

Disclaimer

There may be typographic errors in this paper, which evolves continually. I may alter the contents at any time. If you find an error, please contact me at billw@jajafinance.com so I can correct it and save other people going through the same problems. There are no guarantees that the code in this paper will be suitable for what you want to do. After adapting an example to your code, please check that it performs in the way you want and returns sensible results.

If you have a general VBA query, either regarding content in this paper or otherwise, please consult other sources of information.

This resource is not endorsed by Jaja.

Contents

Introduction	3
1 Set Up	3
1.1 Key Points	3
1.2 Workbooks and Modules	3
1.3 MI Best Practice	4
2 Subroutines	4
2.1 Key Points	4
2.2 Subroutine Example	4
2.3 Running the Subroutine	4
2.4 If Statements	5
2.5 Calling Subroutines	6
2.6 Private and Public Subroutines	6
2.7 Assigning a Button	7
2.8 Enable Commenting Shortcut Keys	7
3 Variables	7
3.1 Key Points	7

3.2	Data Types and Objects	8
3.3	Declaring Variables	8
3.4	For Loops	9
3.5	Arguments in Subroutines	10
3.6	Scope	10
4	Referencing	11
4.1	Key Points	11
4.2	Explicit Referencing	11
4.3	With: End With	12
4.4	Range vs Cells	13
4.5	Additional Cells Usage	14
5	Copying and Pasting	15
5.1	Key Points	15
5.2	Copying and Pasting	15
5.3	Finding Table Dimensions	17
6	Manipulating Workbooks and Worksheets	17
6.1	Key Points	17
6.2	Opening Workbooks	18
6.3	Closing Workbooks	18
6.4	Making a New Workbook	19
6.5	New Worksheets and Position	20
6.6	Copy and Paste Example	21
7	Objects, Properties, and Methods	22
7.1	Key Points	22
7.2	Objects	22
7.3	Properties and Methods	23
8	Error Handling	24
8.1	Error Handling Options	24
8.2	Error Handling Example	24
9	Assign Keys	25
9.1	Assign Keys Example	25

10 Functions	26
10.1 Function Examples	26
11 Outlook	27
11.1 Setting Up The Reference	27
11.1.1 Programmatically Add Outlook Reference	27
11.2 Writing Emails	28
11.2.1 Additional Control: Attaching Documents, Voting	29
11.2.2 Writing HTML	29
Appendix	31

Introduction

This document is an introduction to some useful *Visual Basic for Applications* (VBA) code that can be used to automate many processes, both business as usual and ad-hoc work. Once you start writing VBA yourself, a quick [Google](#) search is likely to provide you with a rich source of solutions to any problem that you have. Some experience using SAS will be useful, but not essential.

1 Set Up

1.1 Key Points

1. Use `.xlsm` Workbooks.
2. Click `Alt + F11` for the code window.
3. Write VBA code inside Modules.

1.2 Workbooks and Modules

To save Workbooks with VBA code, you should use the `.xlsm` file extension which enables macros in the Workbook. You may also choose to save your Workbooks with the `.xls` or `.xlsb` extensions, but note that you should not save the Workbook with the usual `.xlsx` extension.

To write VBA code, you need to open the code editor window. There are two ways to do this:

1. Open the **Developer** tab of your Workbook, and then click **Visual Basic**;
2. Press `Alt + F11`.

If the Developer tab is not visible by default, then right-click on the ribbon (the row of tabs) and choose **Customize the Ribbon...** which will open up a smaller window. On the right-hand-side of this window, tick the Developer box, and then click OK.

Code should usually be written inside *Modules*. To create a new Module, open the code window, click **Insert** in the top ribbon, and then click **Module**. To change the name of the Module, you need the **Properties Window** which can be opened by clicking **View** and then **Properties Window**, or by clicking F4. The top box of this window contains the Module name, which can be changed there.

1.3 MI Best Practice

For automating the main MI, it is best to write the code in a new Workbook to avoid making significant changes to the Workbooks that will be manipulated. It is generally considered bad practice to rely on the `Activate` and `Select` methods as they slow down code and are mostly unnecessary [16], so working from an external Workbook will help enforce correct referencing (more on this in Section 4).

2 Subroutines

2.1 Key Points

1. Write code within subroutines.
2. VBA subroutines perform similarly to SAS macros.
3. Run subroutines with F5.
4. The comment character is the apostrophe (').
5. Line breaks are important.

2.2 Subroutine Example

The majority of VBA code will be contained within *subroutines*. A subroutine in VBA is just like a macro in SAS – you are assigning a series of instructions to a name, and when you call the name the instructions are executed. Just like SAS macros, subroutines can also have arguments passed to them. The code below is an example of a simple subroutine that, when run, generates a message box with the text *Hello World*.

```
Sub PrintHelloWorld()  
    MsgBox "Hello World"  
End Sub
```

Just like SAS macros have to start with `%macro` and end with `%mend`, VBA subroutines must start with `Sub` and end with `End Sub`. The subroutine name must also have the parentheses at the end (even if they are empty).

2.3 Running the Subroutine

There are three main ways to run a subroutine, two of which are from the code window itself.

1. Inside the code window, pressing F5 will do the following:

- (a) if the text cursor is on the same line as a subroutine, then that subroutine will be run;
 - (b) else the list of available subroutines will be generated, and you can choose which to run.
2. Inside the code window, pressing the Run Sub button (the play-looking button in the ribbon) will do the same as pressing F5.
 3. In the Workbook, you can assign subroutines to buttons which run the subroutine when pressed (more on this in Section 2.7).

This means that you cannot run a selection of code inside a subroutine, for example. If you only want to run a portion of the code in your subroutine, one method is to comment the rest of it out – the comment character in VBA is the apostrophe ('), which functions just like the asterisk (*) in SAS by commenting out the remainder of the line.

2.4 If Statements

The syntax for an if-statement in VBA is very similar to the SAS syntax. The following example uses two useful built-in functions: `Time`, which generates the current time, and `TimeValue`, which converts a string (which should be in the hh:mm:ss format) into its corresponding VBA time numeric.

```
Sub TimeSub()  
  
    If Time < TimeValue("12:00:00") Then  
        MsgBox "It is the morning."  
    ElseIf Time < TimeValue("17:00:00") Then  
        MsgBox "It is the afternoon."  
    Else  
        MsgBox "It is the evening."  
    End If  
  
End Sub
```

It is important to note that an if-statement must finish with `End If`, and the `ElseIf` statements cannot have a space between the `Else` and the `If`. Similarly, the line breaks are important – for example, the `ElseIf` statement cannot be on the same line as the `If` statement, nor can the `Else` statement. The code following the `Then` must also be on its own line. To illustrate this, consider the following code.

```
Sub DoesNotWork()  
  
    If 1 = 2 Then MsgBox "Case 1"  
    ElseIf 1 = 1 Then MsgBox "Case 2"  
    Else MsgBox "Case 3"  
    End If  
  
End Sub  
  
Sub DoesWork()  
  
    If 1 = 2 Then  
        MsgBox "Case 1"  
    ElseIf 1 = 1 Then  
        MsgBox "Case 2"  
    Else  
        MsgBox "Case 3"  
    End If  
  
End Sub
```

The first subroutine, `DoesNotWork`, will not work because the required line breaks are missing. The line breaks have been included in the second subroutine, `DoesWork`, so the code does work.

