'-----------------------------------------------------------------------------------------------------------------------------------

'Last Update: Sept 25, 2013

'-----------------------------------------------------------------------------------------------------------------------------------

Option Explicit 'Requires that all variables to be declared explicitly.

'-----------------------------------------------------------------------------------------------------------------------------------

Option Base 1

'Used at module level to declare the default lower bound for array subscripts.

'Because the default base is 0, the Option Base statement is never required. If used, the statement

'must appear in a module before any procedures. Option Base can appear only once in a module and must

'precede array declarations that include dimensions. Note that the to clause in the Dim, Private, Public,

'ReDim, and Static statements provides a more flexible way to control the range of an array's subscripts.

'However, if you don't explicitly set the lower bound with a To clause, you can use Option Base to change

'the default lower bound to 1. The base of an array created with the the ParamArray keyword is zero; Option

'Base does not affect ParamArray (or the Array function, when qualified with the name of its type library, for example VBA.Array).

'The Option Base statement only affects the lower bound of arrays in the module where the statement is located.

'-----------------------------------------------------------------------------------------------------------------------------------

'-----------------------------------------------------------------------------------------------------------------------------------

'-----------------------------------------------------------------------------------------------------------------------------------

'Lesson 3.2.1. Writing Simple Subroutines & User Defined Functions

'-----------------------------------------------------------------------------------------------------------------------------------

'-----------------------------------------------------------------------------------------------------------------------------------

'A User Defined Function (functions that are called directly from worksheet cells) is a Function procedure that typically (but not

'necessarily) accepts some inputs and returns a result. A UDF can only return a value to the cell(s) whence it was called -- it

'must not modify the contents or formatting of any cell and must not modify the operating environment of Excel. If you attempt to

'change anything, the function will terminate immediately and return a #VALUE error to the calling cell. In Excel 97 and 2000, a

'UDF cannot use the Find method of a Range object, even though that method does not change anything in Excel. This was fixed with

'Excel 2002.

'Transferring data from VBA into a worksheet is one of the most frequent tasks carried out in VBA programming.

'Unortunately there isn't a built in Excel function that returned the address of a selected range. But it is possible to create

'the address using a combination of functions and text strings, but in my opinion a much simpler method is available through a very

'short piece of VBA:

Function RNG\_ADDRESS\_FUNC(ByRef SRC\_RNG As Range) As String

RNG\_ADDRESS\_FUNC = SRC\_RNG.Address 'return Absolute ranges; for relative SRC\_RNG.Address(False, False)

'See Relative And Absolute References In Formulas @ http://www.cpearson.com/excel/relative.aspx

End Function

Sub TEST\_RNG\_ADDRESS\_FUNC() 'subroutine

Dim i As Long

Dim NROWS As Long

Dim SRC\_WSHEET As Worksheet

Set SRC\_WSHEET = Worksheets("WSHEET\_TEST")

NROWS = 1000

With SRC\_WSHEET

With .Cells

.Clear

.ColumnWidth = 10

End With

'----------------------------------------------------------------------------------------------------------------------------------------

'The simplest way to transferring data content to a worksheet is to declare the variable as a range:

Dim DST\_RNG As Range 'or Dim DST\_RNG As Excel.Range

Set DST\_RNG = .Cells(1, 2) 'B1

'Any changes made to the DST\_RNG object will now change the worksheet range (since that is what

'the SRC\_RNG refers to), and this changed data will be available from any other routine or module

'within the project (or indeed from another project)

'The problem with this approach is that the transfer of data between VBA and the spreadsheet is very slow,

'and if the data in the range object changes frequently this can result in a significant loss of performance.

Dim SRC\_RNG As Range

Set SRC\_RNG = Range(.Cells(1, 1), .Cells(NROWS, 1)) 'A1 to A1000

For i = 1 To NROWS

.Cells(i, 1) = Rnd()

Next i

DST\_RNG.Value = RNG\_ADDRESS\_FUNC(SRC\_RNG)

'----------------------------------------------------------------------------------------------------------------------------------------

'We can algo get the address of a selected range. The Excel Indirect() function allows other functions to use

'the text in a worksheet cell to define a range (see http://www.cpearson.com/excel/indirect.htm), rather than selecting the range,

'or entering it directly into the function. For instance, if cell B1 contained the text $A$1:$A$1000, then the function

'=Sum(Indirect(B1)) would return the sum of the values in $A$1:$A$1000.

