

2-12: Functions and Procedures

You have already known how to create an empty Sub. We have also used some functions in the previous discussions. In fact writing functions and procedures serves as a systematic approach to writing programs. You use them to separate a large program into small pieces and manage them easily. Now we are going to study the structures of functions and procedures, and what you can do with them.

2-12-1: Procedures

Procedures are also called subroutines. Macros you have recorded are all subroutines. The syntax for a procedure or subroutine is:

```
[Private | Public] [Static] Sub name [(argList)]  
    [statements]  
End Sub
```

- **Private | Public:** Optional element. You can choose either **Private** or **Public** here. **Public** indicates that the procedure is accessible to all other procedures in all modules. On the other hand, when **Private** is selected, the procedure is accessible only to other procedures in the module where it is declared. If omitted, the procedure is **Public**.
- **Static:** Optional element. It is to indicate that the procedure's local variables are preserved between calls of the procedure. If omitted, the values of local variables will be reset each time the procedure ends.
- **name:** Required element. The name of the procedure. The name has the same constraint as a standard variable.
- **argList:** Optional list of arguments or parameters. Arguments / parameters are variables which will be served as an input from the caller when calling the procedure.
- **statements:** a block of valid statements to be executed when the procedure is called.

The argument list should be a list of variables with the type of each variable, separated by commas.

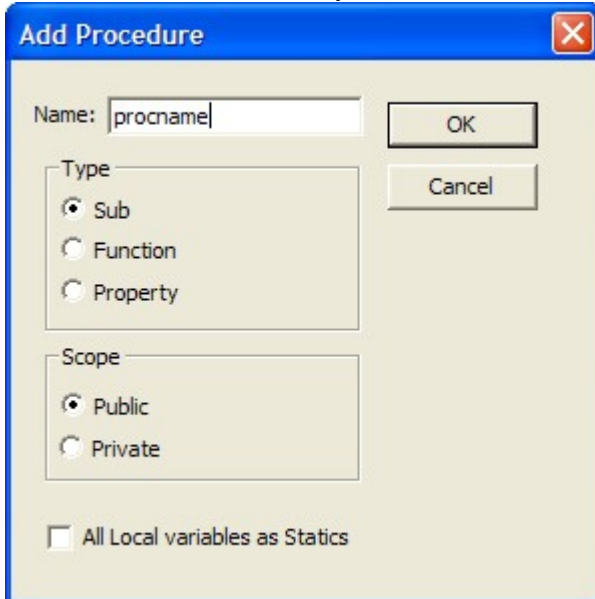
Examples of procedure declarations:

```
Sub demo1()  
    ' Declare Local Variables  
    ' Write your codes  
End Sub
```

```
Sub demo2(item1 As Integer, item2 As String, item3 As Double)  
    ' Declare Local Variables  
    ' Write your codes  
End Sub
```

```
Private Static Sub demo3(item1, item2)  
    ' Declare Local Variables  
    ' Write your codes  
End Sub
```

You can also insert an empty procedure by clicking Insert > Procedures.... You can choose between the **Private** and **Public** keyword, and also choose whether to add the **Static** keyword.



You can call your procedures in other procedures or functions. The syntax for calling a procedure is:

```
procname [arglist]
```

or

```
Call procname[(arglist)]
```

For example,

```
Sub macro1()  
  demo1  
  Call demo2(1, "2", 3#)  
  demo3 "a", #12/1/2008#  
End Sub
```

For the argument list, you should input your variables in the order as they are declared. Otherwise, you can use the “named argument” style to input them. For example,

```
Sub macro2()  
  demo2 item3:=3#, item1:=1, item2:="2"  
  Call demo3(item2:=#12/1/2008#, item1:="a")  
End Sub
```

In this way you can swap the order of the arguments.

One last note for this section is that, if you want to exit the procedure at the middle of your statements, you can use the statement `Exit Sub` at the point where you want to exit.

