## Contents

## cheat sheets

This section will contain various cheat sheets which are helpful.

### keyboard shortcut reference

This will contain a table that gives the equivalent commands for different keyboard shortcuts. It does not intend to be exhaustive but instead to hit all the major ones.

| shortcut | Range function |
|---|---|
| **Range related** | |

| shortcut | Range function |
|---|---|
| SHIFT + Space | `EntireRow` |
| CTRL + Space | `EntireColumn` |
| CTRL + A | `CurrentRegion` |
| CTRL + / | `CurrentArray` |
| Arrows | Offset() |
| SHIFT + Arrow | Resize() |
| CTRL + Arrow | End() |
| CTRL + Plus (+) | Insert() |
| CTRL + SHIFT + L | AutoFilter() |
| ALT + SEMICOLON | `SpecialCells(xlCellTypeVisible)` |
| **Application related** | |
| F9 | Calculate |
| CTRL + ALT + F9 | CalculateFull -or- CalculateFullRebuild |

TODO: finish this table

### VBA Cheat Sheet

This will include a cheat sheet of common VBA functions, control structures and commands on common objects.

TODO: create this cheat sheet

### Excel Object Model Cheat Sheet

This cheat sheet will provide a quick glance at the most commonly used objects in Excel and how they are related. It is meant to be a useful check when you know what you want to work with but are not certain how best to get there.

TODO: finish this list

- Application
- Workbooks -> Workbook
    - Worksheets -> Worksheet
    - Range -> Range
        * Formula
        * Value
        * Address
        * [formatting things]
        * Cells / Rows / Columns
    - Cells -> Range
    - ChartObjects -> ChartObject
        * Chart
            · Series
            · Axes -> Axis
            · ChartArea
            · PlotArea
    - Shapes -> Shape
    - Names -> Name
    - RefersToRange -> Range

## overview of intro, overview

Not sure what to put in the introduction. Think about the goals of this book, its organization, resources on the web, etc.

### comments on knowing Excel before starting VBA

The Basics of Excel chapter may not actually be needed. The problem with working through VBA is that it requires a baseline understanding of how to use Excel. In general, it requires more than a baseline understanding of how to use Excel – it requires an advanced understanding of how Excel works. In particular, it's important to know what happens by default when you do something in Excel. This then makes it much easier to reason through how the VBA is going to function.

TODO: add explanations for other reasons why: using formulas, intuition, knowing what's possible.

An example: if you have never used the keyboard shortcut CTRL+arrow to jump around a block of data, you would never think to look for the Range.End() command which replicates this behavior. Instead, you might be tempted to iterate through every cell calling Offset(1) until the cell is empty.

A second example: most people are familiar with the AutoFill behavior by dragging the corner of the current cell. This is great most of the time but has the bad habit of trying to predict a series when you just want a constant. There is also the Fill command (and a keyboard shortcut `CTRl+D` for fill down and `CTRL+R` for right) that will copy the formula or the value without trying to predict the next value. Fill is much more useful when working with VBA since it's unlikely to secretly ruin your data.

Knowing that there are multiple ways to do something and knowing the quirks of specific commands is invaluable when working with Excel through VBA. You will have a much better intuition for what will happen if you know how Excel normally does things.

If you don't have this intuition, you can still learn VBA and be effective, but you may find yourself falling back on common programming techniques that are not "idiomatic" Excel.

This section might include some the VBA equivalent of common Excel techniques?

TODO: decide if this section is needed

## overview of basics of VBA

section on VBA topics should include:

- basics of variables
  - common variable types
  - difference between value and reference types
- basics of control structure
  - if
  - select case
  - for loop
  - foreach loop
  - do/while loops
  - goto
- error handling
- Subs and Functions

most of this section is going to be boilerplate explanation of these things

consider how to improve on that to avoid saying the same stuff as everyone else (or just power through it to get it on paper)

later on, there should be an advanced VBA section to handle:

- classes
- events
- adding references to other objects (specifically Office and Microsoft Runtime)

The pitfalls of running macros and how they destroy the undo functionality.

- using the debugger
- using the Immediate window

### VBA 101

This section will focus on the basics of opening the Visual Basic Editor and actually getting started with VBA. This is called VBA 101 because these are the steps that you have to know before you can start programming. Hopefully these steps are already done or you can work through them quickly.

There are a handful of ways of getting the VBE to open. Those include:

- Use the keyboard shortcut ALT + F11
- Use the button on the Developer tab of the Ribbon (if enabled) (TODO: include steps for how to enable this menu if needed)
- Hit the Debug button if an existing code sample fails
- Hit the edit button on the run macro button, again off the developer tab (also available with `SHIFT`+`F11`)

Of these, the first two are the more common ways to get the VBE to open.

Several of these methods require the Developer tab to be showing. In general, if you are working with VBA, this tab is useful. To enable it, you need to go through the settings to Customize the Ribbon. On a default install, this tab will not be visible.

TODO: add some steps and images for the Developer tab on the Ribbon

Once you have the VBE open, you can then begin to add or edit code for your spreadsheets. The basics of the simple program are included below.

TODO: add the steps for a Hello World… how to create Sub and output the text

TODO: add pictures for these steps too

This section of the book is one of the few where the steps will be so clearly specified. I am going to this level of detail to ensure that you are able to get things started. This book will not include such detailed steps later for how to type and run code.

### introduction to VBA

This chapter will focus on the basics of VBA that are essential to using VBA to work with Excel. The upside of VBA is that it has a very simple instruction set. The downside of VBA is that it has a very simple instruction set. Fortunately, the vast majority of Excel/VBA interaction can be handled with very simple instructions. Honestly, the real difficulty with using Excel/VBA is not the VBA side of things, it's managing the object model for Excel. This object model does not introduce new commands, but it does add a large number of interrelated objects, properties, and Functions that need to be known at some level to do anything. If you compare the length of this chapter to the length of the book, you will get a sense of what is meant by this.

An important thing to remember about VBA is that it exists outside of Excel, in some sense. VBA (Visual Basic for Applications) is derived from VB6 which is a legitimate programming language that (previously) was used for serious programming. These days (ca. 2017), no one starts a new project looking to use VB6; it just doesn't offer the features of modern programming languages. That VBA exists outside of Excel means that there are certain parts of the language that are independent of anything Excel has to offer. These aspects of VBA are the core parts of the language, and, simply, you have to understand these core parts before you can do anything related to Excel. Technically, you can get by copying code from the internet (or this book) and making simple changes, but you will never truly get good at VBA doing that. Also, doing that for more than a couple tasks is counter productive since learning VBA proper does not involve that many commands.

Having said all of that, VBA consists of several key instructions:

- Declaring and setting variables
- Declaring and calling Subs and Functions
- Logic structures
- Loop structures
- Other control structures (Errors and Goto)

In addition to those aspects of using the language, there are a handful of details related to programming in general that are worth hitting:

- VBA 101, opening the VBE and getting started
- Adding references (how and why)
- Debugging code and using the tools provided

The flow of this chapter will hit on the VBA 101 question first. From there, we'll hit the language basics, and then touch on the 2 more advanced aspects of using VBA and Excel together.

Finally, it's worth noting that this basic overview misses a couple parts of VBA that might come up from time to time. They will be mentioned at the end of the chapter in passing, but this book is not a VBA reference. This book is designed to get you using VBA in a professional setting with confidence. Knowing every nook and cranny of the language is not critical for that goal.

## declaring and setting variables

One of the core tasks when programming via VBA is working with variables. Variables encompass a couple of different topics which makes sense since they are one of two core areas of VBA alongside control structures. That is, your programming exists of two possible categories: variables and control structures. Variables are made of the variables that you will need to delcare and set to make your program work while also including all of the various aspects of the Excel object model. The object model is made of a significant number of variables (e.g. cell value for each cell) and a handful of Subs and Functions. The variables that you declare and use will look very similar to the object that the Excel model is using. There are also a large number of variables that you will create which exist to guide your own control structure or to encompass the algorithms that you need to execute.

To fully work with variables, it helps to split the topic into two areas: declaring variables and setting variables. These two topics are quite simple when it comes time to type out the commands, but variable declaration adn setting is at the core of planning how a program will work. The variable declaration will directly shape how the control strcutures will work. The two go hand in hand and are equally important.

### Declaring Variables

Decalring vairables is a straight forward tasks. VBA offers a simple command to declare a new variable: `Dim`. Note that you can technically use a vriable before declaring it, but you should really avaoid htis practice. It leads to the potential to create all sorts of bugs later. Just don't do it. To better avoid this, setting hte flag in teh settings (TODO: add a picture of that).

When decalring a vairable, there are two compoentns to it: variable name and variable type. Variable name is wholly your decision with only a couple of constraints. You are not allowed to duplicate the name of an internal comannd, and you shoudl go to some lenght ot avoid using the same name as an Excel object model name. Beware that naming a variable ahs certain concentions, but theese do not have any effect on the program execution. The main concern with names is that they will directly affect your ability to work with and maintain your code. Naming things is hard. Pick a strategy that works for you and your coworkers and get on it with it. There is no single answer here about how to name things.

The second part of the puzzle is to declare the type of the variable. This is THE core part of variables. When declaring a vairable, you are essentially deciding if the tpye should be the generic `Variant` or if you should actualyl declare a tpye. Note that there are times when you have ot use Variant, but in general, you should use the most speciifc tpye that is possible. These tpyes can either draw from VBA or from the Object Model, or from your own created types. When thinking of variable types, tehre are two major groups of types:

- Vlaue types = a number, string, or boolean
- Reference types = objects

#### Setting variables

Setting a variable is quite straight forward. The rule is: for reference tpyes, you must use `Set`, for value types, you must not.

The real task then is to determine whether or not you are workign wtih a refernece type. The rule here is: if you are workign with an object, it is a refernece type. If you are working with a value (number, stirng, bool), then you are working with a value. Another approach, if you intend to use a `.` to call out some property of your variable, then it is a reference type and requires `Set`. The one odd excepipon here is arrays: they are declared without using Set.

#### Using Variables

It seems somewhat obvious that you would want ot use a variable after declaring and setting it. This is generally alwayas the case (why else would you create the variable). To that end, there are a pair of ways to use variables depending on whetehr it is a reference or value type. The easier one to understand is the value tpye where you simply ues the variable as is the same as would appear in a mathematical expression. The more complicated example comes with reference types where the variable stores a reference to another object. These variables have the ability to access either a property of the type or the default `Value` of

the type. This distinction adn the overall distinction between reference and value can become incredibly confusing with the Excel Object Model since so many properties of objects reduce to value types. An example is the value of a `Range` which will hold some numbre or string or Error depending on what the cell contains.

When accessing a proprety of the obejct, you will use the `.` and find the property by name. In this way, you can chain together a series of comamdns to access the properties of objects. It is oftentimes the case that the property is itself anotehr object hwich makes it possible to use another `.` to keep going. If you are using the VBE and properly delcaring your variables, the VBE will work to provide helpful suggestions of waht may be possible to use next (this is called Intellisense). The one pitful to Intellisense is when the return from a given property can be Variant or a combination of possible results. When this happens, Intellisense will not offer any suggestions and you are left guessing whether or not the command exsits. This is where it can be quite helpful to do one of two things:

- Create a new variable with the tpye that you know the object will have and Set that reference before using it. This “cheats” and tells Intellisense exactly twhat you expect to exist.
- Read through the documentation and gain an understanding of what ypes are possible and just use them. There is no rule that the type must be suggested by Intellisense for it to be valid.

In general, I take a combination of those two approaches often. If I expect to use the vriable a njmber of times, I will go with the new variable route to avoid guessing properties later. If I only need the variable once or am copying code from somewehre else (and know it works), I will just go with the code as is without Intellisense. The one upside of creating new variables is that it forces you to be more explitcit with your declarations. It also clearly shows your intent to other developers thta may see your code later. Finally, there is nothing worse than missing a typo that Intellisense didn’t catch and having that be a shwo stopping bug at some poinit.

