VBA for Finance

Risk In Finance

ACADEMIC YEAR 2021-2022

Lecturer: Dr. D. O'Kane

Excel

- For this course you need a recent version of Excel
- It does not have to be the latest version – nothing much has changed for the last 10 years – just some menus
- Windows Excel is the best – but Mac is fine too
- There are some minor differences but everything I do here should work on both
- Use the VBA documentation which you should have installed on your machine (or it may be online)

Getting Started

- Make sure you have Excel loaded and running.
- Make sure the Excel language is set to English **NOT** French.
- This is not a bias – I just want everyone to see the same as me!
- Change number formatting so they are expressed with a decimal point **NOT** a comma e.g. 0.567 instead of 0,567.
- Make sure you can save Excel files to a directory of your choice.

Changing to English Display and Settings

- Go to the control panel (*panneau de configuration*)
- Change the display language to English
- Change the location to US or UK or England
- You will have to restart the PC

- **Note that the examined coursework will have to be done in English settings**
- So no commas instead of decimal places,
- And functions must all be in English … *s'il vous plait* …

Black Scholes

- Consider the Black-Scholes equation given the following inputs
  - Stock price S
  - Strike price of option K
  - Time to expiry of option T
  - Risk free rate r and dividend yield q
  - Volatility of stock price σ
- The formula is given below for a call option

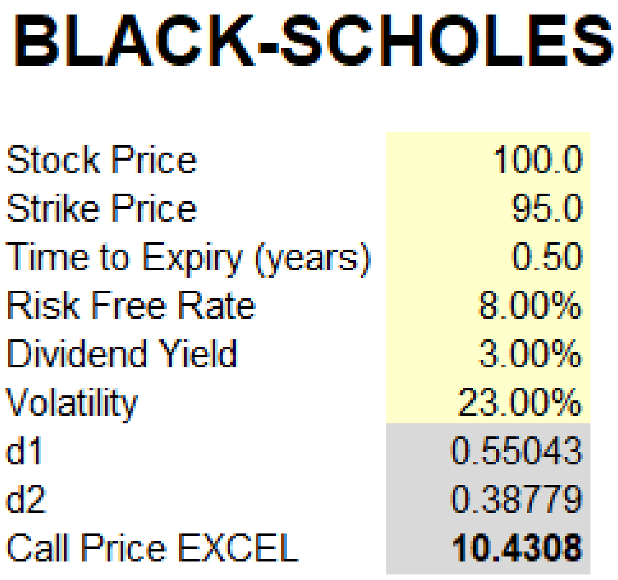
$$C(S,K,r,T,\sigma,q) = S\exp(-qT)N(d_1) - K\exp(-rT)N(d_2)$$

where

$$d_1 = \frac{\ln(S/K) + (r - q + \sigma^2/2)T}{\sigma\sqrt{T}} \text{ and } d_2 = d_1 - \sigma\sqrt{T}$$

Task: Implement this calculation of Black Scholes

- In Excel the function N(x) is NORMSDIST(x)
- Break the calculation into stages to make it simpler
- Calculate d1 and d2 and then the call option price



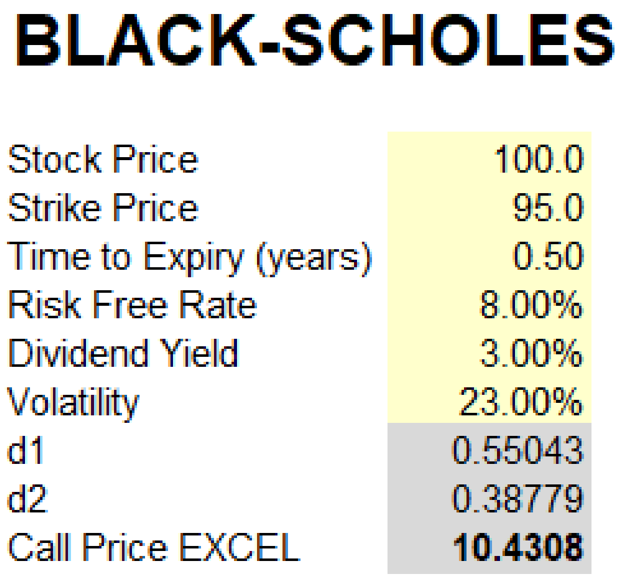
BLACK-SCHOLES

Stock Price	100.0
Strike Price	95.0
Time to Expiry (years)	0.50
Risk Free Rate	8.00%
Dividend Yield	3.00%
Volatility	23.00%
d1	0.55043
d2	0.38779
Call Price EXCEL	**10.4308**

- So now we have the price given the volatility

Formatting Style

- In my spreadsheets I like to impose some formatting rules
- Inputs are labelled and have a yellow background
- Outputs are labelled and have a light grey background
- Show the appropriate number of decimal places
- Highlight an important output using bold font



BLACK-SCHOLES

Stock Price	100.0
Strike Price	95.0
Time to Expiry (years)	0.50
Risk Free Rate	8.00%
Dividend Yield	3.00%
Volatility	23.00%
d1	0.55043
d2	0.38779
Call Price EXCEL	**10.4308**

Do not do Black Scholes in Excel !

- Imagine you want the option price as a function of volatility
- You need to copy the three cells
- Or you can combine them into one long formula
- There is no validation on the inputs – you can make the formula bigger but that just makes it harder to check
- As soon as any calculation becomes complicated, spreadsheet formulas fail – I have seen this myself
- If there is a bug, it has to be fixed everywhere the formula is – this is error-prone and tedious
- And this can cost you money.
- If you want to do a put option, the code is mostly the same - duplication

VBA is the Answer

- Instead we decide to code it up in VBA
- It also means that **ALL OF THE LOGIC IS IN ONE PLACE**
- So if there is a bug, it only needs to be fixed ONCE in ONE place
- You can write clear code that is trivial to understand
- You can have functions that call other functions so you can organise the logic and share common code
- You can make the code robust with validation and error handling
- You can share a library of code as an addin so that you can make your work available to others – very powerful

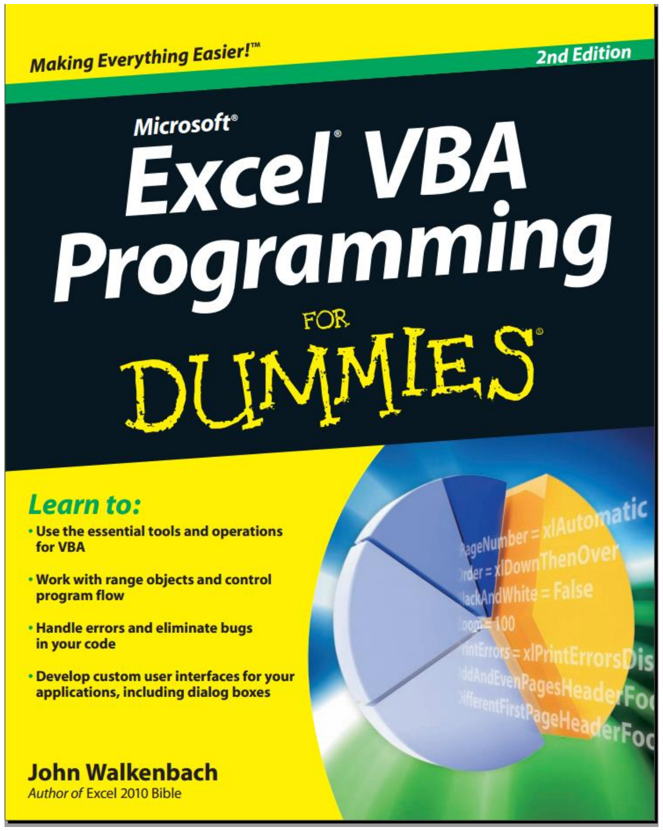
VBA Development

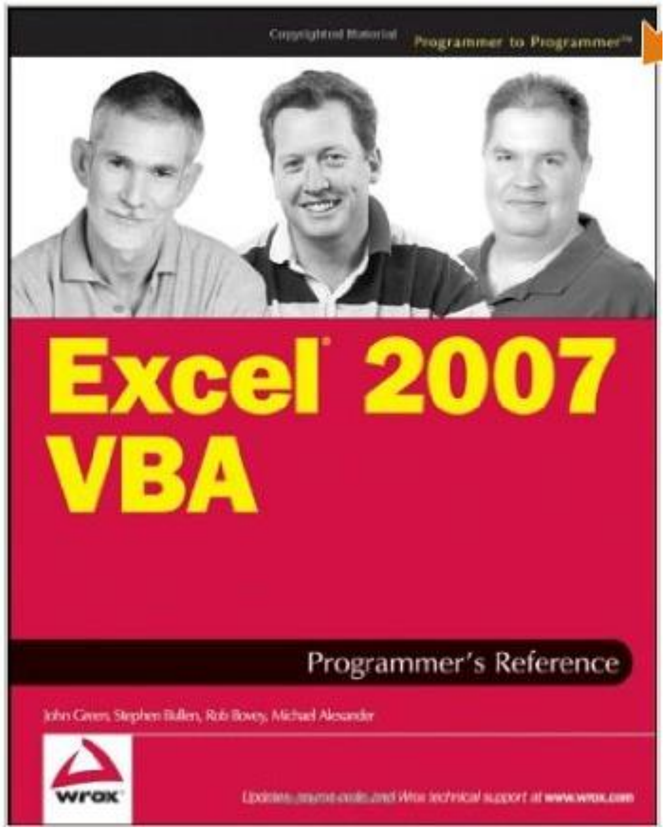
Why VBA?

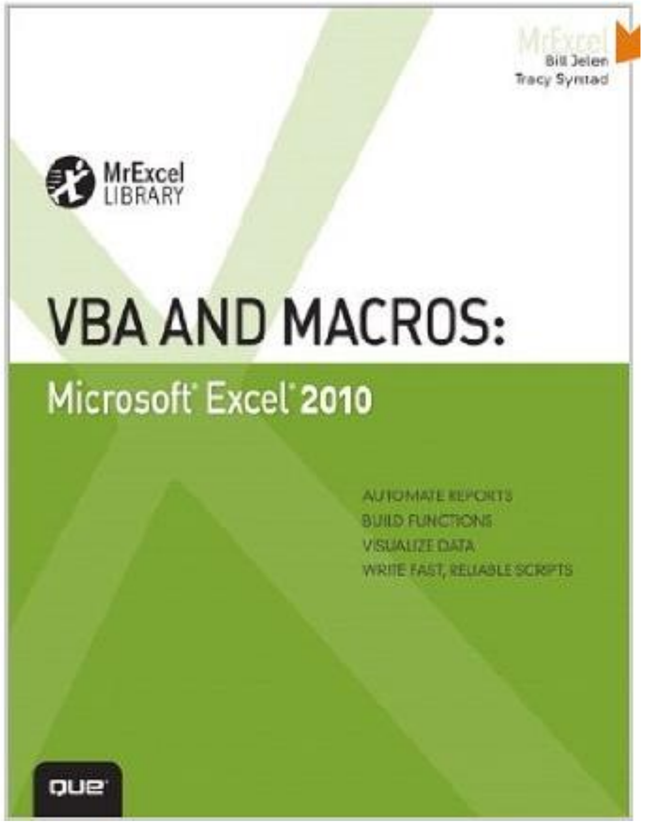
- Excel + VBA is much more powerful than Excel
- VBA is not the fastest code, but is fast enough for many purposes
- The syntax is simple – it is called BASIC for a reason!
- Excel is an easy environment to do complex tasks
- Sometimes I prefer it to other tools like Matlab or Python
- Inputs can be read in easily using all the data import features
- Everyone has Excel so spreadsheets can be shared easily
- Spreadsheets with VBA are more robust than those with formulas
- You can be more productive

Excel VBA Books

- ❑ Excel VBA is not usually well covered in basic Excel books so I would recommend a dedicated Excel VBA book
- ❑ There are several decent books – these are old versions, but you can find newer ones


Making Everything Easier!™
2nd Edition
Microsoft® Excel® VBA Programming FOR DUMMIES®
Learn to:
• Use the essential tools and operations for VBA
• Work with range objects and control program flow
• Handle errors and eliminate bugs in your code
• Develop custom user interfaces for your applications, including dialog boxes
John Walkenbach
Author of Excel 2010 Bible


Programmer to Programmer™
Excel® 2007 VBA
Programmer's Reference
John Green, Stephen Bullen, Rob Bovey, Michael Alexander
wrox

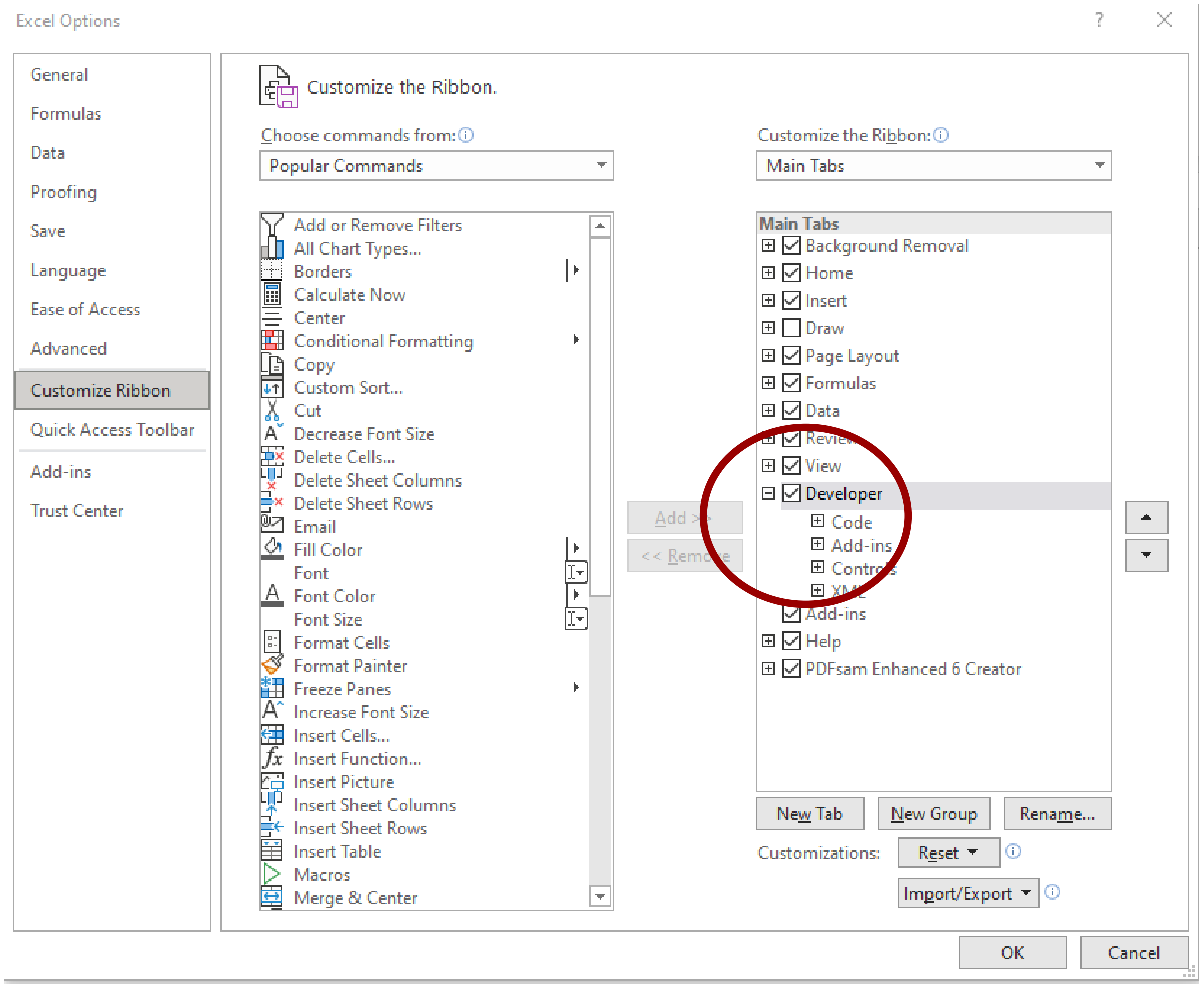


What I will cover

- VBA Programming both procedures and functions
- My focus is on writing good VBA functions
- Also on sharing code via addins
- I imagine you working in a team of finance experts where you can share your code between colleagues
- **I do not cover forms and controls** - they look pretty and are cool but are actually **very hard to do well,** to maintain and really need to be developed by IT professionals
- Usually used for entering inputs safely – but your users will be sophisticated Excel users – and you will do validation in code
- So just use functions with good validation and error handling
- Do not try to do too much in Excel – it has limits

Developer Commands in Excel

- Do File->Options and then select Customize Ribbon
- Make sure the Developer tick box is selected



Excel Options

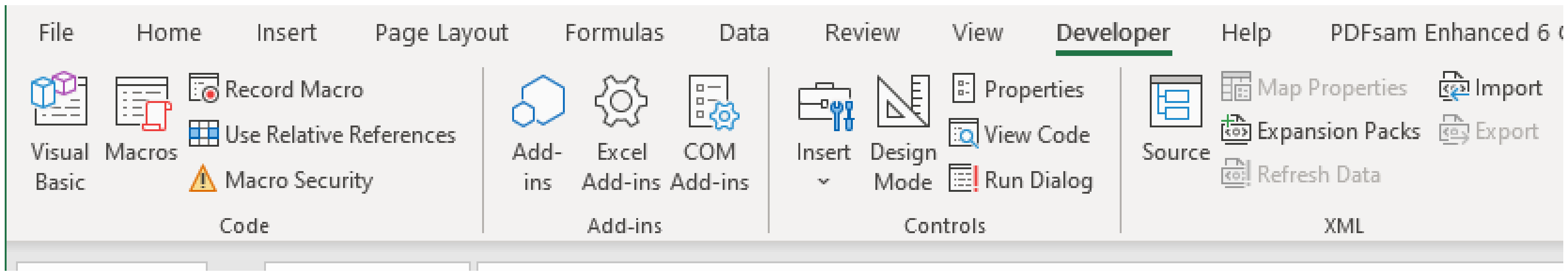
Customize the Ribbon.

Choose commands from: Popular Commands

Customize the Ribbon: Main Tabs

Main Tabs
- [✓] Background Removal
- [✓] Home
- [✓] Insert
- [ ] Draw
- [✓] Page Layout
- [✓] Formulas
- [✓] Data
- [✓] Review
- [✓] View
- [✓] Developer
  - Code
  - Add-ins
  - Controls
  - XML
- [✓] Add-ins
- [✓] Help
- [✓] PDFsam Enhanced 6 Creator

The Developer Tab Commands



- Visual Basic opens the VBA editor
- Macros gives a list of Macros – subroutines in VBA
- You can record a Macro
- Macro security sets what macros can be run
- The rest is too advanced for this course

What is VBA?

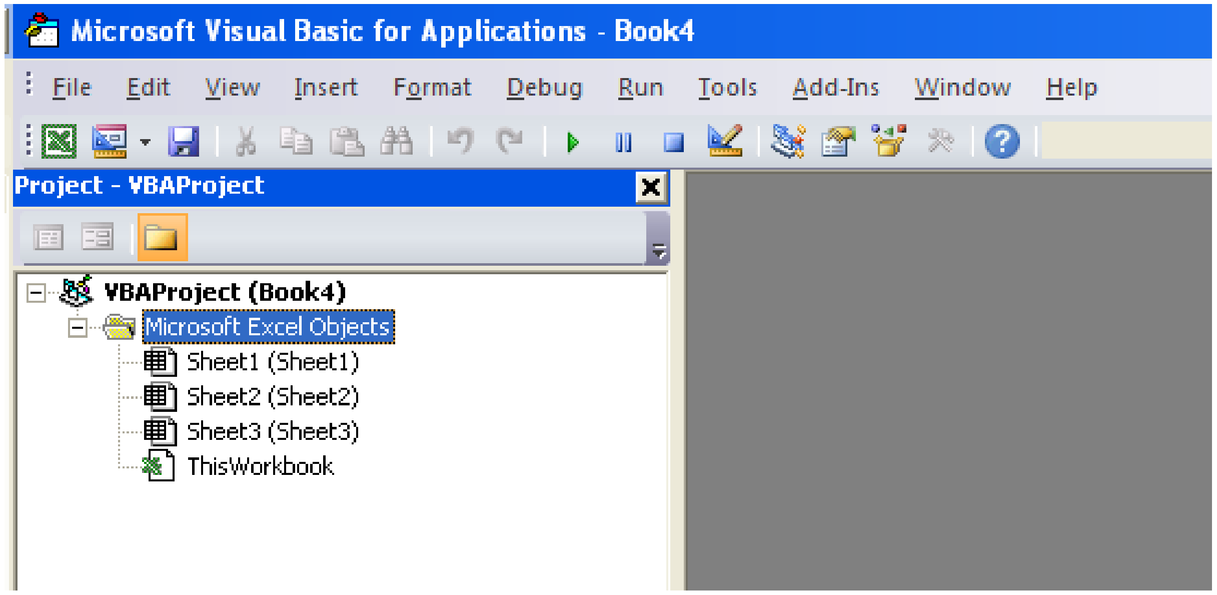
- VBA stands for **Visual Basic for Applications**
- It is **not** a compiled language like C++
- This means that it is not compiled into machine language once and then run many times
- It is an **interpreted language**
- This means that it is converted into machine language one line at a time every time it is run
- This is why it is not as fast as C++
- It runs directly within Microsoft Excel

History - VBA is not VB.Net

- Before around 2007, VB was a language used by Microsoft developers to build applications
- It was actually a very productive language but was old and missed some of the features of modern languages like Java
- Hence MS killed VBA over 10 years ago and replaced it with VB.Net which is very different in many ways
- However the old VB is the same as the VBA language used today in Excel – and you can use it to be more productive
- It does not need the advanced features of Java
- Do not confuse VBA and VB.Net
- They are quite different

The VBA window

- Open Excel, press Alt + F11 key to open the Visual Basic Editor (VBE) – toggle between Excel and VBA using Alt + F11.
- Right click on VBAProject and select: Insert → Module.
- A new Modules directory is created with a blank Module 1 programming sheet.