It is possible to have a single-line if-statement. This type of if-statement cannot have any `ElseIf` statements, and it does not need to end with `End If`. The example below demonstrates this.

```
Sub OneLineIfStatement()  
    If 1 = 2 Then MsgBox "TRUE" Else MsgBox "FALSE"  
End Sub
```

2.5 Calling Subroutines

It is usually convenient to call subroutines as part of another subroutine. This is achieved simply by writing the subroutine that you wish to call, and my preference is to indicate that the subroutine is being called by writing `Call` before the subroutine. The example below is just to demonstrate how calling a subroutine works.

```
Sub FirstSub()  
    MsgBox "This is the first subroutine"  
End Sub  
  
Sub SecondSub()  
    MsgBox "This is the second subroutine"  
End Sub  
  
Sub MasterSub()  
    Call FirstSub()  
    Call SecondSub()  
End Sub
```

When `MasterSub` is run, it first generates a message box that says *This is the first subroutine*, and then generates a second message box that says *This is the second subroutine*.

2.6 Private and Public Subroutines

You may see the words *Private* or *Public* before `Sub`. These just control where the subroutine can be accessed from:

1. *Public* subroutines can be accessed/called from anywhere, such as other modules. Only public subroutines can be assigned to buttons in the Workbook.
2. *Private* subroutines can only be accessed/called from within the module that they are written.

A subroutine written without `Public` or `Private` at the start will be public by default.

2.7 Assigning a Button

We mentioned in Section 2.3 that it is possible to run a subroutine by assigning it to a button. Once a subroutine has been written, this is usually the easiest way to run it alongside the usual updating of a spreadsheet.

To create a button, first click on the **Developer** tab of your Workbook, then click **Insert**, and then click the first (top-left) icon in the **Form Controls** section. The cursor will change to a plus symbol, and you can then draw the button onto the spreadsheet by clicking and dragging the cursor to fill the space that you want the button to occupy. When you release the mouse button, a window will appear with a list of the current *public* subroutines, and you pick the subroutine that you want to assign from here.

Once the subroutine is assigned, simply left-clicking the button will run the subroutine. To edit the button in any way, use right mouse clicks. To get the button to line up with the cells, hold **Alt** while creating, moving, or re-sizing the button.

You can actually assign macros to a number of different objects in your Workbook, such as shapes (created within the **Insert** tab). This can help to make your ‘buttons’ look a little more interesting.

2.8 Enable Commenting Shortcut Keys

By default, there is no keyboard shortcut to comment out or uncomment sections of code. To enable this, we follow the instructions described in the first few answers of [14]:

1. Right-click on the toolbar, select **Customize...**, and then select the **Commands** tab.
2. Under **Categories**, click on **Edit** and then select **Comment Block** in the **Commands** listbox (roughly two thirds down).
3. Drag the **Comment Block** entry onto the menu bar (where the Run Sub button is). Note that you should now see a new icon on the menu bar.
4. With the Customize window still open, make sure that the new icon is highlighted (it will have a black square around it) and then click the **Modify Selection** button in the **Commands** tab.
5. A dropdown box will appear. On the **Name** line, add an ampersand (&) to the beginning of the entry so that instead of *Comment Block* it should read *&Comment Block*. Press Enter to save the change.
6. Click on **Modify Selection** again and select **Image and Text**. Dismiss the Customize window.
7. Repeat the previous steps for *Uncomment Block* instead of *Comment Block*.

You can now comment out code with **Alt + C** and uncomment with **Alt + U**.

3 Variables

3.1 Key Points

1. It is good practice to always declare your variables (**Option Explicit** forces this).
2. Declaring an object requires the **Set** keyword.
3. Arguments in subroutines must have their data type defined as part of the argument.

3.2 Data Types and Objects

Since VBA is (more-or-less) an *Object Oriented* programming language, it has variables that are class-based – if you have not heard of this before, do not worry, as we will explain only the bits that you need to know. This means that it is good practice (and generally required) to write which classes your variables will come from at the start of your subroutines, so that Excel knows what kind of data your variable will be and how to use it. The [Data Type Summary](#) Microsoft Doc [4] contains information on the default Visual Basic data types; it is likely that you will have come across many of these before.

In addition to the data types in [4], there are also *objects* (and *collections*) that you would need to declare at the start of the code. Some common examples of data types and objects are given in Table 1.

Data Types	Objects
String	Workbook
Integer	Worksheet
Date	Range

Table 1: Common Data Types and Objects

At this point, a distinction is not really necessary – you just need to remember that data types and objects need to be assigned values slightly differently (see the next section). Some of their other differences will become clear once you have worked with both a little more.

3.3 Declaring Variables

Declaring a variable has two steps: the first is to define what data type/object the variable will be, and the second is to assign the value/expression to the variable. The first step is achieved by using the [Dim](#) keyword followed by the variable name, then an [as](#) keyword, and then the data type/object.

If the variable is a simple data type, you can assign its value by writing the variable (on a new line), then the equals key (=), and then the value you wish to assign to it. If the variable is an object, you also need to include the [Set](#) keyword before the variable name.

Suppose that we have a Workbook open called `Example Book.xlsm` with one Worksheet called `Example Sheet`. The example below shows how to declare a few objects (a Workbook, a Worksheet, a Range) and a few data types (a date, a string, an integer).

```
Sub DeclareVariables()  
  
    Dim mainBook as Workbook  
    Dim mainSheet as Worksheet  
    Dim mainRange as Range  
  
    Dim today as Date  
    Dim myName as String  
    Dim myAge as Integer  
  
    Set mainBook = Workbooks("Example Book.xlsm")  
    Set mainSheet = Worksheets("Example Sheet")  
    Set mainRange = Range("A1:B2")  
  
    today = #03/14/2019# ' VBA uses American date formats -- this is 14th of March, 2019  
    myName = "Bill"  
    myAge = 23  
  
End Sub
```

You will notice the extra syntax for the objects, the date, and the string. For the simple data types:

1. manual dates must be in the mm/dd/yy format and enclosed in hashes (#);
2. strings must be enclosed in quotation marks (").

The objects themselves are defined using the functions `Workbooks`, `Worksheets`, and `Range` with the corresponding object's name as a string given to the functions as an argument. It is important for you to remember to use these functions whenever you want to use any of the above objects (we will see an alternative to the `Range` function in Section 4.4).

It is also worth noting that you may use the `Sheets` function instead of `Worksheets`: the former allows manipulation of *Chart Sheets*, *Dialog Sheets*, and *Macro Sheets*, in addition to the usual *Worksheets* that we are familiar with. See [10] for more details on these sheet types.

To enforce declaration of variables, including `Option Explicit` at the top of your module will trigger an error any time you try to use a variable that has not been declared.

3.4 For Loops

In addition to if-statements, it is useful to know how to write a for-loop. The syntax for this is very simple, and is similar to the syntax for many other languages. We start by declaring the loop variable, which is `i` in the example below, and then write `For i = m To n` with `m` and `n` replaced by your lower and upper bounds for the iterations, respectively. These numbers must be whole numbers, and the loop will increment by 1 each time. On a new line, write the code to be looped over, and then finish with `Next i`. In the example below, we print the iteration number in a message box for each integer starting at 1 and ending at 5:

```
Sub ForLoop()  
  
    Dim i As Integer  
    For i = 1 To 5  
        Debug.Print "This is iteration number " & i  
    Next i  
  
End Sub
```

The example above also shows how to combine strings and variables in the message box by using the ampersand (&). The spacing either side of the ampersand is important – without it, your code may not perform as intended.