I mentioned it above, but it is worht digging into the default `Value` property a little more. This can be a source of confusiojn because very often, you will accidentally use the name of a variable wiitout calling for a property. In other programming languages, this will result in a compile time or runtime error that is thrown because of the bad call. In VBA, your code will run and even worse will return somethign from teh object that may not be what oyu want. When this happens, it can be incredibly diffciult to track down the soruce of the error. To avoid this, you could never use the variable name as a shortcut ot the `.Value` property. In practice this is a pain to manage and I will often mix and match whether or not Value is called. Sometimes, I am tired of typing out Value and just let the default work. Other times, I am being very diligent about calling evertyhing explicitly to avoid some unforeseen error later. You will find that this comes down to your own preference and hte preferences of others working on your code.

## using Subs and Functions

The basic building blocks of your VBA efforts will be the Sub and the Function. It's possible that they are your only top level components if you do not use Class Modules. In all my years of using VBA, I've used Class modules only a couple of times, so they're not common.

Having said that, Subs and Functions are actually far more similar than different. The only real difference between the two is that Function can return a result back to the caller. A Sub on the other hand is meant to execute without returning anything back to the caller. It's possible to have a Sub manipulate a variable with can approximate returning a value for a little more work. If you're using a Function as a UDF (see chapter XXX, TODO: add link), then there are further limitations on what your Function can do. If you are not using it as a UDF, then there are no limitations that make a Sub distinct from a Function. The only difference is how you call them (if you want the return value) and that a Function is made to return something.

If you have a Function that does not actually return a value, it is the same as a Sub with the same code.

TODO: add an example of a Sub

TODO: add an example of a Function

## declaring the parameters (Subs and Functions)

When creating a new Sub or Function you are able to determine the inputs to your new creation. There are a handful of ways of handling the inputs:

- Put the inputs into the parameters of the Sub/Function and allow the caller to provide them
- Use knowledge of the spreadsheet to determine the inputs (or prompt the user for an input)

The main split here is: do you require the person typing the VBA to give you the inputs? Or, do you use some other approach like asking the user or just pulling the inputs from the spreadsheet.

The most common approach is to pull the inputs out of the spreadsheet. This seems counterintuive, but if you consider that the vast majority of VBA code is purpsoes wirtten for a single use, then it stands to reason that code will not be built on a large nujmber of Subs/Functiojns accepting parameters. The reason for this is that generally someone writes VBA to handle *their* spreadhseet and so the VBA just refelcts that spreadsheet. This works great for indivudal cases but can become a burden when building larger workflows. The main thign to consider for lager workflows is that as the complexity grows, tehre will be a large amount og code that is called multiple times or could be called separately from the main workflow. When thi sis the case, you are often served by pulling that code out into its own Sub/Functiojn wiiht aprameters.

To create a Sub or Function with parmaeters, you simply add them to the defintion line:

```
Sub WithSomeName(firstParameter as String)

End Sub
```

This approach is very simple. You give the parameter a name and a type delcaration. This is very nice becasue it nearly exaclty matches the `Dim` statement with a Sub. That correspondence makes it very easy to start with an internally declared variable and then upgrade it to parmaeter. You can also go the other way: take a parmaeter and inline it into the Sub with some default or determined value. This is less common.

Once the parameter has been given a name and a type, you can simply use it within the Sub like any other variable. In this regard, your code will look teh exact same. IF you are the person typing the VBA to use this Sub, then you will have to provide an appropriate variable as the parameter to make it all work.

#### declaring an Optional parameter

The one additiojnal thign to consider is that of `Optional` parameters. An optional parameter is one who is not strictly requried. In liue of a value, you can either leave the parmaete rmissing or provide a default value. In iether case, you can use the VBA specific function `IsMissing()` to determine if the aprmaeter was entered. An Optiojnal parameter can be a very nice fearture when you are trying to determine whether or not to make a Sub take parameters or just use defaults. You can provide the defaults in the parmaeter declaaration adn then allow the user (person typing the VBA) to override them if needed. This is a very common approach when writing library type code; provide snesible defualts that can be overwritten.

### calling a Sub or Function

When you are calling a Sub or Funciton, there are a couple of ways to do it. The preferred approach is to simply type the name of the Sub/Function aong with any required parameters. This will call the Sub. Another approach is to use `Call SubName` which is the same as `SubName`. This is an older approach that omse people prefer. It can sometimes be the case that Sub names are not particularly clear in the VBA and using `Call` has the effect of making it obvious that code flow is being directed into a Sub.

WHen calling a Funciton, you have the same approaches available. You can just use the Functiojn name or use call (TODO: is that right?).

ONe thing to be aware of with Functiojns is how to properly handle the return from teh functiojn (assuming it actually returns something). This is where VBA gets a bit weird. THe rules here split on whether the Functiojn returns an Object or Value type.

For either type, you are reuiqred to call the Functiojn iwht parenthesesis. This signals to VBA: please retain and use the return of this Functiojn. For a reference tpye, you will need to use `Set` as required. For a value type, you will omit `Set`. See the code example below.

If you ever get the compile time error `Object reference not set` this means that you have not used a `Set` somewhere that is required. A good place to check are spots where you are usign the retunr from a function. The same thing happens if you omit the parentheses. (TODO: is this right?)

```
Sub ExampleOfCallingCode()
    Dim rngReference as Range
    Set rngReference = someFunctionThatReturnsARange()

    Dim dblValue as Double
    dblValue = someFunctionThatReturnsADouble()

End Sub
```

### declaring the return type (Function only)

For a Function, the only extra step is to declare the return type of the Function. This is done after the normal parameters, with an extra `as Type` where `Type` is the actual type that you want to return. Note that this type must be compatible with all possible Types that you could return. Sometimes this means that you need to return a Variant in order to have all possible return Types available to you. There are times where this makes sense (and a large part of the Excel object model does this), but note that using Variant will make it hard to use Intellisense to figure out what your VBA is capable of doing.

TODO: is this a Variant by default?

TODO: give some examples of Function returns (or link to examples of them)

#### returning from a Function

If you want to take advantage of a Function, you need to return a value from your Function. This returned value can then be consumed by the caller (or not). To return a value from a Function, you simply use the

Function name as a variable and set its value appropriate. If the return type is an object or reference type, then you need to use Set to return the object. If it is a value type instead, you can simply set the return with an equal statement like any other value type. Once you have made the return statement, you can call Exit Function to break out of the Function.

For the caller, there are two things to keep in mind when using Functions. The first is that you must call the Function with parentheses in order to access the return value. The corollary of this is that if you call a Function with parentheses, you must use that return value to set the value of a variable. You will get an error if you do not do this correctly. Note that if you do not want the return value for some reason, you can avoid using parentheses in the same way you call a Sub. The second part is that you must call Set if the variable is an object/reference and not a value.

TODO: give an example of the return type and returning

### logic structures

The logic structures are the backbone of nearly all VBA programs. There are a handful of times where you can just run through some commands with no branching logic, but in general, your program will need to make decisions based on some condition that it encounters. In order to make those decisions, you use the logic structures of VBA. There is really only a single logic structure in VBA, the If-Then structure, but VBA provides a handful of useful additions to the basic If-Then to make programming a little easier.

The main logic structures then are:

- If-Then
- If-ElseIf-Then
- Select-Case

The If-Then is the main building block that allows you to do something if a condition is true or do something else otherwise. The If-ElseIf-Then allows you to add additional conditions to check before defaulting to the Else. If any of the ElseIf statements evaluate True, then the branch will stop traversing the conditions. The Select-Case is an extension of the If-ElseIf-Then that always compares a given variable against different possible values.

TODO: add example of the different forms of If-Then

For logic evaluation, there are always a handful of ways to arrive at the same result. You are allowed to evaluate multiple conditions in a single If-Then statement by using And and Or. You can also “nest” different logic blocks inside of each other to create the same sort of logic. In this way, you can either

use If-ElseIf-Then or you can do an If-Then and stick a second If-Then in the Else clause. These will be equivalent. Sometimes one version looks better or makes more sense than the other.

### helpful logic functions

TODO: add an overview of the functions And/Or (is there a XOR?) along with the logic operators <,>,<>, = etc.

### the Select Case

The Select Case makes it possible to compare the value of a variable against multiple values without having to type the variable name every time. This is a purely syntactic feature that makes programming easier in certain cases. You can completely duplicate a Select-Case with an If-ElseIf-Then, but you may have to type more code.

TODO: add an example of a Select Case

Note that this section is fairly short. Logic structures are so prevalent in normal VBA code that should look for examples of these in the respective chapters instead of this section.

### loop structures

The loop structures are an integral part of VBA programming. You are pretty much guaranteed to use them immediately. In some cases, you are more likely to use loops than logic structures. The reason that loops are so critical is that they allow you to perform an action multiple times or across multiple objects. Given the nature of a spreadsheet (where you have a high multitude of cells) and the reasons for using VBA (you want to perform some action multiple times) you really can't avoid loops. Gaining an understanding and comfort with loops is critical to your skill with VBA.

There are several types of loops that work similarly but have different use cases. Those include:

- For Each - useful when you have a collection and want to do something for each object in that collection (this is the most common loop to use since you will nearly always have a collection of Ranges or some other object to iterate through)
- For - useful when you want to do something a specific number of times
- Do/While - run a loop until a condition is met which is useful when you do not know in advance how many times to run the loop and you don't have a finite collection

It is worth noting that all loops can be written as a Do/While loop, but you will nearly never do this. There are good reasons that the For Each and For loops exist.

It is also worth mentioning here that I typically try to avoid For loops whenever possible. I always prefer to use a For Each loop if it is appropriate for the application. This is not an approach that I used from the beginning but have begun to value the use of For Each loops. If you are coming from a programming language that does not value collection iteration, then you might avoid the For Each loop at first. I'd strongly recommend you learn to use the For Each and appreciate it. Your code will be much cleaner and easier to read with For Each loops instead of the alternatives. Especially when dealing with Ranges, it is tempting to iterate through them as a nested For loop. You should really avoid this.

TODO: add an example of a bad For loop

**For Each loop**

It is not traditional to start with the For Each instead of the For loop, but I personally use the For Each far more so I'll start there.

The For Each loop is used whenever you have an iterable collection. An iterable collection can come from either the Excel object model or your own code. In general, most of the Excel object model returns an iterable collection. This is especially true for Ranges.

TODO: add a list of iterable collections that can be used here

You are not required to put the variable name in the Next line. I recommend not including the variable unless you have tons of code in the loop and are nesting loops. Typically you will rename the variable and then get a compile time error because the variable names don't match. I've never found the variable name in the Next line to help much.

TODO: add an example of a For Each loop

**For loop**

Another style of loop that exists is the "bare" `For` loop. This is one of the simplest loops to understand and control. The idea is simple: iterate through a chunk of code a given number of times. The most common forms of the `For` loop work through a fixed number of iterations. One example is easy: if you want to output the numbers 1 through 10 into a column of cells, you can easily use a For loop to output the number. This is a bad example though since it can easily be done with normal Excel functions, but it is quite common to

do the equivlanet sort of task when writing a larger macro. In that sense, it is easy to forget how versatile the For loop can be when needing to do something some number of times.

Compared to a While loop (discussed below) there are a number of advantages to the For loop:

- Much easier to control the "exit" strategy and avoid infinite loops
- Can be wired up with constants that are intutiive and just work
- Can be extended to use variables instead of constants to provide more flexibiilty

When moving beyond the simple 1 to 10 For loop, there are a handful of options which can be pusehd into ever complex strategies:

- Use a variable for either the starting index, increment, or end point
- Use a negative step to go backwards through a list of numbers
- Use `Exit For` statements to control execution and kick out of the loop

On that last point is where you will see VBA is woefully underppowered compared to "modern" programming lanugages. VBA does not provide a simple command to **`continue`** a loop. There is a `Exit For` which can be used to kick out of the loop, but to **`continue`** you must create a `:LABEL` and use a `Goto LABEL` statement to jump there. There is nothign necessarily worng iwth this, but it is an approach that is annoying and prone to some mistakes. The biggest issue is accidentally moving the label or having soem other positon issue. THe annoyance is having to create a label and use a `Goto`. There is nothing inherently wrong with a `Goto` but they provide create power which means awful bugs later.