Modules

- A VBA module holds two types of code:
  - Subroutines/Macros and Functions
  - We will discuss the differences very soon
- A single module can store many subroutines and functions.
- You typically arrange related code in single VBA modules
- It makes sense to split different code into different modules

Subroutines/Macros

- A subroutine/macro is a set of instructions that performs some action – e.g. from doing a calculation to copying and pasting
- The syntax is

```
Sub
....
End Sub
```

- **Subs** are also called **Macros** especially when they automate certain task that you would perform manually
- Excel allows us to write Macros by hand or by recording a sequence of actions that we perform

Functions

- A function is a set of instructions that returns a single value or a matrix of values from of code to the area (cell or range) of the spreadsheet from which it is invoked
- A function starts with

```
Function
....
End Function
```

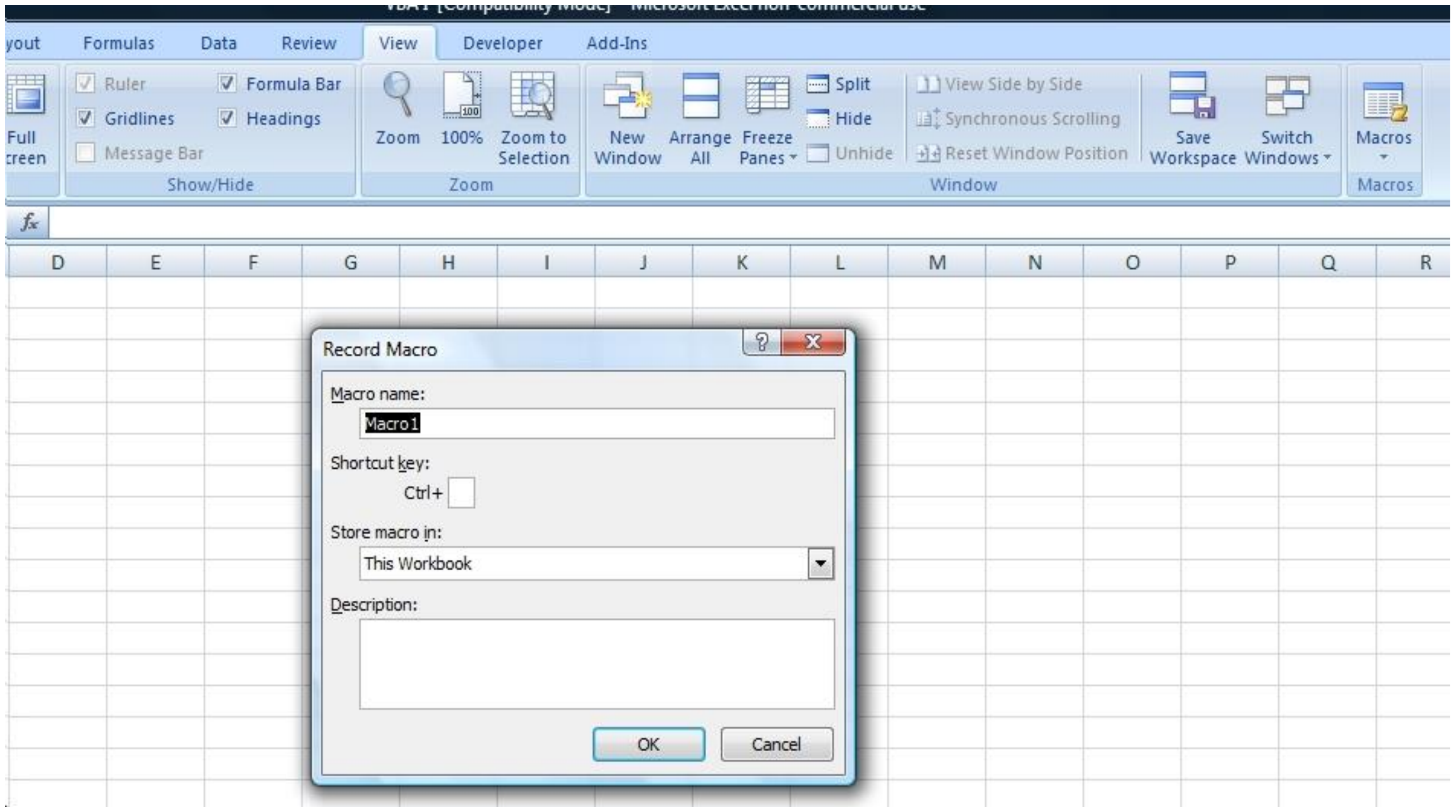
- While subs can be useful, our focus here will be **mostly** on functions

VBA Macros

Recording Macros

- Excel makes it possible to record a macro - makes Excel write for you the code corresponding to actions that you perform
- This can be used to automate some task – but needs to be done carefully
- My main reason to record a macro is to find out how to do something in VBA
- You record the macro and then look at the code
- Suppose I want to know how to set the format of a range of cells so that they have a yellow colour and a larger font
- I record the macro by selecting "Developer" and then clicking on "Record Macro"

Macro Recording



Record Macro

Macro name: Macro1

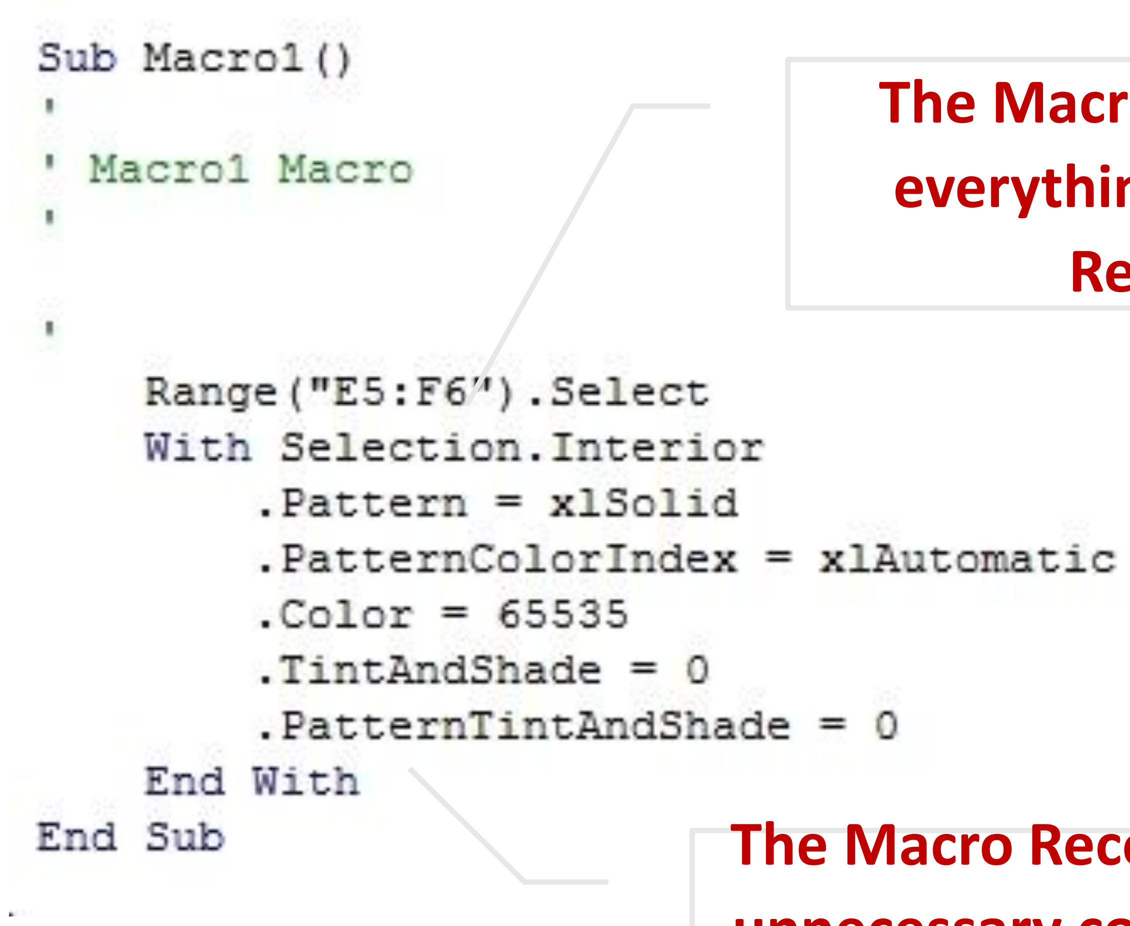
Shortcut key: Ctrl+

Store macro in: This Workbook

Description:

- Give the macro a sensible name
- Then say OK and select some cells and change their colour
- When you are finished click "Stop Recording"

What you see … something like this



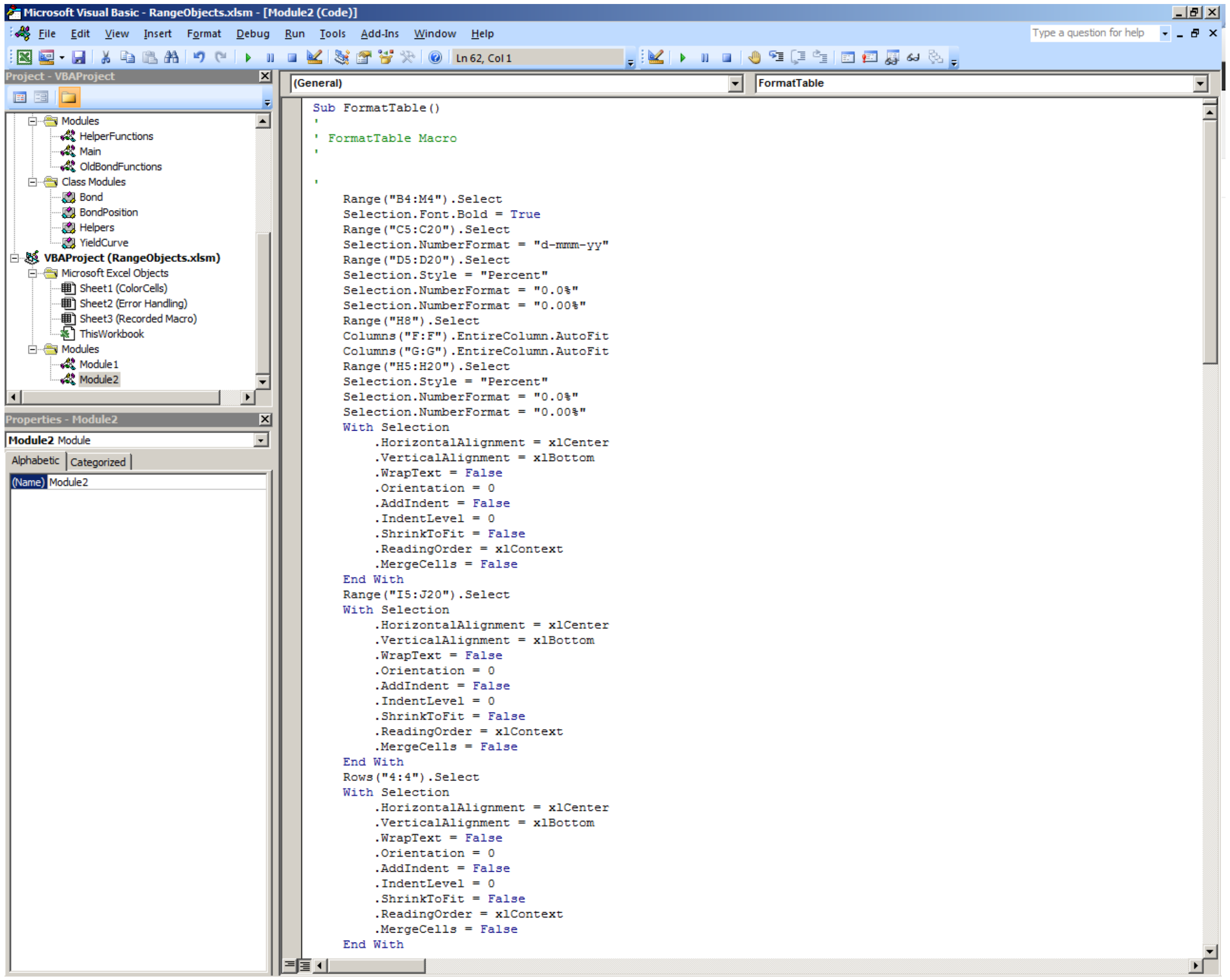
```
Sub Macro1()
'
' Macro1 Macro
'

'
    Range("E5:F6").Select
    With Selection.Interior
        .Pattern = xlSolid
        .PatternColorIndex = xlAutomatic
        .Color = 65535
        .TintAndShade = 0
        .PatternTintAndShade = 0
    End With
End Sub
```

The Macro Recorder records everything you do whilst in Record mode.

The Macro Recorder records lots of unnecessary code, especially when recording large macro programs with many keystrokes and tasks.

Recorded Macros

- When you do a lot and then look at the code it will be messy
- Sometimes you tried to apply different formatting etc…



```
Sub FormatTable()
'
' FormatTable Macro
'

'
    Range("B4:M4").Select
    Selection.Font.Bold = True
    Range("C5:C20").Select
    Selection.NumberFormat = "d-mmm-yy"
    Range("D5:D20").Select
    Selection.Style = "Percent"
    Selection.NumberFormat = "0.0%"
    Selection.NumberFormat = "0.00%"
    Range("H8").Select
    Columns("F:F").EntireColumn.AutoFit
    Columns("G:G").EntireColumn.AutoFit
    Range("H5:H20").Select
    Selection.Style = "Percent"
    Selection.NumberFormat = "0.0%"
    Selection.NumberFormat = "0.00%"
    With Selection
        .HorizontalAlignment = xlCenter
        .VerticalAlignment = xlBottom
        .WrapText = False
        .Orientation = 0
        .AddIndent = False
        .IndentLevel = 0
        .ShrinkToFit = False
        .ReadingOrder = xlContext
        .MergeCells = False
    End With
    Range("I5:J20").Select
    With Selection
        .HorizontalAlignment = xlCenter
        .VerticalAlignment = xlBottom
        .WrapText = False
        .Orientation = 0
        .AddIndent = False
        .IndentLevel = 0
        .ShrinkToFit = False
        .ReadingOrder = xlContext
        .MergeCells = False
    End With
    Rows("4:4").Select
    With Selection
        .HorizontalAlignment = xlCenter
        .VerticalAlignment = xlBottom
        .WrapText = False
        .Orientation = 0
        .AddIndent = False
        .IndentLevel = 0
        .ShrinkToFit = False
        .ReadingOrder = xlContext
        .MergeCells = False
    End With
```

Cleaning up Recorded Code

- Remove selects since these are slow e.g. instead of

```
Range("B4:M4").Select
Selection.Font.Bold = True
```

- Use the following

```
Range("B4:M4").Font.Bold = True
```

Cleaning up Recorded Code

- Do your Copy and paste in one go so not

```
Range(“G16”).select
Selection.Copy
Range(“K19”).select
Selection.Paste
```

- Instead do

```
Range(“G16”).copy Destination:=Range(“K19”)
```

Recorded Macros

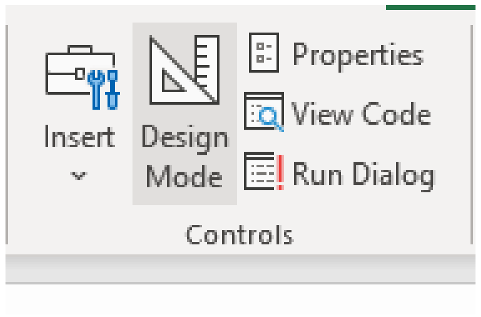
- Useful for tasks that are always the same
- For example loading a file and formatting the data
- As long as the file retains the same structure this should be ok
- Macros cannot take inputs
- Nor is it possible to get a recorded macro to read an input
- Recorded macros have no logic in them
- I tend to record macros to see how to do something and then I copy and paste the code into a subroutine that I hand write
- But in general I avoid recorded macros

Speed Tips

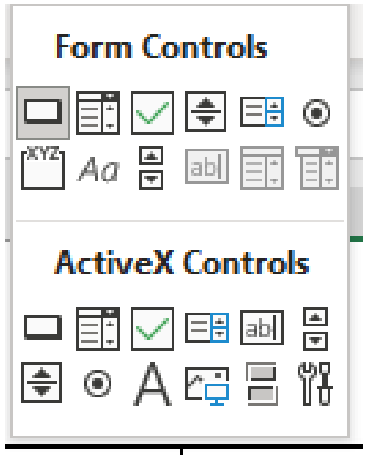
1. **Turn off screen updating if you are changing the screen**
   - Application.ScreenUpdating = False
   - When done, just set this back to True
2. **Turn off Automatic Calc during a Macro**
   - Application.Calculation = xlCalculationManual
   - Application.Calculation = xlCalculationAutomatic
   - If you need to calculate a page then do it using the Calculate method
3. **Use With … End With**
   - It is not just to make your code look nicer
   - It is also faster as the code only needs to find the object once

Add a Button to Run your Macro

- In the developer ribbon click on Design Mode



- Then click on the button in Form Controls



- Drag and draw it onto the worksheet and assign to it your macro

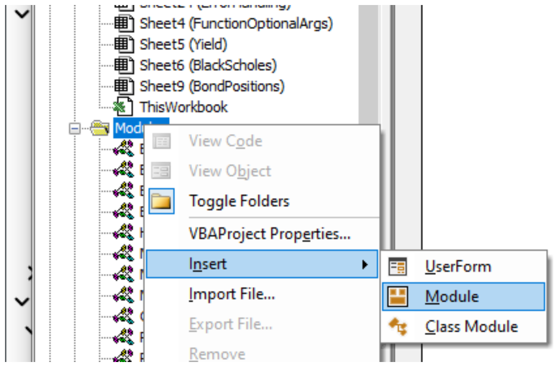
Exercise

- Open the sheet FormatTable.xlsm
- Record a macro to format the table
  - Bold headers
  - Coupon and Yields in % format
  - Dates as dates
  - Prices to 5 decimal places
  - Nominals with comma separation of 000's
- Add a button to automate this
- We'll now consider more advanced subroutines that we actually write by hand before we get to actual user-defined functions

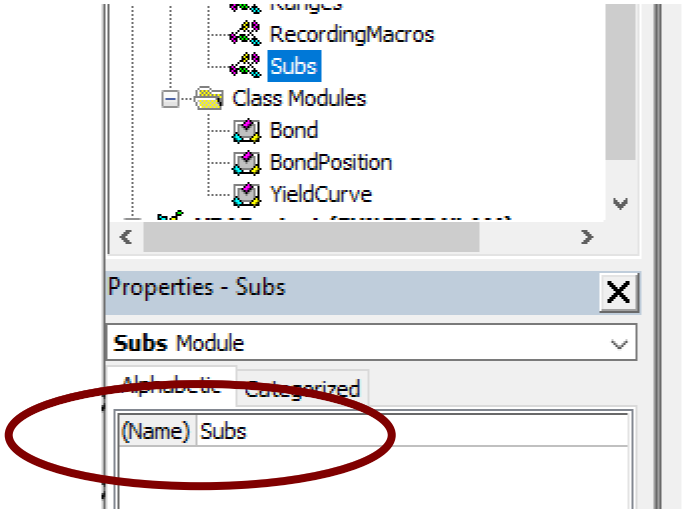
Your first Hand-Written Sub

Inserting and Naming Modules

- In the project explorer window, you right click on the Module
- Then you have a choice from which you choose "Module"

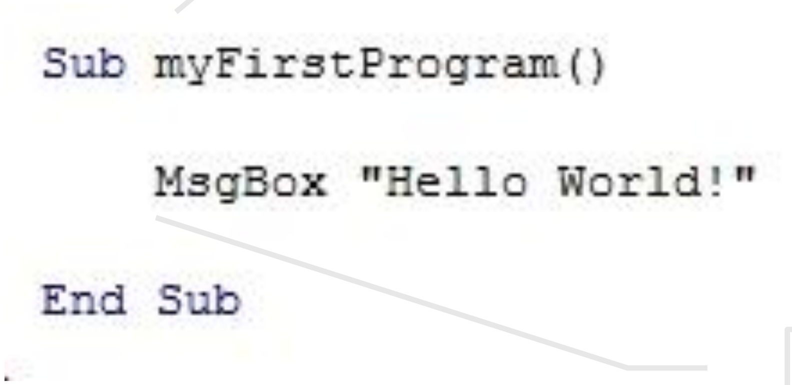


- To rename the module click on the module and change it in the properties tab



(Name) Subs

Your first VBA program (Sub)



```
Sub myFirstProgram()

    MsgBox "Hello World!"

End Sub
```

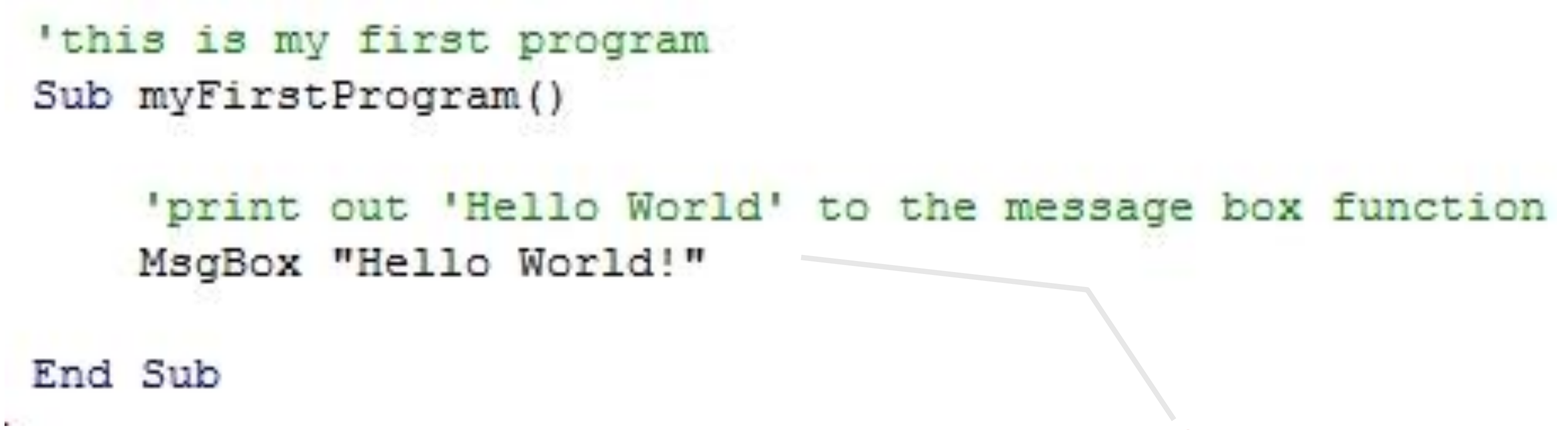
Give the subroutine a sensible name

All subroutines MUST begin with Sub and end with End Sub.