Set DST\_RNG = DST\_RNG.Offset(1, 0) 'B2

DST\_RNG.Formula = "=Sum(Indirect(B1))"

'DST\_RNG.Formula = "=Sum(INDIRECT\_VBA\_FUNC(B1))"

End With

End Sub

'Debug.Print INDIRECT\_VBA\_FUNC("$a$1").Address(True, False) & ":" & \_

INDIRECT\_VBA\_FUNC("$a$1000").Address(True, True)

Function INDIRECT\_VBA\_FUNC(ByVal ADDRESS\_STR As String) As Range

Set INDIRECT\_VBA\_FUNC = Range(ADDRESS\_STR)

End Function

'-----------------------------------------------------------------------------------------------------------------------------------

'-----------------------------------------------------------------------------------------------------------------------------------

'Lesson 3.2.2. Where To Put The Code

'-----------------------------------------------------------------------------------------------------------------------------------

'-----------------------------------------------------------------------------------------------------------------------------------

'The code for a UDF should be placed in a standard code module, not one of the Sheet modules and not

'in the ThisWorkbook module. In the VBA editor, go to the Insert menu and choose Module. This will

'insert a new code module into the project. A module can contain any number functions, so you can

'put many functions into a single code module. You can change the name of a module from Module1 to

'something more meaningful by pressing the F4 key to display the Properties window and changing the

'Name property to whatever you want.

'You can call a function from the same workbook by using just the function name. For example:

'=AVERAGE\_2N(0.015, 0.005)

Sub TEST\_PUT\_CODE\_SAME\_WORKBOOK()

Dim SRC\_WSHEET As Worksheet

Set SRC\_WSHEET = Worksheets("WSHEET\_TEST")

With SRC\_WSHEET

With .Cells

.Clear

.ColumnWidth = 10

End With

With .Range("A1")

.Formula = "=AVERAGE\_2N(0.015, 0.005)"

Debug.Print .Address(False, False), .Value, .Formula

End With

End With

End Sub

'It is possible, but strongly recommended against, to have two functions with the same name is two

'separate code modules within the same workbook. You would call them using the module name from cells

'with formulas like:

'=Module1.AVERAGE\_2N(Parameters)

'=Module2.AVERAGE\_2N(Parameters)

'Doing this will lead only to confusion, so just because it is possible doesn't mean you should do it.

'Don't do it. For further information see "Scope Of Variables And Procedures" @

'http://www.cpearson.com/excel/scope.aspx

'Do not give the same name to both a module and a function (regardless of whether that module contains

'that function). Doing so will cause an untrappable error.

'You can call a UDF that is contained in another (open) workbook by using the workbook name in the

'formula. For example, ='WorkbookName.xls'!AVERAGE\_2N(A1,A2) will call the function AVERAGE\_2N defined

'in the workbook WorkbookName.xls.

Sub TEST\_PUT\_CODE\_NEW\_WORKBOOK()

Dim SRC\_WBOOK As Workbook

Dim DST\_WBOOK As Workbook

Dim DST\_WSHEET As Worksheet

Set SRC\_WBOOK = ThisWorkbook

Workbooks.Add

Set DST\_WBOOK = ActiveWorkbook

Set DST\_WSHEET = DST\_WBOOK.ActiveSheet

With DST\_WSHEET

With .Cells

.Clear

.ColumnWidth = 10

End With

With .Range("A1")

.Formula = "=" & "'" & SRC\_WBOOK.Name & "'!AVERAGE\_2N(0.015, 0.005)"

Debug.Print .Address(False, False), .Value, .Formula

End With

End With

End Sub

'-----------------------------------------------------------------------------------------------------------------------------------

'-----------------------------------------------------------------------------------------------------------------------------------

'Lesson 3.2.3. Input Arguments

'-----------------------------------------------------------------------------------------------------------------------------------

'-----------------------------------------------------------------------------------------------------------------------------------

'Both functions and subroutines can take input arguments. The input argument list, often called the parameters, has its own syntax that

'requires separate consideration. For further reference see Lesson 3.3 ByRef and ByVal.

'We can declare as many input arguments with their respective data types as are needed, provided we separate each parameter with a

'comma. The basic syntax is to specify a local name for the value and a data type. For example, here is a simple function that prints

'out two numbers in the immediate window:

Private Function PRINT\_PRICES\_FUNC(ByVal m\_Price1 As Double, ByVal m\_Price2 As Double)

Dim PRINT\_FLAG As Boolean

On Error GoTo ERROR\_LABEL

PRINT\_FLAG = False

Debug.Print (m\_Price1)

Debug.Print (m\_Price2)

PRINT\_FLAG = True

PRINT\_PRICES\_FUNC = PRINT\_FLAG

Exit Function

ERROR\_LABEL:

PRINT\_PRICES\_FUNC = PRINT\_FLAG

End Function

'We then call the TEST\_PRINT\_PRICES\_FUNC subroutine, we specify the parameters after the name as follows:

Sub TEST\_PRINT\_PRICES\_FUNC()

Debug.Print PRINT\_PRICES\_FUNC(45.23, 65.54)

End Sub

'In this example, the values 45.23 and 65.54 are passed to the variables m\_Price1 and m\_Price2. Within the subroutine definition, the

'values passed in will be known by the local names m\_Price1 and m\_Price2. Since this is a subroutine, there is no return value as was

'the case in the Average() function. The output of this simple program will be: 45.23 and 65.54.