The for-loops do not always need to enumerate through integers. It is possible to also enumerate all of (or a subset of) the items in a collection, say all of the *Worksheets* in a *Workbook*. The next example enumerates through each of the cells in a *Range* (remember, a single cell is still a *Range* object).

```
Sub ObjectForLoop()  
  
    Dim iCell As Range  
    For iCell in Range("A1:D4")  
        Debug.Print iCell.Address  
    Next iCell  
  
End Sub
```

The `Debug.Print` function prints output to the *Immediate Window* instead of to a message box. You can open the Immediate Window with the keyboard shortcut **Ctrl + G**, or by using the **View** tab.

3.5 Arguments in Subroutines

Just like with SAS macros, it is possible to pass arguments to subroutines that have this functionality set up. The names of the arguments must be declared within the parentheses after the subroutine name, rather than in the body of the subroutine. Consider the example subroutine below that adds 1 to the argument passed to it.

```
Sub PlusOne(variable as Long)

    Dim varPlusOne as Long

    varPlusOne = variable + 1
    MsgBox "One plus " & variable & " is " & varPlusOne

End Sub
```

In this subroutine, the argument, namely `variable`, is named and assigned the `Long` data type within the parentheses after the subroutine name. The variable `varPlusOne` is a variable that is defined within the subroutine, so we declare its type inside the subroutine too.

3.6 Scope

Just like with SAS macros, the variables defined within a subroutine are contained within that subroutine. The example below defines the variable `newVar` within the first subroutine, and then attempts to display the variable in a message box in the second and third subroutines.

```
Sub FirstSub()

    Dim newVar As String
    newVar = "In FirstSub"

End Sub

Sub SecondSub()

    MsgBox newVar

End Sub

Sub ThirdSub()

    Call FirstSub()
    MsgBox newVar

End Sub
```

Since `newVar` is not defined within `SecondSub`, an error is generated when we try to use the variable in the `MsgBox` function. The third subroutine is a more interesting case – even though the subroutine `ThirdSub` calls the first subroutine `FirstSub` in which `newVar` is defined, the `MsgBox` function *still* generates an error when we try to use `newVar`.

It is possible to define a variable so that it can be accessed in different subroutines. This is one of the cases where we write code *outside* the subroutines. To define the variable, we write `Public` instead of `Dim` to declare the variable type while outside of a subroutine. Note that the value must still be assigned inside a subroutine, and that subroutine must be run to assign the value. In the example below, we declare the variable `pubVar` outside of the subroutines, give the variable the value in the

AssignVariable subroutine, and then print the variable in the PrintVariable subroutine.

```
Public pubVar As String

Sub AssignVariable()

    pubVar = "In AssignVariable"

End Sub

Sub PrintVariable()

    MsgBox pubVar

End Sub
```

If the public variable will be a constant simple data type, we can assign its value at the same time that we declare it. The example below shows how the example above can be altered to do this – note how the **Public** keyword has changed to **Const**.

```
Const pubVar As String = "Constant Variable"

Sub PrintVariable()

    MsgBox pubVar

End Sub
```

4 Referencing

4.1 Key Points

1. Avoid using **Activate** and **Select**.
2. Use **With: End With** to avoid unnecessary repetition.
3. Combine the **Range** function with the **Cells** function for dynamic selection of Ranges.

4.2 Explicit Referencing

It is common to see many **Activate** and **Select** statements in VBA code, despite the delays they add to the code. Using explicit references is a method around this.

When referring to a **Worksheet**, a **Range**, or a **Cell**, there is the potential for ambiguity if you do not reference properly. Consider a **Workbook** with two sheets in it, say **Sheet1** and **Sheet2**. Suppose that we write the following subroutine.

```
Sub ChangeCell()

    Range("A1").Value = 1

End Sub
```

The subroutine changes the value in cell A1 to 1. However, *which* A1 cell is changed – the one in Sheet1 or the one in Sheet2? By default, Excel will use the last Active sheet (you can make a sheet the active sheet by clicking on it, for example). Thus, it is better to explicitly state the Worksheet in the reference of the cell. The subroutine below uses explicit referencing to change the value of cell A1 in Sheet1 to 10, and the value of A1 in Sheet2 to 20.

```
Sub ChangeCells()  
  
    Worksheets("Sheet1").Range("A1").Value = 10  
    Worksheets("Sheet2").Range("A1").Value = 20  
  
End Sub
```

In fact, as the same Worksheet name might appear in multiple different Workbooks, it is also better to explicitly state the Workbook in the reference of the cell. Assuming that the Workbook is called *Book1.xlsm*, the code can be improved by writing the new subroutine below.

```
Sub ChangeCells()  
  
    Workbooks("Book1.xlsm").Worksheets("Sheet1").Range("A1").Value = 10  
    Workbooks("Book1.xlsm").Worksheets("Sheet2").Range("A1").Value = 20  
  
End Sub
```

Since Excel will not allow you to open multiple Workbooks with the same name, this is as precise as we need to be when using references.

4.3 With: End With

Explicit references can take up a great deal of space on the line, and have the potential to get messy. The **With: End With** statement allows us to write part of the explicit reference once, with everything else contained within the **With: End With** statement understood to start with this reference. Consider the example below.

```
Sub NotUsingWith()  
  
    Workbooks("Book1.xlsm").Worksheets("Sheet1").Range("A1").Value = 10  
    Workbooks("Book1.xlsm").Worksheets("Sheet1").Range("B2").Value = 20  
    Workbooks("Book1.xlsm").Worksheets("Sheet1").Range("C3").Value = 30  
  
End Sub  
  
Sub UsingWith()  
  
    With Workbooks("Book1.xlsm").Worksheets("Sheet1")  
        .Range("A1").Value = 10  
        .Range("B2").Value = 20  
        .Range("C3").Value = 30  
    End With  
  
End Sub
```

Both of the subroutines above produce identical results, but the first subroutine is unnecessarily repeating the first part of the reference. By using **With: End With**, we only need to write that part of the reference once. It is important to use the full-stop before the Range function – without it, the reference after the **With** will not be used.

It is also worth noting that VBA works nicely with nested `With: End With` statements. For example, consider the example below.

```
Sub UsingWithEndWith()  
  
    With Workbooks("Book1.xlsm").Worksheets("Sheet1")  
  
        With .Range("A1")  
            .Value = "Proportion"  
            .HorizontalAlignment = xlCenter  
            .NumberFormat = "@"  
        End With  
  
        With .Range("A2")  
            .Value = 0.01  
            .HorizontalAlignment = xlLeft  
            .NumberFormat = "0.00%"  
        End With  
  
    End With  
  
End Sub
```

This example changes the value of Cell A1 to the string "Proportion", the horizontal alignment to centered, and the format to text. It also changes the value of Cell A2 to 0.01, the horizontal alignment to left justified, and the format to the percentage with two decimal places.

4.4 Range vs Cells

Now that we have covered the `With: End With` statement, the function `Cells` can be introduced which can be used with the `Range` function to dynamically select Ranges.