Let's try! Corner

Create a procedure to output (as a message box) the price of an n -year discount bond with face value R , n -year spot rate of i effective per annum. (Formula: $PV = R(1 + i)^{-n}$)

Then, create another procedure. Holding R fixed to 100, ask the user to input k different n 's and i 's (You need to ask the user to input k in advance). Call the pricing procedure to output the price immediately after each pair of n and i is inputted.

2-12-2: Functions

You can use functions to return a value after a series of statements. You have seen `InputBox` and `MsgBox` earlier. They are both functions. (Recall that `InputBox` function returns a string of the user input, and `MsgBox` returns an integer indicating the type of button the user has pressed.)

The syntax of a function is very similar to that of a procedure:

```
[Private | Public] [Static] Function name [(arglist)] [As type]  
  [statements]  
  [name = returnValue]  
End Sub
```

One added part in the function declaration compared with the procedure declaration is that you can optionally specify the output type of the function after the argument list. If omitted, the output type will be `Variant`.

Another added part is that you can optionally include an assignment statement in the function. (It is often at the end of the function, but not required. You can also put several such statements in an If-Then-Else structure, for example, to return different values in different situations.)

The shorthand form of declaration is something like this:

```
Function func#(a#, b#)
```

Some examples:

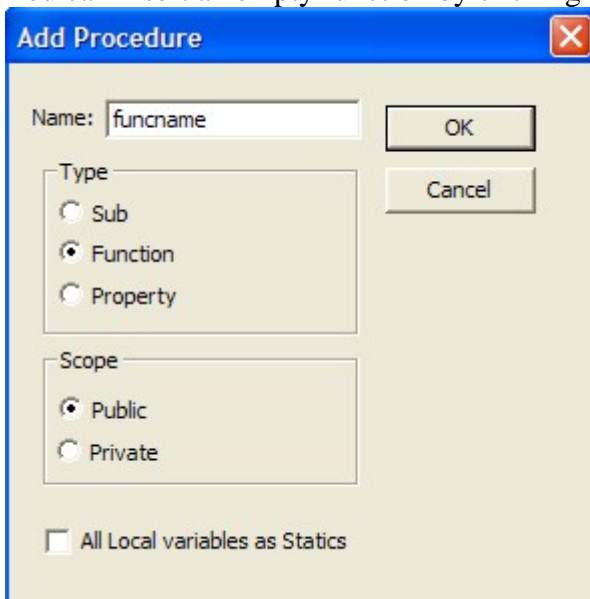
```
Function piOver180() As Double
    piOver180 = WorksheetFunction.pi / 180
End Function
```

```
Function actuallyASub(input1, input2) As Double
    ' nothing here
End Function
```

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

You can choose to exit a function prematurely by inserting the statement `Exit Function`.

You can insert an empty function by clicking `Insert > Procedures...` and choose the appropriate options.



Like procedures, you can call functions in other procedures and functions. You can use the function value returned like other simple values. You can also choose to discard the return value and treat the function as a procedure. For example:

```
Sub macro3()
    Dim someNumber As Double

    MsgBox piOver180 'outputs 1.7453...E-02
    MsgBox actuallyASub(1, 2) 'outputs 0

    actuallyASub 1, 2
    Call actuallyASub(1, 2)

    someNumber = sum(1.2, 3.4) 'stores 4.6
    MsgBox sum(sum(1.2, 3.4), -someNumber) 'outputs 0
End Sub
```

You will be able to use your function in worksheets. Treat them as normal functions and use “`=formulaName(parameters)`” to invoke the function.

Let's try! Corner

- 1 Change the previous pricing procedure (price of a zero coupon bond) into a function so that it returns the price (instead of outputting to a message box). Name the function ZeroPrice.
- 2 In an Excel worksheet, input R , n , i and calculate the zero coupon bond price by using the function. You should type something like this to use the function:

= ZeroPrice(100,10,5%)

	A	B	C
1	R	100	
2	n	10	
3	i	5%	
4	ZeroPrice	=ZeroPrice(B1,B2,B3)	

2-12-3: Passing by Value vs Passing by Reference

You can pass your parameters by value or by reference. Passing by value means that when a variable is passed to the function or procedure, a copy of the value is passed. Passing by reference means the original variable is passed. The difference between these two methods of passing is that, if you pass by reference, your original variable may be changed in the procedure or function; whereas if you pass by value, the original variable cannot be changed.