It is worht noting that the `For Each` loop is a simplification of a `For` loop for a number of instances. In most cases, you coudl create an index and then iterate through an object by index, storing a reference to the object being stored. Behidn the scenes, I believe this is how the majority of internal commands are hanlded. Depsite the 1:1 translation between the two loops, it is typically MUCH simpler to use a `For Each` loop if you just need access to the underlying object in the collection. I will nearly always create an index outside of the loop and use it alongside a `For Each` instead of creating a `For` loop and storing a reference to an object. It always seems vastly simpler to store an Integer than an object. Aside from the marginal advantage of not calling `Set`, there is a immediate payoff of a `For Each` loop that if you name the variables correctly, you can typically read exaclty what the code will do. This pays dividends for yourself and others later when reviewining the code.

It is also worth metnioning that there are a handful of instances where you are typically required to use a For loop even if you want to use the object being used. The standard example here is if you will eb modified the colleciton that you are iterating. In this case, you will rapidly create iteration issues trying to modify the collecion inside the loop using it. In some of those cases, you will get runtime errors, but in others

you will just get unintended consequences. The most common example of doing this is when you want to delete items from a colleciton while iterating through it. Let's say you need to check whether an item meets some critertia before deleting it. There are two ways to handle this:

- Use a For loop adn run throught eh collection BACKWARDS. The direction is crtiical because it means that at wrost, you are working at the end of the list which will nto affect future operations.
- Use a "dual loop" approach.

An example of item 1 is sjown below.

TODO: add example of backward loop

The dual loop approach is worth mentionign further since sometimes it can give you an elegant way out of a bind. The idea is that instead of modifying the colleciton while you iterate it, you store some amount of information outside of the collection and then use that to determine what to delete. This typically only works if the items being deleted exist independetly of the colleciton that holds them. This happens often enough with Excel that it is worht giving a concrete example: deleting Rows from a Worksheet. To handle this dual loop approach, there are two possible options:

- Use one loop to create a colleciton that stores the rows to be deleted and then iterate that collection in a new loop
- Build a larger Range to delete as you go and then use an Excel funciton to handle the actual deletion

The latter option is only technically a "dual" loop. Tehcnically Excel will use some sort of internal loop to actually delete the Range. You are only required to cleverly create the Range which allows this internal process to be kicked off.

TODO: add an example of the Collection approahc for deleting Ranges

TODO: add an example of the UNION-DELETE approach for dleeting ranges.

TODO: add some examples of For loops and how they might be used

**Do/While loop**

The final style of loop is the Do/While loop. Although it is mentioned last, it is ultimately the simplest type of loop that exists. The idea is: run until some condition is meeting. This loop matches very nicely when your looping strategy involves some condition. You simply put tha tocndifiton int he loop adn let it work. The downside to a Do?while loop comes down to the possibility of an infinte loop. This leads to the common problem of a macro that hangs Excel and requires intevention to shut down. Infinite loops are

technically easy to avoid, but it is far more common in practice to skip the steps that help avoid infintie loops.

It is worth mentioning at this point that all of the loop varieties can be recreated from teh other loop varieties. From this standpoint, there are slight advantages to one style over anotehr, but at the end of hte day, you simply write the loop that works for the task at hand.

For a Do/While loop, there are two possible ways of writing it. You can either do a `Do...While` or a `While...Loop`. THe main difference is whether or not the loop will execute before the condition is checked. There are instances where one style makes more sense over the other. Typically you can always use the `While...Loop` variety, but you may be required to type an intialization statement before the loop that is repeated within the loop.

Some common examples where a While loop make sense include:

- Iterate down through a column of cells until some condition is met (typically a blank or non-blank cell). This is quite helpful when it is difficult to create the `Range` that might be used for a `For Each` loop.
- Iterate thruogh the file system usign the `Dir` command to find files to open and process

The WHile loop tends ot make the most sense when you are not iterating through a fixed colleciton of objects bcause the `For Each` does a better job there. You also would avoid using it when you have a fixed nujmber of iterations to run where a `For` loop makes a lot more sense. That then leaves the instances where you want to loop through some action some nujmber of times, but you're not sure how many times untli you start going.

If you are particualr adventurous, you can make use of the `Exit Do` command to exit out of the loop mid iteration. This pairs nicle with a `While True` at the start of hte loop to ensure that nothign else will kick you out of hte loop. There are instances where this can be a simple way to loop, but you have to be absolutely certain your `Exit Do` command will be triggedered at some ppint or else you guarantee an infinite loop.

TODO: add an example of looping use `Dir`

TODO: add an example of a loop that works through a range using Offset

#### which loop and why

There are a handful of common reasons you might go for one loop instead of another. Worth knowing is that in general you can use any of the loops and get the same result. One approach is typically much easier

to program, understand, and maintain.

A couple of good things to remember:

If you are going to modify a collection in the course of iterating through it, you should not use a For Each loop. The For Each does not update the iterable collection if you modify it during a loop. This is particularly important if you are looping through a collection to identify items to delete from the collection. You should never do this in a For Each loop. When deleting, you should typically use a For loop and iterate through the collection in reverse order. This makes it easy to handle deleting items since you cannot get out of order. You can achieve the same result with a Do/While, but I won't cover that.

TODO: add an example of deleting via a For loop

If you need an index/incrementing variable alongside your loop, you are not required to use a For loop. You can always create a new variable and increment it yourself inside the loop. This is sometimes preferable to switching to a For loop solely to get the counter/index variable.

If you are using a Do/While loop, you should give serious consideration to adding a counter and breaking the loop if the counter gets too large. It happens far too often where I use a While loop and end up freezing Excel because the loop never terminates. You can sometimes break the code and get Excel to respond, but that does not always work. This is especially important if you are generating code that others will use since they may be less familiar with how to break out of an infinite loop.

You may need to break out of a loop. Unfortunately, VBA does not have the normal Break and Continue commands that you might be familiar with from another language. The only way to break out of a loop on the spot is to add a label use a Goto command unless you are able to break out of the Function/Sub completely using Exit. This always feels dirty to me so instead I will typically structure the loop with a Boolean that can detect whether the next iteration should continue. This works for Continue, but it is not a good solution for getting a Break. The only way to do this is via a Goto. Just do it.

## other control structures

### With command

THe `With` command allows you to place a given variable within "scope" and avoid repeatedly typing that variable's name for each required call. The `With` command exists solely to reduce the nujmber of times that a givne object or variable name is typed. You are never required to ues a With command to accomplish a goal, but it can be helpful to clarify or avoid having too long of a code block. Having said that, a With block can be incredibly confusing to read especially when mixed with the always in scope function calls

like `Range` or `Cells`. It is incredibly easy to avoid typing the required `.` to start a new line and accidentally refer to the globally scope object instead of your With scoped object. For this reason, I very rarely use the With command. When I do use it, I will tpyically only use it when I am workign with a nested object that might be several levles deep. Having said that, I mostly avoid the With block by creating a variable which holds the object in question adn using that instead. I have found that parsing a With block later can quickly become a confusing mess becuase of the difficulty of spotting the `.` which is critical.

If you read through some of the most common questons on teh interent about “why my VBA no work?” you will quickly find issues with With blocks accidentally calling a globally scoped command. I have never asked those questions on the internet, but I have definitely been bittne by teh same errors where a `.` is missed and the commang goes bonkers. It happens but is easily avoided by not using `With`.

#### GoTo staements

`GoTo` statements are used ot force execution to jump to a speciifc Label regardless of anythign else that the progrma is doing. A `GoTo` statement is requried for error handling but is otherwise frowned upon by programmers with expereince in other languages. The rpoblem is that a bad `GoTo` statement allows you to do much damage within a program because you can quickly corrupt your program state by jumping around. Also, other programming languages tend to include all fo the nice features that have replaced places where `GoTo` was prevously required. A good example of this is breaking out of a loop or skipping ot the next item in a loop. The latter is tpyically handled with a **`continue`** statement in other langugaes. In VBA, this statemnet does not exist and you are reuqired to use a `GoTo` if you want the functionality.

To make a GoTo statement work, you need ot have a Label that the GoTo points to. An example looks like this:

```
Sub GoToExample()
    'doing some stuff

    If someConditiojn Then
        GoTo EndOfCode
    Else
        ' do some other sutff
    End if

EndOfCode:

End Sub
```

The rule for labels is that they are reuqired to occur at the front of the lien (no indenting), they must be a single vairable name without sapces, and they must end with a colon.

You should go to reasonable lengths to avoid using GoTo statements for anything other than error handling. They are the root of a lot of problems as execution order is concerned.

### Error Handling

One final control structure that exists is related to error handling. It is an inevitable consequence that computer programs will eventually throw errors. There are a lot of techniques and good practice that can avoid errors, but sometimes you will be forced to deal with an error. The alternative to error handling is usually a pop up that informs the user that something went wrong. For an experienced user, they may be able to handle the `Debug` or `Continue` or `End` decision, but your typical user will assume that your code has failed catastrpohically. It's entirely possible that the error has no effect on your intended outcome, or that the error could be resolved if the user just hit `Contineu` but the take home message is that *if* something *has* to happen to respond to an error (or a possibly error), then you need error handling.

The elements of error handling are simple:

- Determine when to allow an error to be thrown
- Determine what happens with execution when an error occurs
- Determine where to go back to once the error state has been addressed

The first decision to make is whether or not to allow errors to interrupt execution. By default, the answer here is "yes", an error will interrupt execution. If you want to hadnle this differnetly or reset it back to default, there are a pair of commands that can be used:

- `On Error Resume Next`, ignore all future errors, just keep trucking
- `On Error Goto 0`, stop execution immediately at the next error

If you are savy about searching online for solutions to your problem, you will often see option 1 listed as the "go to" (or is it `GoTo`, ha!) solution for getting around an error. In the technical sense, yes, `On Error Resume NExt` will absolutely get you around an error. It will by definition ignore hte error and just keep going with execution. For the vast majority of workflows, this is an awful approach. Very often an error is indciating that somethign has gone awry from your expectations. If those exepctations were reasonable, then it is very liekly that future code will not work as intended. Therefor,e if you are getting an erorr, you

should give serious consideration to finding the soruce of it before oyu `Resume NExt` through it. Ignoring an error that hsould have been addressed nearly always causes more pain later.

THe other harsh approach to respond to an eror is to force execution to stop immediately. This prompts the user with the popup about how to proceed. THis prompt is helpful because it gives two options that may allow you to solve the problem. THe first is `Continue` which will attempt to run the line of code again that casue the issue. If the error stil persists, then you will simply get it again. No harm. However, it is also possible to change the state of Excel while the prompt is visible. T his means that if your code was relying on an `ActiveChart` and you did not select one; you will be able to select a chart before hitting `Continue`. This can be a quick way out of a problem if you are confident where the error occured. If you are programming onyl for yourself, this can also be a clean way aroudn dealing with watiing for user input using another `GoTo` approach down below. Having said that, allowing a user to deal with an error prompt is absolutelu awful in terms of usability.

The second way you can deal with these error prompts is by hitting `Debug`. This is likely the first respnse when an error occurs because you are very unlikely to know where the exact error occurs. Once you've seen it however, then you may be able to contue above. THe nice thing about debugging the error is that you get some powerful tools to try and solve the problem. For a full overview of debuggin, check out the other sectiojn (TODO: add link). The specific features that are nice for dealing with error sinclude:

- Locals window, which will provide an overiview of all the local variables and their current state
- Set next statement, which will allow you to skip over an error or rerun a line of code whose state may have changed between executions
- Immediate window, whcih will allow you to either run arbitrary commands or possinly output infomration about the program state.