'-----------------------------------------------------------------------------------------------------------------------------------

'-----------------------------------------------------------------------------------------------------------------------------------

'Lesson 3.2.4. UDFs And Calculations

'-----------------------------------------------------------------------------------------------------------------------------------

'-----------------------------------------------------------------------------------------------------------------------------------

'As a general rule, you should pass into the function all the values it needs to properly calculate the result. That means

'that your UDF should not make explicit refences to other cells. If you reference other cells directly from within the

'function, Excel may not recalculate the function when that cell is changed. For example, a poorly written UDF is as

'follows:

Function APPLICATION\_VOLATILE\_FUNC1(ByVal m\_Return1 As Double, \_

Optional ByVal ADDRESS\_STR As String = "A1") As Double

APPLICATION\_VOLATILE\_FUNC1 = (m\_Return1 + Range(ADDRESS\_STR).Value) / 2

End Function

'In this function, the second return is assumed to be in cell A1. The problem here is that Excel doesn't

'know that this function depends on cell A1 and therefore will not recalculate the formula when A1 is changed.

'Thus, the cell calling the function call will not contain the correct result when cell A1 is changed. You can

'force Excel to recalculate a UDF whenever any calculation is made by adding the line Application.Volatile True

Function APPLICATION\_VOLATILE\_FUNC2(ByVal m\_Return1 As Double, \_

Optional ByVal ADDRESS\_STR As String = "A1") As Double

Application.Volatile True

APPLICATION\_VOLATILE\_FUNC2 = (m\_Return1 + Range(ADDRESS\_STR).Value) / 2

End Function

'This has the drawback, however, that the function is recalculated even if it doesn't need to be recalculated, which can cause a

'performance problem. In general, you shouldn't use Application.Volatile but instead design your UDF to accept as inputs everything

'it needs to properly caclulate the result.

'Or:

Function APPLICATION\_VOLATILE\_FUNC3(ByVal m\_Return1 As Double, \_

Optional ByVal ADDRESS\_STR As String = "A1", \_

Optional ByVal VOLAT\_FLAG As Boolean = False) As Double

If ADDRESS\_STR = "" Then

APPLICATION\_VOLATILE\_FUNC3 = m\_Return1

Else

Application.Volatile VOLAT\_FLAG

APPLICATION\_VOLATILE\_FUNC3 = (m\_Return1 + Range(ADDRESS\_STR).Value) / 2

End If

End Function

Sub TEST\_APPLICATION\_VOLATILE\_FUNC()

Dim i As Long

Dim DST\_RNG As Range

Dim SRC\_WSHEET As Worksheet

Set SRC\_WSHEET = Worksheets("WSHEET\_TEST")

With SRC\_WSHEET

With .Cells

.Clear

.ColumnWidth = 10

End With

.Cells(1, 1).Value = Rnd() 'A1

Debug.Print "APPLICATION\_VOLATILE\_FUNC1: False"

.Cells(1, 3).Formula = "=APPLICATION\_VOLATILE\_FUNC1(0.02, " & """" & "A1" & """" & ")"

Debug.Print "a1=" & .Cells(1, 1).Value, "f(a1)=" & .Cells(1, 3).Value

.Cells(1, 1).Value = Rnd() 'A1

Debug.Print "a1=" & .Cells(1, 1).Value, "f(a1)=" & .Cells(1, 3).Value

Debug.Print

Debug.Print "APPLICATION\_VOLATILE\_FUNC2: True"

.Cells(1, 3).Formula = "=APPLICATION\_VOLATILE\_FUNC2(0.02, " & """" & "A1" & """" & ")"

Debug.Print "a1=" & .Cells(1, 1).Value, "f(a1)=" & .Cells(1, 3).Value

.Cells(1, 1).Value = Rnd() 'A1

Debug.Print "a1=" & .Cells(1, 1).Value, "f(a1)=" & .Cells(1, 3).Value

Debug.Print

Debug.Print "APPLICATION\_VOLATILE\_FUNC3: False"

.Cells(1, 3).Formula = "=APPLICATION\_VOLATILE\_FUNC3(0.02, " & """" & "A1" & """" & ", False)"

Debug.Print "a1=" & .Cells(1, 1).Value, "f(a1)=" & .Cells(1, 3).Value

.Cells(1, 1).Value = Rnd() 'A1

Debug.Print "a1=" & .Cells(1, 1).Value, "f(a1)=" & .Cells(1, 3).Value

Debug.Print

Debug.Print "APPLICATION\_VOLATILE\_FUNC3: True"