So far, we have been using the `Range` function in the same way each time – to reference a single cell or a range of cells by using the corresponding Excel name for the `Range` (for example, A1 or A1:C3). We can also define a `Range` by supplying two opposing corners to the `Range` function. For example, instead of writing `.Range("A1:C3")`, we could write `.Range(.Range("A1"), .Range("C3"))`. Note that we have again kept the leading full-stop as a reminder that full references are preferred.

In fact, the following five expressions are equivalent:

1. `.Range("A1:C3")`
2. `.Range(.Range("A1"), .Range("C3"))`
3. `.Range(.Range("C3"), .Range("A1"))`
4. `.Range(.Range("C1"), .Range("A3"))`
5. `.Range(.Range("A3"), .Range("C1"))`

These have been included to emphasise that it does not matter in which order you supply the opposing corners – either opposing corners given in either order works the same.

The `Cells` function becomes more useful for referencing single cells as part of a subroutine, as it only needs numeric arguments. The most common way of using `Cells` is to write `.Cells(m, n)`, where `m` is the row number of the cell and `n` is the column number of the cell. For example, the code `.Cells(1, 2)` refers to cell B1, and `.Cells(2, 1)` refers to cell A2. When we combine this with the code for table

dimensions (Section 5.3), we can dynamically reference cells without having to convert cell references into text for the Range function.

One of the other common ways of using the Cells function is to completely omit the arguments, which selects *all* of the cells in the referenced worksheet. For example, the code below copies every cell in the Sheet1 Worksheet, and pastes them into the Sheet2 Worksheet:

```
Sub SelectWholeWorksheet()  
  
    With Workbooks("Example Book.xlsm")  
        .Worksheets("Sheet1").Cells.Copy  
        .Worksheets("Sheet2").Cells.PasteSpecial xlPasteAll  
    End With  
  
End Sub
```

We will see how to use this copy and paste method more in the following section.

It is also worth mentioning another way of using the Cells function that is less common. Before we introduce it, we need to note that in the current latest version of Excel, there are 16,384 (2^{14}) columns and 1,048,576 (2^{20}) rows, so that there is a total of 17,179,869,184 (2^{34}) cells. These cells are numbered sequentially from left to right, and top to bottom. That is, cell A1 is cell 1, cell B1 is cell 2, cell A2 is cell 16,385, and so on. With this in mind, we can use a single numeric argument in the Cells function to reference a cell. For example, the following three are equivalent:

1. .Cells(16386)
2. .Cells(2, 2)
3. .Range("B2")

The final thing to note is that we can combine the Range and the Cells function to select a range of cells like we did at the start of this section using multiple Range functions. This is typically the most versatile way of selecting a range of cells, as we can specify the opposing corner cells using the row and column numbers which are usually easier to work with than the cell addresses as text. This will become more apparent after reading through Sections 5.3 and 6.6. For example, the following are all equivalent:

1. .Range("A1:C3")
2. .Range(.Range("A1"), .Range("C3"))
3. .Range(.Cells(1, 1), .Cells(3, 3))
4. .Range(.Cells(1, 1), .Range("C3"))
5. .Range(.Range("A1"), .Cells(3, 3))

4.5 Additional Cells Usage

In fact, the Cells property belongs to a number of different objects, not just to Worksheets. For example, it also belongs as a property to the Range object, which means that we can write something like the following:

```

Sub RangeDotCells()

    With ThisWorkbook.Worksheets("Sheet1")
        .Range("C3:E5").Cells(2, 2).Value = 1
    End With

End Sub

```

Can you guess which cell will have its value changed? Try running this code to find out, noting that `ThisWorkbook` will always refer to the Workbook that the sub exists in.

5 Copying and Pasting

5.1 Key Points

1. Copy with the `.Copy` method, and paste using the `.PasteSpecial` method (with an option).
2. Clear the clipboard with `Application.CutCopyMode = False`.

5.2 Copying and Pasting

Knowing how to copy and paste in VBA is a very useful part of VBA. This includes, but is not limited to, copying and pasting the following:

1. SAS outputs into corresponding input Workbooks (note that we have to download the SAS outputs to an Excel Workbook first);
2. Monthly columns with formulas in summary tables, which can include diagonals of cells like in the delinquency tables;
3. Cell formats, which includes cell colours, font colours, border colours, number formats, and so on.

For this section, we will restrict our attention to copy and pasting Ranges (a selection of cells, which may be just a single cell). There are two main ways to do this, one of which is analogous to pressing `Ctrl + C` and then `Ctrl + V`, and the other which has a lot more pasting options.

The first way is to simply reference the Range that you want to copy, write the `.Copy` method, and then write the reference of the destination of the paste. If you are copying a Range that consists of more than a single cell, it is enough to write the top-left cell of the paste destination rather than write the entire Range that is being pasted to. The example below demonstrates how to copy and paste a single cell and a range of cells using this method:

```

Sub CopyAndPaste()

    With Workbooks("Example Book.xlsm").Worksheets("Sheet1")

        ' Copy cell A1 into cell A2
        .Range("A1").Copy .Range("A2")

        ' Copy the Range A1:B2 into the Range C1:D2
        .Range("A1:B2").Copy .Range("C1:D2")
    End With
End Sub

```

```

' The same as the above, but specifying the top-left cell in the paste destination
.Range("A1:B2").Copy .Range("C1")

End With

End Sub

```

The second method is very similar, but offers more flexibility. The copy part is the same, but instead of writing the paste destination on the same line, we write the paste destination on a new line (being on a new line is important) and use the `.PasteSpecial` method, followed by the pasting option that we want. For example, if we just want to paste the values, then we would follow `.PasteSpecial` with `xlPasteValues`.

The example below shows how to perform the operations in the above example using this second method:

```

Sub CopyAndPaste()

    With Workbooks("Example Book.xlsm").Worksheets("Sheet1")

        ' Copy cell A1 into cell A2
        .Range("A1").Copy
        .Range("A2").PasteSpecial xlPasteAll

        ' Copy the Range A1:B2 into the Range C1:D2
        .Range("A1:B2").Copy
        .Range("C1:D2").PasteSpecial xlPasteAll

        ' The same as the above, but specifying the top-left cell in the paste destination
        .Range("A1:B2").Copy
        .Range("C1").PasteSpecial xlPasteAll

    End With

End Sub

```

Some of the more useful options for `PasteSpecial` are:

1. `xlPasteAll`, which pastes everything;
2. `xlPasteFormats`, which pastes just the formatting of the cell;
3. `xlPasteFormulas`, which pastes just the formulas;
4. `xlPasteValues`, which pastes just the values;
5. `xlPasteValuesAndNumberFormats`, which pastes the values and the formats of the numbers.

The list of all options can be found in [9].

It is worth noting that it is usually a good idea to clear the clipboard whenever you copy and paste something (especially something large), otherwise you may get a warning that starts with *There is a large amount of information on the Clipboard*. To do this, write the following on a new line immediately after pasting:

```
Application.CutCopyMode = False
```

After getting more familiar with VBA, you will be able to write code that does not rely on the clipboard at all, so this step would not be necessary.

For more on `Application.CutCopyMode = False`, see [12] and the answers in [17].

5.3 Finding Table Dimensions

In many cases, the sizes of tables are constantly changing. Because of this, it is important to know how to determine the size of the table, so that we can copy and paste a correct range, or so that we can add more data to a table correctly, or many other things.