VBA by default pass your variables by reference. If you want to pass by value, you need to add `ByVal` keyword before the parameter name.

Study the following example.

```
Sub PassByValue(ByVal var1 As Boolean)
    var1 = True
End Sub

Sub PassByReference(var1 As Boolean)
    var1 = True
End Sub

Sub ch2_12_3()
    Dim booleanVar1 As Boolean, booleanVar2 As Boolean
    booleanVar1 = False
    booleanVar2 = False
    PassByValue booleanVar1
    PassByReference booleanVar2
    MsgBox booleanVar1 'outputs False
    MsgBox booleanVar2 'outputs True
End Sub
```

2-12-4: Optional Arguments

You can also set the argument as optional, i.e., user may choose to skip the parameter. Use the **Optional** keyword before the parameter you wish to set optional. Programmers may choose to provide a default value to the optional parameter.

```
Sub OptionalParameter1(Optional var1 As Integer)
    MsgBox var1
End Sub

Sub OptionalParameter2(Optional var1 As Integer = 100)
    MsgBox var1
End Sub

Sub ch2_12_4_1()
    OptionalParameter1 10 'Outputs 10
    OptionalParameter1 'Since the parameter has no default value,
                        'the default value of integer, 0, is outputted.
    OptionalParameter2 20 'Outputs 20
    OptionalParameter2 'Outputs 100
End Sub
```

Note that, you can provide some required parameters as well as some optional parameters. However, all the required parameters must be declared preceding the optional parameters.

```
Sub OptionalParameter3(var1 As Integer, Optional var2 As Integer)
```

is correct, but

```
Sub OptionalParameter4(Optional var1 As Integer, var2 As Integer)
```

is not correct.

2-13: Scoop of Variables

There are generally two scoops: module-wide and procedure-wide. Any global variables (declared outside any procedures) are module-wide, and any local variables and parameters (declared within procedures or functions) are procedure-wide.

We can declare a module-wide variable or constant as **Public** or **Private**. The concept is the same as that of procedures: **Public** variables can be accessed by other modules whereas **Private** variables cannot. The default is **Public**. You declare a **Public** and a **Private** variable like this:

```
Public var1 As Integer
Private var2 As Integer
```

This is similar for constants.

```
Public Const const1 As Integer = 1
Private Const const2 As Integer = 2
```

```

Dim var1
Dim var2

Sub sub1()
  Dim var1
  Dim var2
  var1 = 10
  var2 = 20
  MsgBox var1
  MsgBox var2
End Sub

Sub sub2()
  MsgBox var1
  MsgBox var2
  var1 = 50
  var2 = 60
End Sub

Sub sub3(var1, var2)
  MsgBox var1
  MsgBox var2
  var1 = 100
  var2 = 200
End Sub

Sub mainProgram()
  var1 = 1
  var2 = 2

  sub1      ? outputs 10,20
  MsgBox var1 ? outputs 1
  MsgBox var2 ? outputs 2

  sub2      ? outputs 1,2
  MsgBox var1 ? outputs 50
  MsgBox var2 ? outputs 60

  sub3 var1, var2 ? outputs 50,60
  MsgBox var1      ? outputs 100
  MsgBox var2      ? outputs 200

  sub3 var2, var1 ? outputs 200,100
  MsgBox var1      ? outputs 200
  MsgBox var2      ? outputs 100
End Sub

```

var1 and var2 are procedure-wide

var1 and var2 are procedure-wide

var1 and var2 are module-wide

For local variables, normally the value of a variable will be destroyed if the procedure has ended. However, if you wish to preserve its value, you can use the **Static** keyword to declare the variable. For example,

```

Sub RunningTotal(intValue)
  Static intTotal As Integer
  intTotal = intTotal + intValue
  MsgBox intTotal
End Sub