.Cells(1, 3).Formula = "=APPLICATION\_VOLATILE\_FUNC3(0.02, " & """" & "A1" & """" & ", True)"

Debug.Print "a1=" & .Cells(1, 1).Value, "f(a1)=" & .Cells(1, 3).Value

.Cells(1, 1).Value = Rnd() 'A1

Debug.Print "a1=" & .Cells(1, 1).Value, "f(a1)=" & .Cells(1, 3).Value

'Union(.Cells(1, 1), .Cells(1, 3)).Clear

End With

End Sub

'-----------------------------------------------------------------------------------------------------------------------------------

'-----------------------------------------------------------------------------------------------------------------------------------

'Lesson 3.2.5. Returning Errors From Functions

'-----------------------------------------------------------------------------------------------------------------------------------

'-----------------------------------------------------------------------------------------------------------------------------------

'If you use VBA to create User Defined Functions (functions that are called directly from worksheet cells) in a module or add-in, you

'likely will need to return an error value under some circumstances. For example, if a function requires a positive number as a parameter

'and the user passes in a negative number, you should return a #VALUE error. You might be tempted to return a text string that looks

'like an error value, but this is not a good idea. Excel will not recognize the text string, for example #VALUE, as a real error, so

'many functions and formulas may misbehave, especially ISERROR, ISERR, and IFERROR, and ISNA. These functions require a real error

'value.

'VBA provides a function called CVErr that takes a numeric input parameter specifying the error and returns a real error value that

'Excel will recognize as an error. The values of the input parameter to CVErr are in the XLCVError Enum and are as follows:

' xlErrDiv0 (= 2007) returns a #DIV/0! error.

' xlErrNA (= 2042) returns a #N/A error.

' xlErrName (= 2029) returns a #NAME? error.

' xlErrNull (= 2000) returns a #NULL! error.

' xlErrNum (= 2036) returns a #NUM! error.

' xlErrRef (= 2023) returns a #REF! error.

' xlErrValue (= 2015) returns a #VALUE! error.

'The only legal values of the input parameter to CVErr function are those listed above. Any other value causes CVErr to return a #VALUE.

'This means, unfortunately, that you cannot create your own custom error values. In order to return an error value, the function's return

'data type must be a Variant. If the return type is any other data type, the CVErr function will terminate VBA execution and Excel will

'report a #VALUE error in the cell.

'Note that these errors are meaningful only to Excel and have nothing at all to do with the Err object used to work with runtime errors

'in VBA code.

'Example1: The following is a example using CVErr.

Sub Test\_CVErr\_Func()

Debug.Print CVErr\_Func(10)

Debug.Print CVErr\_Func(-10)

End Sub

Function CVErr\_Func(d As Double) As Variant

If d < 0 Then

CVErr\_Func = CVErr(xlErrValue)

Else

CVErr\_Func = d \* 10

End If

End Function

'This function will return a #VALUE! error if the input parameter is less than 0. Note that the return type of the function is Variant.

'You can also use CVErr to test whether a cell has a specific error value in it. However, you must first test whether the cell contains

'any sort of error, and then, if it does contain an error, test which type of error. For example,

Sub Test\_Error\_Value()

Dim SRC\_RNG As Range

Dim SRC\_WSHEET As Worksheet

Set SRC\_WSHEET = Worksheets("WSHEET\_TEST")

With SRC\_WSHEET

With .Cells

.Clear

.ColumnWidth = 10

End With

End With

Set SRC\_RNG = Range("A1")

SRC\_RNG.Offset(0, 1) = "x"

SRC\_RNG.Formula = "=3+B1" 'Create xlErrValue (= 2015 = #VALUE! error ) by adding a letter.

'Debug.Print SRC\_RNG.Value, SRC\_RNG.Formula, IsError(SRC\_RNG.Value)

If IsError(SRC\_RNG.Value) = True Then

If SRC\_RNG.Value = CVErr(xlErrValue) Then

Debug.Print "#VALUE error"

Else

Debug.Print "Some other error"

End If

End If

Set SRC\_RNG = Range("A2")

SRC\_RNG.Formula = "=B2+3"

Columns("B:B").Delete Shift:=xlToLeft 'Create xlErrRef (= 2023 = #REF! error) by removing Column B

'Debug.Print SRC\_RNG.Value, SRC\_RNG.Formula, IsError(SRC\_RNG.Value)

If IsError(SRC\_RNG.Value) = True Then

If SRC\_RNG.Value = CVErr(xlErrRef) Then

Debug.Print "#REF error"

Else

Debug.Print "Some other error"

End If

End If

Set SRC\_RNG = Range("A3")

SRC\_RNG.Formula = "=3/0" 'Create xlErrDiv0 (= 2007 = #DIV/0! error ) by / 0.