There are two particularly useful pieces of code that will help with this. They are:

1. `.Cells(1, .Columns.Count).End(xlToLeft).Column`
2. `.Cells(.Rows.Count, 1).End(xlUp).Row`

Note that both are assumed to be within a `With` statement, and used as properties of a `Worksheet`.

The first piece of code returns the *column number* that corresponds to the first non-empty cell on row 1, from the right. This is analogous to going to the right-most cell in the first row, then pressing `Ctrl` and the left arrow, and making a note of the column number. By changing the 1 in the first argument of the `Cells` function to a different positive whole number, say `n`, the code will return the column number of the first non-empty cell on row `n` from the right.

The second piece of code works the same as the first, but returns the *row number* and is analogous to going to the bottom cell in the first column, then pressing `Ctrl` and the up arrow, and making a note of the row number. Similarly, by changing the 1 in the second argument of the `Cells` function to a different positive whole number, say `n`, the code will return the row number of the first non-empty cell on column `n` from the bottom.

For example, consider a table that occupies every cell within the range `A1:C10`. The following code will determine the column number of the first non-empty cell on row 1 (from the right), the row number of the first non-empty cell on column 1 (from the bottom), and then prints a message box with the text *The table consists of 3 columns and 10 rows*.

```
Option Explicit

Sub TableDimensions()

    Dim colNum As Long, rowNum As Long

    With Workbooks("Example Book.xlsm").Worksheets("Sheet1")
        colNum = .Cells(1, .Columns.Count).End(xlToLeft).Column
        rowNum = .Cells(.Rows.Count, 1).End(xlUp).Row
    End With

    MsgBox "The table consists of " & colNum & " columns and " & rowNum & " rows."

End Sub
```

Of course, the code above assumes that the table is rectangular for the message box to be correct, and the code would have to be tweaked if the top-left cell of the table was not `A1`.

6 Manipulating Workbooks and Worksheets

6.1 Key Points

1. Open and close Workbooks with the `.Open` and `.Close` methods.
2. Add Workbooks and Worksheets with the `.Add` method.

6.2 Opening Workbooks

By supplying the file path of a Workbook, we are able to open the Workbook as part of a subroutine. There are also a number of options that can be used to control some aspects of how the Workbook is opened (see [7]).

Suppose that the file path for a Workbook is C:\Documents\Book1.xlsx. The subroutine below opens this Workbook.

```
Sub OpenWorkbook()  
    Workbooks.Open Filename:="C:\Documents\Book1.xlsx"  
End Sub
```

When the Workbook is already open, this does nothing and does not produce an error. If we wanted to open this Workbook as Read-Only, then we use the `ReadOnly:=True` option.

```
Sub OpenWorkbookReadOnly()  
    Workbooks.Open Filename:="C:\Documents\Book1.xlsx", ReadOnly:=True  
End Sub
```

It is important to note that the options use the combination of a colon and equals sign (`:=`) when giving them a value. If just an equals sign (`=`) is used, then the options are likely to not function as intended.

We saw in Section 3.3 how to assign a Workbook to a variable. We can use a combination of these codes to open a Workbook and simultaneously assign it to a variable to make it easier to work with – see the example below, which opens the Workbook `Book1.xlsx` and assigns it to the variable `mainBook`, and then changes the value of Cell 1 in the `Sheet1` Worksheet to `Test`.

```
Sub WorkbookOpener()  
    Dim mainBook As Workbook  
    Set mainBook = Workbooks.Open(Filename:="C:\Documents\Book1.xlsx")  
    mainBook.Worksheets("Sheet1").Range("A1").Value = "Test"  
End Sub
```

Note that there is a slight syntactic difference in this example – the parentheses have been included around the `Filename` argument. This is important: usually, the parentheses are *excluded* when a function is run without the output being assigned to anything like in the `OpenWorkbook` subroutine, and the parentheses are *included* when the output of a function is assigned to something.

6.3 Closing Workbooks

Closing a Workbook is even easier than opening it – all you need to write is a `.Close` after the Workbook. The example below closes the Workbook `Book1.xlsx` explicitly, and also by using the variable that the Workbook is assigned to.

```

Sub WorkbookCloser()

    Workbooks("Book1.xlsx").Close

End Sub

Sub SecondWorkbookCloser()

    Dim mainBook As Workbook
    Set mainBook = Workbooks("Book1.xlsx")
    mainBook.Close

End Sub

```

It is important to note that the .Close method will *only* work if the Workbook is open. If you try to close a Workbook that is not open (or that doesn't exist, either), you will get the error “Run-time error '9': Subscript out of range”.

A particularly useful closing option is the SaveChanges option. By specifying True, any changes made to the Workbook will be saved when the Workbook is closed. An example of how to use this option is given below.

```

Sub WorkbookCloser()

    Workbooks("Book1.xlsx").Close SaveChanges:=True

End Sub

```

Note the use of the colon with the equals rather than just an equals. More closing options can be found in [6].

6.4 Making a New Workbook

There are two main ways to make a new Workbook. The first is using the Workbooks.Add method, which is analogous to opening a fresh Excel manually; the second is an option when copying a Worksheet from an open Workbook.

The code below is an example of the first method which opens a new blank Excel Workbook when run.

```

Sub AddWorkbook()

    Workbooks.Add

End Sub

```

It is usually convenient to assign the new Workbook to a variable so that you can use it immediately. The example below creates a new Workbook and assigns it the the variable newBook. Try to figure out what the rest of the code is doing – it should be fairly intuitive.

```

Sub AddWorkbook()

    Dim newBook As Workbook
    Set newBook = Workbooks.Add

    With newBook

        With .Worksheets("Sheet1")

```

```

        .Name = "Summary Sheet"
        .Tab.Color = RGB(146, 208, 80) 'Light Green

        With .Range("A1")

            .Value = Now
            .NumberFormat = "dd/mm/yyyy"

        End With

    End With

    .SaveAs Filename:="New Book.xlsx"
    .Close

End With

End Sub

```

The second method just uses the `.Copy` method for a Worksheet that is currently open, without specifying where to copy the Worksheet to. When no destination is specified, the default action is to open a new book whose only sheet is the sheet just copied. The example below creates a copy of the sheet `Sheet1` in a new Workbook:

```

Sub SheetCopier()

    Workbooks("Example Book.xlsx").Worksheets("Sheet1").Copy

End Sub

```

The name of the newly created Workbook will be `BookN.xlsx`, with `N` replaced by a number corresponding to however many new Workbooks have already been opened during the session. This has the disadvantage of making it more difficult to work with the newly created Workbook.

6.5 New Worksheets and Position

We can create new sheets using the `Worksheets.Add` method. The example below creates a new sheet after the active sheet in the Workbook `Book1.xlsx`. The code is analogous to clicking the plus button at the bottom of the Workbook to add a new sheet.

```

Sub AddWorksheet()

    Workbooks("Book1.xlsx").Worksheets.Add

End Sub

```

You can specify where in the order of Worksheets you would like the new sheet to be created. For example, we can create the new Worksheet before or after the Worksheet `ExampleSheet` by writing the code below.

```

Sub AddWorksheet()

    With Workbooks("Book1.xlsx")

        ' Add sheet before
        .Worksheets.Add Before:=Worksheets("ExampleSheet")

    End With

End Sub

```

```

' Add sheet after
.Worksheets.Add After:=Worksheets("ExampleSheet")

End With

End Sub

```

There are two things to note: firstly, we do not need to specify which Workbook the ExampleSheet Worksheet is contained in, as we have already specified that we are adding the worksheet to the Workbook Book1.xlsx; secondly, it is important that there are no parentheses surrounding the .Add arguments in this particular example, as this would produce an error. We will require the parentheses in the next example.