All of htose tools combined should make it possible for you to determine the source of an error. Once you have determine teh soruce of an error, you can then set about resolving the rror, again using the debug tools. Once you have solved the problem, you should give serious ocnsideration to then adding that solutiojn to the code using proper error handling technqes. Again, it is absolutely awful to present the user with an erorr dialog adn epxect them to be able to ifgure it out. Even if you are the user, you will absolutely tire of dealign ith error prompts that cna be handled with proper handling.

If you want to addres an error, there are a couple of ways to handle that. They all rely on using the `On Error Goto LABEL` tehcnique. This allows the code executojn to jump to a speciif cplace in your code. That area in your code is then able to do a couple of helpfl things:

- Query the state of the `Err` object
- Attempt to addres the error and then kick code back to the previous spot

- Provide the user with proper feedback before killing execution
- Log the issue accordingly before failing or prompting the user

With all of these approaches, the idea is simple: redirect execution to a known spot when the error has occurred. Once you are in a known spot, you can then step through possible problems and possible solutions. If you want, you are then able to send execution back to another spot to advance. IF oyu cannot resolve the error (or determine what caused it), you can then end execution all the same. Ideally you end execution with a better message than the normal prompt.

TODO: give an exmaple of some error handling code

**avoiding errors** Although this section is about error handling, the best error handling is an approahc hta tmake is very difficult for an error to occur in teh first place. As you call into specific VBA and Excel Subs, you will gain a feel for which ones can cause problems. On the VBA side, there are a nujmber of specific calls that will lead to errors:

- Indexing into an array with a index that is not valid: `Sheets("SomeSheetThatIsMissing")`
- Attempting to use a property on an object that does not exist
- Sending invalid parameters to a function

All of those items above have the nice poprerty that you may be able to provide checks for when you will enter an error state. The upside of this pproach is that oyu can use an `If...Then` statement to check for an error causing state and then step around it. Before using `Range.Value`, you can check that `If Not Range Is Nothing`. `Nothing` is the default value for a reference type before it has been set to a poper reference. You are always going ot get an error if you attempt to use a `Nothign`. You can avoid a ton of errors being thrown by simply checking for Nothing and avoiding its use when it appears.

For a lot of arrays and other iterable objects, you have different approaches for checking inf someting is a valid index before accessing it. For a `Dictionary`, there is the `Exists` method. For `Worksheets` and other Excel arrays, you are always able ot iterate through all of the items to check for existing before then using the index. TODO: add example of iterating sheets. It is very rare for the performacne of VBA to be affected by these tpyes of checks. There ar einstances where it is not appropriate, but in general, these tehcniqeus work fine.

**Application.XXX functions** In some instances, it is possible to trade a runtime error for a return value that has a tpye of error. This occurs with the Application.XXX functiojns where XXX includes items in the list:

- Match

- TODO: any others?

THis can be benficial because when the functojn returns an error, you can then turn around and deal with it by checking `IsError`. If the functiojn throws an error instead, you are forced to use proper error handling to catch the error and attempt to resujme state.

**common VBA errors** TODO: add section about 1004

TODO: add infomration about ocmpile time errors vs. run time errors.

**common Excel errors** In additiojn to the VBA errors, there are also a number of Excel specific errors that happen often enough that they shoudl ba addressed. Some of those common examples include:

- Using `ActiveXXX` without have `XXX` selected. This is most common with `ActiveChart` where it is possible to not have a Chart selected. This is not possibly with ActiveWorkbook or ActiveSheet since one will always be active. TODO: what about ActiveCEll?
- Using `Selection` when the "wrong" thing is selected. It is quite common to `Set` some vriable equal to Selection. If the wrong thing is selected, you will get an error about `Type Mismatch`
- Attemptign to make a selection when it is not valid per the UI. THis is most often the case when you attempt to Select a cell when its Parent Worksheet is not selected.
- Attempting to build a Range across Worksheets using `Union`
- Attempting to iterate thorugh a Range of cells by checking `Range.Value` if the Range can contain errors. If this is possibl you will instead have ot check for errors first.
- Attempting to access or change the `AutoFilter` if is has not been enabled first

Tehre are also a ton of instances where some functiojn retunrs `Nothing` and you do not check for it. T his most commonly occurs with:

- `Range.Find` where nothing was found
- `Intersect` where the two Ranges do not overlap
- TODO: add some others?

As a ifnal note, it is owrht metnioning that the sign of a good programmer is one who has a feel for when errors can and cannot occur. You will begin to appreciate when it is needed to add error handling code versus when you know you will not need it. Too often as a beginner, you will be excluding error handling because you are unaware of what can go wrong. As you get better, you will start to exclude error handlign because you actually know that no erorrs can occur. Until you get good, the result may look teh same (no error handling code) but the result ot the user is prompts and halted execution in one case.

### where Excel and VBA meet

The previous sections focused on the aspects of VBA that exist independent on Excel. It is worth ending this chapter with a section that discusses the general theme of where VBA and Excel actually do intersect. The main thing to remember is that VBA provides the programming constructs and language, while Excel exposes an object model to VBA that can be programmed against.

A good rule of thumb is that anything you can do in Excel can be done via VBA. This is probably not an exaggeration. The Excel object model is incredibly detailed and provides access to every nook and cranny of an Excel spreadsheet. This gives you enormous power to manipulate a spreadsheet in whatever way you can imagine, but it also means that it is easy to be overwhelmed by sheer volume of commands and objects that exist. Fortunately, there are only a handful of common/useful objects to start with and within those objects there is a significant amount of overlap. For example, the Range and Chart both expose formatting related properties (e.g. Border colors) but the ways of editing those are the same on both objects.

TODO: is that true about the Borders being the same?

TODO: finish this section… not sure where it's going

### adding references to external code

This section will cover how to add References to other files and programming components. There are 2 main reasons for why you would need to do this:

- You want to access some code from an Excel file that you or someone else created
- You want to access code from an existing component on your computer

The latter reason on the list is the more common reason for adding a Reference. There are a handful of common references that are added if you want additional components that are not available by default. Of these, the most common include the Microsoft Scripting Runtime and the references to other Office programs. For example, if you want to create a Dictionary, you will need to reference the Scripting Runtime. In general, there are a number of references that are nearly guaranteed to exist on all Windows computers. Having said that, there are also a handful of references that are commonly made where the required file may not be available. This uncertainty about the files available on a system is the major downside of using these references.

TODO: add a list with other common references and what they might include

For the first item, there are times where you have created some code that would be useful to use somewhere else but that you don't want to copy. This can be common for using helper code that you know is included

in another file. The major drawback to this approach is that you are creating a permanent link between the file and the reference. This means that the one file will quit working if the reference ever moves or becomes unavailable. Despite this drawback, there are times where this can be convenient and the drawbacks less significant.

To add a reference is relatively simple. You simply go to Tools -> References. You can then check the boxes for any references that you would like to add. To add a reference to an existing Excel file, you will have to browse to the file and select it that way.

TODO: add some images of how to add a component

## debugging your VBA code

One of the most useful features of VBA and the VBE is the ability to debug your code simply and in place. It is easy to take for granted the power of the VBE debugger, but it is worth mentioning that it is a solid debugger. The debugger has a handful of specific uses related to debugging your code:

- Stepping through excecution and watching the movement of values into and out of variables
- Using the Immediate Window to execute arbitrary code or output the results of some value
- Setting teh next instruction to force VBA to jump to an arbitrary point in your code
- Viewing teh call stack ot see how you reached a given spot
- Breaknig at an arbitrary breakpoint or after an error was thrown

### entering the debugger

To enter the debugger, you need to eitehr set a breakpoint, hit Step Into, hit the Break key, or have an error thrown that prompts for debugging.  By default, you will not be using the debugger while your code is running. This si actually a good thing since debuggin code adds a large overhead which will kill performance. The most common approaches to entering the debugger are to set a breakpoint or via an error. This lines up with teh idea that you either want to debug a specific point in your code or that you want to be able to see what wnet wrong when an error is thrown.

When setting a breakpoint, tehre are a handful of reasons for choosing where to set one:

- Right before an important step so that oyu can see the before adn after state
- Inside of a control structure so that you can see whether or executiojn enters that structure. Sometimes there is infomration to be had when the code does *not* reach a breakpoint.

When breakpoints, you can technically disable them instead of removing them if you do nto wnat them to tirgger. I never use that feature.

If you are entering the debugger through an error, you simply hit `Debug` on the prompt. You will be starting on teh line that threw the error ready to execute it again.

The other ways to enter teh debugger are by hitting the CTRL+BREAK shortcut. If the VBA is at a stoppable point, this will cause an interrupt which gives the same prompt as the error prompt. From here, you can hit `Debug`.

The final approach is to use the Step Into button on the code ot run. TODO: is this true?

#### stepping through code

Once you have entered the debugger, tehre are a handful of ways to affect execution. They are:

- Run
- Step Into
- Step Over

TODO: add a picture of the toolbar icons

TODO: explain how to reach these comamdns along with the shortcuts

Run will tell teh debugger to just keep running until it hits another error or breakpoint. This is the same as normal execution.

Step Into and Step Over do the same thing with one difference. They both tell VBA to execute the current instruction adn then resume debugging after it. The difference is how they hadnle whether or not to enter a `Sub` or `Function`. If you have a written a Sub or Functojn of your own adn then call it, you ahve tow options while debugging. You can either enter that Sub and step through the commands in tehre. Or, you can treat that line with teh Sub as a single step which can be processed as a single instruction. If you do that, you will `Step Over` all of the intermediate execution and reusme deugging once code returns back to the level you started at. This is very important if you ahve a large number of nested Subs and Functiojns. The debugging steps allow you to decided how “deep” into the call stack you will go to pursue your deugggin. Soemtimes, you will know that a givne Sub works as intended and you do not want to step into it. Other times, you will reach a Sub being called and want to know exactly how it arrived at its output.

If you want to step through to a specific spot but cannot get there easily with the commands above, you can always just set a new breakpoint right there and hit `Run`. This will run until that line. You can also right click on a line nad do `Run until this point` and you will get the same effect. TODO: is that right?

**viewing the state of your code**

The whole point of debugging is generally to view the state of oyur code (or the Excel side of things) in process. The idea of viewing teh state menaes a couple of concrete things:

- What are the values of specific variables?
- What was the order of execution? Which control structures were processed and in what way?
- What happens if I do “this” instead of “that”?

Each of those is hit below:

**values of variables** Typically, the most important aspect of debugging is seeing whcih variables hold which values. The idea is taht if you can see waht the variables hold at runtime, you can cehck that against your expecatations and then gain insight into why your program is behaving the way it does. Other times, you want to see the values of things so that you can decide how to proceed from your current ppint. VBA provides a number of ways to check the value of a variable:

- Hover over teh variable and allow the VBE to see you the value
- Using the Locals window
- Using the Immediate window with `?` added to the start (TODO: is that the same as Debug.pRint?)
- Using the Watch window after creating a watch
- Running a command where you put the value into the spreadsheet

The VBE is fairly helpful when debgging comapred to other debuggers. It does about what you would expect. This means that you will get tooltips when you hover over variables. This works well for variables that hold a value and not an object. For an object, if you hover, you will get the `.Value` property of the object and not a drop down to explore. IN this regard, the debugger is inferior to a modern Visual Studio instance.

If you want to explore the properties of an object, or see a persistent value without hovering, you can use the Loacls or Watch widnow. They do the same thing: show the values of variables while also alllowing you to click down into Objects and their properties. The Locals windwo works by giving you a list of all the local variables automatically. T eh Watch window works by requirng you to provide the vairable name or caommnad that you watn to watch. I always start with teh Locals window since tpyically local variabel are what I want to see.

When reviewing the contents of an object, beware that VBA will not show you all of the properties of the object. In particular, it will not show you properties that are the result of a Function instead of a nomral

property. For a lot of Excel Object Model objects this is a key point. THere are a large nujmber of properties that you will need to add ot the Watch window or query directly with teh Immediate window to see their value. A common example: `Range.Address`.