```

If you want every local variable in the procedure to be static, you can just add the **Static** keyword with the procedure. For example,

```

Static Sub RunningTotal2(intValue)
  Dim intTotal As Integer
  intTotal = intTotal + intValue
  MsgBox intTotal
End Sub

```

2-14: Recursive Function Calls (Advanced Topic)

A procedure or function can call itself. This action is called recursive procedure call. This is particularly useful if you can specify some recursive relationships between related variables. One example is the factorial function: $n! = n(n-1)(n-2)\dots 3 \cdot 2 \cdot 1$. We can write $n! = n \cdot (n-1)!$ with initial condition $0! = 1$. We can construct the recursive function like this:

```
Function Fact(x As Integer)
  If x < 0 Then
    Err.Raise 5 'Invalid argument
    Exit Function
  End If
  If x = 0 Then Fact = 1 Else Fact = x * Fact(x - 1)
End Function
```

Let's try! Corner

1 Construct a recursive function to calculate the Fibonacci number sequence defined by:

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n > 1 \end{cases}$$

2 Reconstruct the program using For loop.

3 Compare the complexity of the program code of the two functions. Which one is shorter and easier to write?

4 Compare the computational time of the two functions around $n = 30$. Which one is faster?

2-15: Arrays

An array is a collection of variables. You access variables through indices so that it provides a way to loop through and process different groups of related variables.

2-15-1: One-Dimensional Arrays

You can declare a one-dimensional array like this:

```
Dim varname(upperIndex) As vartype
```

In this way the array will contain spaces to hold *upperIndex* + 1 variables of *vartype*. You can access the variables with *varname(0)*, *varname(1)*, ..., *varname(upperIndex)*.

Note that the lower index is 0. If you prefer something other than 0, you can declare like this:

```
Dim varname(lowerIndex to upperIndex) As vartype
```

Arrays are looped in a style like this:

```
Dim intArray(9) As Integer, i As Integer
```

```
For i = 0 To 9  
    intArray(i) = i  
Next i
```

Also, recall that we can use For Each...Next loop to run through the array.

Let's try! Corner

Use the For Each...Next loop to initialize a one-dimensional array with values of all the elements be -1.

An element in an array can also contain other arrays:

```
Dim arrayInArray(3) As Variant  
  
arrayInArray(0) = Array(1, 2, 3)  
arrayInArray(1) = Array(2, 3)  
arrayInArray(2) = Array("a", "b")  
arrayInArray(3) = Array(#1/1/1900#, True)
```

In the above example, the function `Array()` returns an array constructed from the data provided.

To determine the upper limit and lower limit of the array index, use the function `UBound(arrayVar)` and `LBound(arrayVar)`.

2-15-2: Multidimensional Arrays

You can declare a multidimensional array like this:

```
Dim varname(upperIndex1, upperIndex2, ..., upperIndexN) As vartype
```

or

```
Dim varname(lowerIndex1 to upperIndex1, lowerIndex2 to upperIndex2, ..., _  
lowerIndexN to upperIndexN) As vartype
```

Typically you declare 1 or 2 dimensional arrays.

You can check the upper bound and lower bound of the *i*th index by `UBound(arrayVar, i)` and `LBound(arrayVar, i)`. For example, to check the upper bound of the 3rd column of an array named `stuScores`, we use `UBound(stuScores, 3)`.

2-15-3: Dynamic Arrays

If you want to declare an array where the boundary is not fixed, you may first declare an array without a boundary:

```
Dim varname() As vartype
```

Next, you use the `ReDim` statement to redimension the array to a certain size.

```
ReDim varname(newUpperIndex)
```

or

```
ReDim varname(newLowerIndex To newUpperIndex)
```

Normally, `ReDim` statement resets the elements in the array. If, after some data is stored in the array, you wish to increase the number of elements of the array, you can use the `Preserve` keyword to make the array preserve its values.

```
ReDim Preserve varname(newUpperIndex)
```

Note that you cannot preserve the elements in an array if you decrease the number of elements.