MsgBox is a built-in function automatically recognised by VBA

Adding comments to a program

Comments are added to a program using the apostrophe (') sign and highlighted automatically in green.

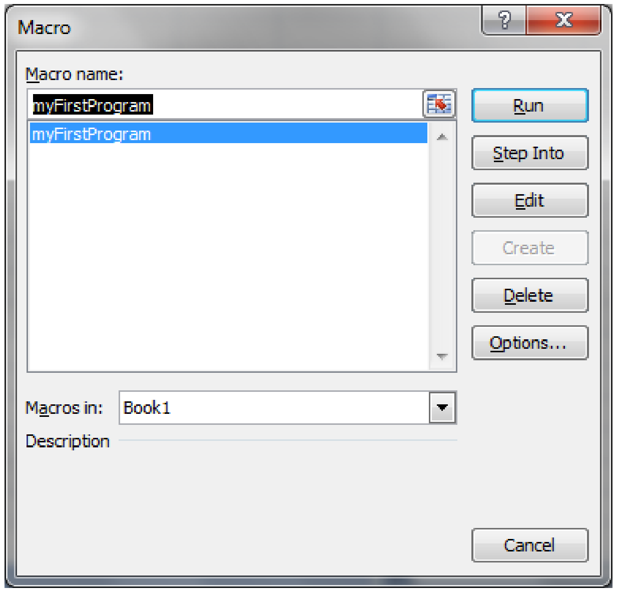


```
'this is my first program
Sub myFirstProgram()

    'print out 'Hello World' to the message box function
    MsgBox "Hello World!"

End Sub
```

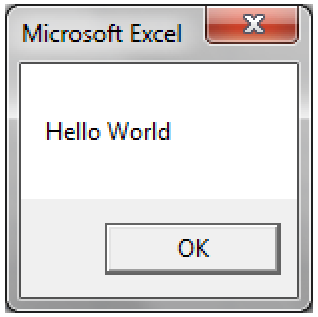
Use lots of comments to make your programs easier to follow – not just for others but also for you when you come back in 6 months to make a change

Running the Program

- You can run your program from Excel in a number of ways
- Click on the Macros button and a window like this will appear



- Select the "myFirstProgram" Macro and then click on "Run" – you will see this



Hello World

Exercise

- Add a button to the sheet to run the Hello World Macro
- Add the label "Hello" to the button

Program Control Logic

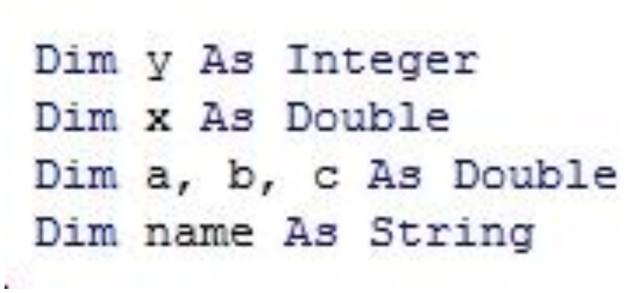
Logical operators

- VBA has all of the mathematical and logical operators of Excel

+ -	plus/minus
* /	multiplied by/divide by
^	power
=	equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
<>	not equal to
AND	
OR	

Variable declarations

- All variables used in a subroutine should be declared with a certain type e.g. **integer**, **double**, **string** etc.
- Doubles are numbers with precision, integers are whole numbers
- Variables can have any name and generally programmers develop their own style and they must be declared



```
Dim y As Integer
Dim x As Double
Dim a, b, c As Double
Dim name As String
```

Several variables of the same type can be declared in the same line

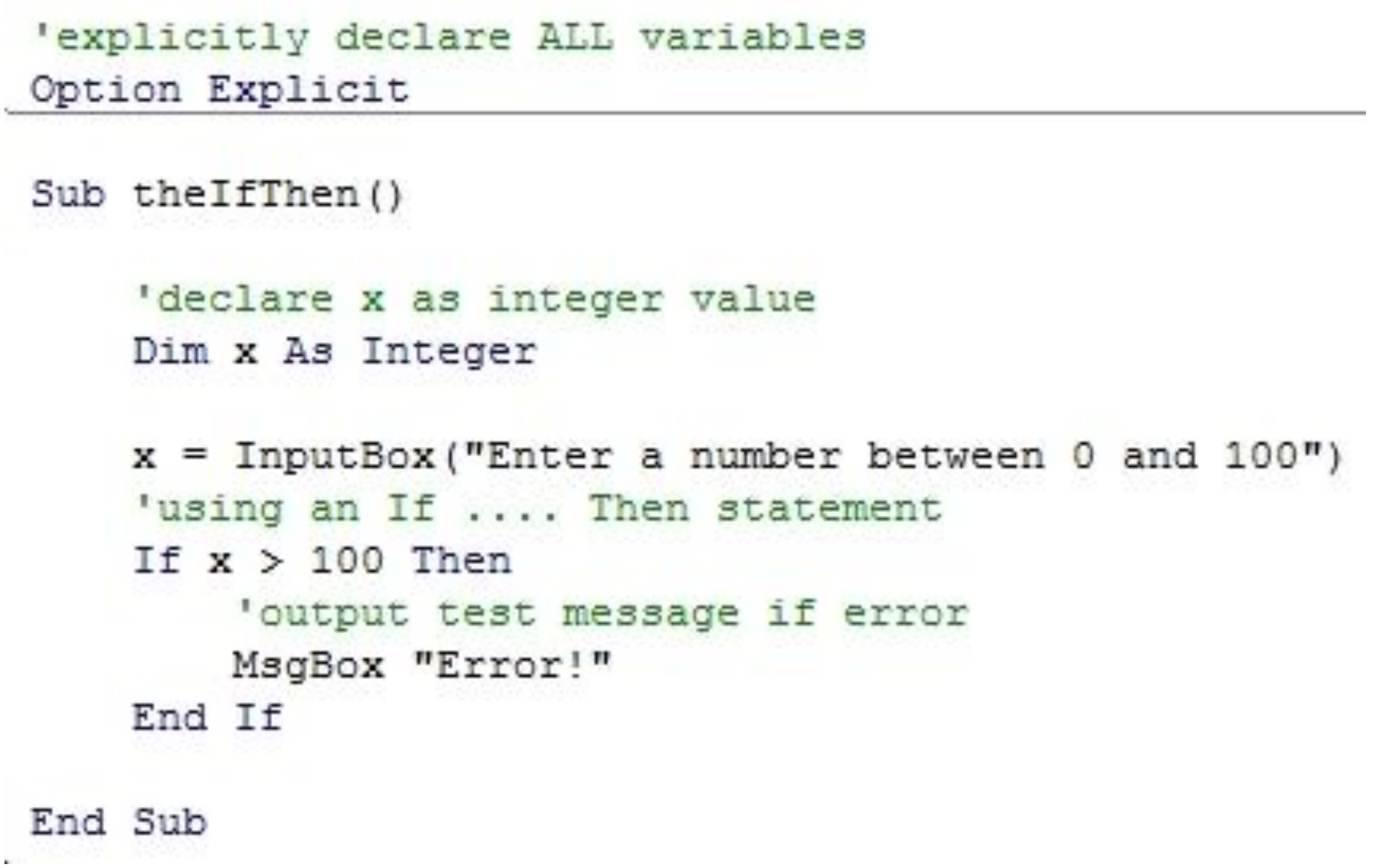
- I prefer variables with meaningful names and a strict style
- For example **optionVolatility** and **portfolioReturn**

The Option Explicit statement

- **Option Explicit** forces ALL variables to be explicitly declared
- Declaring types up front makes the code slightly faster as there is no need to convert them from one data type to another
- You save more time avoiding simple errors in your code when you add option explicit than you do writing the declarations
- It is good programming practice and you **must use it in all modules in your examined coursework**

If .... Then statement

- The If .... Then statement gives the program decision-making capabilities

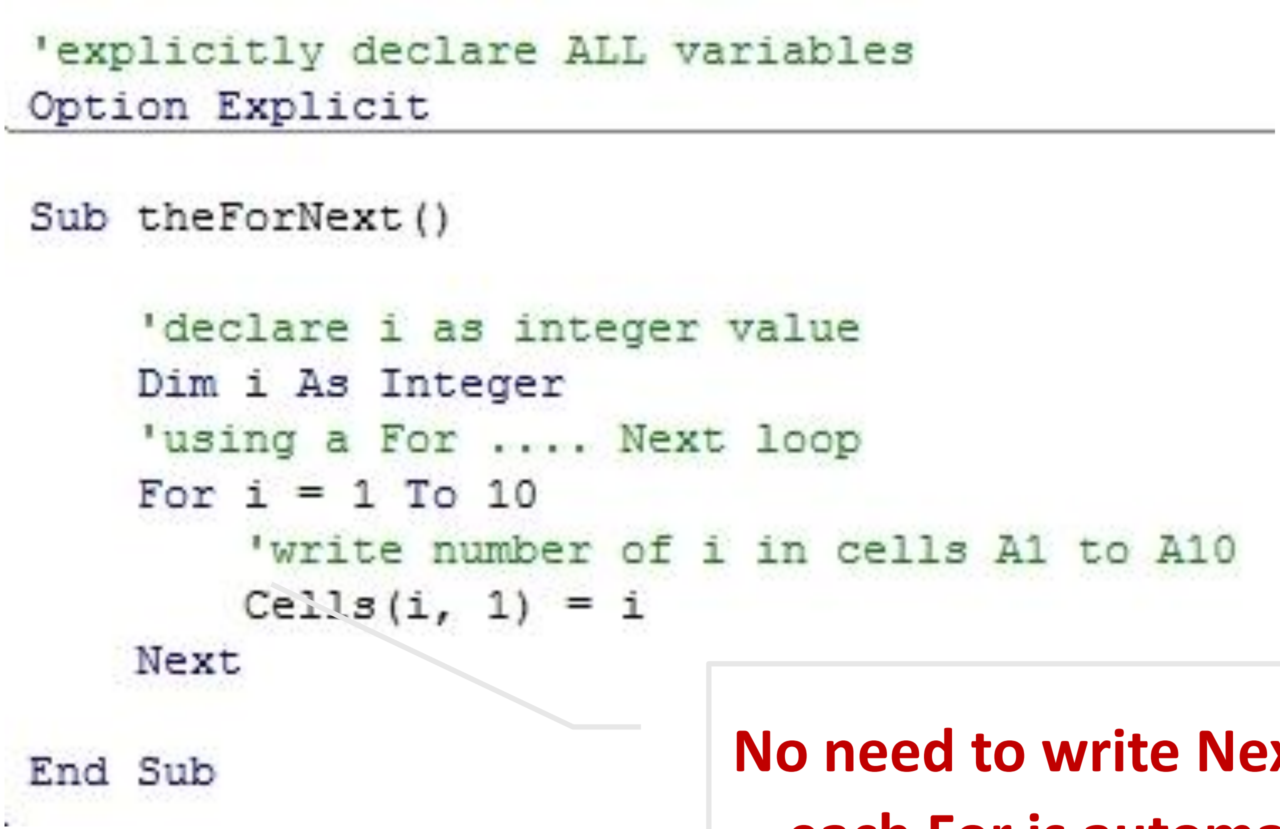


```
'explicitly declare ALL variables
Option Explicit

Sub theIfThen()

    'declare x as integer value
    Dim x As Integer

    x = InputBox("Enter a number between 0 and 100")
    'using an If .... Then statement
    If x > 100 Then
        'output test message if error
        MsgBox "Error!"
    End If

End Sub
```

- Can you get it to output a message with the input value in it

The For .... Next statement

- We can repeat an action a fixed number of times using this

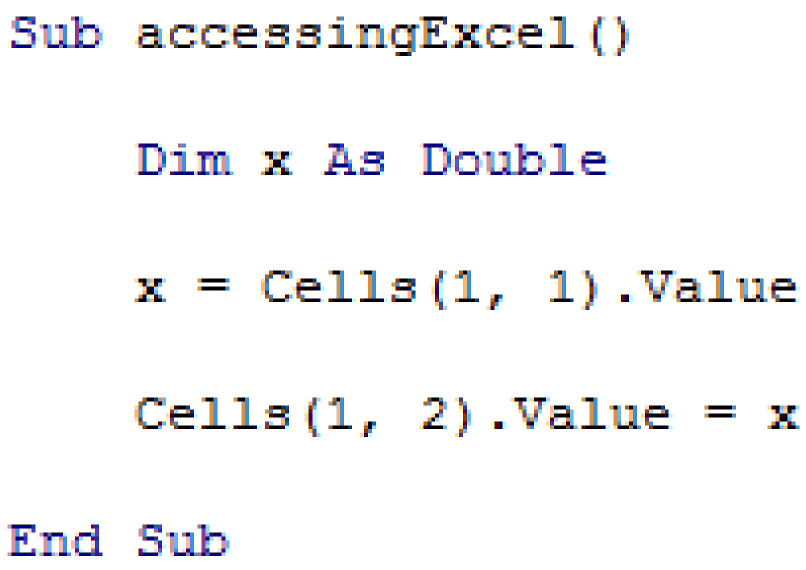


```
'explicitly declare ALL variables
Option Explicit

Sub theForNext()

    'declare i as integer value
    Dim i As Integer
    'using a For ..... Next loop
    For i = 1 To 10
        'write number of i in cells A1 to A10
        Cells(i, 1) = i
    Next

End Sub
```

No need to write Next i since each For is automatically linked to the relevant Next.

Communicating with an Excel Sheet

- To get a value from an Excel sheet you can get and set the value of a cell using Cells(row,column) to identify which cell you want

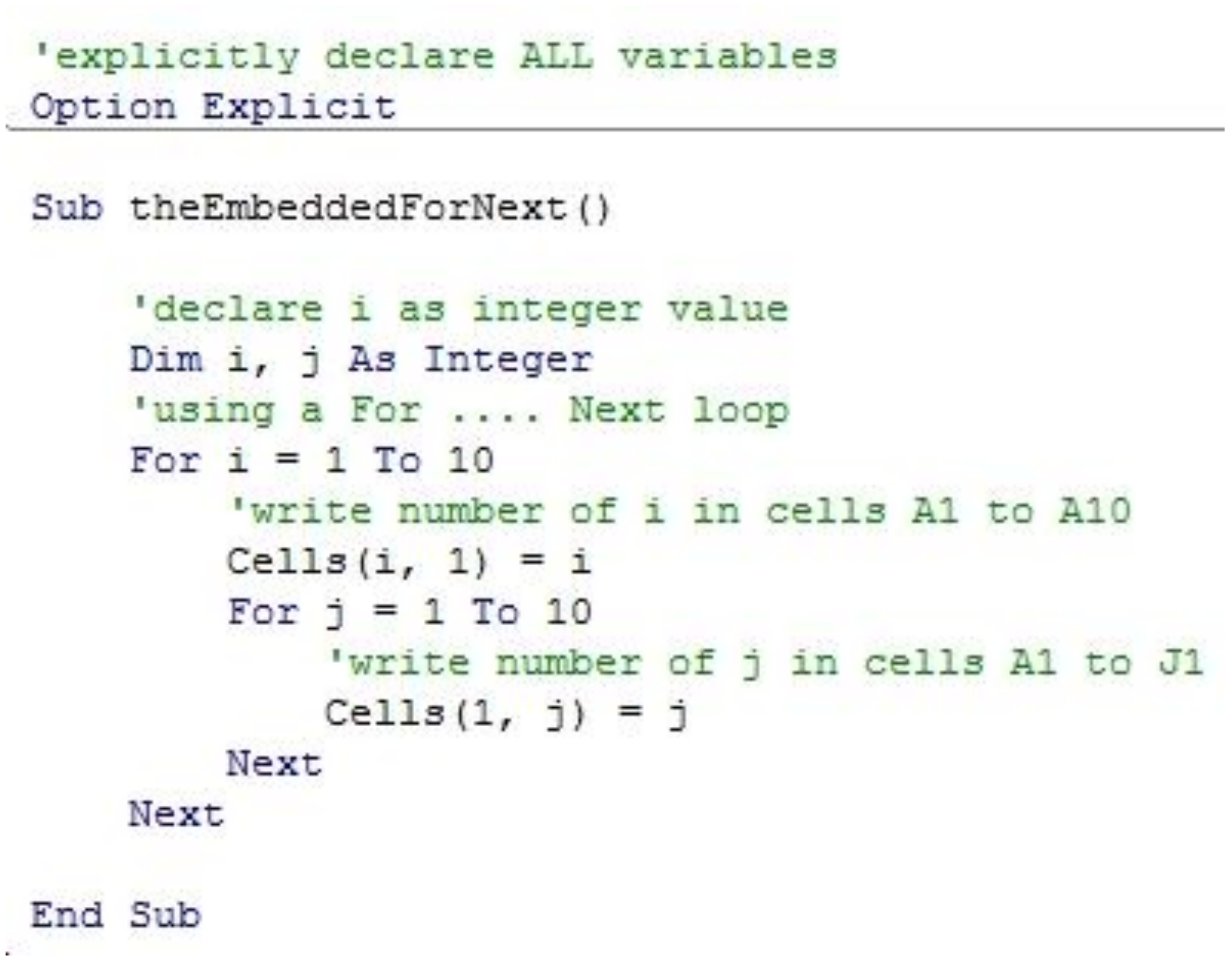


```
Sub accessingExcel()

    Dim x As Double

    x = Cells(1, 1).Value

    Cells(1, 2).Value = x

End Sub
```

- This will be on the active worksheet when you call this macro

Nested For .... Next loops

- We can loop over an array by having a nested loop



```
'explicitly declare ALL variables
Option Explicit

Sub theEmbeddedForNext()

    'declare i as integer value
    Dim i, j As Integer
    'using a For .... Next loop
    For i = 1 To 10
        'write number of i in cells A1 to A10
        Cells(i, 1) = i
        For j = 1 To 10
            'write number of j in cells A1 to J1
            Cells(1, j) = j
        Next
    Next

End Sub
```

Accessing a specific Range

- If you want it to call a specific sheet then you need to specify that using the sheet name or a specific range name

```
Sub accessingExcelSheets()

    Dim p As Double
    Dim rate As Double

    p = Range("Price").Value

    rate = Worksheets("Rates").Cells(1, 1).Value

End Sub
```

Using Subs

- To be honest, I rarely use them
- However, they may be useful as part of a smart macro that does formatting or data importing or repetitive calculations
- Too fragile – you can amend cells anywhere
- What I do use a lot are user-defined functions

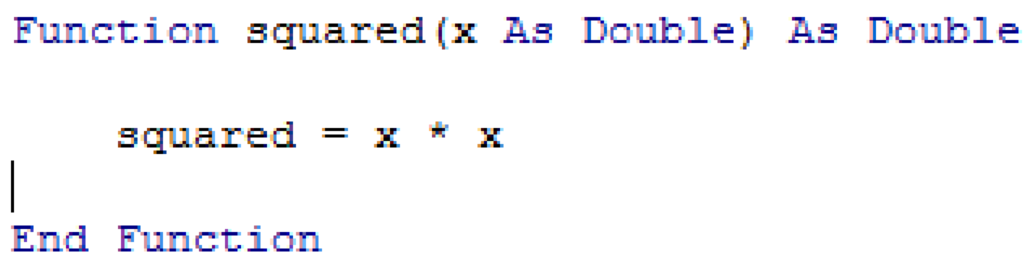
VBA Functions

User Defined Functions

- Powerful feature of Excel VBA is user defined functions (UDF)
- A function receives inputs and returns a data type
- A function cannot change the value of a cell except those from which it is being called
- Functions are robust and look like ordinary Excel functions
- User-Defined Functions make Excel very powerful

The Basics of a VBA User-Defined Function

- A function receives inputs and returns a data type
- We have to respect the correct syntax – here is an example



```
Function squared(x As Double) As Double

    squared = x * x

End Function
```

- The function has to have a name
- We declare the type of data it returns
- We declare the list of function arguments
- We declare the type of each argument – if known in advance
- A variable with the function name represents the return value
- Use indentation to make it easier to see the function

Some Warnings

- Don't put a function of the same name in more than one modules
- A function can return a value or date or a string or an array etc…
- It can also return a Variant - a variant can be any data type
- Use a variant if the return type can change
- For example a function that returns a value or an error message should be defined as returning a variant

