extending toward more positive values, where as negative **skewness** extending toward more negative values.

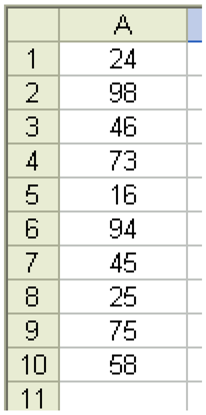
$$Skewness = \sum \frac{\left(X - \overline{X}\right)^3}{\delta} \times \frac{N}{(N-1)(N-2)}$$

**Kurtosis** measures the degree of peakedness or flatness of a distribution compared with normal distribution. Positive **kurtosis** indicates a relatively peaked distribution. Negative **kurtosis** indicates a relatively flat distribution.

$$Kurtosis = \left[\sum \frac{\left(X - \overline{X}\right)^4}{\delta} \times \frac{N(N+1)}{(N-1)(N-2)(N-3)}\right] - \frac{3(N-1)^2}{(N-2)(N-3)}$$

Often, these two estimates along with mean and standard deviation are used to test to see if the simulated data from a distribution is sound (if the data represents the distribution).

The following sub routine, compute(), reads the following data in column one from row 1 to 10 (of the active sheet) into the array,



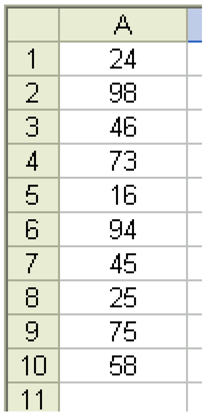
| | A |
|---|---|
| 1 | 24 |
| 2 | 98 |
| 3 | 46 |
| 4 | 73 |
| 5 | 16 |
| 6 | 94 |
| 7 | 45 |
| 8 | 25 |
| 9 | 75 |
| 10 | 58 |
| 11 | |

calls both functions by passing the arguements, computes the four moments (namely mean, standard deviation, skewness, and kurt) and returns the values in a message box.

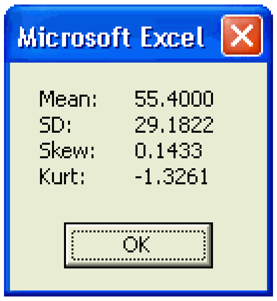
```
Sub compute()
    Dim arr(10) As Single

    For i = 1 To 10
        arr(i) = Cells(i, 1)
    Next i

    MsgBox "Mean:" & vbTab & Format(Mean(arr), "0.0000") & vbCrLf & _
    "SD:" & vbTab & Format(Var(arr) ^ 0.5, "0.0000") & vbCrLf & _
    "Skew:" & vbTab & Format(Skew(arr), "0.0000") & vbCrLf & _
    "Kurt:" & vbTab & Format(Kurtosis(arr), "0.0000")
```

```
End Sub

Function Skew(arr() As Single)
    Dim i As Long, n As Long
    Dim avg As Single, sd As Single, SumTo3 As Single

    n = UBound(arr)
    avg = Mean(arr)
    sd = (Var(arr)) ^ 0.5

    SumTo3 = 0
    For i = 1 To n
        SumTo3 = SumTo3 + ((arr(i) - avg) / sd) ^ 3
    Next i

    Skew = SumTo3 * (n / ((n - 1) * (n - 2)))
End Function

Function Kurtosis(arr() As Single)
    Dim i As Long, n As Long
    Dim avg As Single, sd As Single, SumTo3 As Single

    n = UBound(arr)
    avg = Mean(arr)
    sd = (Var(arr)) ^ 0.5

    SumTo4 = 0
    For i = 1 To n
        SumTo4 = SumTo4 + ((arr(i) - avg) / sd) ^ 4
    Next i

    Kurtosis = SumTo4 * (n * (n + 1) / ((n - 1) * (n - 2) * (n - 3))) - (3 * (n - 1) ^ 2 / ((n - 2) * (n - 3)))
End Function

Function Mean(arr() As Single)
    Dim Sum As Single
    Dim i As Long, k As Long

    k = UBound(arr)
    Sum = 0
    For i = 1 To k
        Sum = Sum + arr(i)
    Next i

    Mean = Sum / k
End Function

Function Var(arr() As Single)
    Dim i As Long
    Dim avg As Single, SumSq As Single

    k = UBound(arr)
    avg = Mean(arr)
    For i = 1 To k
        SumSq = SumSq + (arr(i) - avg) ^ 2
    Next i
    Var = SumSq / (k - 1)
End Function
```