TODO: add an example of using the Watch window

TO use the Immediate iwndow, you first need to enable it via View (TODO: add this for others). Once enabled, you can use the Immediate window as a palce to execute whatever code you want. It works by executing single lines at a time. IF you want the output of a command, use `?` at the start to print the result. You can use the Immeidate window whevnerm, including duirng normal development (i.e. even when code is not running).

TODO: add an example of using the Immedaite window

One particular thing that can be done (although not often) is that you can use the spreadsheet as a place to dump the results of oyur debugging. Sometimes, you will need to inspect some oibject and find that the VBE is just not tha thelpful. Maybe you have an array whose values you want to hceck. The simpple approach here is to dump that array to the spreadsheet using teh Immediate windwo (or actual code) and then set a breakppint to inspect it. This igves a nice back and forth between Excel and VBA that simply does not exist in other programming environments. ONce you see Excel as a huge playgournd to dump arrays, you will find all sorts of using for that while programming.

#### forcing execution

In addition to watching the executiojn fo a programm, you also ahve the ability to chagne the execution. T his si done by usign the `Set next command` TODO: name? while running. This is the “nuclear” option of debugging becasue it does exaclty what it says. It will tell VBA to execute *whatever* line you want next. This allows you to completely ruin your execution while also providing you the power to step to a given spot. It’s always the case when writing code that you end up on the wrong side of an If/Else while developing a loop. Sometimes, you just want to see what happens if you go down teh other branch. T his option allows you to test that alternative execution path without having to modify your code. You just tell the debugger to execute that bracnh next and things will work. Very often howveer, you can using this feature to accidetanlly skip over code where variables are decalred or Set and then you will ahve all sorts of errors because objects are set to `Nothing` instead of the values that are required.

Despite the pitfalls of moving executiojn arbitrarily, most people who know this feature exists are capable of using it appropraitely. They typically are not surpised when things break.

#### viewing the call stack

One final feature which is useful is to check the Call Stack. The Call Stack is a list of all the proceduring Subs or Functiojns that are “active” preceding the current command. It gives you a list of all the places that came before your current line of code. The Call Stack is invaluable when you have started debuggin following an erorr because oftentimes you will nto know how you reached a given spot. This is epsecially true if you are debugging code that is used in multiple places.

To see teh Call Stack do View->Call Stack. You can then double click on an item adn jump back to that spot. Note that the VBE will attempt to show you the vales of variables at that location whcih can be very helpful.

The Call Stack can be very helpful if you are using recursive code that calls itself. This code can be very hard to debug because oftentimes a breakpoint will trigger more than you want. IF you are waiting for an error on the 8th time thruogh a FUnctiojn, then you don’t want to skip the breakpoint 7 times. Instead, you can wait for the error, then use teh Call Stack to step back through teh previosu iteratiojns adn see what happeneded.

TODO: add a picture of the call stack

## summary of Selections

The selections chapter needs to focus on the ways to access Ranges from VBA. These should cover the various ways to mimic normal ways of selecting along with VBA special stuff.

#### ways to get a Range object

- calling `Range`
- calling `Cells`
- from an existing Range
  - Cells
  - Rows
  - Columns
  - SpecialCells
  - Offset
  - Resize
  - EntireRegion

    - End
- from a Name object
- using `Selection`
- using `ActiveCell`
- using `Union` and `Intersect`
- using `Find`
- using `UsedRange`
- using `Application.Index` (or is it only WorksheetFunctions?)
- using `CurrentArray`

### some common patterns combining these techniques

- the Offset-Intersect approach (move a block down and intersect with the original)
- the Offset-Resize pattern when you move to a cell and expand the selection based on something
- the Union-Delete approach to getting a Range to delete

### introduction to selections

Identifying and using a `Range` object in VBA is the most critical aspect of building usable macros and helpful code. This point can be missed since you always have access to `ActiveCell` or `Selection`, but you will quickly reach the limits of VBA if you only use those functions.

This chapter will focus on the myriad ways to access a `Range`. A `Range` represents any (and every) cell in a `Worksheet`. The power of the `Range` is that it can represent a single cell, a row, a column, all cells, or a discontinuous collection of any combination of those options. From the `Range` you can then have access to the core functions of Excel/VBA.

The motivation for finding a `Range` is simple: the cell is the core entity of a spreadsheet, and presumably you're using the spreadsheet for some reason. You can technically write VBA code that executes without ever touching the underlying spreadsheet – and this can be useful at times – but more likely, you are using Excel and VBA because your data or use case is in Excel. If you want to access and work with the data in an existing spreadsheet, you will do so using a `Range`. If you want to put new data into a spreadsheet, you will use a `Range` to do that. If you want to use the more advanced features of Excel (e.g. Charting, PivotTables, etc.) you will use a `Range` to tell Excel how to drive those features.

Simply put, you will not be writing useful and maintainable VBA code unless you've got a strong command of working with the `Range`. To that end, this chapter will describe the ways to get a `Range`.

When thinking of the `Range`, you should think in terms of strategies for navigating `Ranges` and the actual code to execute those strategies. In some cases, the strategy is as simple as using the right command, but, often, you are required to think a step or two in advance about how to get the `Range` you want based on the nature of the spreadsheet and the actual task to be completed. For example, you will handle a block of data that is largely blank cells (sparse) different than a fat chunk of data with no missing values (dense). For the latter, you can quickly navigate the block of data with `Range.End`; not true for the former.

When thinking of the different strategies, the major split is whether you are starting with a blank `Worksheet` or if you are working with data in an existing `Worksheet`. If the `Worksheet` is blank, the main task is managing the `Ranges` that you are creating to place data on the sheet. If the `Worksheet` is contains data that needs to be processed, the goal is to identify the parts of the data you need and understand their relationship to other parts of the `Worksheet`. Often, you will be combining both of these workflow (i.e. process data into a new form) and will require both ways of thinking, possibly interleaved throughout the same code.

When the term `Selection` is used here, it refers generically to getting a `Range` reference. That `Range` could actually be `Select`ed, but the goal is generally to avoid selecting cells. Instead, the reference is used directly to do some processing.

### strategies and methods for selections, existing Worksheet

When working with data in an existing `Worksheet`, the main goal is to find the section of the data that you actually want to process. This task can range from trivial to the bulk of the VBA code. A rough overview, starting with trivial is:

- Use the selection – `Selection`
- Use the ActiveCell – `ActiveCell` (see later for why these are different)
- Hard-code the address of a single cell – `Range("A1")` or `Cells(1,1)` (please don't use the latter)
- Name a cell and use that name directly – `Range("CellName")`
- Iterate through all cells – `Cells`, `UsedRange`
- While iterating through cells, use some logic to identify if a `Range` is the one you want:
  - Check the `Value` of the cell
  - Check if the cell has some property (e.g. `HasFormula`, `HasArray`, etc.) * Check the `Style` of the cell

- Take an existing `Range`, possibly all cells, and pare it down using:
  - Move from a known cell to a new spot – `Offset()`, `End()`
  - Take a subset of an existing `Range` – `Cells`, `Rows`, `Columns`, `Areas`
  - Take a an existing `Range` and change its size – `Resize()`
  - Take a super set of an existing `Range` – `EntireColumn`, `EntireRow`, `CurrentRegion`, `CurrentArray` * Allow Excel to filter the `Range` based on things it tracks (e.g. value, blank, hidden, etc.) – `SpecialCells()`
- Identify several `Ranges` and combine them – `Union()`
- Identify several `Ranges` and use only the common cells – `Intersect()`
- Pull the `Range` reference from some other object
- Name a cell and use that name indirectly – `Names("CellName")`
- Ask the user to select the `Range` to use
- Use a function to get a reference – `Application.Index`
- Search for the cell based on its function or value – `Find()`
- Process a formula to determine the `Range` it depends on

In addition to those "simple" techniques above, there are more advanced techniques available. Those advanced techniques all rely on some combination of the above options, along with additional logic to manipulate the `Worksheet`. A couple of combination techniques would include:

- Use the Offset-Intersect technique to get a block of data without its header
- Use the `AutoFilter` to filter a data set and then get the visible cells with `SpecialCells()`
- Use one of the techniques above to get a `Range` on one `Worksheet`; grab the corresponding `Range` on a another `Worksheet` to do some processing

#### common aspects to working with a Range

There are several common aspects of working with `Ranges`. The most important thing is to remember the difference between using the `Range` as a reference or as a `Value`. The problem comes because VBA will work really hard to allow your code to execute regardless of whether the Value/reference part is done correctly.

The difference is best explained with an example. In this example, you can see that when the reference is stored, you must use the `Set` command. If you want the `Value` of a `Range`, you can either use `Value` or rely on VBA calling it implicitly otherwise. If you attempt to assign the `Value` of a `Range` to a `Range` object, you will get an error. If you attempt to assign the `Value` of a `Range` to a `Variant` variable, it will work, but

the variable will only hold the `Value`. That is, you cannot make further calls from the `Range` object model. This should highlight the importance of declaring variables with the tightest scope on the variable type. If everything is a `Variant`, VBA will let you get away with a lot; sometimes that flexibility will bite you.

TODO: add an example here

#### some simple techniques for finding a Range

The simple selection techniques consist of:

- Use the ActiveCell – `ActiveCell` (see later for why these are different)
- Use the selection – `Selection`
- Hard-code the address of a single cell – `Range("A1")` or `Cells(1,1)` (please don't use the latter)
- Name a cell and use that name directly – `Range("CellName")`

These are considered simple, but their simplicity means they are commonly used. These techniques can return a `Range` that represents either a single cell or multiple cells or a group of discontinuous cells. The one exception to this is the `ActiveCell`; it is always a single cell.

**Selection and ActiveCell** The `Selection` and `ActiveCell` commands both work based on what is currently going on with the active spreadsheet. In particular, they work on the current selection of the `ActiveSheet` in the `ActiveWorkbook`. For a normal workflow, the active sheet and workbook are the ones with focus (or that last had focus). When working through an involved workflow, you can control the `ActiveSheet` and `ActiveWorkbook`. In general, you should not use these commands in an involved workflow without a very good reason.

**Selection** `Selection` is a catch all object that refers to anything that is selected. If the current selection is a group of cells, then you get a `Range`. If instead the selection is a Chart, Shape, button, or some other non-`Range`, then you will get an error if you assume that it has type `Range`. When working with the `Selection`, it is always good to assign a new `Range` variable equal to the `Selection`. This ensures that you get Intellisense for commands and also ensures that VBA will throw an error if the `Selection` is something other than a `Range`.

**ActiveCell** The `ActiveCell` always refers to a single cell. If the current `Selection` is a single cell, then these will refer to the same `Range`. If the current `Selection` is a multi-cell `Range`, then the `ActiveCell`

is the cell that currently has focus. When normally editing cells, you have some control over which cell in a multi-cell `Range` is active. This can be changed by hitting `CTRL+.`, `SHIFT+Enter`. This functionality in Excel is what allows an array formula to be applied to a larger range. You select a multi cell `Range` and then enter the formula with `CTRL+SHIFT+Enter`. This in turn will apply the formula to all cells.

TODO: what happens when the `Selection` is not a `Range`? Does this still work?

#### Hard-code a cell reference

The second most common way of getting access to a `Range` is to simply give Excel the address of the `Range` to work with. This is a convenient way of working with `Ranges` because it can be easily checked against normal Excel formulas and addresses. The common ways of doing this are using the `Range` and `Cells` functions with the appropriate parameters.