2-15-4: Passing Arrays into Procedures and Functions

You can pass arrays in procedures and functions. For example,

```
Sub aSub(anArray() As Integer, anotherArray() As Integer)
```

You can also return an array from a function (note the parenthesis at the return type):

```
Function aFunction(anArray() As Integer) As Integer()
    Dim anotherArray() As Integer
    ' Do some works
    aFunction = anotherArray
End Function
```

2-15-5: Indefinite Number of Arguments (Advanced Topic)

You can specify that a procedure or function will accept an arbitrary number of arguments. Use the `ParamArray` together with a Variant-type array parameter to do so. The `ParamArray` argument must be put at the end of the parameter list.

```
Function Sum(ParamArray intNums())
    Dim x, y
    For Each x In intNums
        y = y + x
    Next x
    Sum = y
End Function

Sub ch2_15_5()
    MsgBox Sum(2, 1, 5, 5, 7) 'outputs 20
End Sub
```

Let's try! Corner

In the regression analysis (or a “best fit line” problem), we wish to find a regression equation:

$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \dots + \hat{\beta}_r x_r$, where x_1, x_2, \dots, x_r are vectors containing data, such that the equation is

“best fit” to all the data. The solution to $\hat{\beta} = (\hat{\beta}_0, \hat{\beta}_1, \hat{\beta}_2, \dots, \hat{\beta}_r)^T$ can be found by $\hat{\beta} = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{y}$, where

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \text{ and } \mathbf{A} = \begin{pmatrix} 1 & x_{11} & \cdots & x_{r1} \\ 1 & x_{12} & \cdots & x_{r2} \\ \vdots & \vdots & & \vdots \\ 1 & x_{1n} & \cdots & x_{rn} \end{pmatrix}.$$

For example, given the data,

x	0	3	6
y	1	4	5

$$\text{We compute } \hat{\beta} = \left[\begin{pmatrix} 1 & 1 & 1 \\ 0 & 3 & 6 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 1 & 3 \\ 1 & 6 \end{pmatrix} \right]^{-1} \begin{pmatrix} 1 & 1 & 1 \\ 0 & 3 & 6 \end{pmatrix} \begin{pmatrix} 1 \\ 4 \\ 5 \end{pmatrix} = \begin{pmatrix} \frac{4}{3} \\ \frac{1}{3} \end{pmatrix},$$

and so the regression line is $\hat{y} = \frac{4}{3} + \frac{1}{3} x$.

We can use a 2-dimensional array to represent a matrix.

- 1 Write a function to return the coefficients of the regression with inputs y , x_1 , x_2 , ..., x_n . You can use `WorksheetFunction.Transpose(matrix)` to return the transpose of a matrix, and `WorksheetFunction.Inverse(matrix)` to return the inverse of a matrix. For matrix multiplication, you can use `WorksheetFunction.MMult(matrix1, matrix2)` to do so.
- 2 Test your function using the data from the example above.

2-15-5: Passing procedure names into another procedure (Advanced Topic)

You can pass procedure names into a function and call the function. The trick is, you pass the name as string, and use `Application.Run` function to invoke the function.

The `Application.Run` function has the following syntax:

*Function Application.Run(Optional Macro As Variant, _
Optional ParamArray Arg() As Variant) As Variant*

- *Macro*: Optional Variant. The macro to run. This can be either a string with the macro name, a Range object indicating where the function is, or a register ID for a registered DLL (XLL) function.
- *Arg()*: Optional Variant. The arguments to be passed into the macro.

Let's try! Corner

We try to write a function to numerically integrate a function in a range, i.e. to approximate $\int_a^b f(x)dx$.

The computer is not capable to do real integration. Therefore, a numerical method can be adopted as follows:

$\int_a^b f(x)dx \approx \sum_{x \in (a,b)} f(x)\Delta x$. In our illustration, we can take the increment $\Delta x = 1/100000$.