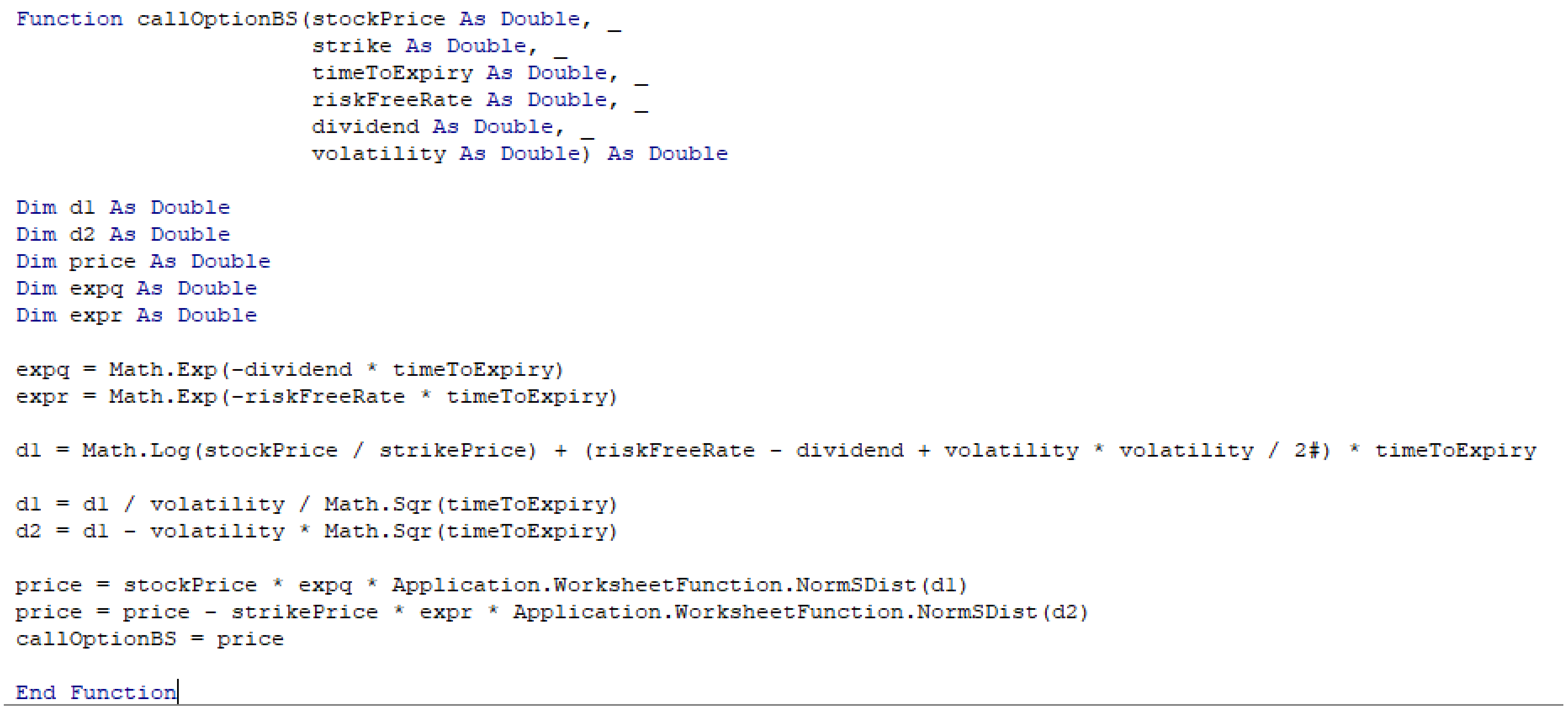
VBA Applications: Black Scholes

Black Scholes in VBA

- We can finally implement Black-Scholes in VBA
- Use Option Explicit to force you to define all variables
- Use good names for inputs for users
- N(x) is simply the **NORMSDIST(x)** function in Excel
- Remember that it is an Excel function – so how do you call it ?
- You use **Application.WorksheetFunction.Normsdist(x)**
- Make sure you name the result correctly

Black-Scholes Call Option in VBA

- Can you find my deliberate bug – using option explicit to find it
- Now do you see why option Explicit is important ?



```
Function callOptionBS(stockPrice As Double, _
                      strike As Double, _
                      timeToExpiry As Double, _
                      riskFreeRate As Double, _
                      dividend As Double, _
                      volatility As Double) As Double

Dim d1 As Double
Dim d2 As Double
Dim price As Double
Dim expq As Double
Dim expr As Double

expq = Math.Exp(-dividend * timeToExpiry)
expr = Math.Exp(-riskFreeRate * timeToExpiry)

d1 = Math.Log(stockPrice / strikePrice) + (riskFreeRate - dividend + volatility * volatility / 2#) * timeToExpiry

d1 = d1 / volatility / Math.Sqr(timeToExpiry)
d2 = d1 - volatility * Math.Sqr(timeToExpiry)

price = stockPrice * expq * Application.WorksheetFunction.NormSDist(d1)
price = price - strikePrice * expr * Application.WorksheetFunction.NormSDist(d2)
callOptionBS = price

End Function
```

Black Scholes Put Option

- The pricing equation for a put option is

$$P(S,K,r,T,\sigma,q) = K\exp(-rT)N(-d_2) - S\exp(-qT)N(-d_1)$$

- What is the best way to extend our VBA code to price a put option
- We have 2 choices
  - **Write a new function to do put options**
  - **Write a new function which can handle both call and put options**
- Let us consider both of these

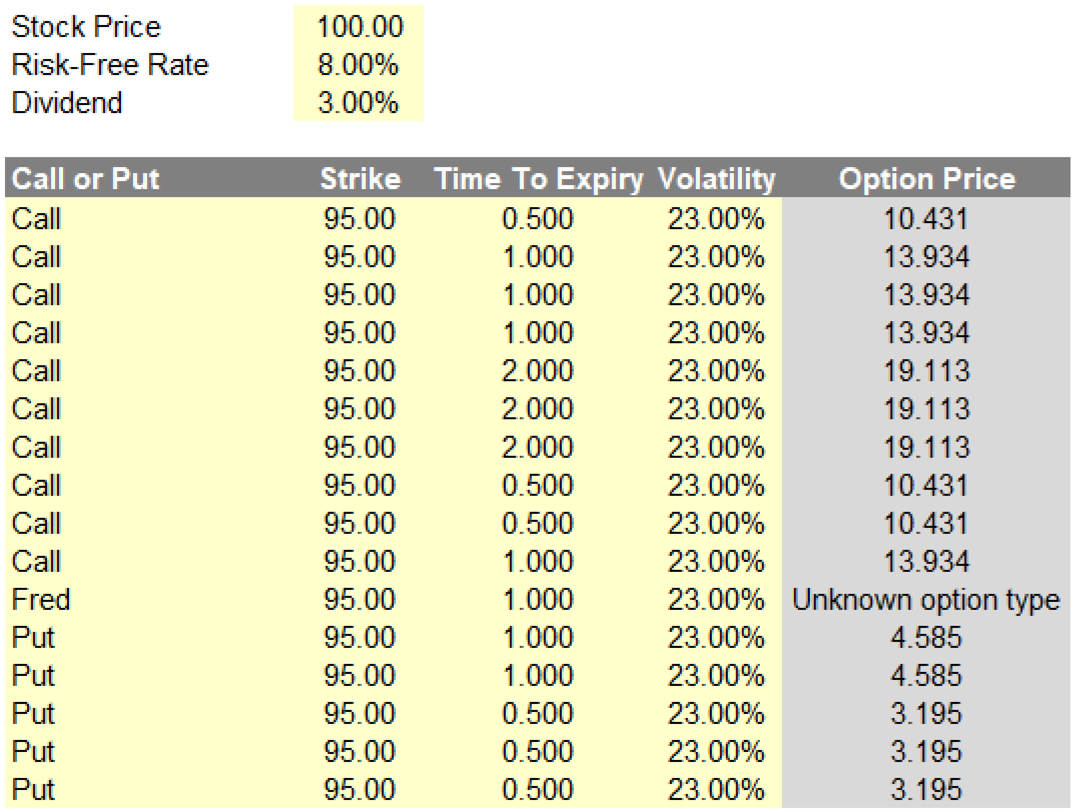
Choice 1: Write a new function

- We can copy and paste and amend to write a new function to price put options - this is very easy to do

```
Function putOptionBS(stockPrice As Double, _
                     strikePrice As Double, _
                     timeToExpiry As Double, _
                     riskFreeRate As Double, _
                     dividend As Double, _
                     volatility As Double)

Dim d1 As Double
Dim d2 As Double
Dim price As Double
Dim expq As Double
Dim expr As Double
Dim drift As Double

expq = Math.Exp(-dividend * timeToExpiry)
expr = Math.Exp(-riskFreeRate * timeToExpiry)

' sometimes it is useful to define intermediate variables to simplify complex formulae
drift = (riskFreeRate - dividend + volatility * volatility / 2#)

d1 = Math.Log(stockPrice / strikePrice) + drift * timeToExpiry
d1 = d1 / volatility / Math.Sqr(timeToExpiry)
d2 = d1 - volatility * Math.Sqr(timeToExpiry)

price = strikePrice * expr * Application.WorksheetFunction.NormSDist(-d2)
price = price - stockPrice * expq * Application.WorksheetFunction.NormSDist(-d1)
putOptionBS = price

End Function
```

But suppose I have a portfolio of options

- I do not want to change my sheet when I buy a call or put
- I want the pricing code to just figure that out from the deal description which would be sucked from my position database



Stock Price	100.00
Risk-Free Rate	8.00%
Dividend	3.00%

Call or Put	Strike	Time To Expiry	Volatility	Option Price
Call	95.00	0.500	23.00%	10.431
Call	95.00	1.000	23.00%	13.934
Call	95.00	1.000	23.00%	13.934
Call	95.00	1.000	23.00%	13.934
Call	95.00	2.000	23.00%	19.113
Call	95.00	2.000	23.00%	19.113
Call	95.00	2.000	23.00%	19.113
Call	95.00	0.500	23.00%	10.431
Call	95.00	0.500	23.00%	10.431
Call	95.00	1.000	23.00%	13.934
Fred	95.00	1.000	23.00%	Unknown option type
Put	95.00	1.000	23.00%	4.585
Put	95.00	1.000	23.00%	4.585
Put	95.00	0.500	23.00%	3.195
Put	95.00	0.500	23.00%	3.195
Put	95.00	0.500	23.00%	3.195

Choice 2: Write one function for calls and puts

```
Function optionBS(stockPrice As Double, _
                  strikePrice As Double, _
                  timeToExpiry As Double, _
                  riskFreeRate As Double, _
                  dividend As Double, _
                  volatility As Double, _
                  optionType As String) As Variant

Dim d1 As Double
Dim d2 As Double
Dim price As Double
Dim expq As Double
Dim expr As Double

If volatility < 0 Then
    optionBS = "Volatility must be greater than zero"
    Exit Function
End If

expq = Math.Exp(-dividend * timeToExpiry)
expr = Math.Exp(-riskFreeRate * timeToExpiry)

d1 = Math.Log(stockPrice / strikePrice) + (riskFreeRate - dividend + volatility * volatility / 2#) * timeToExpiry
d1 = d1 / volatility / Math.Sqr(timeToExpiry)
d2 = d1 - volatility * Math.Sqr(timeToExpiry)

If optionType = "Call" Then
    price = stockPrice * expq * Application.WorksheetFunction.NormSDist(d1)
    price = price - strikePrice * expr * Application.WorksheetFunction.NormSDist(d2)
    optionBS = price
ElseIf optionType = "Put" Then
    price = strikePrice * expr * Application.WorksheetFunction.NormSDist(-d2)
    price = price - stockPrice * expq * Application.WorksheetFunction.NormSDist(-d1)
    optionBS = price
Else
    optionBS = "Unknown option type"
End If

End Function
```

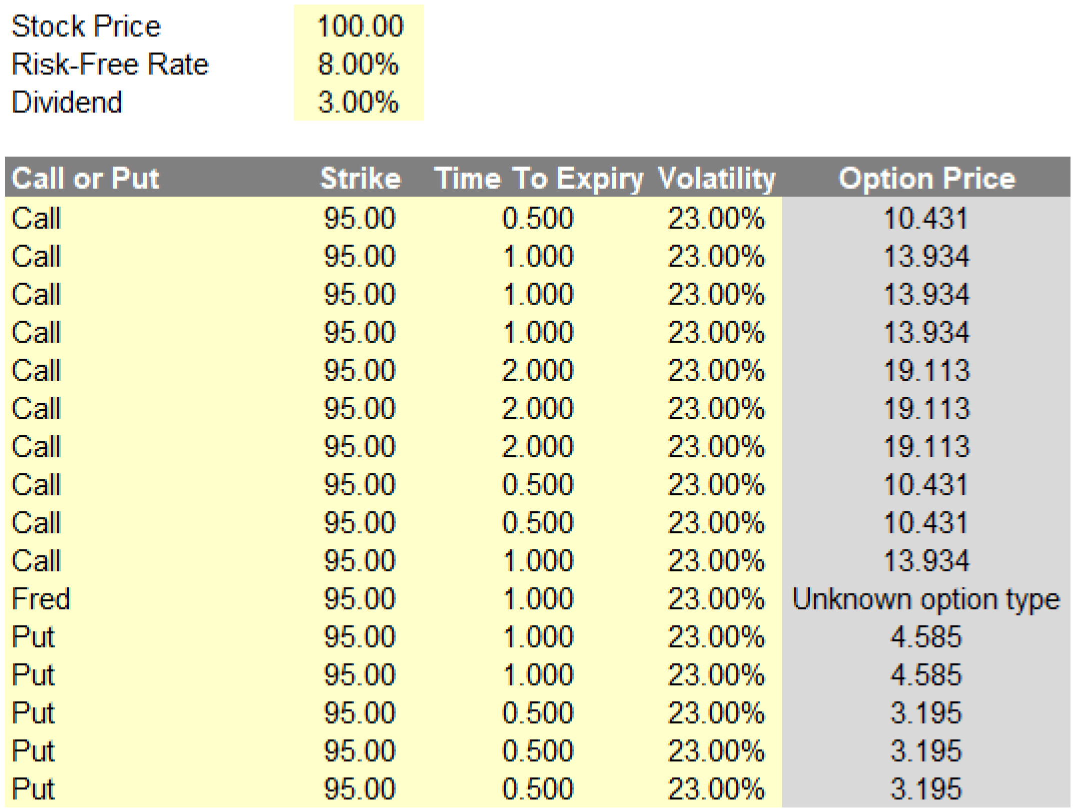
Return a variant in case of error so that a string can be returned otherwise you get an error

Read in the option type as a string

Check for error and return error message

Validation of inputs is important

- Error messages must be returned, and they must be informative
- Type in a silly name for the option type



Stock Price	100.00
Risk-Free Rate	8.00%
Dividend	3.00%

Call or Put	Strike	Time To Expiry	Volatility	Option Price
Call	95.00	0.500	23.00%	10.431
Call	95.00	1.000	23.00%	13.934
Call	95.00	1.000	23.00%	13.934
Call	95.00	1.000	23.00%	13.934
Call	95.00	2.000	23.00%	19.113
Call	95.00	2.000	23.00%	19.113
Call	95.00	2.000	23.00%	19.113
Call	95.00	0.500	23.00%	10.431
Call	95.00	0.500	23.00%	10.431
Call	95.00	1.000	23.00%	13.934
Fred	95.00	1.000	23.00%	Unknown option type
Put	95.00	1.000	23.00%	4.585
Put	95.00	1.000	23.00%	4.585
Put	95.00	0.500	23.00%	3.195
Put	95.00	0.500	23.00%	3.195
Put	95.00	0.500	23.00%	3.195

- Avoid use of Message Box. Why ?

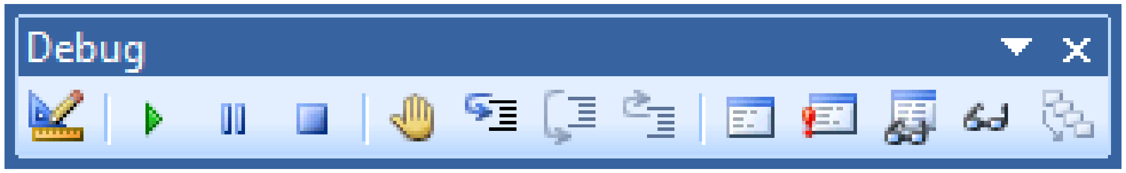
Using the Debugger

Debugging

- When the code becomes complicated, bugs get harder to find
- We need tools to understand what is happening
- One of the most powerful tools is the debugger
- Allows you to step through the code to find errors while it is running

Controlling the debugger and Watch

- We must first set a break point in the VBA
- **To do this you click in the gray margin at the left side on the line you want to have a breakpoint**
- Use the following toolbar to control the debugging

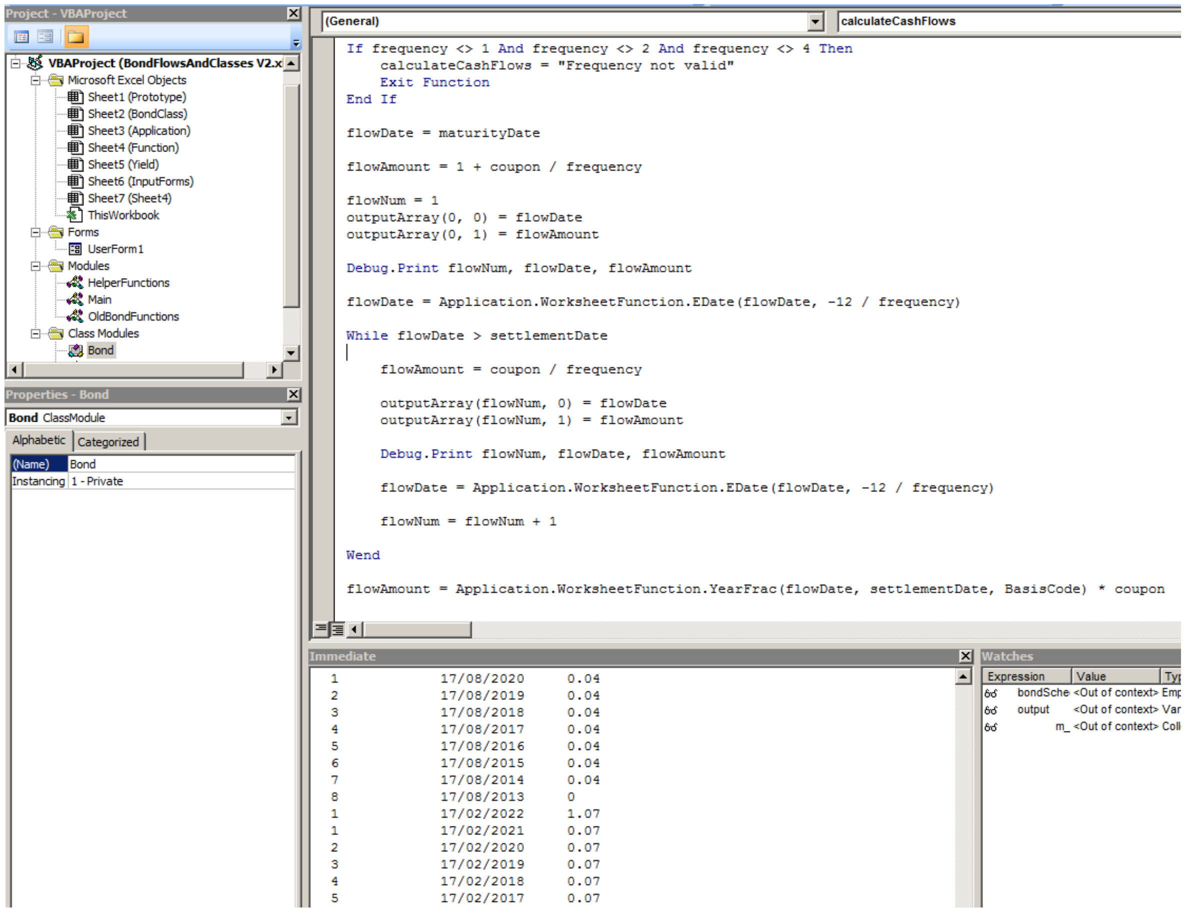


- If you do not see it go to View->ToolBars->Debug
- The step over, step into and pause buttons are important ones
- We observe the values of variables using the watch window
- **Select the variable and then right click on it**
- Select "Add watch …"

Debug.Print

- For complex or lengthy calculations, it helps to print out intermediate results
- In this case, it becomes useful to use the **Debug.Print** command

Use CTRL+G to make the immediate window visible



```
If frequency <> 1 And frequency <> 2 And frequency <> 4 Then
    calculateCashFlows = "Frequency not valid"
    Exit Function
End If

flowDate = maturityDate

flowAmount = 1 + coupon / frequency

flowNum = 1
outputArray(0, 0) = flowDate
outputArray(0, 1) = flowAmount

Debug.Print flowNum, flowDate, flowAmount

flowDate = Application.WorksheetFunction.EDate(flowDate, -12 / frequency)

While flowDate > settlementDate

    flowAmount = coupon / frequency

    outputArray(flowNum, 0) = flowDate
    outputArray(flowNum, 1) = flowAmount

    Debug.Print flowNum, flowDate, flowAmount

    flowDate = Application.WorksheetFunction.EDate(flowDate, -12 / frequency)

    flowNum = flowNum + 1

Wend

flowAmount = Application.WorksheetFunction.YearFrac(flowDate, settlementDate, BasisCode) * coupon
```

```
 1             17/08/2020     0.04
 2             17/08/2019     0.04
 3             17/08/2018     0.04
 4             17/08/2017     0.04
 5             17/08/2016     0.04
 6             17/08/2015     0.04
 7             17/08/2014     0.04
 8             17/08/2013     0
 1             17/02/2022     1.07
 1             17/02/2021     0.07
 2             17/02/2020     0.07
 3             17/02/2019     0.07
 4             17/02/2018     0.07
 5             17/02/2017     0.07
```

Managing Arrays in VBA Functions

Inputting and Outputting Arrays

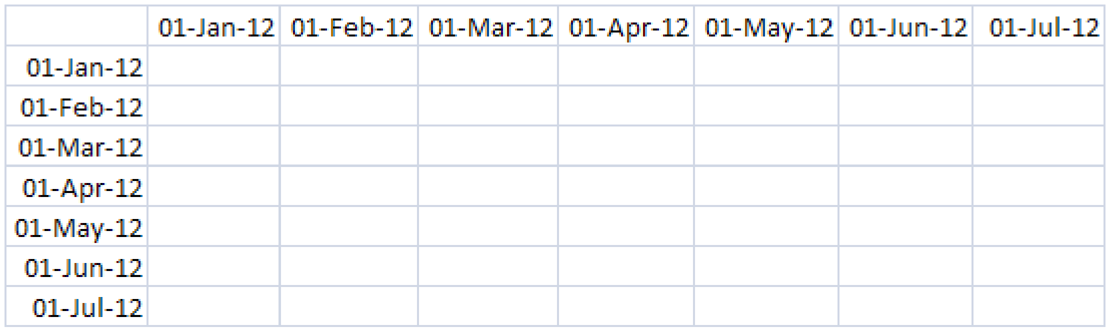
- So far, all our functions return just one value – a number or text
- Sometimes we want to get more than one value back
- We need to return an array
- Sometimes we also want to input an array
- This is called a Range
- **Being able to do both of these is very useful, safer, more powerful and more efficient**

Returning an Array from a Function

- Before we can discuss arrays, we need to learn how to get an array back from a function
- To try this, let's use the Transpose function which takes a range

= TRANSPOSE(range)

Type function then Ctrl + Shift + Enter pressed simultaneously.