When working with these functions, it is possible to use them "bare" or unqualified. That is, you can just type `Range()` or `Cells()` and it will work. Specifically, it will work on the `ActiveSheet` of the `ActiveWorkbook`. This can lead to some difficulties when working with multiple `Worksheets` or `Workbooks`. If you are working across contexts (`Worksheets` or `Workbooks`), you should generally qualify your reference to the widest context required. This is done by calling the appropriate function on the appropriate object/context. If you have multiple `Worksheets`, you would call `Worksheet.Range()` or specifically `Sheets("SheetName").Range()` in order to access a `Range` on that specific `Worksheet`. If you are working with multiple `Workbooks`, you still only need a reference to the `Worksheet`, but you will have to go through the correct `Workbook` first. This looks like: `Workbooks(1).Worksheets(1).Range`. If you've previously stored a reference to a `Worksheet`, you do not have to use the `Workbook` also; it is very common when working across `Workbooks` to store a `Worksheet` reference as you go (for this reason).

This caveat about qualifying a reference brings up an important point: a `Range` can only refer to cells that are on the same `Worksheet`. You are not allowed to create a `Range` across multiple `Worksheets`. (TODO: what happens if you try this?). If you want to work with `Ranges` on multiple `Worksheets`, you will need to iterate through the `Worksheets`.

**`Range()`** The `Range()` function is the powerhouse of cell referencing. It works hard to take whatever you give it and return a valid cell reference. It can process the same commands as the address bar in Excel. That is, it will parse:

- a cell reference (`A1`)

- a multi-cell reference (`A1:B5`)
- a discontinuous reference using a union (`(A1, B1, C1)`)
- a discontinuous reference using an intersect (`(A:A 1:1)`) – Note this will return the cell `A1` which is at the intersection of the two given references. Also note that this way of referencing cells is incredibly rare (I've never used it in a real application).
- a named range (`some_named_range`)
- any application of the multi cell references with named ranges

TODO: can the Range handle a function in it?

Alongside that power of the `Range()`, you can also use it to refer to a group of cells using the corners of the `Range`. This can be used to either return a group of cells in the same row/column, or it can be used to grab a block of data. You are free to give the cells in whatever order you'd like (not required to be top left and bottom right).

This multi-cell version of the `Range()` function is quite powerful when you know or can determine the corners of the `Range` you want. In particular, this works well with the `End()` and `Offset()` functions to build `Ranges` from a single starting point.

If you thought the `Range()` couldn't get any better, it has one last trick up its sleeve. It can also take parameters that are of the `Range` type when building a multi cell `Range`. This is quite powerful because it means you can use any of the techniques to find a `Range` and then get a block of data by feeding them to the `Range()` function. This saves the hassle of calling `Range(someRange.Address, someOtherRange.Address)` just to build the block.

There is one approach to using the `Range` function that is effective but can be a bad crutch. It involves building a `String` to feed to the `Range()` function. This usually looks like `Range("A"& Cells(1,1).Column)` or something similar. There are legitimate cases where this is a quick and easy way out of a problem. It generally involves knowing that you want a cell from a specific row or column while also knowing the other piece (column or row) from an existing cell. You can quickly combine the two to get your reference. There is nothing wrong with building a `String` here, but it might be a sign that there was a better way to get the reference from the start. It can be helpful when working with far to the right columns that are not easily thought of as a number; what column is `AB6` again?

When considering whether and how to use the `Range()` function, the main things to consider are:

- How stable does this code need to be?
- How likely am I to change the address of the cell I want?
- Will a given cell always be in the same place?

- Will a given name always exist?

This questions are pointing to some of tbe downfalls of `Range()`. The biggest downfall is that if you are going to use `Range("A1")` to refer to cell `A1`, your VBA code will not work if that cell moves for some reason. Furthermore, it can be a real pain to identify when code is failing because of a bad cell reference. I've had it happen countless times now where I hard-code a cell reference, use that in VBA, and then break things completely by adding a row or column somewhere. This is akin to using `VLOOKUP` and inserting a column in the middle of the lookup range; your code will not know or adjust to the new reference. Even worse, depending on what your code does, it's entirely likely that it will run just fine with the mistake. This is the most pernicious type of error to debug in a complicated program.

The upside of this dilemma is that you can quickly remedy the situation by using a named range to refer to the cell. If you name the cell on the Excel side of things, you get the benefit of Excel moving the reference around if the underlying cell moves. This is an incredibly powerful technique. More emphatically, this is the fastest way to "level up" your VBA if you are just getting started. Robust VBA generally relies on named ranges on the underlying spreadsheet. It takes very regular spreadsheets to get away hard-coded references. As a tip, the second time you manually increment 10+ `Range("A1")` calls because of a new row is the last time you want to do that.

A common technique for building macros quickly is to start with hard coded references and convert them to named ranges once the spreadsheet takes form. There is nothing wrong with naming ranges early and not needing them, but it can take more time than it's worth to name the ranges instead of hard-coding a reference. Again, this can burn you quickly if you have to manually change several of those references.

**Cells()** A convenient but less powerful version of `Range()` is the `Cells()` function. `Cells()` is much simpler since it only requires a row or column number for the reference. This can be useful to quickly grab a reference if you know the row or column number (or both). It's far more likely that you know the Excel reference you want – `A1` – than that you know the exact row and column number. It's the column number that is always a pain to determine. Some folks try to get around this by using the `Asc()- 65` approach to get the number for the letter and send that into `Cell1()`. Once you know about the `Range()` function, you'll never touch that madness again.

So, if the `Range()` function is typically more useful and powerful than `Cells()`, why would you ever use `Cells()`? Well, `Cells()` is the entry point for iterating through the cells in a multi-cell `Range`. This use of `Cells` will be covered later on, but it's mentioned here because it's incredibly powerful in that context. Specifically, if you have a `Range` already, you can use `Range.Cells()` to grab a cell within that `Range` at the specific spot. In this way, `Cells()` is actually useful because the indices are smaller and typically

correspond to the actual application at hand. Again, this is covered later.

TODO: add a link to the section where iteration is covered

#### some simple techniques for finding a multi-cell Range

The simple selection technique for working with multiple cells consist of:

- Iterate through all cells – `Cells`, `UsedRange`
- Building a range from the corners – `Range()`

The previous section identified the simplest techniques for obtaining a reference to a `Range`. Those techniques touched on single and multi-cell `Ranges`. There are a couple of additional techniques for obtaining a multi-cell `Range` that are used commonly.

The typical goal of these multi-cell calls is to take the reference and iterate through the cells. To iterate through the cells, there are two techniques, `For Each` and `For` loops. The former is vastly preferred to the latter in nearly all cases. I'll say that again, if you're iterating through cells, you should strongly prefer to use a `For Each` loop instead of a simple `For` loop. Those two examples look like:

TODO: add code samples for `For` and `For Each` loops

**`Cells`** The `Cells` call exists on several different objects. The easiest way to access it is via the bare, unqualified, reference – just type `Cells`. It applies to the `ActiveSheet` of the `ActiveWorkbook`. Typically, you should avoid iterating all `Cells` unless you know you will break out of the loop at some point. There are a lot of cells in a `Worksheet`, and your code will grind to a halt working through rows 10100 to 132000 doing a bunch of nothing on empty cells.

**`UsedRange`** `UsedRange` is available on a `Worksheet`. It also exists as a bare unqualified reference applying to the `ActiveSheet` of the `ActiveWorkbook`. The `UsedRange` is a slightly complicated function but its goal is to provide you a `Range` that provides a bounding box on all of the used cells in the current `Worksheet`. The complication of `UsedRange` comes when determining what is a "used" cell. Excel will consider a cell used if it has a non default property for its value or formatting. The formatting part of the definition can throw you for a loop because it's possible to change the formatting in a non-obvious way (e.g. it's impossible to spot the font size of an empty cell). There are several well-regarded folks who will advocate against the `UsedRange` in all cases. Their argument is that the `UsedRange` is too undependable because it can be thrown off too easily. In my experience, the `UsedRange` is a powerful way

to leverage Excel tracking the internal state of the spreadsheet. You can also avoid most of the issues with the `UsedRange` not matching expectations by taking care of the state of the spreadsheet. If a `Worksheet` was under your control, there's no reason to avoid the `UsedRange`. As a first tip, the `UsedRange` matches the scrollbars around the spreadsheet. If the scrollbars stop scrolling when you reach the "end of the spreadsheet", then the `UsedRange` is good to go. You can also do a quick test with `UsedRange.Address` or `UsedRange.CountLarge` to see what it refers to. Again, I think the arguments against the `UsedRange` are overly cautious, and it's a great command in a well managed spreadsheet.

TODO: is `UsedRange` available bare?

### finding a Range while iterating through a Range

One technique for working with Ranges is to start with one Range, iterate though it, can build a new Range based on some criteria. Alternatively, you may jsut act immdiatel on teh Range as you are iterating through it. This approach is dead simple and is used in abudnance htoruhgout good workflows. As long as there is some meaninful logic which can be applied to identify whether or not a subset of a Range is interesting, you can use this tehcnque. SOme common logical stepst aht are chcked:

- Check the `Value` of the cell
- Check if the cell has some property (e.g. `HasFormula`, `HasArray`, etc.)
- Check the `Style` of the cell

The idea is simple: check some property while iterating and act on it. THis is obvious once you have been programming for a bit, but sometimes oyu just need to be told htat this is an acceptable way of doing things. You do not always need to use `Find` to search for a cell that contians some value. You can always just iterate all the cells and see if a cell matches that value (or contains it with `InStr`).

TODO: find some code related to this?

### finding a Range by paring down (or up) an existing Range

One of the key ways to access a `Range` is to use an existing `Range` and modify it slightly. This might prompt the question: how do I get the first `Range` in order to use that? Well, check the previous section for the most common techniques. You can always start with `ActiveCell` if you just want to see these in action.

Using a `Range` to get the next `Range` really is the bread and butter of serious VBA development. It is a very common pattern to identify a single `Range` in a `Worksheet` that is critical to the rest of the spreadsheet

and use that as an “anchor” to access the rest of the cells. This is particularly common when the data is structured in some way that can be utilized.

When using these techniques, there are a couple of common strategies. They work by either paring down the current `Range`, moving the current `Range`, or using the current `Range` as the start of some expansion. Of course, since a `Range` can be used to access a `Range`, you will quickly find yourself chaining these functions together. That is the true power of these techniques. Very often you will use 2 or 3 to take a single cell, move to a new spot, resize to cover all of the data and then move over a column to do something.

- Take an existing `Range`, possibly all cells, and pare it down using:
  - Move from a known cell to a new spot – `Offset()`, `End()`
  - Take a subset of an existing `Range` – `Cells`, `Rows`, `Columns`, `Areas`
  - Take a an existing `Range` and change its size – `Resize()`
  - Take a super set of an existing `Range` – `EntireColumn`, `EntireRow`, `CurrentRegion`, `CurrentArray` * Allow Excel to filter the `Range` based on things it tracks (e.g. value, blank, hidden, etc.) – `SpecialCells()`

**move to a new spot, `Offset()` and `End()`** There are two simple ways to “move” from a given `Range` to a new `Range`, namely using `Offset()` and `End()`. Both of these take an existing `Range` and return a new one. `Offset()` will not modify the size of the current `Range`; it will just move it. `End()` will always return a single cell even if the starting `Range` was multi-cell.

**`Offset()`** `Offset(rows, columns)` works by moving the given `Range` over by the parameters given to it. The nice thing about `Offset()` is that the parameters can be negative to move backwards. There are a couple of simple use cases for `Offset()`:

- Work your way down or across a group of cells, by `Offsetting()` and setting a reference to the new cell
  - This is often paired with a `While` loop to work down a `Range`
  - This is also helpful when you are not exactly sure what `Range` you want (maybe it’s dependent on cell values) so you can’t simply assign the correct multi-cell `Range` at the start.
- Use an existing `Range` to get the starting point for a `Range` and move over to a neighbor cell or a blank area to do something
  - THis is common when using one cell’s value to determine the value of the next one (e.g. splitting on a delimiter)

  - THis is also common when adding formulas to a spreadsheet. Find the current data, `Offset()` over a column and apply the formula to all cells. * Also helpful when you "just know" that a desired `Range` is some distance away from the `Range` you've got. This is not the most elegant code at times (since it breaks easily), but it works reliably when you control the spreadsheet.