The figures below show the data and the result.



| | A |
|---|---|
| 1 | 24 |
| 2 | 98 |
| 3 | 46 |
| 4 | 73 |
| 5 | 16 |
| 6 | 94 |
| 7 | 45 |
| 8 | 25 |
| 9 | 75 |
| 10 | 58 |
| 11 | |



```
Microsoft Excel

Mean:    55.4000
SD:      29.1822
Skew:    0.1433
Kurt:    -1.3261
```

(These functions are similar to the **SKEW()** and the **KURT()** functions provided by Excel.)

---

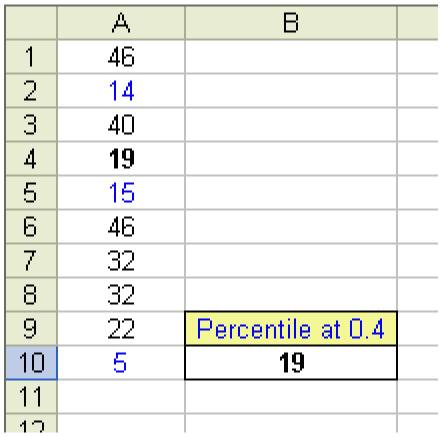
**Percentile and Confidence Interval**

Percentile returns the k-th percentile of values in a range. A confidence interval is the interval between two percentiles. For example: if a set of data has 20 numbers ranging from 2.5 to 50 with an increment of 2.5 (2.5, 5, ...., 50), the 80th percentile would be 40. This means that 80% of the elements from the set will be equal to or below than 40. If the alpha value is 10%, for a two tails test, the lower percentile should be set to 5% (alpha/2) and the upper percentile should be set to 95% (1 - alpha/2).

In order to get the percentile, the data needs to be sorted. In the sub routine (GetPercentile()) below, 10 random numbers between 1 to 50 are assigned to an array. The sub routine calls the percertile function (u_percentile()). The function calls the Sort sub routine to sort the array. The function gets the value from the array based on the percentile (40%), and returns the percentile value back to the sub routine.

Notice that Application.Max(Application.Min(Int(k * n), n), 1) in the percentile function makes sure that first, the array index is an integer and second, the maximum value and the minimum value for the array index will not excess the number of elements in the data set or below 1, respectively.

The data and the result are as followed:



| | A | B |
|---|---|---|
| 1 | 46 | |
| 2 | 14 | |
| 3 | 40 | |
| 4 | **19** | |
| 5 | 15 | |
| 6 | 46 | |
| 7 | 32 | |
| 8 | 32 | |
| 9 | 22 | Percentile at 0.4 |
| 10 | 5 | **19** |
| 11 | | |

The numbers in blue are below the 40% percentile. Nineteen (19), in this case, is the value that the function returns at 40% percentile.