'Debug.Print SRC\_RNG.Value, SRC\_RNG.Formula, IsError(SRC\_RNG.Value)

If IsError(SRC\_RNG.Value) = True Then

If SRC\_RNG.Value = CVErr(xlErrDiv0) Then

Debug.Print "#DIV/0 error"

Else

Debug.Print "Some other error"

End If

End If

Set SRC\_RNG = Range("A4")

With SRC\_RNG

.Formula = "=UNKNOWN\_FUNC()" 'Create xlErrName (= 2029 = #NAME? error ) by typing the name of an unkwnown function.

'You can use CVErr in a Select Case statement to test the various error types. For example,

If IsError(.Value) = True Then

Select Case .Value

Case CVErr(xlErrValue)

Debug.Print "#VALUE error"

Case CVErr(xlErrDiv0)

Debug.Print "#DIV/0 error"

Case CVErr(xlErrName)

Debug.Print "#NAME? error"

Case Else

Debug.Print "Some other error"

End Select

End If

End With

'If you attempt to compare a cell's value to a value produced by CVErr, and the cell does not contain an error value, you will get a

'run-time error 13, Type Mismatch. For example, the following code will fail if A5 does not contain an error value.

Set SRC\_RNG = Range("A5")

If SRC\_RNG.Value = CVErr(xlErrDiv0) Then: Debug.Print "#DIV/0 error" ' error 13 if A5 has no error

End Sub

'You can also return an error value from a UDF if an incorrect input parameter is passed in. To do this, the function must return a

'Variant data type and use the CVErr function to create an error-type Variant result. For example, the function Divide\_Error\_Func

'below will return a #DIV/0 error if the divisor is 0.

Sub Test\_Divide\_Error\_Func()

Debug.Print Divide\_Error\_Func(3, 2)

Debug.Print Divide\_Error\_Func(3, 0)

End Sub

Function Divide\_Error\_Func(A As Double, B As Double) As Variant

If B = 0 Then

Divide\_Error\_Func = CVErr(xlErrDiv0)

Else

Divide\_Error\_Func = A / B

End If

End Function

'Remember that you can use any of the following error constants with the CVErr function to return an error to Excel:

'xlErrDiv0, xlErrNA, xlErrName, xlErrNull, xlErrNum, xlErrRef and xlErrValue. If any other value is passed to CVErr, Excel

'will treat it as a #VALUE error. It is generally good practice to validate the input parameters and return an error value

'with CVErr rather than letting the VBA code error out with #VALUE errors. If a run-time error occurs in your code, or you

'attempt to change anything in Excel, such other cells, VBA terminates the function and returns a #VALUE error to Excel.

'For further reference see Section "Error Handling".

'-----------------------------------------------------------------------------------------------------------------------------------

'-----------------------------------------------------------------------------------------------------------------------------------

'Lesson 3.2.6. Determining The Range From Which Your UDF Was Called

'-----------------------------------------------------------------------------------------------------------------------------------

'-----------------------------------------------------------------------------------------------------------------------------------

'Under nearly all circumstances, it is not necessary to know the actual address of the range from which your UDF was called.

'Indeed, you should avoid have the need for such information. Your function should work the same regardless of where it was

'called from. However, you may well need to know the size of the range from which your UDF was called if it was array entered

'into a range of cells. The Application.Caller object will return a reference to the range from which your function was called,

'regardless of whether that range is a single cell or a range of cells. For further reference see "Choosing The Right Return

'Array Size" in Lesson 3.7.3. Returning Arrays From VBA User Defined Functions

'CAUTION: Application.Caller will be a Range object only when the function in which it appears was called from a worksheet cell.

'If the function was called from another VB procedure, Application.Caller will be an Error-type Variant and most any attempt to

'use it will result in a Type Mismatch (13) error. If the code containing Application.Caller was called via the OnAction property

'of a Shape object on a worksheet, Application.Caller will be a String containing the name of the sheet. Therefore, if your

'function might be called from another VB procedure rather than only from a worksheet cell, you should test Application.Caller with

'the IsObject function to ensure that it is indeed an object before attempting to access any of its properties.

'CAUTION: In Excel 2003, a new object, Application.ThisCell, was introduced. It is similar in nature to Application.Caller, but

'differs when a UDF is array entered into a range of more than one cell. Application.Caller will return the a Range reference to

'the entire range in which the UDF was array-entered. Application.ThisCell returns a reference to the first (upper left) cell in

'the range from which the UDF was called. Frankly, I'm not sure why Application.ThisCell was introduced in the first place.