We can give the new Worksheet a name in the same step by accessing the .Name property of the Worksheet. In this example, the parentheses are important.

```

Sub AddWorksheet()

    With Workbooks("Book1.xlsx")

        ' Add sheet before
        .Worksheets.Add(Before:=Worksheets("Example Sheet")).Name = "BeforeSheet"

        ' Add sheet after
        .Worksheets.Add(After:=Worksheets("Example Sheet")).Name = "AfterSheet"

    End With

End Sub

```

More information on adding sheets can be found in [8]. We can access other properties of the sheet when we first add it, like the colour of the tab. The code below adds a new Worksheet called New Sheet, and also makes the tab colour for the Worksheet green.

```

Sub AddWorksheetWithColour()

    With Workbooks("Book1.xlsx").Worksheets.Add
        .Name = "New Sheet"
        .Tab.Color = RGB(146, 208, 80)
    End With

End Sub

```

6.6 Copy and Paste Example

This section is an example that utilises a few of the concepts above.

For this example, we will assume that we have some raw data in a Workbook that we need to copy and paste into a different Workbook, and that we need to paste the raw data underneath the raw data that is already there. We will assume that both Workbooks have only three columns, and that the tables both start at A1 in the Workbooks. Try to determine what each step of the example is doing.

```

Sub TableCopierExample()

    ' Set up the dimensions
    Dim srceBook As Workbook
    Dim destBook As Workbook

```

```

Dim srceRows As Long
Dim destRows As Long

' Open the Workbooks
Workbooks.Open Filename:="C:\Documents\Destination Book.xlsx"
Workbooks.Open Filename:="C:\Documents\Source Book.xlsx", ReadOnly:=True

' Set up the Workbook variables
Set destBook = Workbooks("Destination Book.xlsx")
Set srceBook = Workbooks("Source Book.xlsx")

' Copy the raw data
With srceBook.Worksheets("Sheet1")
    srceRows = .Cells(.Rows.Count, 1).End(xlUp).Row
    .Range(.Cells(2, 1), .Cells(srceRows, 3)).Copy
End With

' Paste into the destination Workbook
With destBook.Worksheets("Sheet1")
    destRows = .Cells(.Rows.Count, 1).End(xlUp).Row
    .Cells(destRows + 1, 1).PasteSpecial xlPasteValues
End With

' Clear the clipboard
Application.CutCopyMode = False

' Save and close the Workbooks
destBook.Close SaveChanges:=True
srceBook.Close

End Sub

```

7 Objects, Properties, and Methods

In this paper so far, we have made reference to *properties* and to *methods* without defining them explicitly. This section will explain what these are in some detail, as having an understanding of how these work will improve your ability to use them. However, this section may be skipped without any detriment to the understanding of the remaining sections.

There are a number of resources available that explain these concepts (and others): the pages [2], [1], and [5] are three such examples.

7.1 Key Points

1. A *property* of an object is exactly that – it can be something as simple as the colour of the object.
2. A *method* is any kind of action that can be performed, either with the object or on the object.

7.2 Objects

To understand properties and methods, you need to know what an object is. We briefly mentioned in Section 3.2 that VBA is an Object Oriented programming language and we gave three examples of objects, namely Workbooks, Worksheets, and Ranges.

If we consider how a computer program usually works, there are a collection of variables which are related and manipulated using functions. In an Object Oriented language, some predefined variables

and functions are bundled together to create an *object* – the variables that make up the object are its properties, and the functions are its methods.

For example, a cell in a Worksheet is an object (specifically, a single cell is still a Range object). Some examples of its properties are:

1. Address (such as A1);
2. Value (either text or numeric);
3. NumberFormat (which is just the format of the cell value);
4. HorizontalAlignment (left justified, center, right justified);
5. Formula (such as =SUM(A1:B2), which would also change the Value).

Some examples of its methods are:

1. Copy (which copies a selection of the properties);
2. PasteSpecial (which we have already seen);
3. ClearContents (clears formulas and values);
4. AddComment (adds text as a comment to the cell).

The full list of all properties and methods for Range objects can be found in [13].

7.3 Properties and Methods

Given an object, the easiest way to determine the value of one of its properties is to use the `Debug.Print` function and display the property itself. For example, we may write the following:

```
Sub DisplayProperty()  
    Debug.Print Workbooks("Example Book.xlsm").Worksheets("Sheet1").Range("A1").Value  
End Sub
```

In this case, the object is referenced by writing

```
Workbooks("Example Book.xlsm").Worksheets("Sheet1").Range("A1")
```

and the property that has been called is the `Value` property. Supposing that we put the text *This is cell A1* into the cell referenced above, the subroutine above would produce the text *This is cell A1* in the Immediate Window.

This can be particularly helpful for when you start using more variables in your code, particularly when you start to define cell references numerically using variables. For example, in the following piece of code we include the `Debug.Print` function at the bottom to check the address of the cell defined by the variables.

```

Sub VariableCellReference()

    Dim rowNum As Long, colNum As Long
    Dim newCell As Range

    rowNum = 2
    colNum = (2 * rowNum) + 3

    Set newCell = ThisWorkbook.Worksheets("Sheet1").Cells(rowNum, colNum)

    Debug.Print newCell.Address

End Sub

```

Here, we have utilised the Address property of the cell. When we run this code, we get the message \$G\$2 in the Immediate Window.

8 Error Handling

In most cases, getting an error in your code is an indication that something is not working as expected. Thus, the usual error handling behaviour is to stop the code and display the error as soon as one has been triggered. However, there are some cases where it is beneficial to have more control over what happens when an error has been triggered.

For more on error handling, visit the pages [\[3\]](#) and [\[11\]](#).

8.1 Error Handling Options

The error handling options are set by first writing `On Error`, and then the error handling option which is any of the options (descriptions taken from [\[3\]](#)) listed in Table 2.

Option	Description
<code>Goto 0</code>	When error occurs, the code stops and displays the error
<code>Goto -1</code>	Clears the current error setting and reverts to the default
<code>Resume Next</code>	Ignores the error and continues on
<code>Goto [Label]</code>	Goes to a specific label when an error occurs

Table 2: Error Handling Options

Note that `[Label]` is replaced by the label that you wish to use.

8.2 Error Handling Example

One example of where control over error handling is beneficial is when you want to ungroup a column by writing `Columns(1).Ungroup` as part of a usual process. Consider this in the following code:

```

Sub UngroupColumn()

    Columns(1).Ungroup

End Sub

```

This will work fine when column 1 is grouped, and will ungroup the column as expected. However, if column 1 has been ungrouped at some point (manually or otherwise) before the `UngroupColumn` subroutine runs, an error will be generated by this `Ungroup` method and will cause the code to stop running.

One way to work around this could be to write an if-statement to check whether column 1 is ungrouped or not, and to only attempt to ungroup it if it is still grouped. An alternative method is to change the error handling options so that if an error is generated, we ignore the error and continue running the code:

```
Sub UngroupColumn()  
    On Error Resume Next  
        Columns(1).Ungroup  
    On Error Goto 0  
  
End Sub
```

When the subroutine above is run, column 1 will be ungrouped if it is grouped, or nothing will happen (with no error) if it is already ungrouped. Since we only want to ignore errors for the `Ungroup` method, we have to set the error handling back to its default by writing the `On Error Goto 0` line.