Here is the complete program for the above example:

```
Sub GetPercentile()
    Dim arr(10) As Single

    For i = 1 To 10
        arr(i) = Int(Rnd * 50) + 1
        Cells(i, 1) = arr(i)
    Next i

    Cells(10, 2) = u_percentile(arr, 0.4)
End Sub

Function u_percentile(arr() As Single, k As Single)
    Dim i As Integer, n As Integer

    n = UBound(arr)
    Call Sort(arr)
    x = Application.Max(Application.Min(Int(k * n), n), 1)
    u_percentile = arr(x)
End Function

Sub Sort(ByRef arr() As Single)
    Dim Temp As Single
    Dim i As Long
    Dim j As Long

    For j = 2 To UBound(arr)
        Temp = arr(j)
        For i = j - 1 To 1 Step -1
            If (arr(i) <= Temp) Then GoTo 10
            arr(i + 1) = arr(i)
        Next i
        i = 0
10      arr(i + 1) = Temp

        If j Mod 100 = 0 Then
            Cells(26, 5) = j
        End If
    Next j
End Sub
```

Similar concept from this tutorial is used in many of our simulation examples.

(This function is similar to the **PERCENTILE()** and the **QUARTILE()** functions provided by Excel.)

---

**Profitablity**

The previous percentile example shows how to get the value that corresponds to a specific percentile. In this example, we will show you on how to get the percentile with a given value.

We are going to start this tutorial by showing you a very simple simulation. However, simulation is not necessary to get the answer in this example because we are using very loss assumptions. The result can actually be computed in your head if your math is that good.

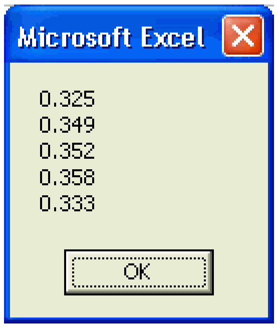
Assume your profit is distributed uniformly.  From the past records, you know that your annual average profit flucturates between -$100,000 to $500,000.  We want to know what is the probabilty that you will be making over $300,000 next year holding all other things constant. Interesting enough?  Now watch this:

1 - (300,000-(-100,000))/(500,000-(-100,000)) = 1 - 0.666 = 0.333

The probabilty will be 33%.

Now, let's run the simulation and see what will happen.

Five simulations were ran, each with 1000 iterations.  The result shows 5 probability values in a message box.  Each result is closed to the mathematic computation of 33%.



```
0.325
0.349
0.352
0.358
0.333
```

Here is the sub routine that runs the simulation:

```
Sub GetProb()
   Dim high As Single, low As Single, profit As Single
   Dim counter As Integer
   Dim str As String

   high = 500000
   low = -100000
   profit = 300000

   srt = ""
   For j = 1 To 5
      counter = 0
      For i = 1 To 1000
         If profit <= Rnd * (high - low + 1) + low Then
            counter = counter + 1
         End If
      Next i
      str = str & counter / 1000 & vbCrLf
   Next j

   MsgBox str
End Sub
```

This example is also implemented in the Monte Carlo Simulation tutorial.

(This function is similar to the **PERCENTRANK()** function provided by Excel.)

## Creating a Histogram

A histogram from a simulation shows the graphical representation of the derived probability distribution.
The following sub procedure is an improved model for generating a histogram. The first parameter, M, is the number of bins (breaks) that you want to have for the histogrm. The second parameter is the array that contains that values for the histogram.

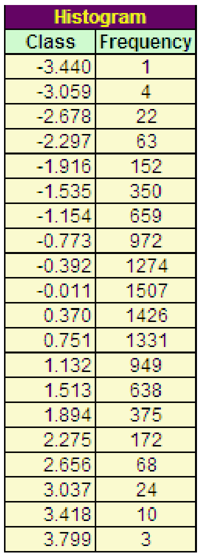
In order for this procedure to work properly, the array needs to be sorted for calling the histogram procedure. This way, the maximum and the minimum values can be derived and used for setting up the bin values. Please see the following examples for the implementation:

Normal Distribution Random Number Generator, Bootstrap - A Non-Parametric Approach, and Monte Carlo Simulation.