'You can get the properties of Application.Caller with code like the following:

Sub TEST\_PROPERTIES\_APPLICATION\_CALLER()

Dim i As Long

Dim j As Long

Dim l As Long

Dim NROWS As Long

Dim NCOLUMNS As Long

Dim DST\_RNG As Range

Dim SRC\_WSHEET As Worksheet

Set SRC\_WSHEET = Worksheets("WSHEET\_TEST")

With SRC\_WSHEET

With .Cells

.Clear

.ColumnWidth = 10

End With

NROWS = 5

NCOLUMNS = 4

Set DST\_RNG = Range(.Cells(1, 1), .Cells(NROWS, NCOLUMNS))

DST\_RNG.FormulaArray = "=PROPERTIES\_APPLICATION\_CALLER\_FUNC1()"

End With

End Sub

Function PROPERTIES\_APPLICATION\_CALLER\_FUNC1()

Dim CallerRows As Long

Dim CallerCols As Long

With Application.Caller

CallerRows = .Rows.Count

CallerCols = .Columns.Count

Debug.Print .Address

End With

PROPERTIES\_APPLICATION\_CALLER\_FUNC1 = CallerRows \* CallerCols

End Function

'-----------------------------------------------------------------------------------------------------------------------------------

'-----------------------------------------------------------------------------------------------------------------------------------

'Lesson 3.2.7. Using A Variable Number Of Parameters

'-----------------------------------------------------------------------------------------------------------------------------------

'-----------------------------------------------------------------------------------------------------------------------------------

'You can define a function to accept a variable number of parameters in one of two somewhat different ways. You can use a

'specified number of optional parameters, or you can allow the function to accept any number of parameters, including none at

'all, using a ParamArray Variant parameter. The two methods are mutually exclusive. You cannot use both optional parameters

'and a ParamArray in the same function.

'-----------------------------------------------------------------------------------------------------------------------------------

'i) Optional Variant Parameters

'-----------------------------------------------------------------------------------------------------------------------------------

'You can define one or more parameters as Optional Variant types. For example:

Function Optional\_Variant\_Parameters\_Func(d As Double, Optional B As Variant) As Variant

If IsMissing(B) = True Then

Optional\_Variant\_Parameters\_Func = d

Else

If IsNumeric(B) = True Then

Optional\_Variant\_Parameters\_Func = d + B

Else

Optional\_Variant\_Parameters\_Func = CVErr(xlErrNum)

End If

End If

End Function

Sub Test\_Optional\_Variant\_Parameters\_Func()

Debug.Print Optional\_Variant\_Parameters\_Func(2)

Debug.Print Optional\_Variant\_Parameters\_Func(2, 3)

Debug.Print Optional\_Variant\_Parameters\_Func(2, "x")

End Sub

'This function defines the parameter B as an optional Variant and uses the IsMissing function to determine whether the parameter

'was passed. The IsMissing function can be used only with Variant type parameters. If IsMissing is used with any other data type

'(e.g., a Long), it will return False. More than one parameter may be Optional, but those parameters must be the last parameters

'accepted by the function. That is, once one parameter is specified as Optional, all the parameters that follow it must also be

'optional. You cannot have a required parameter following an optional parameter. If a parameter is declared as Optional but is

'not a Variant (e.g, it is a String or a Long) and that parameter is omitted, the IsMissing function will return False and the

'default value for that data type (0 or empty string) will be used. You can specify a default value for an optional parameter that

'should be used if the parameter is omitted. For example, the parameter B in the function below is optional with a default value

'of 2.

Function DIV\_LONG\_FUNC(A As Long, Optional B As Long = 2) As Variant

If B = 0 Then

DIV\_LONG\_FUNC = CVErr(xlErrDiv0)

Else

DIV\_LONG\_FUNC = A / B

End If

End Function

'In this code, the value 2 is used for the default value of B if B is omitted. When using a default value for a parameter, you don't

'call the IsMissing function. Your code should be written to use either a passed in parameter value or the default value of the parameter.

'With the code above, the following two worksheet functions are equivalent:

Sub TEST\_DIV\_LONG\_FUNC()

Debug.Print DIV\_LONG\_FUNC(1, 2)

Debug.Print DIV\_LONG\_FUNC(1)

Debug.Print DIV\_LONG\_FUNC(1, 0)

End Sub

'ii) Variant ParamArray

'The second method for working with optional parameters is to use a ParamArray Variant parameter. A ParamArray allows any number of parameters,

'including none at all, to be passed to the function. You can have one or more required parameters before the ParamArray, but you cannot have

'any optional parameters if you have a ParamArray. Moreover, the ParamArray variable must be the last parameter declared for a function. The

'ParamArray variables must be Variant types. You cannot have a ParamArray of other types, such as Long integers. If necessary, you should

'validate the values passed in the ParamArray, such as to ensure they are all numeric. If your function requires one or more inputs followed

'by a variable number of parameters, declare the required parameters explicitly and use a ParamArray only for the optional parameters. For

'example, the function SUM\_PARAMARRAY\_FUNC below accepts any number of inputs and simply adds them up:

Function SUM\_PARAMARRAY\_FUNC(ParamArray DATA\_ARR() As Variant) As Variant 'Add up the numbers in DATA\_ARR