- 1 Using For...Next loop with `Step` equal to $1/100000$, together with the `Application.Run` function, write the `numericalIntegrate` function which takes in a custom function name, a lower bound (a) and an upper bound (b).
- 2 Construct some custom functions and test the `numericalIntegrate` function. (You can try integrating

$\sin x$, e^x , polynomials, etc.)

Exercise 2

- Write a program to calculate $\sum_{n=1}^{100} \binom{n+k-1}{n} (1-p)^n p^k$, where $k = 50$, $p = 0.3$, $\binom{n}{r} = \frac{n!}{r!(n-r)!}$. (Ans: 0.2101)

- Suppose the credit risks of 10 types of bonds (Call them “Bond 1”, ..., “Bond 10”) are uniformly distributed in $[0,0.2)$. First, simulate the credit risk of each of the bonds. Then, according to the following table, generate a single report in a message box showing all the credit ratings of the bonds.

Credit Risks	Rating
[0,0.005)	AAA
[0.005,0.02)	AA
[0.02,0.05)	A
[0.05,0.1)	B
[0.1,0.2)	C

- Write a function to calculate the dot product of two vectors. (The dot product of two $n \times 1$ vectors \mathbf{x} and \mathbf{y} is $\mathbf{x} \cdot \mathbf{y} = \mathbf{x}^T \mathbf{y}$).
- Write a function to calculate the semi-variance for a sample size n . The formula is:

$$sv^2 = \sum_{i=1}^n \frac{(\max(x_i - \bar{x}, 0))^2}{n}$$

- Write a function to calculate the Black-Scholes Call Price:

$$C(S_0, T, K, r, \delta, \sigma) = S_0 e^{-\delta T} \Phi(d_1) - K e^{-rT} \Phi(d_2), \text{ where } d_1 = \frac{\ln(S_0/K) + (r - \delta + \frac{1}{2} \sigma^2) T}{\sigma \sqrt{T}},$$

$d_2 = d_1 - \sigma \sqrt{T}$, Φ is the standard normal cumulative distribution function. You can use `WorksheetFunction.NormSDist(d)` to calculate the standard normal cumulative distribution.

6. Write a function to find the solution of the following system of linear equations:

$$\begin{cases} a_1x + a_2y + a_3z = a_0 \\ b_1x + b_2y + b_3z = b_0 \\ c_1x + c_2y + c_3z = c_0 \end{cases}$$

7. Make a bubble sort function to sort an array of numbers or strings in increasing order.
Step-by-step example of bubble sort:

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort algorithm. In each step, elements written in bold are being compared.

First Pass:

(**5** 1 4 2 8) to (**1** 5 4 2 8) Here, algorithm compares the first two elements, and swaps them.

(1 **5** 4 2 8) to (1 4 5 2 8)

(1 4 **5** 2 8) to (1 4 2 5 8)

(1 4 2 **5** 8) to (1 4 2 5 8) Now, since these elements are already in order, algorithm does not swap them.

Second Pass:

(1 4 2 5 8) to (1 4 2 5 8)

(1 4 2 5 8) to (1 2 4 5 8)

(1 2 4 5 8) to (1 2 4 5 8)

Now, the array is already sorted, but our algorithm does not know if it is completed. Algorithm needs one whole pass without any swap to know it is sorted.

Third Pass:

(1 2 4 5 8) to (1 2 4 5 8)

(1 2 4 5 8) to (1 2 4 5 8)

Finally, the array is sorted, and the algorithm can terminate.

8. Use recursion technique to write a binary search function to find the index of a value in an array.

The binary search is a method to search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

An algorithm:

```
BinarySearch(A[0..N-1], value, low, high) {
    if (high < low)
        return -1 // not found
    mid = (low + high) / 2
    if (A[mid] > value)
        return BinarySearch(A, value, low, mid-1)
    else if (A[mid] < value)
        return BinarySearch(A, value, mid+1, high)
    else
        return mid // found
}
```