TODO: add a while loop example

TODO: add a formula example

**`End()`** `End(xlDirection)` is a powerful function for its specific use case. It replicates the functionality of the `CTRL+Arrow` keyboard shortcuts. It will move from the current `Range` as far as possible in a given direction so long as the cells are contiguous. Contiguous in this sense refers to the fact that the cells must not have a blank cell in between them. A blank cell is any cell that does not have a value *or* a formula. The formula part is important because you can use a formula to return `""` while still counting as a contiguous `Range`.

`End()` takes a parameter which is the direction to travel in. You can go all 4 directions, up/down and left/right.

`End()` will always return a single cell as the reference. This often means that `End()` is used alongside a `Range(Range, Range)` to get a multi-cell `Range` that spans from the start cell to the end cell. This is so common of a pattern, that I typically add a UDF that handles this logic directly.

TODO: add the function that is used `RangeEnd`

There are a few common patterns when working with `End()`:

- Use a `Range` that you know is at the top of a block of data and use `End(xlDown)` to get to the bottom of the column.
  - This can be combined with `Range(Range, Range)` to get the full multi-cell `Range` to work through
  - THis technique is very powerful when redefining the `Ranges` of a chart to include all of the cells (this can also be used for formulas too).
- If you know your data has blanks, you can use `End()` to jump to the next non-blank cell. * This is helpful if you are trying to fill in blank cells (TODO: add the Waterfall fill here)

**RangeEnd.md**

```
Public Function RangeEnd(ByVal rangeBegin As Range, ByVal firstDirection As
    XlDirection, Optional ByVal secondDirection As XlDirection = -1) As Range
```

```
2
3       If secondDirection = -1 Then
4           Set RangeEnd = Range(rangeBegin, rangeBegin.End(firstDirection))
5       Else
6           Set RangeEnd = Range(rangeBegin, rangeBegin.End(firstDirection).End(
                secondDirection))
7       End If
8   End Function
```

**RangeEnd_Boundary.md**

```
1   Public Function RangeEnd_Boundary(ByVal rangeBegin As Range, ByVal
        firstDirection As XlDirection, Optional ByVal secondDirection As
        XlDirection = -1) As Range
2
3       If secondDirection = -1 Then
4           Set RangeEnd_Boundary = Intersect(Range(rangeBegin, rangeBegin.End(
                firstDirection)), rangeBegin.CurrentRegion)
5       Else
6           Set RangeEnd_Boundary = Intersect(Range(rangeBegin, rangeBegin.End(
                firstDirection).End(secondDirection)), rangeBegin.CurrentRegion)
7       End If
8   End Function
```

**Take a subset of an existing `Range` – `Cells`, `Rows`, `Columns`, `Areas`** The subset functions work by providing you with a Range that is created from another Range based on some condition. They can be quite useful for building a workflow that makes it very explicit how you are trying ot iterate through a Range or waht you are searching for. The idea is that you know your starting Range contains some pieces that you would like to tierate through. The grouping goes from smallest unit to largest:

- Cells will return a “flat” list of all cells with in the Range. No grouping left.
- Rows and Colujmns will each return a new iterable object built of the previous Range sliced into its Rows or Columns. If call them in order, it will look the same as iterating through Cells except that the order may be difference (TODO: how does this work?). Be sure that if oyu want to yuse htese, avoid the properties with the “s”. If you call Row ro Column, you will just get a number instead of a group of Ranges
- Areas will return a group of cells that may contain groups of Rows or Columns or just individual Cells.

Areas are commonly built by users using `CTRL` to select multiple things or by VBA which uses `Union` to build Ranges.

TODO: add some specific code related to Columns and Rows… that code is quite useful as a replacement to Cells(i,j)

TODO: give an example of using Areas

**Take a an existing Range and change its size – `Resize()`** `Resize()` is a straightforward function that does exactly what you expect. It takes a current `Range` and resizes it to contain the number of rows and columns specified. The most common uses of a `Resize()` are:

- You know where you want some output to start and its size, so you `Resize()` to get a `Range` that will hold all of the data.
- You know that some data starts at a given cell and its size, so you `Resize()` and call `Value` to get an array of that data.
- You would like to extend or change a formula based on some condition, so you `Resize` and apply the formula down the line

In general, these uses follow a pattern: you know what size you want the `Range` to be (or can compute the size) and `Resize` gives you the `Range` back. This is one of the least controversial of the `Range` methods. Enough said.

TODO: how does this handle negative numbers

TODO: how does this handle a multi-cell range, does it always pick top left?

**Take a super set of an existing Range – `EntireColumn`, `EntireRow`, `CurrentRegion`, `CurrentArray`** These "super set" functions work by taking a starting point and expanding it to include more cells. These will grow the `Range`. Of the four listed above, `CurrentArray` is the only one that requires some special case. That is, the current cell must be a part of an array formula. The others will always work. These functions are best thought of with their keyboard shortcut equivalents:

TODO: extract this table along with others and make a single big table somewhere

| shortcut | Range function |
|---|---|
| SHIFT + SPACE | `EntireRow()` |
| CTRL + SPACE | `EntireColumn()` |

| shortcut | Range function |
|---|---|
| CTRL + A | `CurrentRegion()` |
| CTRL + / | `CurrentArray()` |

`CurrentRegion` is really only as useful as the data on the spreadsheet. If you have a large block of data, it works well to get the entire region. If you have blanks in your data, it's a bit of an unknown to know in advance what `CurrentRegion` will give you. Typically, if you know you have a block of data, it can be a quick shortcut to using `End()` twice. In general, I avoid it.

`EntireRow` and `EntireColumn` are somewhat special because they can be used to make modifications to the rows and columns in Excel. In particular, they are needed if you want to insert a row/column, delete a row/column, change the row/column formatting, or change the height/width of the row/column. You can also use `Range("A:A")` or similar ot get a reference to the entire column, but it is much simpler to have a reference to a `Range` of a single cell and work out from there. Even better, if you have a multi-cell `Range`, the `Entire` functions will return the combination of all the rows or columns contained in the `Range`.

In addition to modifying the rows/columns of a `Worksheet`, the `Entire` functions also work very nicely with `Intersect()` to get group of cells that are in a specific row/column. The `Entire` functions are generally much nicer than trying to build the `Range` from address or any other technique.

TODO: is this true? Does it work for a multi-cell in this way?

**Allow Excel to filter the Range based on things it tracks (e.g. value, blank, hidden, etc.) – `SpecialCells()`** The final function in this round up is also the most powerful at times: `SpecialCells()`. This function works by taking a parameter how which "special" cells to return. Special is a bad name here, because the most common uses of `SpecialCells` are to grab cells that are formula, values, blanks, or visible. These are some of the more mundane properties of a cell. Name aside, `SpecialCells()` can really take your VBA to the next level with very little effort.

An example: if you have ever iterated through `UsedRange` or `Cells` with something that checks for `rng.Value = ""` then you could have saved a loop by using `SpecialCells(xlCellTypeBlanks)` instead. This will return a new `Range` that only contains the blank cells. There are similar special types for other things that commonly come up.

One particular application of `SpecialCells` is when working with the `AutoFilter` which will cause rows to be `Hidden`. You can get a `Range` that contains all of the visible rows which is the same as the rows

which satisfy the filter. If your data is well structured or can be filtered, this ends up being a great way to push the burden of filtering onto Excel instead of having all that logic in VBA.

You can also use `SpecialCells` to quickly return a list of those cells which have a value (or formula) if you have a large block of sparse data. Once you have all of those cells, you can `Intersect()` the `EntireColumn` (or row) with the header of the data. This allows you to move quickly through data without having ot build addresses or remember where specific things are. In general, this highlights an important strategy: if you can obtain `Ranges` with the areas that are critical, you cna quickly manipulate those `Ranges` to perform some action. You can spend less time building finding cells and `Ranges` once you know how to work and combine these functions.

TODO: add the table manipulation code here to give an example of that

TODO: consider adding an example of using SpecialCells with filtering

#### working with a Range via `Union` and `Intersect`

You can perform set operations on multiple Ranges using Union and Intersect. Like all set operations, they correspond to differnet sections of a Venn Diagram. The simpler example is using `Union` since it will always return a new valid Range if it was fed valid Ranges to start. It works by growing the Range into a new Range that includes all previous objects rerferenced.

Intersect is a different beast because it is possible for it to return `Nothing` if the givne Ranges do not actually intersect. This is actually a very useful property if you are trying ot confirm whether or not a given cell is within in another Rnage.

TODO: add a pciture of set operatiojns

Some common examples of where these functiojns come up:

- Intersect is used with Events and other usability tasks to determine if a givne or slected Cell is within a target Range
- Interect is very useful with Offset and Resize to grab a new Range that contains a subset of data of the orignal Range wihtout having to worry about creating a new Range that includes cells not previosuly indlucded. IN this sense, Intersect only alllows a Range to get smaller.
- Uniojn can be very helpful when building a larger group to change all of their properties at once. This is quite nice because Excel will “batch” the caluclations if oyu change the `Value` all at once. This sam etechnique can b used to build a Rnage to delete

TODO: add Uniojn-Dlete example

TODO: add Intersect exampe to remove headers

TODO: add Intersect technque for Events and Sleection changed

#### the kitchen sink of remaining Range ideas

- Pull the `Range` reference from some other object
- Name a cell and use that name indirectly – `Names("CellName")`
- Ask the user to select the `Range` to use
- Use a function to get a reference – `Application.Index`
- Search for the cell based on its function or value – `Find()`
- Process a formula to determine the `Range` it depends on

TODO: look into the Trace functions to see what they return

**Objects that will return a Range** One of the greatest consistencies throughout VBA and the Object Model is how various objects will return a new object or reference to a useful property. At times, this can save you a large chunk of time trying to recreate that access from scrath. The key then is knowing when these proeprties exist and how ot use them.

Below is a rough summary of objects that will give you access to a Range.

- TODO: create this list
- TODO: consider making this a cheat sheet or similar since it covers most of the sections in this chapter

In addition to objects that will return a Range, there are also objects which will not return a Range but should. These include:

- TODO: create the rest of this list
- Chart Series info related to the Name, Values, and XValues. You are required to work through the `=SERIES` formula instead

**Using `Names().RefersToRange`** There are two ways to work with named ranges. One of them is quite simple: `Range("SomeNamedRange")`. This works well in a couple of cases:

- You know the exact name you want to use or can prompt the user for it
- You are using the `Range` call on an object that has proper scope.

For the latter point, the default named ranges have `Workbook` scope and the `Range` call works across the board. This becomes more of an issue when you are using the same name across multiple Worksheets with a Worksheet level scope. You can still access the named ranged, but now your call to `Range`, needs to be `Worksheet.Range` from the correctly scoped Worksheet.

The former ppint about needing to know the name is more often the problem. Sometimes you want to help someone use a named range, but you simply do not know what they are named. One trivial example is creating an addin that outputs all of the named rnages in the Workbook. You cannot iterate them through `Range` because you want to know what they are!

When you are in a posiiton where you want to use the named ranges but do not know or want to use the actual names, you can go directly thrgouh the `Names` object. There are two ways to do this:

- Iterate the `Names` with no knoweldge of them
- Use an index, i.e. the `Name` and call into `Names(index)`

Once you have access to a valid `Name`, you can then access the `RefersToRange` which will return a Range that cna be used. There are few instances where this is ever going to be better if you already have the name. The one exception to this is if you are wanting to change some of the metadata associated with the Name. This mainly includes the comment on the name since there is not much else. another otpion is that you can copy the named Range as a new range with a slightly different name. I have done this before to process all of the named ranges into some new named Rnage based on a fomrula which included the previous one. This cna be a critical step to improving the performance of arrya formulas that previosuly pointed to entire columns. The problem is that create the dynamically named ranges is an absolute pain without VBA.