Dim N As Long

Dim d As Double

For N = LBound(DATA\_ARR) To UBound(DATA\_ARR)

If IsNumeric(DATA\_ARR(N)) = True Then

d = d + DATA\_ARR(N)

Else

SUM\_PARAMARRAY\_FUNC = CVErr(xlErrNum)

Exit Function

End If

Next N

SUM\_PARAMARRAY\_FUNC = d

End Function

Sub TEST\_SUM\_PARAMARRAY\_FUNC()

Debug.Print SUM\_PARAMARRAY\_FUNC(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

Debug.Print NPARAM\_PARAMARRAY\_FUNC(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

End Sub

Function NPARAM\_PARAMARRAY\_FUNC(ParamArray DATA\_ARR() As Variant) As Variant

Dim NSIZE As Long

NSIZE = UBound(DATA\_ARR) - LBound(DATA\_ARR) + 1

'to determine how many parameters were passed in the ParamArray variable DATA\_ARR. This will be 0 if no parameters

'were passed as the ParamArray. Of course, the code above counts the number of parameters within the ParamArray, not

'the total number of parameters to the function. See Lesson 3.6. Optional Parameters To Procedures for a more in depth

'discussion of optional parameters and ParamArray parameter type.

NPARAM\_PARAMARRAY\_FUNC = NSIZE

End Function

'-----------------------------------------------------------------------------------------------------------------------------------

'-----------------------------------------------------------------------------------------------------------------------------------

'Lesson 3.2.8. Returning Arrays From Functions

'-----------------------------------------------------------------------------------------------------------------------------------

'-----------------------------------------------------------------------------------------------------------------------------------

'Your function can return an array of values so that it can be entered as an array formula, either entered into an array of cells or

'to return an array to be aggregated by a function like SUM. (See Lesson 3.7.3. Returning Arrays From VBA User Defined Functions for

'information about returning arrays as the result of your User Defined Function.) The ADD\_INTEGERS\_FUNC function below returns an

'array of the integers from 1 to the input parameter L. For simplicity, L must be between 1 and 5. The function also requires that

'if the function is array entered, it must be in either a single row or a single column. A range with more than one row and more

'than one column will result in a #REF error. This restriction applies to this example only; it is not a limitation on UDF array

'functions in general. See the next section for example code that return values to a two dimensional range of cells.

'In a UDF, Application.Caller (see "Referencing Worksheet From Formulas" @ http://www.cpearson.com/excel/sheetref.htm) returns

'a Range type object that references the cell(s) from which the formula was called. Using this, we can test whether we need a

'row array or a column array. If the function is called from a column of cells (e.g., array entered into A1:A5), the VBA array

'must be transposed before returning it to Excel. Note that there is also an object named Application.ThisCell that references

'the cell from which a function is called. In functions called from a single cell, Application.Caller and Application.ThisCell

'work the same. However, they differ when a function is called as an array formula. You should use Application.Caller, not

'Application.ThisCell.

Function ADD\_INTEGERS\_FUNC(l As Long) As Variant 'Add up the integers from 1 To L

Dim V() As Long

Dim ArraySize As Long

Dim N As Long

Dim ResultAsColumn As Boolean

If (l > 5) Or (l < 1) Then 'Allow inputs only between 0 and 5.

ADD\_INTEGERS\_FUNC = CVErr(xlErrValue)

Exit Function

End If

If Application.Caller.Rows.Count > 1 And Application.Caller.Columns.Count > 1 Then 'Allow only one columns or one row

ADD\_INTEGERS\_FUNC = CVErr(xlErrRef)

Exit Function

End If

If Application.Caller.Rows.Count > 1 Then ' Test whether the result should be returned as a columns or row array.

ResultAsColumn = True

Else

ResultAsColumn = False

End If

ReDim V(1 To l) ' ReDim the array to hold L elements.

For N = 1 To UBound(V) ' Fill up the array

V(N) = N

Next N

If ResultAsColumn = True Then ' Return the result, transposing if necessary.

ADD\_INTEGERS\_FUNC = Application.Transpose(V)

Else

ADD\_INTEGERS\_FUNC = V

End If

End Function

'If the ADD\_INTEGERS\_FUNC function is called from a range that has more than one row, the array must be transposed before it

'is returned, using the Application.Transpose function. The result of the function is an array of L integers from 1 to L. If

'the range from which the function is called has N cells, and N is less than L (the size of the result array), elements at

'the end of array are discarded and only the firt N elements are sent to the cells. If L is less than N (the function is entered

'into an array of cells larger than L), #N/A errors fill out the ending elements of the range on the worksheet. Since the result

'of ADD\_INTEGERS\_FUNC is an array, it can be used in an array formula, such as Sum (ADD\_INTEGERS\_FUNC(5)) which returns 15, the

'sum of the numbers from 1 to 5.