	01-Jan-12	01-Feb-12	01-Mar-12	01-Apr-12	01-May-12	01-Jun-12	01-Jul-12
01-Jan-12							
01-Feb-12							
01-Mar-12							
01-Apr-12							
01-May-12							
01-Jun-12							
01-Jul-12							

- Convert the column of dates into a transposed row of dates as shown

Outputting an Array: The Detailed Steps

- Here are the steps:

1. Type the function into the top left cell of where you want the output to go
2. Put the cursor in this cell and drag it to select the output range
3. Put the cursor at the end of the formula in the formula window where the formula is showing
4. Holding down both Ctrl + Shift at the same time Hit the Enter key

Let's Write Our own Matrix ADD function

- Excel does not do have matrix addition and subtraction functions
- Shame as it would be quite nice to have some validation etc….
- We want to write a simple function to add two matrices
- It will be smarter than simply adding since it will check matrix sizes so adds some safety
- Let us call the function **MADD** for Matrix Add
- It takes two **Ranges** and then outputs an array with the sum

Range versus Array Element Numbering

- By default, Excel automatically starts all arrays at 0.
- But when dealing with Excel Ranges, indexing always starts at 1
- So always reference the first element (1,1) and not (0,0)
- **To be consistent if we want to start them at 1, we need to add the command Option Base 1 at the start of the VBA module**

Excel Range

1,1	1,2	1,3
2,1	2,2	2,3
3,1	3,2	3,3

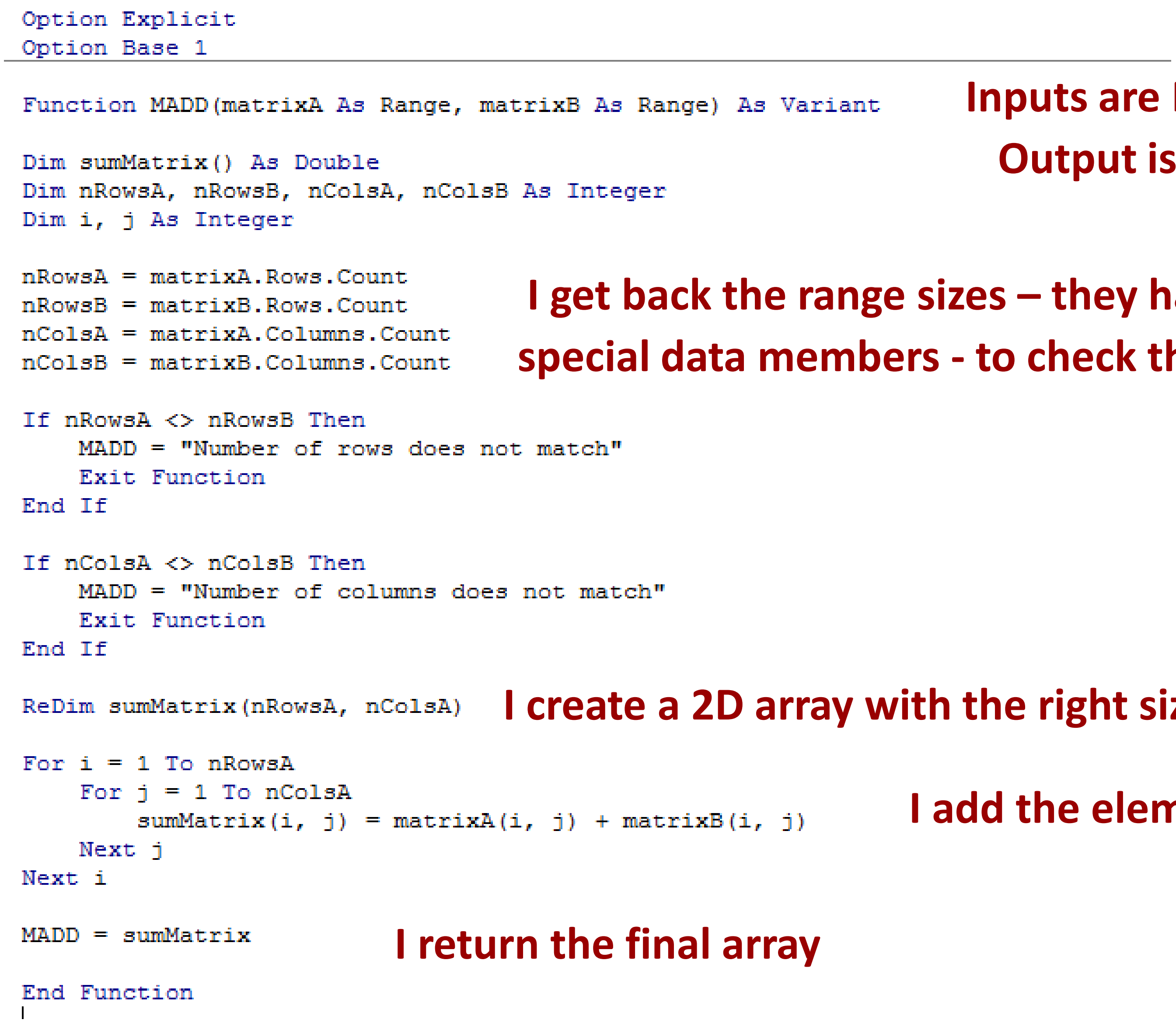
VBA 2D Array with Option Base 0

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

VBA 2D Array with Option Base 1

1,1	1,2	1,3
2,1	2,2	2,3
3,1	3,2	3,3

Matrix Addition – our implementation



```
Option Explicit
Option Base 1

Function MADD(matrixA As Range, matrixB As Range) As Variant

Dim sumMatrix() As Double
Dim nRowsA, nRowsB, nColsA, nColsB As Integer
Dim i, j As Integer

nRowsA = matrixA.Rows.Count
nRowsB = matrixB.Rows.Count
nColsA = matrixA.Columns.Count
nColsB = matrixB.Columns.Count

If nRowsA <> nRowsB Then
    MADD = "Number of rows does not match"
    Exit Function
End If

If nColsA <> nColsB Then
    MADD = "Number of columns does not match"
    Exit Function
End If

ReDim sumMatrix(nRowsA, nColsA)

For i = 1 To nRowsA
    For j = 1 To nColsA
        sumMatrix(i, j) = matrixA(i, j) + matrixB(i, j)
    Next j
Next i

MADD = sumMatrix

End Function
```

Inputs are Ranges and Output is a Variant

I get back the range sizes – they have special data members - to check them

I create a 2D array with the right size

I add the elements

I return the final array

Adding Matrices

- Create some examples to test.
- The arrays being added must have the same dimensions
- Try some examples with the wrong dimensions to see that you get an error message

1.00	2.00	4.00		5.00	7.00	-2.00		6.00	9.00	2.00
2.00	7.00	2.00	+	3.00	2.00	4.00	=	5.00	9.00	6.00
4.00	-3.00	1.00		2.00	3.00	4.00		6.00	0.00	5.00

VBA Applications: Monte Carlo

What is Monte Carlo ?

- Monte Carlo is an algorithm which relies on the repeated sampling of a probability distribution to compute an expectation
- The term "Monte Carlo method" was coined in the 1940s by physicists who used it to analyse nuclear physics at Los Alamos
- A reference to the roulette wheels of Monte Carlo which are a sort of random number generator !

The Principle behind Monte Carlo

- The MC principle is that

Present value of payoff in scenario p

$$V_{Exact} = E\left[\Phi(\omega)\right] = \lim_{P \to \infty} \frac{1}{P} \sum_{p=1}^{P} \Phi(\omega_p)$$

- where
  - Φ is some function which represents the discounted payoff of the security e.g. for a European call option it is Max(S(T)-K,0)
  - ω is a set of random variables which represent the underlying risks of the asset which determine its payoff
  - P is the number of repeated samplings of the probability distribution from which the ω is drawn and
- In practice P is finite so we have

$$V_{Approx} = \frac{1}{P} \sum_{p=1}^{P} \Phi(\omega_p)$$

Pricing a Call Option

- The payoff of a call option on a stock with expiry time T is

$$Max(S(T)-K,0)$$

- We need to draw S(T), the price of the stock at option expiry from a probability distribution in each path
- We need first to specify the random process for S(T) which in Black-Scholes is lognormal
- Mathematically the distribution of the time T stock price is

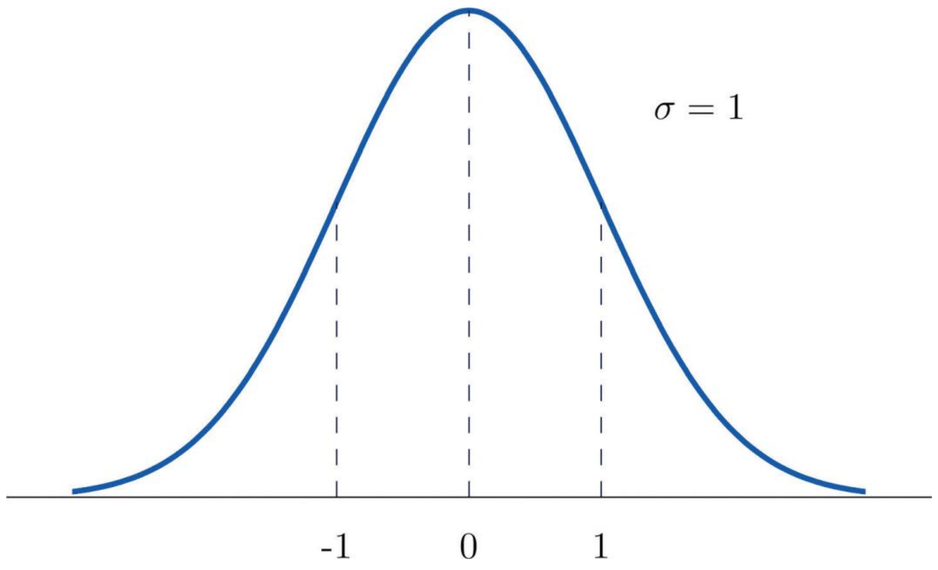
$$S(T)=S(0)\exp\left((r-\sigma^2/2)T+\sigma g\sqrt{T}\right)$$

where g is a Gaussian random variable with mean zero and standard deviation of 1 and you know the other parameters

Random Number Generation

- One way to get truly random numbers is with a random process e.g. radioactive decay produced by a small amount of plutonium.
- Not a practical solution to pricing options on a trading floor !!
- Random numbers generated by a computer are almost random
- Excel VBA has two uniform random number generators!
  - In Excel the rand() function gives us a uniform in [0 ,1]
  - In VBA the rnd() function does the same thing
- A random number generator always needs a seed which initialises it – this is important if we are to compare output

But we need a Gaussian Random Draw



- We can generate a uniform random number using rand()
- But generating a Gaussian random number is a bit more tricky
- One approach is to invert the cumulative normal distribution
- In Excel we can simply write "=NormsInv(Rand())"
- In VBA, we must write

```
g = Application.WorksheetFunction.NormSInv(Rnd())
```

Monte Carlo Code for Pricing a Call Option in VBA

```
Option Explicit

Function callPriceMC(s0 As Double, _
                     k As Double, _
                     T As Double, _
                     r As Double, _
                     q As Double, _
                     vol As Double, _
                     numTrials As Long, _
                     seed As Long) As Double

Rnd (-20)
Randomize (seed)

Dim i As Long
Dim g, st As Double

callPriceMC = 0

For i = 1 To numTrials
    g = Application.WorksheetFunction.NormSInv(Rnd())
    st = s0 * Exp((r - q - vol * vol / 2) * T + g * vol * Sqr(T))
    callPriceMC = callPriceMC + Application.WorksheetFunction.Max(st - k, 0)
Next i

callPriceMC = callPriceMC * Exp(-r * T) / numTrials

End Function
```

To save typing I am using short names for variables – this is not always recommended but, in this case, they are very well known

This resets the random number sequence, so you always get the same result for the same seed – important

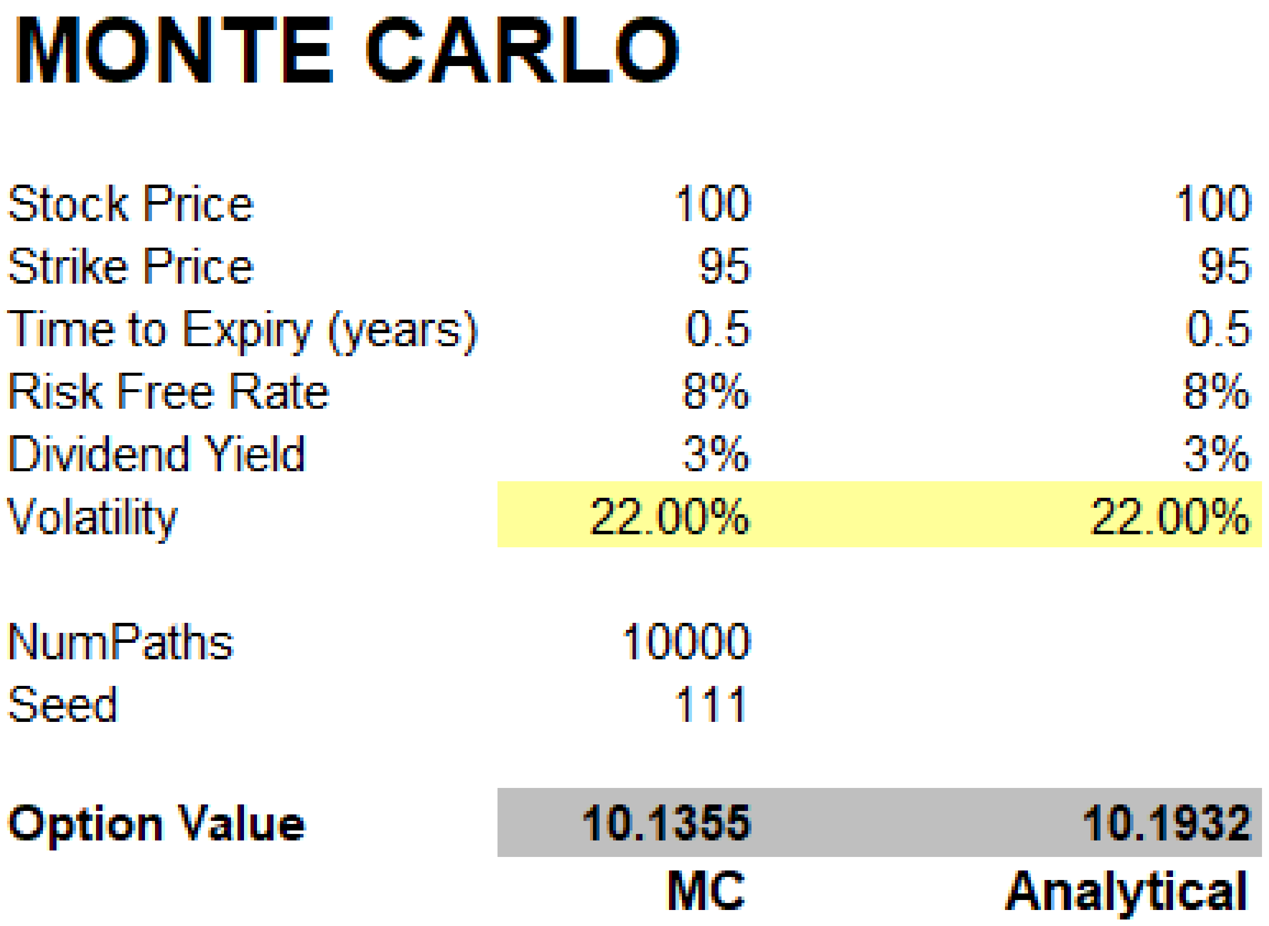
The Problem with Monte Carlo

- The problem with MC is that there is statistical noise - as long as P is finite the results of the Monte-Carlo pricing are not exact
- If the random numbers across each path are independently generated, the error in our estimate of the price scales as

$$\varepsilon = \left| V_{Exact} - V_{Approx} \right| \sim \frac{1}{\sqrt{P}}$$

- This means you need a lot of paths to do MC with high accuracy
- In VBA you can do 20,000 paths for a 1d model – not bad!
- For some products you need 100 times this – in this case you need C or Python/Numba which is 100 times faster

Compare MC and Black-Scholes Analytical



MONTE CARLO

Stock Price	100	100
Strike Price	95	95
Time to Expiry (years)	0.5	0.5
Risk Free Rate	8%	8%
Dividend Yield	3%	3%
Volatility	22.00%	22.00%
NumPaths	10000	
Seed	111	
Option Value	**10.1355**	**10.1932**
	MC	**Analytical**

- I have a color-coding scheme – yellow = input, gray = calculation

VBA Application: Bond Analytics

Bond Analytics

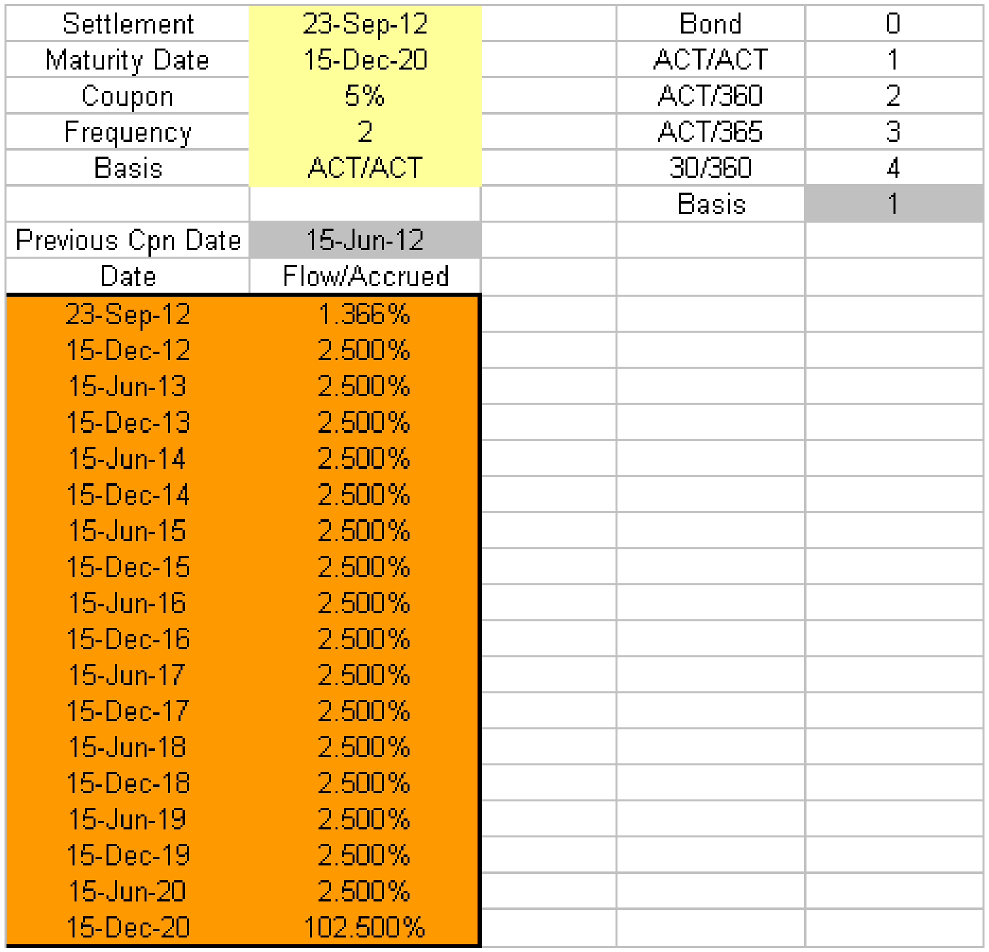
- We wish to program some bond analytics
- To begin, we first need to build a function which gives us the dates and payments of a standard bond
- We will then use this function to do other work
- So we will have a function which calls another function

Must calculate the Coupon Schedule first

- The first function takes in
  - Settlement date
  - Maturity date
  - Coupon
  - Frequency
  - Accrual basis convention
- And then outputs two vectors (in a matrix)
  - **Vector 1 is the set of coupon and principal dates with the first date being the bond settlement date**
  - **Vector 2 is the set of cash flow payments with the first cash flow on the bond settlement date being used to hold the value of the accrued interest even though no flow actually occurs on that date**

Calculating Bond Flows

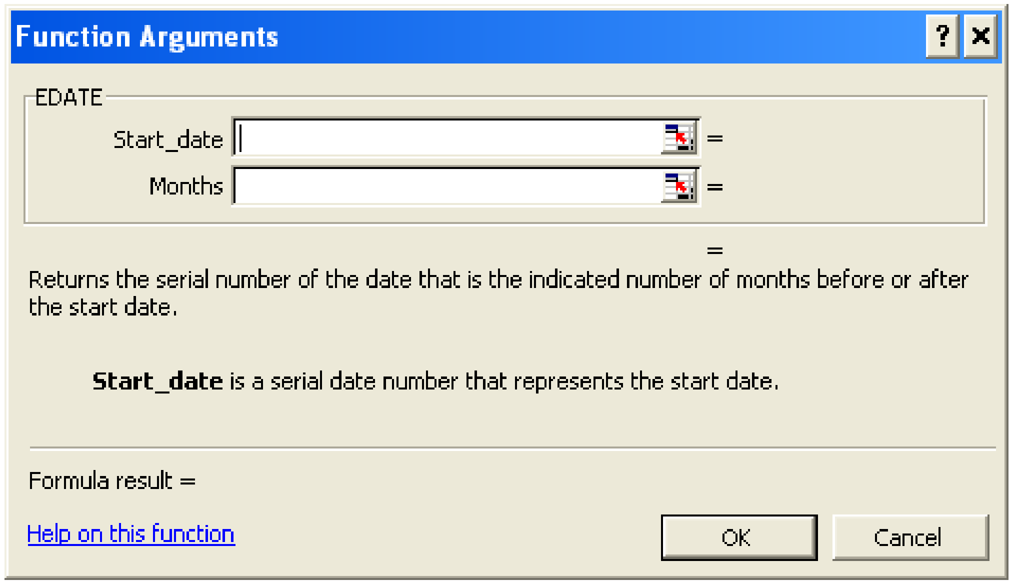
- The best way to do this is to start at the maturity date and to step backwards
- We stop as soon as we are at a coupon date which is before the settlement date
- This allows us to calculate the accrued interest too
- This is how we want the function to look – the block in orange