9 Assign Keys

It is possible to assign subroutines to keys so that we can exploit custom keyboard shortcuts by using the `Application.OnKey` method. The exact syntax required to define the key combination is very specific, so if you intend to assign your own keys, you should read [\[12\]](#).

9.1 Assign Keys Example

Suppose that we want to write a subroutine that adds 1 to the current selection when we press `Alt` and the `up` key. We do this in two parts: firstly, we need to write the subroutine that adds one to the selection; secondly, we then need to assign this subroutine to the key combination `Alt` and `up`.

We can achieve the first part by writing the following subroutine [\[15\]](#):

```
Sub PlusOneToSelection()  
    Dim r As Range, c As Range  
    Set r = Selection  
  
    For Each c In r  
        c.Value = c.Value + 1  
    Next  
  
End Sub
```

This is then assigned to the key combination `Alt` and `up` by writing the following subroutine:

```
Sub AssignKeys()  
    Application.OnKey "%{UP}", "PlusOneToSelection"  
  
End Sub
```

Once this `AssignKeys` subroutine has been run, pressing the keys `Alt` and `up` will add 1 to the current selection as desired. It should be noted that this shortcut will work across all open instances of Excel, and that when all instances of Excel have been closed, this assignment will be reset. This means that the `AssignKeys` subroutine will have to be run again to assign the `PlusOneToSelection` subroutine to the key combination `Alt` and `up` any time that Excel is opened fresh.

If you want this assignment to be run every time you open the workbook with the above subroutines in, you can write a subroutine which runs as soon as the workbook it is written in is opened. To do this, we have to use a specific name for the subroutine, which is `Auto_Open` (there is also `Workbook_Open`, as well as `Workbook_BeforeClose`, but we will not cover them in this paper).

The following code will run whenever the Workbook is opened, which will run the `AssignKeys` subroutine, which will assign the `PlusOneToSelection` subroutine to the key combination `Alt` and `up`.

```
Sub Auto_Open()  
    Call AssignKeys()  
End Sub
```

10 Functions

Another major advantage of using VBA is the ability to write your own functions. These have a very similar syntax to subroutines, but we need to make sure that we explicitly define what the output of the function should be.

We also do not 'run' functions, so we cannot assign them to keys or to buttons. Instead, we use custom functions exactly the same way that we would use the default functions, but with the added advantage that we can also use custom functions within subroutines (normal Excel functions cannot usually be used within subroutines).

10.1 Function Examples

Defining a function is just like defining a subroutine, except that the `Sub` text is replaced by `Function`, and the output of the function is defined by writing the name of the function, followed by an equals sign (=) and the desired output. It is also good to declare the data type that the output of the function will be after writing the function name. For example, the function defined below adds 1 to the number provided as an argument to the function:

```
Function PLUSONE(value As Long) As Long  
    PLUSONE = value + 1  
End Function
```

By utilising if-statements, you can conditionally assign function outputs depending on the conditions that you provide:

```
Function IFZERO(value As Variant, value_if_zero As Variant) As Variant  
    If value = 0 Then
```

```

        IFZERO = value_if_zero

    Else

        IFZERO = value

    End If

End Function

```

The functions can get as complex as you like, and they can be used as soon as they are written. It should be noted that custom functions are *only* accessible from the Workbook that they are written in (unless you save them into an Add-In, but that is not covered in this resource).

11 Outlook

It is possible to control the Outlook application using VBA code. Actually, Outlook has its own VBA editor which we can utilise, but this paper will only cover controlling Outlook from within Excel's VBA editor.

11.1 Setting Up The Reference

In the previous sections, we have declared variables as a number of different types of objects such as Range, Worksheet, and Workbook. These object classes are available by default, but there are numerous object classes available that need to be added before they can be used. To add object classes, we need to enable the corresponding **Reference**.

To open the list of available references, open the VBA window, then click the **Tools** tab followed by **References...** This will open a new window with a long list of all of the available references.

The Outlook reference that we need to enable to make the Outlook object classes available is **Microsoft Outlook 16.0 Object Library**. Check the tickbox next to this reference and then click **OK** to enable the reference.

11.1.1 Programmatically Add Outlook Reference

Adding this reference can be done programmatically, rather than checking the reference manually. To do this, we can write the code below.

```

Sub AddReferenceGUID()

    ' Outlook Reference
    ThisWorkbook.VBProject.References.AddFromGuid _
        GUID:="{00062FFF-0000-0000-C000-000000000046}", _
        Major:=0, _
        Minor:=0

End Sub

```

The GUID supplied is the GUID corresponding to the **Microsoft Outlook 16.0 Object Library** reference. To get the GUID for all enabled references, we can write the code below.

```

Sub GetReferenceGUID()

    Dim i As Integer
    For i = 1 To ThisWorkbook.VBProject.References.Count
        With ThisWorkbook.VBProject.References(i)
            Debug.Print .Name, .GUID, .Description, vbNewLine
        End With
    Next i

End Sub

```

11.2 Writing Emails

With the Outlook reference set up, we have control over many aspects of the Outlook application. One of the most useful aspects is writing emails.

To write an email, we need to use the Outlook.Application and Outlook.MailItem object classes. See the code below.

```

Sub WriteNewEmail()

    Dim OutlookApp As New Outlook.Application
    Dim OutlookMail As Outlook.MailItem
    Set OutlookMail = OutlookApp.CreateItem(olMailItem)

    With OutlookMail
        .BodyFormat = olFormatHTML
        .Display

        .To = "john.doe@example.com; joe.bloggs@example.com"
        .CC = "jane.doe@example.com"

        .Subject = "Email from VBA"
        .HTMLBody = "This is some text."
    End With

End Sub

```

There are a few things going on here, but we'll only focus on the OutlookMail part. It is important to include the Display property at the start as this displays the email (so that we can see it). However, the BodyFormat property can be omitted if you want.

The To and CC properties should be easy to understand – just remember to delimit separate email addresses with a semicolon. Similarly, the Subject property should also be easy to understand. Finally, the HTMLBody property corresponds to the content of the email – this is where the body of the email can be written.

The HTMLBody property actually comes with a non-trivial default value, namely your email signature. Thus, it is usually preferable to define the HTMLBody property with itself concatenated onto the end, as in the code below.

```

.HTMLBody = "This is some text." & HTMLBody

```

There is one drawback to using the HTMLBody property in this way: it will cause Outlook to generate a security prompt when this code is run, and this is currently unavoidable (it's a countermeasure against malicious code).

The properties (To, CC, Subject, and HTMLBody) can all be set up manually, or by linking them to calculated fields in, say, the corresponding workbook – that’s up to you. Once you’ve set up how your email will be constructed, you may wish to link the subroutine to a button (Section 2.7) or find some other way to run it easily. The steps to send the email will be to run the subroutine however is convenient for you, allow the VBA to run in the security prompt, double check that the email has been created correctly, and then click the send button.

It should be noted that there is a .Send method that you can put inside the end of the `With OutlookMail` statement, but this will also always generate the security prompt. More importantly, including this method will not allow you to give the email a quick look over, so you may accidentally email out a nonsensical email if something went wrong in the code.