Here are the codes that generate a histogram:

```
Sub Hist(M As Long, arr() As Single)
    Dim i As Long, j As Long
    Dim Length As Single
    ReDim breaks(M) As Single
    ReDim freq(M) As Single

    For i = 1 To M
        freq(i) = 0
    Next i

    Length = (arr(UBound(arr)) - arr(1)) / M

    For i = 1 To M
        breaks(i) = arr(1) + Length * i
    Next i

    For i = 1 To UBound(arr)
        If (arr(i) <= breaks(1)) Then freq(1) = freq(1) + 1
        If (arr(i) >= breaks(M - 1)) Then freq(M) = freq(M) + 1
        For j = 2 To M - 1
            If (arr(i) > breaks(j - 1) And arr(i) <= breaks(j)) Then freq(j) = freq(j) + 1
        Next j
    Next i

    For i = 1 To M
        Cells(i, 1) = breaks(i)
        Cells(i, 2) = freq(i)
    Next i
End Sub
```

The following is an example output from the procedure:

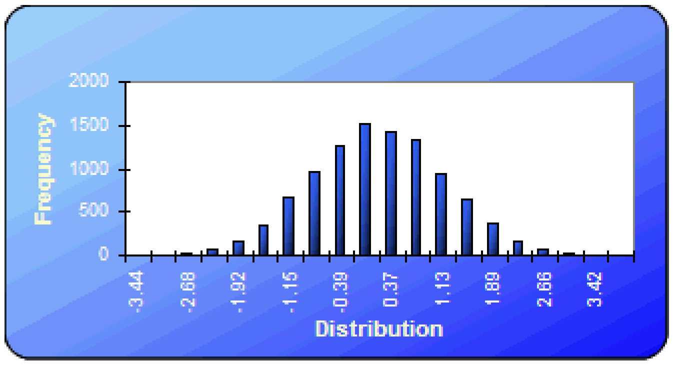


Histogram

| Class | Frequency |
|---|---|
| -3.440 | 1 |
| -3.059 | 4 |
| -2.678 | 22 |
| -2.297 | 63 |
| -1.916 | 152 |
| -1.535 | 350 |
| -1.154 | 659 |
| -0.773 | 972 |
| -0.392 | 1274 |
| -0.011 | 1507 |
| 0.370 | 1426 |
| 0.751 | 1331 |
| 1.132 | 949 |
| 1.513 | 638 |
| 1.894 | 375 |
| 2.275 | 172 |
| 2.656 | 68 |
| 3.037 | 24 |
| 3.418 | 10 |
| 3.799 | 3 |

The class is the bins or the breaks. The frequency contains the number of simulated values for each of the classes.

Here is the histogram chart from this example:



## Excel VBA Statistics and Mathematics Examples

This page contains simple Excel VBA Statistics and Mathematics examples.  Many of the examples used are already available in functions that come with Excel.  Users can use these Excel functions as tools to check against the results that come from the examples.  These examples require basic programming skills in VBA.  Users are encouraged to read the Simulation Based Tutorial 101 if they have problem understanding the programming concepts and terms used on this page.

This document contains information about the following topics.

### Finding Median

To find the median from an array, two steps are required.  First the array needs to be sorted (in either order), then a decision structure needs to be used.

Step 1.  Sort the array.  See example on sorting.

Step 2.  If the total elements in the array is an odd number (defined by Ubound(Arr) Mod = 1), then the median is the middle number (defined by Arr(Int(Ubound(Arr) / 2) + 1) ).
If the total elements in the array is an even number then take the average of the two middle
numbers.

```
Function u_median(Arr() As Single)    Call Sort(Arr)

   If UBound(Arr) Mod 2 = 1 Then
      u_median = Arr(Int(UBound(Arr) / 2) + 1)
   Else
      u_median = (Arr(UBound(Arr) / 2) + Arr(Int(UBound(Arr) / 2) + 1)) / 2
   End If
   End Function
```

This function is also implemented in the Bootstrap - A Non-Parametric Approach example.

(This function is similar to the **MEDIAN()** function provided by Excel.)