Settlement	23-Sep-12		Bond	0
Maturity Date	15-Dec-20		ACT/ACT	1
Coupon	5%		ACT/360	2
Frequency	2		ACT/365	3
Basis	ACT/ACT		30/360	4
			Basis	1
Previous Cpn Date	15-Jun-12			
Date	Flow/Accrued			
23-Sep-12	1.366%			
15-Dec-12	2.500%			
15-Jun-13	2.500%			
15-Dec-13	2.500%			
15-Jun-14	2.500%			
15-Dec-14	2.500%			
15-Jun-15	2.500%			
15-Dec-15	2.500%			
15-Jun-16	2.500%			
15-Dec-16	2.500%			
15-Jun-17	2.500%			
15-Dec-17	2.500%			
15-Jun-18	2.500%			
15-Dec-18	2.500%			
15-Jun-19	2.500%			
15-Dec-19	2.500%			
15-Jun-20	2.500%			
15-Dec-20	102.500%			

How to calculate bond flows

- We start at the maturity date and step backwards in steps of n months where n = 12 / frequency
- We cannot simply subtract days from the date to do this
- We need to make sure for example that 15 November steps back to the 15 March
- We use the EDATE function whose arguments are shown below

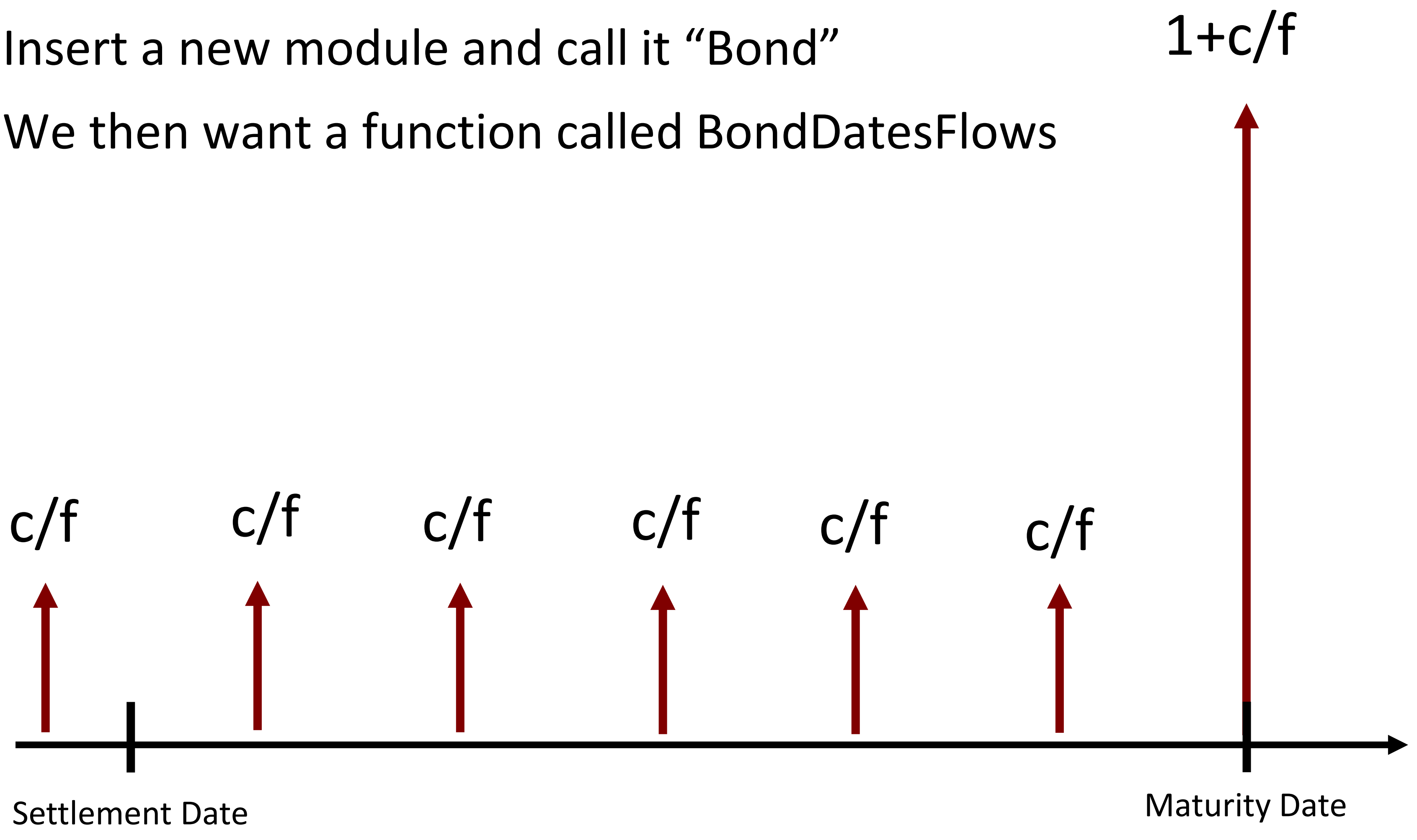


Returns the serial number of the date that is the indicated number of months before or after the start date.

Start_date is a serial date number that represents the start date.

To Create the Function

- Open the Visual Basic editor
- Insert a new module and call it "Bond"
- We then want a function called BondDatesFlows



Getting the basis code

- We write a function, and that function checks the input string

```
Function getBasisCode(basisStr As String) As Integer

If basisStr = "Bond" Then
    getBasisCode = 0
ElseIf basisStr = "ACT/ACT" Then
    getBasisCode = 1
ElseIf basisStr = "ACT/360" Then
    getBasisCode = 2
ElseIf basisStr = "ACT/365" Then
    getBasisCode = 3
ElseIf basisStr = "30/360" Then
    getBasisCode = 4
Else
    getBasisCode = -1
    Exit Function
End If

End Function
```

- If there is a problem, we send back a value of -1

Top of the Function – Declaration and Validation

```
Option Explicit
Option Base 0
```

Make sure all arrays start at zero

```
Function bondDatesFlows(settlementDate As Date, _
                        maturityDate As Date, _
                        coupon As Double, _
                        frequency As Integer, _
                        basisString As String) As Variant

Dim basisCode As Integer
Dim outputArray(100, 2) As Double

Dim flowDate As Date
Dim flowAmount As Double
Dim flowNum, i As Integer

If frequency <> 1 And frequency <> 2 And frequency <> 4 Then
    bondDatesFlows = "Frequency not valid"
    Exit Function
End If

basisCode = getBasisCode(basisString)

If basisCode = -1 Then
    bondDatesFlows = "Unknown basis string"
    Exit Function
End If
```

We validate the inputs – always saves time in the long run

Schedule Generation Logic I

```
Dim flowDate As Date
Dim flowAmount As Double
Dim flowNum, i As Integer

If frequency <> 1 And frequency <> 2 And frequency <> 4 Then
    bondDatesFlows = "Frequency not valid"
    Exit Function
End If

basisCode = getBasisCode(basisString)

If basisCode = -1 Then
    bondDatesFlows = "Unknown basis string"
    Exit Function
End If

flowDate = maturityDate

flowAmount = 1 + coupon / frequency

outputArray(0, 0) = flowDate
outputArray(0, 1) = flowAmount
```

Validation checks

Initialise flow dates to maturity date and payment to par plus coupon so we can work backwards

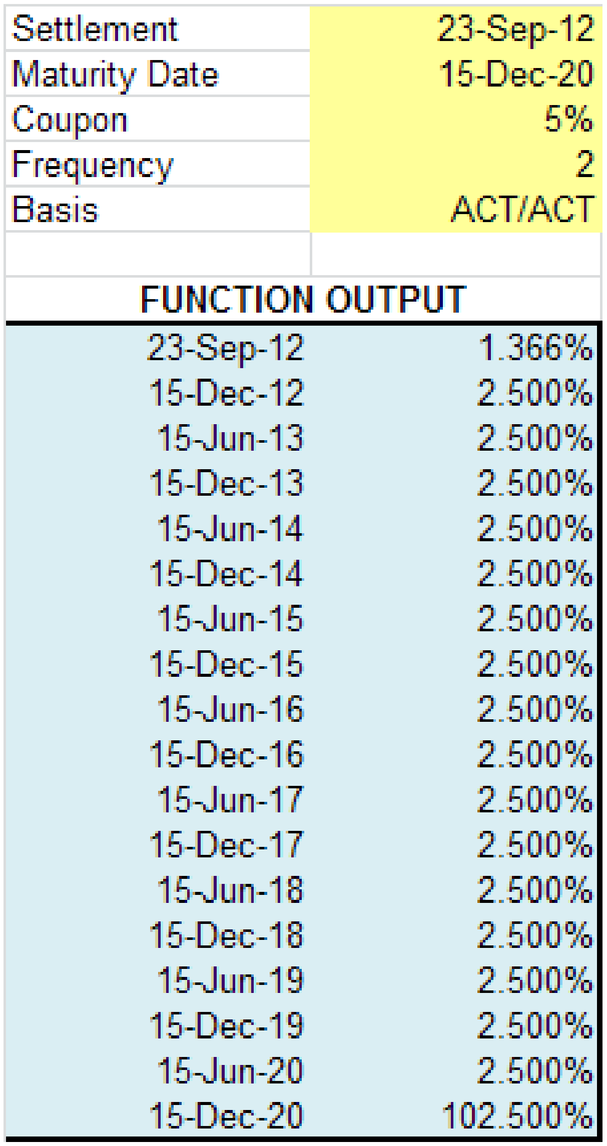
Schedule Generation Logic II

```
flowNum = 1
flowDate = Application.WorksheetFunction.EDate(flowDate, -12 / frequency)

While flowDate > settlementDate

    outputArray(flowNum, 0) = flowDate
    outputArray(flowNum, 1) = coupon / frequency

    flowDate = Application.WorksheetFunction.EDate(flowDate, -12 / frequency)

    flowNum = flowNum + 1

Wend

outputArray(flowNum, 0) = settlementDate
outputArray(flowNum, 1) = Application.WorksheetFunction.YearFrac(flowDate, settlementDate, basisCode) * coupon

Dim orderedArray() As Double
ReDim orderedArray(flowNum, 2)

For i = 0 To flowNum

    orderedArray(i, 0) = outputArray(flowNum - i, 0)
    orderedArray(i, 1) = outputArray(flowNum - i, 1)

Next i

bondDatesFlows = orderedArray

End Function
```

Start at maturity date and step backwards

Reverse all the dates so we have them in chronological order

Function Output



Settlement	23-Sep-12
Maturity Date	15-Dec-20
Coupon	5%
Frequency	2
Basis	ACT/ACT
FUNCTION OUTPUT	
23-Sep-12	1.366%
15-Dec-12	2.500%
15-Jun-13	2.500%
15-Dec-13	2.500%
15-Jun-14	2.500%
15-Dec-14	2.500%
15-Jun-15	2.500%
15-Dec-15	2.500%
15-Jun-16	2.500%
15-Dec-16	2.500%
15-Jun-17	2.500%
15-Dec-17	2.500%
15-Jun-18	2.500%
15-Dec-18	2.500%
15-Jun-19	2.500%
15-Dec-19	2.500%
15-Jun-20	2.500%
15-Dec-20	102.500%

=bondDatesFlows(C2,C3,C4,C5,C6)

Remember that the output must be returned as an array

Calculating a Bond Price-Yield

- The price to yield calculation for a bond is given by

$$P = \sum_{t=1}^{Tf} \frac{F_t}{\left(1 + y/f\right)^t}$$

- We need to take in the bond details and the yield to get the price
- Remember that this will be the full/dirty price not the clean price
- So, we will output clean, accrued and dirty together

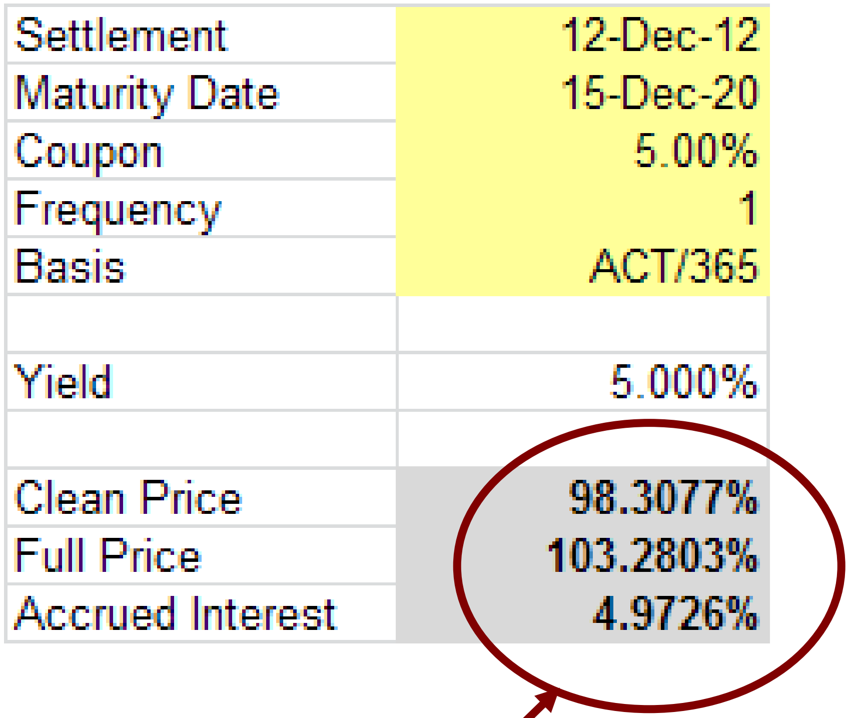
Bond Price Yield – VBA Code – Part 1

```
Function bondPriceFromYield(settlementDate As Date, _
                            maturityDate As Date, _
                            coupon As Double, _
                            frequency As Integer, _
                            basisString As String, _
                            yield As Double) As Variant

Dim bondSchedule() As Double
Dim numFlows, i, basisCode As Integer
Dim discountFactor, fullprice, firstPeriodFraction, accruedCoupon As Double
Dim nextcouponDate As Date
Dim flow As Double

basisCode = getBasisCode(basisString)

If basisCode = -1 Then
    bondPriceFromYield = "Unknown basis string"
    Exit Function
End If

bondSchedule = bondDatesFlows(settlementDate, maturityDate, coupon, frequency, basisString)
numFlows = UBound(bondSchedule)
fullprice = 0#
nextcouponDate = bondSchedule(1, 0)
accruedCoupon = bondSchedule(0, 1)
firstPeriodFraction = (1# - frequency * accruedCoupon / coupon)
```

Bond Price Yield – VBA Code – Part 2

```
discountFactor = Application.WorksheetFunction.Power(1 + yield / frequency, -firstPeriodFraction)

For i = 1 To numFlows

    flow = bondSchedule(i, 1)
    fullprice = fullprice + discountFactor * flow
    discountFactor = discountFactor * (1# / (1# + yield / frequency))

Next i

Dim output(3, 1)
output(0, 0) = fullprice - accruedCoupon
output(1, 0) = fullprice
output(2, 0) = accruedCoupon

bondPriceFromYield = output

End Function
```

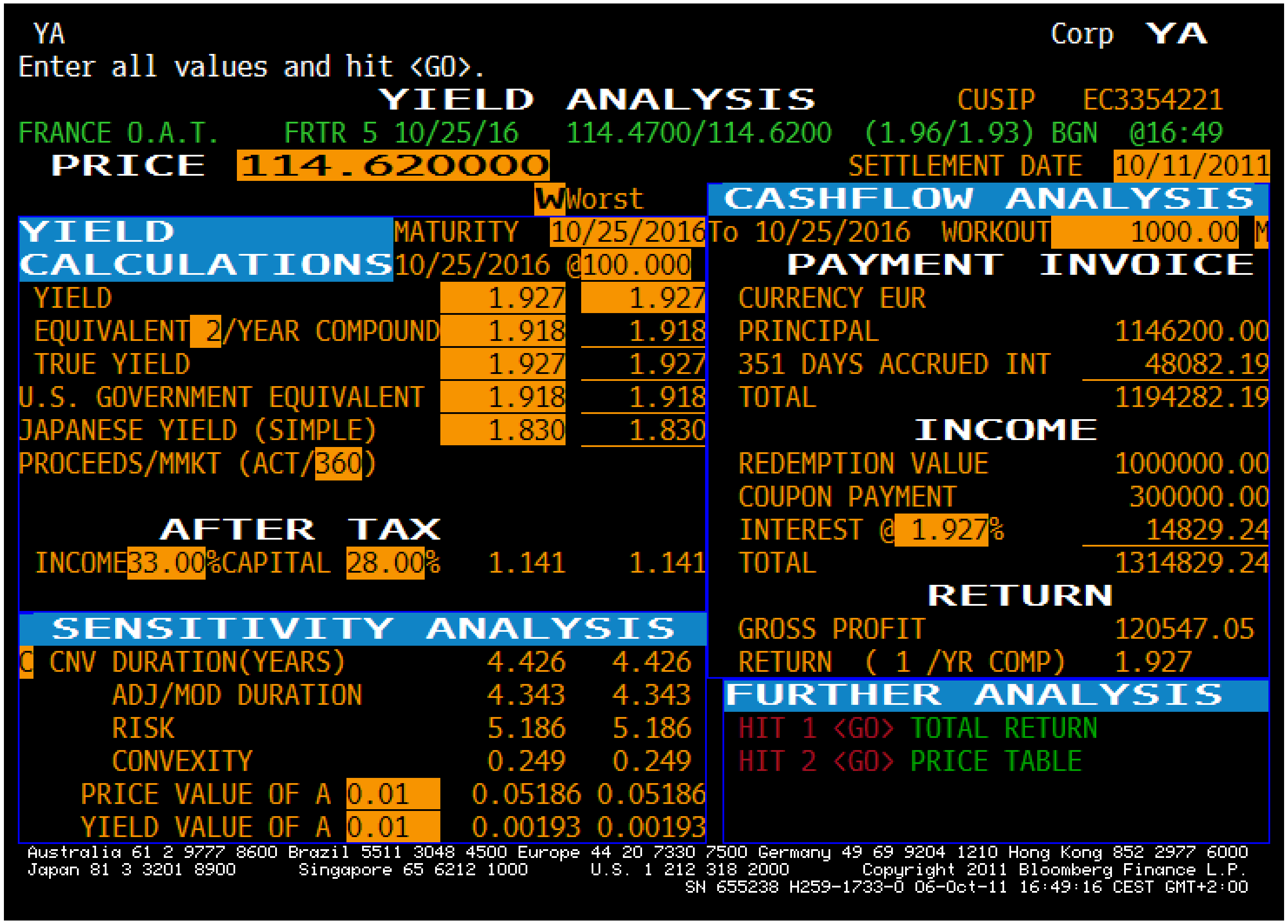
Example of Output



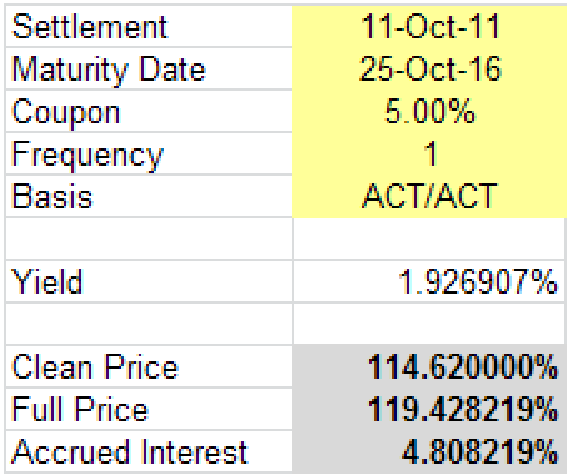
Settlement	12-Dec-12
Maturity Date	15-Dec-20
Coupon	5.00%
Frequency	1
Basis	ACT/365
Yield	5.000%
Clean Price	**98.3077%**
Full Price	**103.2803%**
Accrued Interest	**4.9726%**

=bondPriceFromYield(C2,C3,C4,C5,C6,C8)

Check against Bloomberg - French OAT Bond



```
 YA                                                              Corp  YA
Enter all values and hit <GO>.
                        YIELD  ANALYSIS          CUSIP    EC3354221
FRANCE O.A.T.    FRTR 5 10/25/16   114.4700/114.6200  (1.96/1.93) BGN  @16:49
  PRICE  114.620000                                 SETTLEMENT DATE  10/11/2011
                                  WWorst        CASHFLOW  ANALYSIS
YIELD                    MATURITY   10/25/2016To 10/25/2016  WORKOUT     1000.00 M
CALCULATIONS10/25/2016 @100.000       PAYMENT  INVOICE
 YIELD                          1.927    1.927  CURRENCY EUR
 EQUIVALENT 2/YEAR COMPOUND     1.918    1.918  PRINCIPAL              1146200.00
 TRUE YIELD                     1.927    1.927  351 DAYS ACCRUED INT     48082.19
U.S. GOVERNMENT EQUIVALENT      1.918    1.918  TOTAL                  1194282.19
JAPANESE YIELD (SIMPLE)         1.830    1.830              INCOME
PROCEEDS/MMKT (ACT/360)                         REDEMPTION VALUE       1000000.00
                                                COUPON PAYMENT          300000.00
           AFTER  TAX                           INTEREST @ 1.927%        14829.24
 INCOME33.00%CAPITAL 28.00%     1.141    1.141  TOTAL                  1314829.24
                                                             RETURN
  SENSITIVITY  ANALYSIS                         GROSS PROFIT            120547.05
C CNV DURATION(YEARS)           4.426    4.426  RETURN  ( 1 /YR COMP)  1.927
      ADJ/MOD DURATION          4.343    4.343   FURTHER  ANALYSIS
      RISK                      5.186    5.186   HIT 1 <GO> TOTAL RETURN
      CONVEXITY                 0.249    0.249   HIT 2 <GO> PRICE TABLE
    PRICE VALUE OF A 0.01     0.05186 0.05186
    YIELD VALUE OF A 0.01     0.00193 0.00193
 Australia 61 2 9777 8600 Brazil 5511 3048 4500 Europe 44 20 7330 7500 Germany 49 69 9204 1210 Hong Kong 852 2977 6000
 Japan 81 3 3201 8900        Singapore 65 6212 1000       U.S. 1 212 318 2000        Copyright 2011 Bloomberg Finance L.P.
                                                  SN 655238 H259-1733-0 06-Oct-11 16:49:16 CEST GMT+2:00
```