11.2.1 Additional Control: Attaching Documents, Voting

Two more useful properties that we can use when writing emails are .Attachments and .VotingOptions, which allow the manipulation of files attached to the email and the inclusion of voting options, respectively.

To add an attachment to a file, we need to use the .Add method of the .Attachments property and supply the full file path for the file to attach to the Source argument. An example of how this might look is below.

```
.Attachments.Add Source:="C:\Documents\Test.xlsx"
```

To add voting buttons to the email, we need to use the VotingOptions property. The voting options are input the same as they would be if you were to add them in the email itself – each of the items should be listed on the same line, and delimited by semicolons. An example of how this might look is below.

```
.VotingOptions = "Option 1;Option 2;Option 3"
```

There are loads of other properties that can be utilised – these are just two that I’ve personally used fairly frequently.

11.2.2 Writing HTML

The start of this section showed that the body of the email can be defined in the HTMLBody property. This property, as the name suggests, allows the body to be supplied in a HTML format so that we can add colour, change font size and type, add rich text, add line breaks, and so on to the body of the email. This guide isn’t a HTML guide, so we’ll only supply some basic HTML and describe how to see the HTML of an email.

It’s important to understand that the majority of the HTML elements (code of the form ``) need to be closed after they’re opened, much like a do loop needs an end statement, or a `Sub` statement must end with an `End Sub` statement. An HTML element opens with the angle brackets and the element tag, say ``, and closes with the angle brackets and the same element with a slash before the element tag, like ``. In Table 3, the . . . is to be replaced by whatever you want to contain within the element.

The following code example shows how these would be used in the body of an email – try running this code to see the email that is created.

Outcome	HTML
Line Break	
Bold	...
Italic	<i>...</i>
Underlined	<u>...</u>
Font Size (14pt)	...

Table 3: Basic HTML

```

Sub AddingRichText()

    Dim OutlookApp As New Outlook.Application
    Dim OutlookMail As Outlook.MailItem
    Set OutlookMail = OutlookApp.CreateItem(olMailItem)

    With OutlookMail
        .BodyFormat = olFormatHTML
        .Display

        .HTMLBody = "" &
            & "<b>Bold</b>" & "<br><br>" &
            & "<i>Italic</i>" & "<br><br>" &
            & "<u>Underlined</u>" & "<br><br>" &
            & "<span style='font-size:14pt'>Large</span>" & "<br><br>" &
            & "<span style='font-size:9pt'>Small</span>" & "<br><br>"
    End With

End Sub

```

To see the underlying HTML of an email that you've received, open the email in a new window (pop-out or double click). In the **Message** tab, there is an **Actions** drop-down box in the **Move** section. Click this, then **Other Actions**, and then **View Source**. The window that opens is the underlying HTML for the email. This will usually come with a lot of MS 'junk', but the body content of the email will be right at the very bottom in the <body> element (there will still be a lot of 'junk' to sort through, but it can be a good starting place to figure out the HTML that you need to format a particular piece of text).

Appendix

Useful Characters

Character	Effect
Apostrophe (')	Comments out the remainder of the line
Colon (:)	Allows multiple statements to be written on one line
Underscore (_)	Used to break a line onto a new line

Table 4: Useful Characters

Useful Subroutine Code

```
Sub ExampleSub()  
    ' Speeds up processing  
    With Application  
        .EnableEvents = False  
        .ScreenUpdating = False  
    End With  
  
    ' To refresh PivotTables  
    Workbooks("Book With PivotTables.xlsx").RefreshAll  
  
End Sub
```

References

- [1] Alexander Petkov. *How to explain object-oriented programming concepts to a 6-year-old*. freeCodeCamp News, 2018. Available at: <https://www.freecodecamp.org/news/object-oriented-programming-concepts-21bb035f7260>, accessed 01-06-2019.
- [2] Corporate Finance Institute. *VBA Methods*. Available at: <https://corporatefinanceinstitute.com/resources/excel/study/vba-methods-list>, accessed 28-05-2019.
- [3] Paul Kelly. *A Quick Guide to Error Handling*. Excel Macro Mastery. Available at: <https://excelmacromastery.com/vba-error-handling/>, accessed 18-06-2019.
- [4] Microsoft. *Data Type Summary (Visual Basic)*. Microsoft Docs, 2015. Available at: <https://docs.microsoft.com/en-gb/dotnet/visual-basic/language-reference/data-types/index>, accessed 21-04-2019.
- [5] Microsoft. *Understanding Objects, Properties, and Methods*. Microsoft Docs, 2017. Available at: <https://docs.microsoft.com/en-us/office/vba/word/concepts/objects-properties-methods/understanding-objects-properties-and-methods>, accessed 01-06-2019.
- [6] Microsoft. *Workbooks.Close method (Excel)*. Microsoft Docs, 2017. Available at: <https://docs.microsoft.com/en-us/office/vba/api/excel.workbook.close>, accessed 08-05-2019.
- [7] Microsoft. *Workbooks.Open method (Excel)*. Microsoft Docs, 2017. Available at: <https://docs.microsoft.com/en-us/office/vba/api/excel.workbooks.open>, accessed 24-04-2019.
- [8] Microsoft. *Worksheets.Add method (Excel)*. Microsoft Docs, 2017. Available at: <https://docs.microsoft.com/en-us/office/vba/api/excel.sheets.add>, accessed 11-05-2019.
- [9] Microsoft. *XlPasteType enumeration (Excel)*. Microsoft Doc, 2017. Available at: <https://docs.microsoft.com/en-us/office/vba/api/excel.xlpastetype>, accessed 24-06-2019.
- [10] Microsoft. *XlSheetType Enumeration (Excel)*. Microsoft Docs, 2017. Available at: <https://docs.microsoft.com/en-us/office/vba/api/excel.xlsheettype>, accessed 23-04-2019.
- [11] Microsoft. *On Error statement*. Microsoft Docs, 2018. Available at: <https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/on-error-statement>, accessed 18-06-2019.
- [12] Microsoft. *Application.OnKey method (Excel)*. Microsoft Docs, 2019. Available at: <https://docs.microsoft.com/en-us/office/vba/api/excel.application.onkey>, accessed 18-06-2019.
- [13] Microsoft. *Range object (Excel)*. Microsoft Docs, 2019. Available at: [https://docs.microsoft.com/en-us/office/vba/api/excel.range\(object\)](https://docs.microsoft.com/en-us/office/vba/api/excel.range(object)), accessed 04-06-2019.
- [14] RemarkLima. *How to comment and uncomment blocks of code in the Office VBA Editor*. Stack Overflow, 2012. Available at: <https://stackoverflow.com/q/12933279/8213085>, accessed 25-06-2019.
- [15] rory.ap. *Plus One To Selection*. Stack Overflow, 2015. Available at: <https://stackoverflow.com/a/28587328/8213085>, accessed 18-06-2019.
- [16] Siddharth Rout. *How to avoid using Select in Excel VBA*. Stack Overflow, 2013. Available at: <https://stackoverflow.com/a/10718179/8213085>, accessed 29-04-2019.
- [17] user4039065. *Should I turn .CutCopyMode back on before exiting my sub procedure?* Stack Overflow, 2015. Available at: <https://stackoverflow.com/q/33833318/8213085>, accessed 26-06-2019.