Sub TEST\_ADD\_INTEGERS\_FUNC()

Dim i As Long

Dim j() As Long

Dim NROWS As Long

Dim DATA\_RNG As Range

Dim SRC\_WSHEET As Worksheet

Set SRC\_WSHEET = Worksheets("WSHEET\_TEST")

With SRC\_WSHEET

With .Cells

.Clear

.ColumnWidth = 10

End With

NROWS = 5 'If (NROWS > 5) Or (NROWS < 1) Then CVErr(xlErrValue) -> Allow inputs only between 0 and 5

With Range(.Cells(1, 1), .Cells(NROWS, 1)) 'ResultAsColumn = True

'Since Rows.Count > 1 then the result should be returned as a column array.

.FormulaArray = "=ADD\_INTEGERS\_FUNC(" & NROWS & ")"

Debug.Print .Address

End With

With Range(.Cells(1, 1 + 1), .Cells(1, 1 + NROWS)) 'ResultAsColumn = False

'Since Rows.Count = 1 then the result should be returned as a row array.

.FormulaArray = "=ADD\_INTEGERS\_FUNC(" & NROWS & ")"

Debug.Print .Address

End With

'If Rows.Count > 1 And Columns.Count > 1 Then CVErr(xlErrRef) -> Allow only one columns or one row

With Range(.Cells(NROWS + 2, 1), .Cells(NROWS \* 2 + 1, NROWS))

.FormulaArray = "=ADD\_INTEGERS\_FUNC(" & NROWS & ")"

Debug.Print .Address

End With

End With

End Sub

'To return an array to a range that contains more than one row and more than one column, create a two dimensional array with the

'first dimension equal to the number of rows in the range and the second dimension equal to the number of columns in the range.

'Then load that array, looping through the rows and columns and then return the array as the result.

'The function ACROSS\_THEN\_DOWN\_FUNC below loads the calling cells with sequential integers, moving across each row and then moving

'down to the next row. The function DOWN\_THEN\_ACROSS\_FUNC below loads the calling cells with sequential integers, moving down each

'column then moving right to the next column. The difference between the two function is in the For loops, whether the outer loop

'is for Rows or Columns. As noted before, use Application.Caller not Application.ThisCell to get a reference to the range of cells

'calling the function.

Function ACROSS\_THEN\_DOWN\_FUNC() As Variant

Dim NumCols As Long

Dim NROWS As Long

Dim RowNdx As Long

Dim ColNdx As Long

Dim Result() As Variant

Dim N As Long

' Get the number of rows and columns in the range that is calling this function.

NumCols = Application.Caller.Columns.Count

NROWS = Application.Caller.Rows.Count

' ReDim the Result array to the number of rows and columns in the calling range.

ReDim Result(1 To NROWS, 1 To NumCols)

For RowNdx = 1 To NROWS

For ColNdx = 1 To NumCols

N = N + 1

Result(RowNdx, ColNdx) = N

Next ColNdx

Next RowNdx

ACROSS\_THEN\_DOWN\_FUNC = Result

End Function

Sub TEST\_ACROSS\_THEN\_DOWN\_FUNC\_AND\_DOWN\_THEN\_ACROSS\_FUNC()

Dim i As Long

Dim j() As Long

Dim NROWS As Long

Dim DATA\_RNG As Range

Dim SRC\_WSHEET As Worksheet

Set SRC\_WSHEET = Worksheets("WSHEET\_TEST")

With SRC\_WSHEET

With .Cells

.Clear

.ColumnWidth = 10

End With

NROWS = 5

With Range(.Cells(1, 1), .Cells(NROWS, NROWS))

.FormulaArray = "=ACROSS\_THEN\_DOWN\_FUNC()"

Debug.Print .Address

End With

With Range(.Cells(1 + 1 + NROWS, 1), .Cells(1 + 1 + NROWS \* 2, NROWS))

.FormulaArray = "=DOWN\_THEN\_ACROSS\_FUNC()"

Debug.Print .Address

End With

End With

End Sub

Function DOWN\_THEN\_ACROSS\_FUNC() As Variant

Dim NumCols As Long

Dim NROWS As Long

Dim RowNdx As Long

Dim ColNdx As Long

Dim Result() As Variant

Dim N As Long

' Get the number of rows and columns in the range that is calling this function.

NumCols = Application.Caller.Columns.Count

NROWS = Application.Caller.Rows.Count

' ReDim the Result array to the number of rows and columns in the calling range.

ReDim Result(1 To NROWS, 1 To NumCols)

For ColNdx = 1 To NumCols

For RowNdx = 1 To NROWS

N = N + 1

Result(RowNdx, ColNdx) = N

Next RowNdx

Next ColNdx

DOWN\_THEN\_ACROSS\_FUNC = Result

End Function