Settlement	11-Oct-11
Maturity Date	25-Oct-16
Coupon	5.00%
Frequency	1
Basis	ACT/ACT
Yield	1.926907%
Clean Price	114.620000%
Full Price	119.428219%
Accrued Interest	4.808219%

Optional Arguments

Optional Arguments

- Sometimes a function may have a large number of arguments
- But some may be fairly standard and default to known values
- For example you may be in a country where all bonds are semi-annual and all basis codes are "ACT/360"
- So you could combine both
  - Allow user to specify all parameters for non-standard cases
  - Use default parameters for the standard case
- Here is the syntax

```
Function bondDatesFlows(settlementDate As Date, _
                        maturityDate As Date, _
                        coupon As Double, _
                        Optional frequency As Integer = 2, _
                        Optional basisString As String = "ACT/360") As Variant
```

Optional Arguments for Functions

Specified Arguments

Settlement	11-Oct-11
Maturity Date	25-Oct-16
Coupon	5%
Frequency	1
Basis	ACT/ACT

FUNCTION OUTPUT

11-Oct-11	4.808%
25-Oct-11	5.000%
25-Oct-12	5.000%
25-Oct-13	5.000%
25-Oct-14	5.000%
25-Oct-15	5.000%
25-Oct-16	105.000%
#N/A	#N/A
#N/A	#N/A
#N/A	#N/A
#N/A	#N/A
#N/A	#N/A
#N/A	#N/A
#N/A	#N/A
#N/A	#N/A
#N/A	#N/A
#N/A	#N/A
#N/A	#N/A

Optional Arguments

Settlement	11-Oct-11
Maturity Date	25-Oct-16
Coupon	5%

FUNCTION OUTPUT

11-Oct-11	2.347%
25-Oct-11	2.500%
25-Apr-12	2.500%
25-Oct-12	2.500%
25-Apr-13	2.500%
25-Oct-13	2.500%
25-Apr-14	2.500%
25-Oct-14	2.500%
25-Apr-15	2.500%
25-Oct-15	2.500%
25-Apr-16	2.500%
25-Oct-16	102.500%
#N/A	#N/A
#N/A	#N/A
#N/A	#N/A
#N/A	#N/A
#N/A	#N/A
#N/A	#N/A

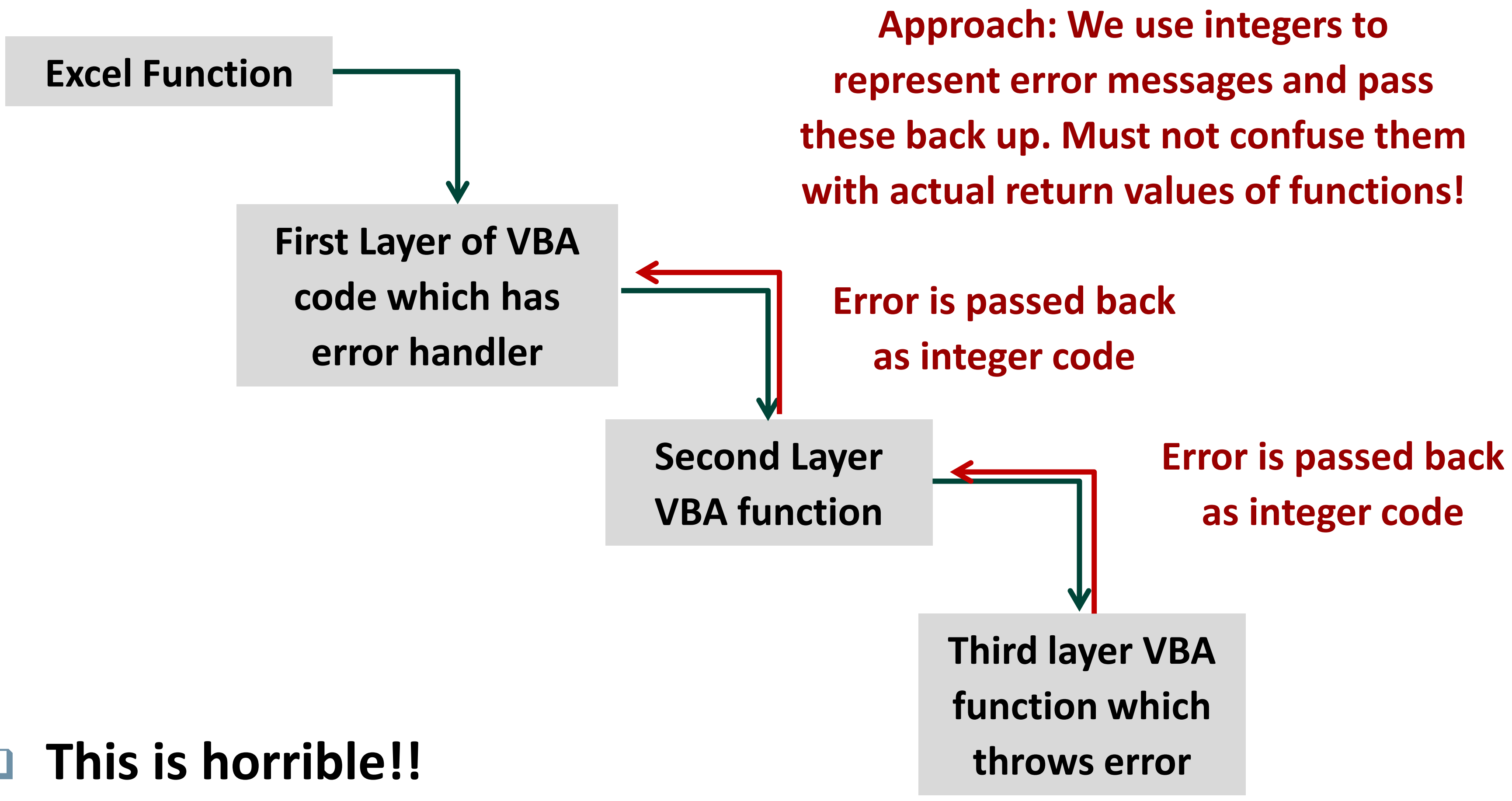
- On the right the frequency is the default "semi-annual"

Handling Errors

Error Handling

- Earlier I used a negative value to signal a basis code error
- I need a different value for each possible error
- This means we need to add checks to the code at the calling function which needs to know what all the codes mean
- Also, it is tedious to write the code to pass this error up to the previous function and to the top function
- Easy to make an error !

The Previous Approach with Multilayer Functions


Excel Function
First Layer of VBA code which has error handler
Second Layer VBA function
Third layer VBA function which throws error
Approach: We use integers to represent error messages and pass these back up. Must not confuse them with actual return values of functions!
Error is passed back as integer code
Error is passed back as integer code

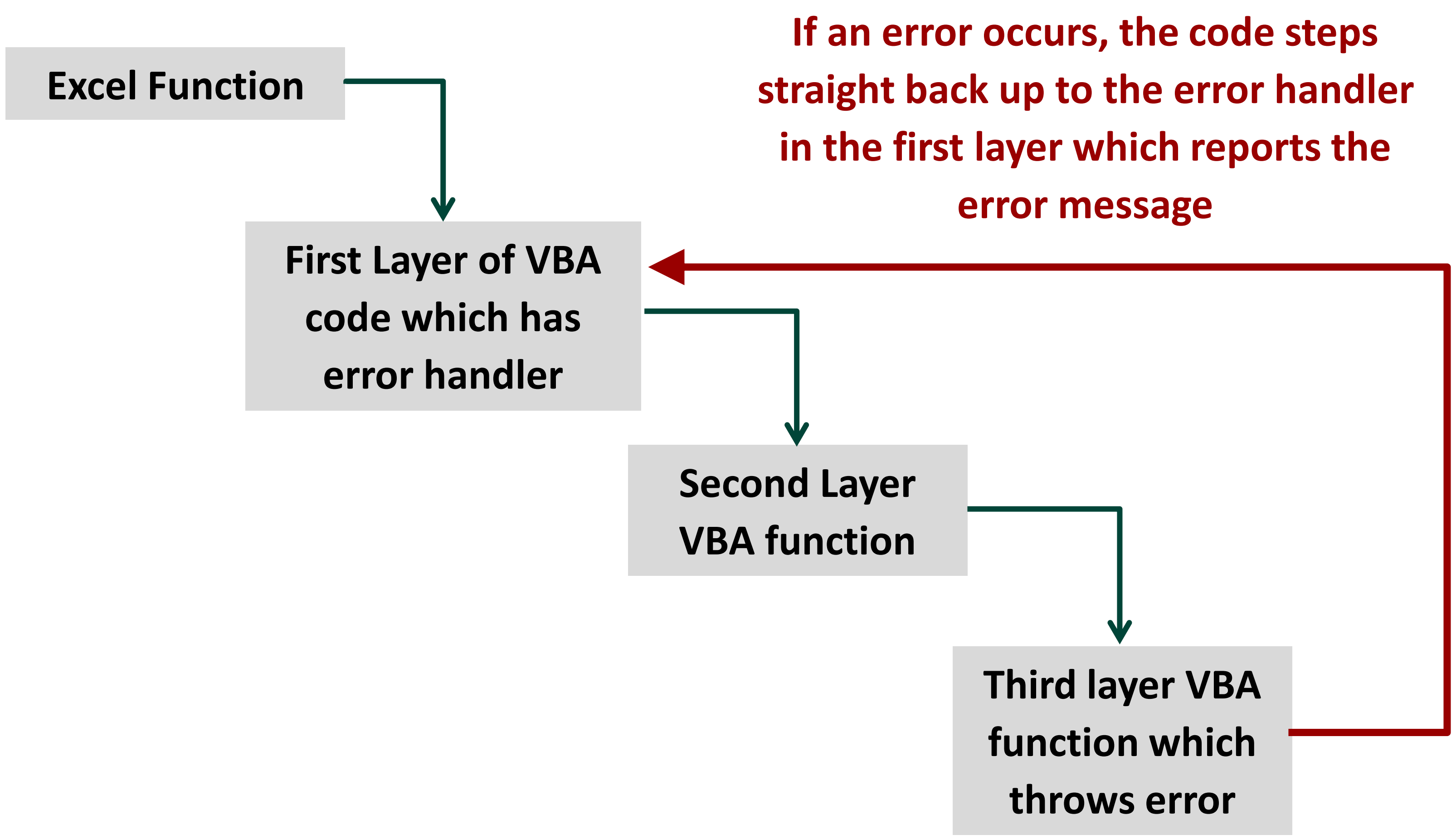
- **This is horrible!!**
- First layer needs to know what error message corresponds to what code
- Means having to carry up more information – tedious to write

The OnError Statement

- You add the statement **On Error …** before the error-prone code
- There are three ways to use the On Error statement
- **On Error Goto XXX**
  - If an error occurs after this statement, then the code will jump to the line which has a label **XXX:** at the start
- **On Error Resume Next**
  - If an error occurs it is ignored, and the code continues
- **On Error Goto 0**
  - This returns VBA back to its normal error handling behaviour

- **We will use On Error Goto XXX**

What is happening using VBA Error Handling


Excel Function
First Layer of VBA code which has error handler
Second Layer VBA function
Third layer VBA function which throws error
If an error occurs, the code steps straight back up to the error handler in the first layer which reports the error message

- This is very nice because it saves us from having to pass back text messages or error codes – everything is handled in one place

Error Handling

- How we handle errors depends on whether we can test for them
- If we can test for them we should, but this is not always possible
- Errors should also be captured and described where they occur
- The question is how do we trigger a custom error that is not an error for VBA

We can create our own Excel Errors!

- We add this line to trigger our error at the place where the error occurs

```
Err.Raise vbObjectError+513, Source, Message
```

- This takes us to the error handler – or whatever we put at the start of the error handler
- Source is the name of the function that caused the error
- Message is where we put the error message
- **This is very elegant – no error codes !**

Customised Error Handling

```
Function bondPriceFromYieldErr(settlementDate As Date, _
                               maturityDate As Date, _
                               coupon As Double, _
                               frequency As Integer, _
                               basisString As String, _
                               yield As Double) As Variant

On Error GoTo ErrorHandler

Dim bondSchedule() As Double
Dim numFlows, i, BasisCode As Integer
Dim discountFactor, fullPrice, firstPeriodFraction, accruedCoupon As Double
Dim nextcouponDate As Date
Dim flow As Double

BasisCode = getBasisCodeErr(basisString)
bondSchedule = bondDatesFlowsErr(settlementDate, maturityDate, coupon, frequency, basisString)
numFlows = UBound(bondSchedule)
fullPrice = 0#
nextcouponDate = bondSchedule(1, 0)
accruedCoupon = bondSchedule(0, 1)
firstPeriodFraction = (1# - frequency * accruedCoupon / coupon)
discountFactor = Application.WorksheetFunction.Power(1 + yield / frequency, -firstPeriodFraction)

For i = 1 To numFlows
    flow = bondSchedule(i, 1)
    fullPrice = fullPrice + discountFactor * flow
    discountFactor = discountFactor * (1# / (1# + yield / frequency))
Next i

Dim output(3, 1)
output(0, 0) = fullPrice - accruedCoupon
output(1, 0) = fullPrice
output(2, 0) = accruedCoupon

bondPriceFromYieldErr = output
Exit Function

ErrorHandler:
    bondPriceFromYieldErr = Err.Description

End Function
```

Tell VBA to watch for errors and what to do

Exit if all OK

Error handling

Local Raising of Error

- Use this special syntax to create a customised error

```
Function getBasisCodeErr(basisStr As String) As Integer

If basisStr = "Bond" Then
    getBasisCodeErr = 0
ElseIf basisStr = "ACT/ACT" Then
    getBasisCodeErr = 1
ElseIf basisStr = "ACT/360" Then
    getBasisCodeErr = 2
ElseIf basisStr = "ACT/365" Then
    getBasisCodeErr = 3
ElseIf basisStr = "30/360" Then
    getBasisCodeErr = 4
Else
    Err.Raise vbObjectError + 513, "getBasisCodeErr", "Unknown basis code"
End If

End Function
```

- Adding another validation check somewhere else is trivial

```
If frequency <> 1 And frequency <> 2 And frequency <> 4 Then
    Err.Raise vbObjectError + 513, "bondDatesFlowErr", "Unknown frequency"
End If
```

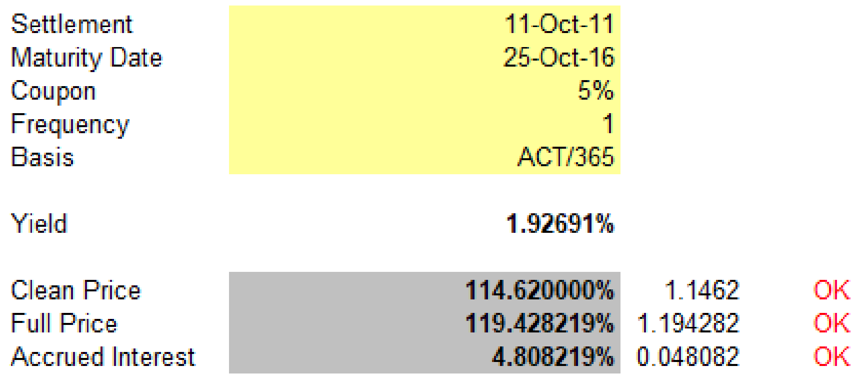
Building an Addin

Creating Custom Addins in Excel

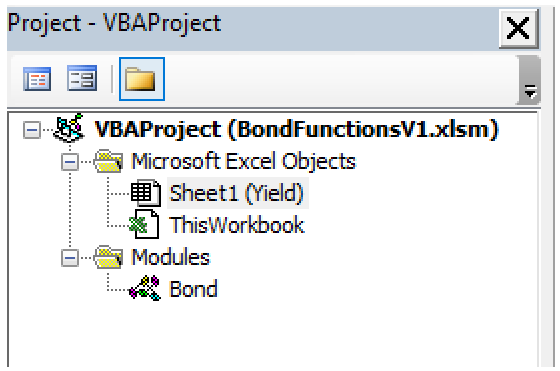
- Suppose you want to share your VBA functions
- These functions are to be used across various people and spreadsheets
- How can you do this ?
- The answer is to use an Addin.

Removed Everything Apart from One Sheet

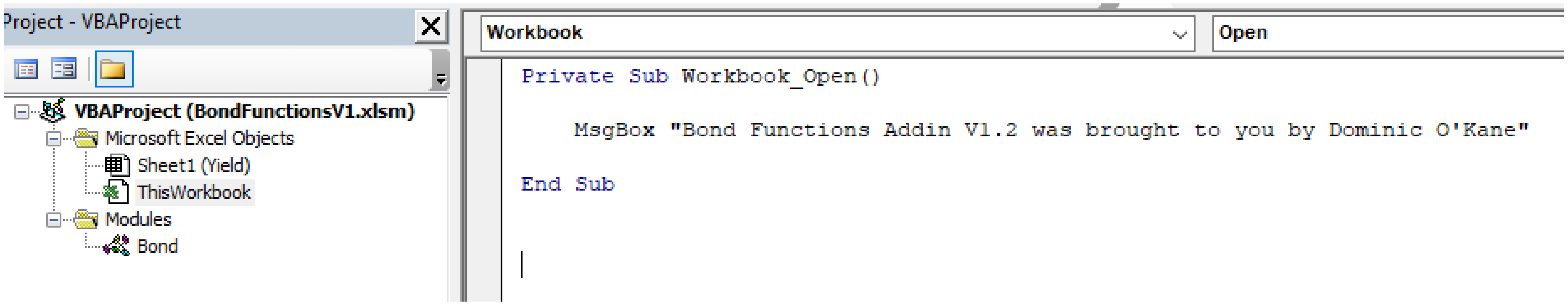
- Remove all the sheets (except one) and save as an xlsm as usual
- Use the remaining one sheet or sheets for testing functions



Settlement	11-Oct-11		
Maturity Date	25-Oct-16		
Coupon	5%		
Frequency	1		
Basis	ACT/365		
Yield	**1.92691%**		
Clean Price	**114.620000%**	1.1462	OK
Full Price	**119.428219%**	1.194282	OK
Accrued Interest	**4.808219%**	0.048082	OK



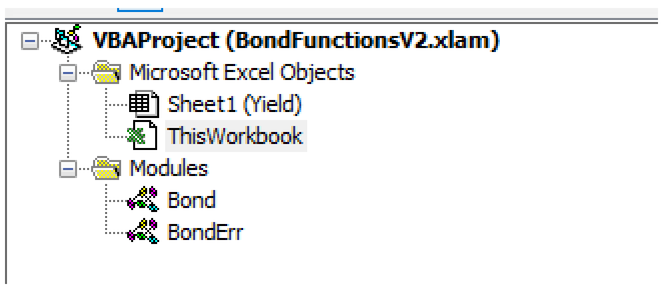
- Add an event which triggers and informs when addin is loaded



```
Private Sub Workbook_Open()

    MsgBox "Bond Functions Addin V1.2 was brought to you by Dominic O'Kane"

End Sub
```

Add an Event to Inform you which version is loaded

- Sometimes you want to be sure what addins have been loaded
- It may help for troubleshooting to know which version is loaded
- Double click on "ThisWorkbook"

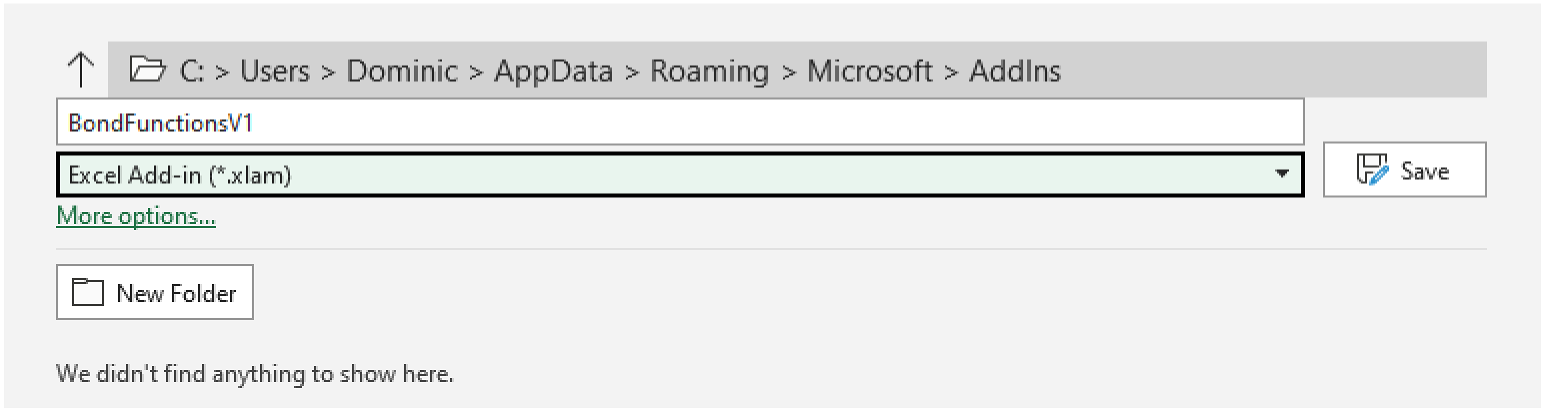


- Add an event trigger which runs when the workbook is opened.
- This will also run whenever the addin is loaded.

```
Private Sub Workbook_Open()

    MsgBox "Bond Functions Addin V2 was brought to you by Dominic O'Kane"

End Sub
```

We create an Addin version of the file

- Remove all the sheets (except one)
- Use the remaining one sheet or sheets for testing
- Then save it as an Excel Addin
- Excel will try to save it in the folder with the other addins or you can give it your own folder
- I prefer the latter as I like to have it close to my work



C: > Users > Dominic > AppData > Roaming > Microsoft > AddIns

BondFunctionsV1

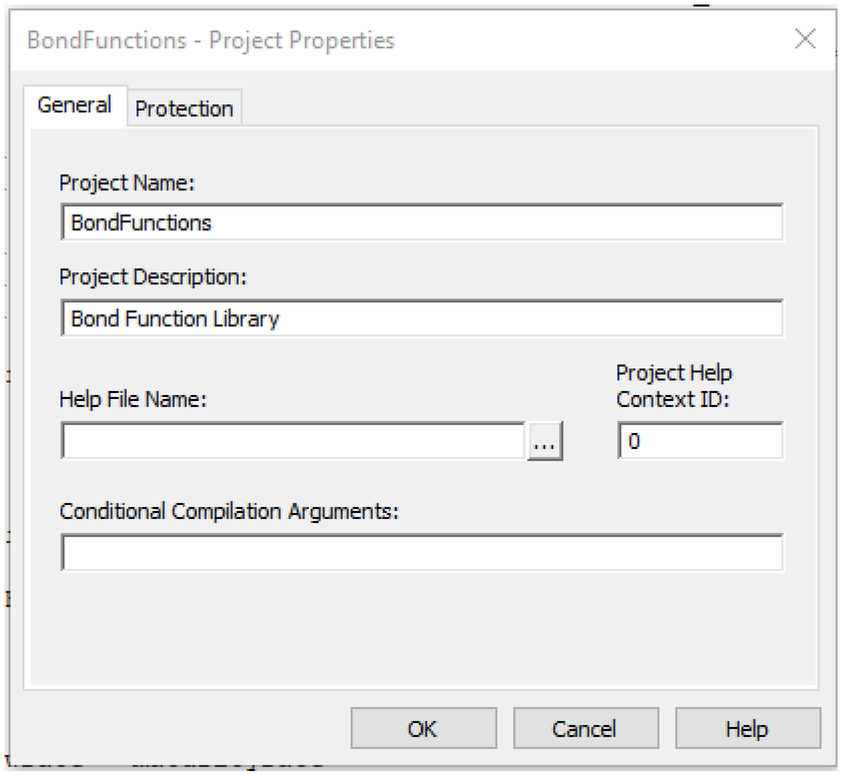
Excel Add-in (*.xlam)

We didn't find anything to show here.