### Generate Random Numbers From Uniform Distribution

This function provides an uniform distribution random number between a specified range.

```
Function UniformRandomNumner(Low As Single, High As Single)

    UniformRandomNumner = Rnd * (High - Low + 1) + Low

End Function
```

For example, the following function returns a random number between 10 and 100:
UniformRandomNumner(**10**, **100**)

(This function is similar to the **RANDBETWEEN()** function provided by Excel.)

---

**Sum Numbers**

This function reads an array, and then returns the total number of the elements in the array.

```
Function u_sum(Arr() As Single)

  For i = 1 To UBound(Arr)
    u_sum = u_sum + Arr(i)
  Next i

End Function
```

Here is a sub routine that calls the u_sum function and returns the sum in a message box.

```
Sub computeSum()

  Dim arr(3) As Single
  arr(1) = 5
  arr(2) = 4
  arr(3) = 10

  MsgBox u_sum(arr)

End Sub
```

The message box will return 19.

(This function is similar to the **SUM()** function provided by Excel.)

---

**Compute Factorial**

To initiate the loop,  we assign u_fact, the function, an initial value of 1.  Then we multiple the new number (i) with the current number (u_fact) until i = Int(number).  Note that the Int function is require to make sure the number is an integer or becomes an integer.

```
Function u_fact(number As Single)
  u_fact = 1
  For i = 1 To Int(number)
    u_fact = u_fact * i
  Next i

End Function
```

For example, the following function returns a 6:
u_fact(**3**)

(This function is similar to the **FACT()** function provided by Excel.)

## Binomial Coeffieient

$$\binom{n}{j} = \frac{n!}{j!(n-j)!}$$

```
Function u_binoCoeff(n, j)

        Dim i As Integer
        Dim b As Double
        b = 1
        For i = 0 To j - 1
                b = b * (n - i) / (j - i)
        Next i
        u_binoCoeff = b

End Function
```

The following function compute all the possible combination on 5 items choosen from 10 items. This function returns 252:
u_binoCoeff(**5**, **10**)

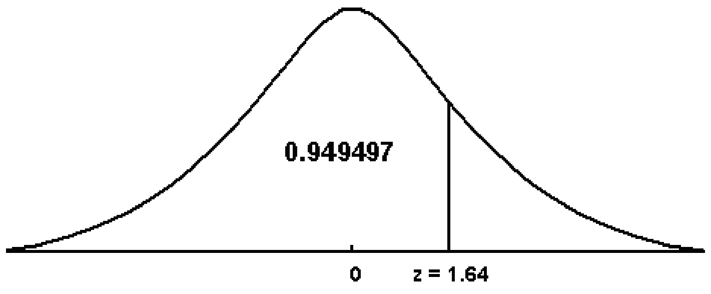
This function is also implemented in the Bootstrap - A Non-Parametric Approach example.

(This function is similar to the **COMBIN()** function provided by Excel.)

## Cumulative Standard Normal Distribution

This function computes the area under the left hand side of a specified value (the z value) from a standard normal distribution density function curve. In plain English, it returns the probabilty of X that is smaller than a specific value.

If you do not know what a normal curve looks like or have already forgotten about it, here is a sample:



In this example, the probabilty of X smaller than 1.64 (z) is 94.9497%.

```
Function u_SNorm(z)

    c1 = 2.506628
    c2 = 0.3193815
    c3 = -0.3565638
    c4 = 1.7814779
    c5 = -1.821256
    c6 = 1.3302744
    If z > 0 Or z = 0 Then
              w = 1
    Else: w = -1
    End If
    y = 1 / (1 + 0.2316419 * w * z)
    u_SNorm = 0.5 + w * (0.5 - (Exp(-z * z / 2) / c1) * _
            (y * (c2 + y * (c3 + y * (c4 + y * (c5 + y * c6))))))

End Function
u_SNorm(1.64) = 0.949497
```

This function is also implemented in the Black-Scholes Option Pricing Model - European Call and Put example.

(This function is similar to the **NORMSDIST()** function provided by Excel.)