- So I would not use the folder shown!

Right Click on the Top Line to Set Properties

- We can add some information and also lock the code so it cannot be seen by others



BondFunctions - Project Properties

General | Protection

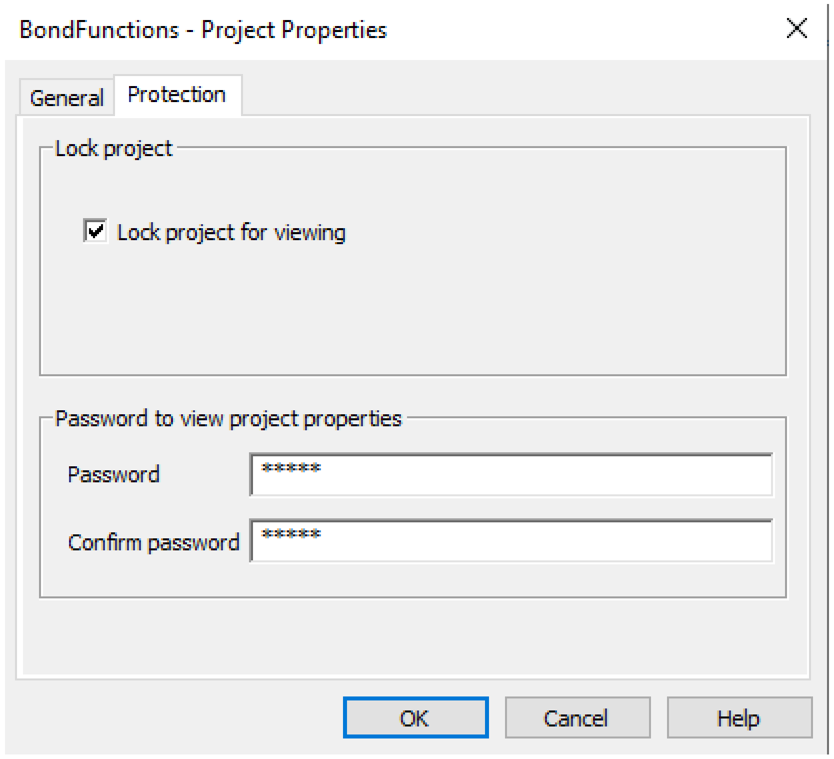
Project Name: BondFunctions

Project Description: Bond Function Library

Help File Name:

Project Help Context ID: 0

Conditional Compilation Arguments:



BondFunctions - Project Properties

General | Protection

Lock project

[✓] Lock project for viewing

Password to view project properties

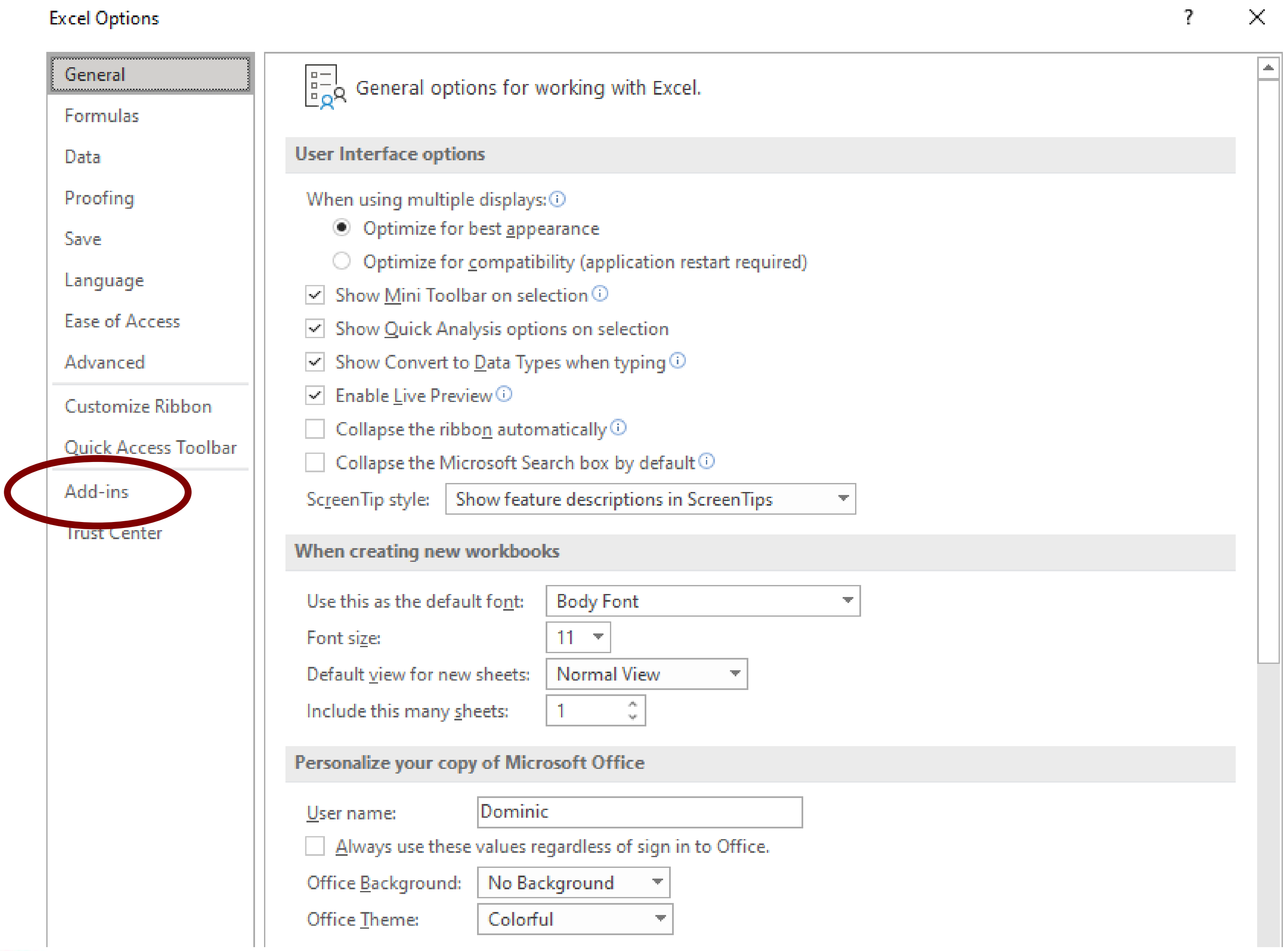
Password: *****

Confirm password: *****

- Password is “edhec”

Create a New Excel Workbook without any VBA code

- The do File->Options and select Add-ins



General options for working with Excel.

User Interface options

When using multiple displays:
- ● Optimize for best appearance
- ○ Optimize for compatibility (application restart required)
- ✓ Show Mini Toolbar on selection
- ✓ Show Quick Analysis options on selection
- ✓ Show Convert to Data Types when typing
- ✓ Enable Live Preview
- ☐ Collapse the ribbon automatically
- ☐ Collapse the Microsoft Search box by default

ScreenTip style: Show feature descriptions in ScreenTips

When creating new workbooks

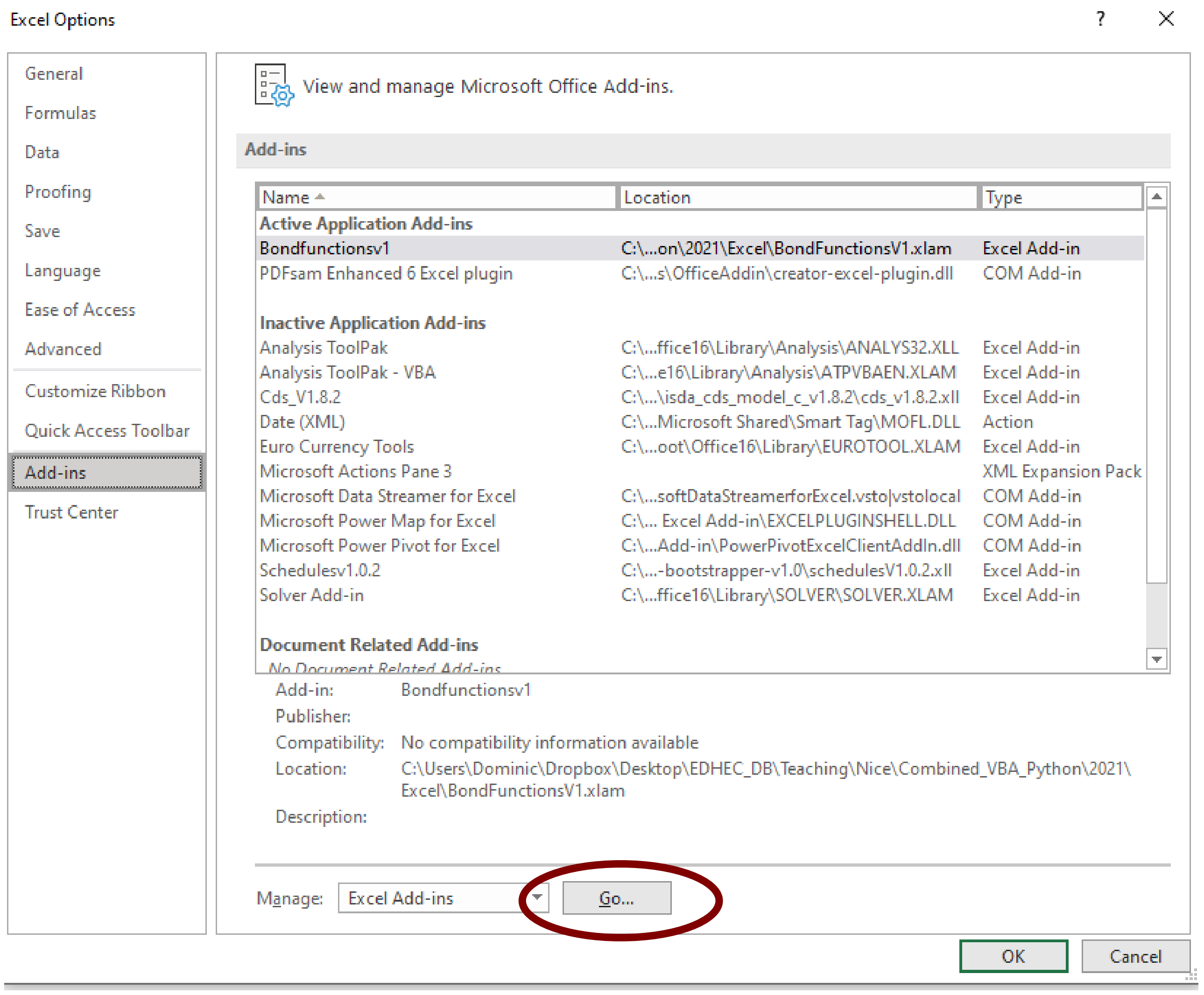
Use this as the default font: Body Font
Font size: 11
Default view for new sheets: Normal View
Include this many sheets: 1

Personalize your copy of Microsoft Office

User name: Dominic
- ☐ Always use these values regardless of sign in to Office.

Office Background: No Background
Office Theme: Colorful

Then Open the Addin List and Select the Addin



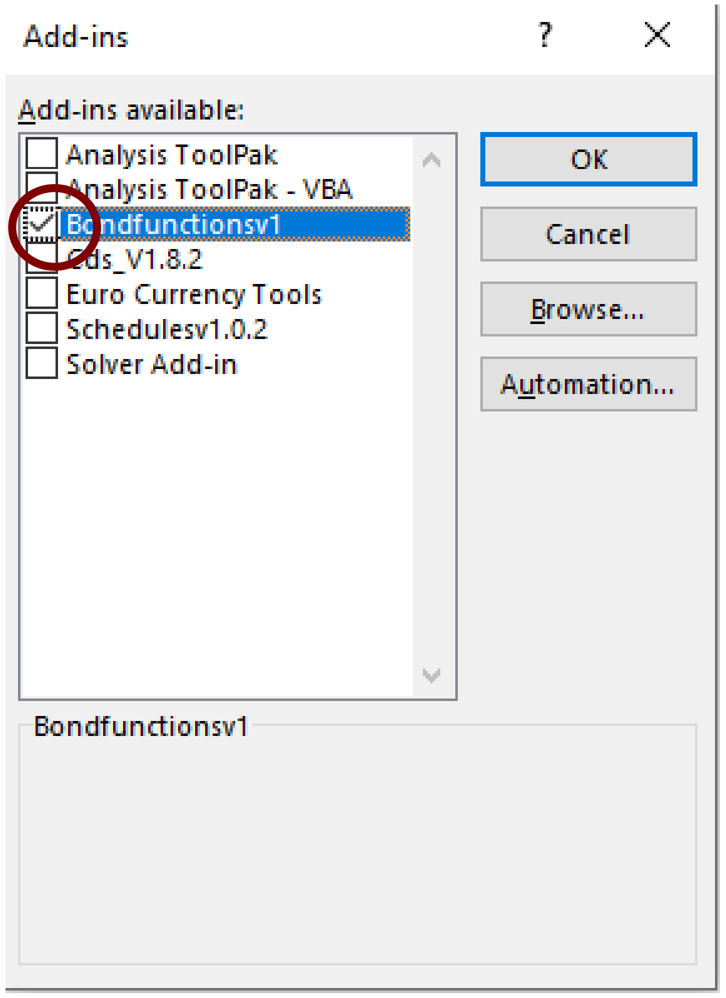
View and manage Microsoft Office Add-ins.

Add-ins

Name	Location	Type
Active Application Add-ins		
Bondfunctionsv1	C:\...on\2021\Excel\BondFunctionsV1.xlam	Excel Add-in
PDFsam Enhanced 6 Excel plugin	C:\...s\OfficeAddin\creator-excel-plugin.dll	COM Add-in
Inactive Application Add-ins		
Analysis ToolPak	C:\...ffice16\Library\Analysis\ANALYS32.XLL	Excel Add-in
Analysis ToolPak - VBA	C:\...e16\Library\Analysis\ATPVBAEN.XLAM	Excel Add-in
Cds_V1.8.2	C:\...\isda_cds_model_c_v1.8.2\cds_v1.8.2.xll	Excel Add-in
Date (XML)	C:\...Microsoft Shared\Smart Tag\MOFL.DLL	Action
Euro Currency Tools	C:\...oot\Office16\Library\EUROTOOL.XLAM	Excel Add-in
Microsoft Actions Pane 3		XML Expansion Pack
Microsoft Data Streamer for Excel	C:\...softDataStreamerforExcel.vsto|vstolocal	COM Add-in
Microsoft Power Map for Excel	C:\... Excel Add-in\EXCELPLUGINSHELL.DLL	COM Add-in
Microsoft Power Pivot for Excel	C:\...Add-in\PowerPivotExcelClientAddIn.dll	COM Add-in
Schedulesv1.0.2	C:\...-bootstrapper-v1.0\schedulesV1.0.2.xll	Excel Add-in
Solver Add-in	C:\...ffice16\Library\SOLVER\SOLVER.XLAM	Excel Add-in
Document Related Add-ins		
No Document Related Add-ins		

Add-in: Bondfunctionsv1
Publisher:
Compatibility: No compatibility information available
Location: C:\Users\Dominic\Dropbox\Desktop\EDHEC_DB\Teaching\Nice\Combined_VBA_Python\2021\Excel\BondFunctionsV1.xlam
Description:

Manage: Excel Add-ins Go...



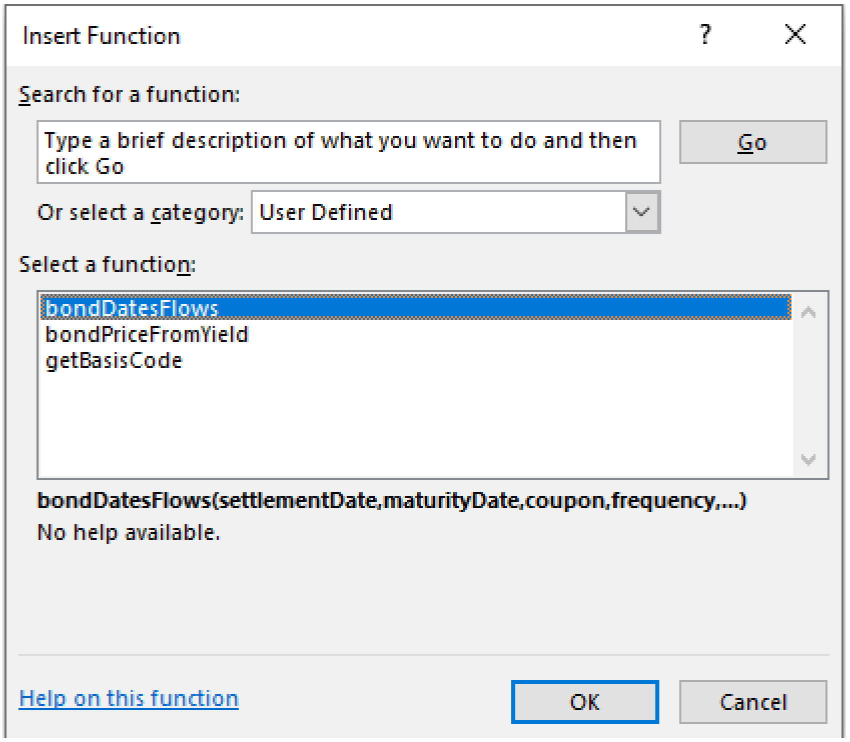
Add-ins available:

- [ ] Analysis ToolPak
- [ ] Analysis ToolPak - VBA
- [✓] Bondfunctionsv1
- [ ] Cds_V1.8.2
- [ ] Euro Currency Tools
- [ ] Schedulesv1.0.2
- [ ] Solver Add-in

Bondfunctionsv1

You can then access the functions from Excel

- The functions are in the function wizard under User Defined



Or select a category: User Defined

bondDatesFlows
bondPriceFromYield
getBasisCode

bondDatesFlows(settlementDate,maturityDate,coupon,frequency,...)
No help available.

- You can now access them without having them in your workbook
- You can share your addin with others – email it or shared drive
- Give it a version number or else you will get confused

Evolution of the Addin

- There are two ways to manage the addin. First approach is:

- Edit the .xlam file and redistribute (changing version number)
- This is simpler for quick edits but not a great protocol

- Second approach is to edit the .xlsm and save it with a new name
- This may be more time consuming but better as you can keep tests and function in the same place
- When tests work you are ready to deploy a new addin
- This is often fragile – if Excel copies the addin to a local folder it can get confusing
- But in general, this approach is powerful and scalable

Final Warning - Excel has limits

Don't build big applications in Excel VBA (even if you're in IT)

- Work has to be done removing/hiding parts of Excel
- Speed becomes important - VBA is not as fast as C# or C++
- Where is the data to be stored ? If the data is stored in Excel sheets, who has the working version ? Big sheets are slow.
- There are also big issues about who should maintain the application and what if there are multiple copies
- A good Excel user has to know Excel's limitations!
- If you want to build large GUI applications learn C# or Java
- If you want to have process tracking and be able to handle large amounts of data – use Python
- Python can also go GUI's but it's not as easy as C#