TODO: add an example of the dynamic name creation

Once you are comfortable accessing named ranges, you may find that it is heplful to create them from time to time from VBA. This can be a helpful way of storing a complicated Rnage that your VBA created wihtout having to select the cells and hope you can type the name correctly.

**Using `Application.InputBox(, Type:=8)`** One very useful technique for obtianing a Range is to ask the user for one. This is one of the fastest ways to level up your VBA game because it provides the user control while also making your VBA look pretty slick with the Range picker. The other upside here is that the InputBox Range picker generally works better than the RedEdit version on a form. The odd thing here is two-fold:

- You have to know that InputBOx exists on teh Application alone. IF you use the other verison, then you cannot supply the Type

- YOu have to know that Type:=8 allows for a Range selection

ONce oyu have two those things down (because you read this book!) then you are able to ask teh user to pick a Range with ease. The other very nice thing about the InputBox approach is that you can supply a default address (not Range) and it will automatically be selected at the start. I have used this approach to get effect in bUTL to allow the VBA ot process the Selection (by default) or to allow the user to select somethign different. This is a very clean solution to sneivle defulats hwile also allowing the user to do somethign different once they read your initial prompt. It is also dead simple to upgrade your current `Set rng = Range()` to `Set rng = Applicaiton.InputBox("Select a cell", Type:=8)` instead. For utility tpye code, the difference in immense in terms of not having to hard code or guess Ranges. Or you can still guess them but provide the user a chance to change the guess.

TODO: move that Funciton here form bUTL GetOrSelect...

**Using `Application.Index`** The `=INDEX` formula is the most potent formula in Excel. Its couterpart in the VBA world is also powerful but less impressive compared to real programming. Having said that, the `Index` function works exactly as expected in VBA and is a very nice tool to have if you are comfortable using INDEX in a normal spreadsheet. The real power of Index is that you can use it to replace a lot of the common code where you iterate through a Range until you find given value. One potential upside of Index is that you can upgrade an Excel only methodology over to VBA with minial change to formulas. Once you have the work converted over, you can then set about addin teh details that VBA alone can provide.

TODO: does this work any different than Cells? is it really that useful?

**Using `Range.Find()`** I seldom use `Range.Find()`, but it can be a powerful addition when you know what you want to search for. My problem with .Find is that it is incredibly rare that I have some free text I am searching for and want to find using VBA. Generally speaking, Find becomes useful when you are processing a somewhat arbitrary Worksheet which may contain certain data you want. In my experience, I am far more likely to use an AutoFilter or something other than Find. Part of hte problem for me is that I have never had a problem usign some other method htan Find. I also generally find myself somewhat confused by the parameters and the general executon of Find. Typically, you will need to create a While loop to search for the next found items.

I also have the (probably unfair) view that Find is a crutch to not being able to use other methods to Find a given Range. I generally prefer to iterate through cells and check values. My mind is built around building a Rnage and processing it rather than attempting to find a Range and then process it. Your mileage may vary.

TODO: add an example of using Find correctly

**Pulling a Range from a Formula with string processing** One of the next level things to do with VBA is to start processing your Formulas to drive your VBA. There are a couple of places where this might be useful:

- You are dealing with a Chart Series Formula which must be parsed
- You want to Trace the precedent cells but don't want to deal with TracePrecedents
- You want to modify some part of the fomrula (e.g. take `A1` and surroudn it with an `ABS(A1)`)
- You want to make all of the cells in a speciifc formula a specific color (like a permentant version of hitting `F2`)

Whatever your motivation, it's good to rememebr that the formulas in a spreadsheet are generally the most important infomration aside from the actual data. IN some spreadsheets, the formulas are the only improtant part. If you want to extract and use htis informaiton, then it is helpful to be able to parse the formulas and identify the Rnages.

There are a couple of approaches to parsing Ranges from formulas, depending on what you need to do and what you start with:

- Your formulas contain only A1 style references without sheet names
- Your formulas may contain a sheet name too
- You want to extract non-range formula informatuon

For the first two, you can build relatively simple parsers whcih can extract the Range infomraiton which good accuracy. The key here is to understand exactly what your formuals look like. The worst case is having to build a full out formulas parser which is a non-trvial exercise. Hadnling all possible Excel syntaxs is a mess.

If you can settle for somethign less, then you have a couple of approaches at hand:

- Use a Regular Expression keyed in to Range options
- Use your knoweldge of the possible formulas to extract the relevant parts iwth string funcitons

TODO: add an example of some Regex which work here… expanding complexity

TODO: add an example of parsing out with Split and Left or something

**working with Ranges via advanced techniques**

- Use the Offset-Intersect technique to get a block of data without its header
- Use the `AutoFilter` to filter a data set and then get the visible cells with `SpecialCells()`
- Use one of the techniques above to get a `Range` on one `Worksheet`; grab the corresponding `Range` on a another `Worksheet` to do some processing

**Offset-Intersect** The Offset-Intersect is one of the most useful and simple approaches to creating a Range. The idea is that by using Intersect, you will avoid ever creating a Range that is bigger than some starting point. THis means that you will not be able ot accidetanlly add a blank or neighboring column to your Range. KNowing this, you can then take whatever steps are necessary to "remove" bad sections fromy our Range. This is most comonly used to remove a header row from teh top of a Range. If you are using Offset, the only rule is that you must make a valid move before calling INtersect. To remove a header, assuming you have a range which is a block of data iwth headers, simply do: `Set rng = Intersect(rng, rng.Offset(1))`. This gives you a new Range which has all of the cells of the first oen except for the first row.

TODO: add an image of how this works

Intersect used in this fashion is incredibly pwoerful. You can do all sorts of wacky steps to filter out a Range and then Intersect against the orignal Range to ensure that you have not accidetanlly stepped outside your starting box.

**AutoFilter and then SpecialCells** This approach is straight forward adn mirrors a cmomon operation in non VBA Excel. You use an AutoFilter to filter out specific cells adn tehn you can select ony the visible cells. In Excel, you can use `ALT+SEMICOLON` to only select visible cells. Often times, you will not need to actually do this since Excel tries to help you when dealing with Hidden rows and colujmns. Tpyically Excel will nto apply formatting to hidden cells and will also not fill a fomrula through them (assuming you used the Fill command).

In VBA, things are often more difficult because you are working with the underlying Range indepednet of whether or not the cells are hidden. To get around this, Excel provides the SpecialCells function which allows you to select a subset of cells based on some crtieria. wehn using the AutoFilter, the most common criteriojn to use is that of visibilty. You can call `Range.SpecialCells(xlCellTypeVisisble)` to obtain a new Range which only coantians visible cells.

If you haev ever written a loop which does a `If rng.Hidden = True Then...` then you will be grateful to know that Excle VBA provides this feature automtically. SpecialCells really is one of the most powerful ways to access Ranges in an intuitive fahsion that matches normal Excel.

**The Duplicated Range on another Sheet** If you are working with multiple sheets that are the same, similar, or related, you will often find yourself using inforamtiom about one sheet to build a Range on another or several others. THe probelm with Ranges however is that they are not allowed to span multiple Worksheets. This means that if you want to apply some actiojn ot each `A1:A10` Range on each Worksheet, you will need to do it iteratively. This can be a pain however if you built your Range usign code and not a direct address. To get aroudn this, you can use the `Range.Address()` function to obtain an address for the Range. The trick here is to use the `Address` function without parameters which will give you the local address without a Worksheet name. Y ou can then use that address on each of the other Worksheets, you access the given cells on that Worksheet.

This is anice way to replicate the functioanlity of Excel where you can sleect multiple Worksheets with CTRL or SHIFT and then apply some acitojn to all of them. The realyl nice thing about VBA however is that you can apply an action that is aware of the Worksheet on which it is acting. This si quite nice becasue the normal muti edit feature do the exact same steps to all spredahseets whereas you may want to use `End` or soemthign in your code.

#### Range via user input: InputBox

This section will focus on obtaining a Range from user input via the Application.InputBox

TODO: clean up this code

**GetInputOrSelection.md**

```
Public Function GetInputOrSelection(ByVal userPrompt As String) As Range

    Dim defaultString As String

    If TypeOf Selection Is Range Then
        defaultString = Selection.Address
    End If

    On Error GoTo ErrorNoSelection
```

```
    Set GetInputOrSelection = Application.InputBox(userPrompt, Type:=8,
        Default:=defaultString)

    Exit Function

ErrorNoSelection:
    Set GetInputOrSelection = Nothing

End Function
```

## overview of values and formulas

Values and formulas will focus on what to do with a Range after you have it. This typically falls into a couple of categories:

- Do some control logic based on the value of the Cell
- Apply some formatting to the cell
- Modify the formulas of the cell
- Manipulate the cell or its neighbors in order to produce a more useful result
- Transform the cells based on their content
- Do something specific to Excel with the `Range`: conditional formatting, data validation, comment, hyperlink etc.
- adding or deleting a `Range` or possibly just using one of the clear function

In addition to those basic tasks, also include:

- Working with Conditional Formatting
- Combining some more advanced topics like using the data in a `Range` to manipulate something about a `Chart`

TODO: run through bUTL and see what other category of things there are

### chapter 2 - 1, introduction to manipulations

This chapter will focus on the actual work of using `Ranges` for some purpose. This chapter is predicated on the previous one which focused on obtained a `Range`. When talking of “using” a `Range`, the goal usually takes one of the following forms:

- Work through a spreadsheet of data, processing it from one format into another. This can be to pull data out, do calculations on a subset of data, change the formatting, aggregate data, summarize data into a new spreadsheet, etc. The options here are really endless, but the main idea is that you have an existing spreadsheet of data to do something with.

The next category of work is to process some small amount of data in place, typically to clean up data or convert it to some other form. A lot of this type of work is providing some functionality that would be great to have in Excel by default. This work also includes a lot of very specific types of functions that only make sense with your data. In that sense, these types of actions can be the quickest hitters; they are specific to your task and easy to program.

Another category of work is to run through an existing worksheet and perform some amount of checking on it. These checks do not necessarily need to modify the spreadsheet, they can be checking for formula errors, bad values, etc.

Another type of work that can be done is to modify the spreadsheet to make it easier to do work or to manage a workflow. These types of things are often implemented as events, but they can just be stray macros as well. When modifying the spreadsheet, you are often showing/hiding columns or worksheets. You can also be sorting `Worksheets`. You might be moving some number of `Worksheets` over from a "template" and setting up a common work environment. You may also be d

As we progress down this list, things are becoming increasingly complicated. At some point, the work involved will progress from a couple of simple tasks to a much more involved workflow. It's generally the nature of a complicated workflow that it is simply doing a long string of simple tasks all at once. In that sense, if oyu can learn these techniques, you can start to become comfortable combining them in more complicated fashions.

### simple manipulations (one steppers)

This section will focus on simple manipulations. Simple manipulations generally take a two step process: identify a `Range` to work with and then do something to that `Range`. In a lot of cases, the `Range` contains multiple cells and may be iterated through a cell at a time to apply the action.

These simple manipulations truly are simple. It includes things like:

- Change the value of a cell
- Change the value of a group of cells
- Change the formula of a cell

- Change the formatting of a cell
- Clear the formatting or value from a cell
- Return some piece of information about a cell

TODO: deal with these later

Name a group of cells

Add a hyperlink to the current cell

TODO: add a couple examples of the simple manipulations

## slightly more complicated manipulations (the two steppers)

This section will on the so called “two steppers”. I call them that because these manipulations typically involve two commands after identifying a `Range`. the first command is usually a logic or loop, and the second command is the actual work ot be done. Two steppers are important because a large number of complicated tasks involve nesting and combining these two steps.

Some examples of two step manipulations includes;

- Run through a list of cells, if the text is numeric, convert to a number
- Run through a list of cells, if the cell is blank, fill with the value from above
- Run through some cells, check if the row is odd or even, and color the row from one of two colors
- Run through one list of cells, apply the formatting to the same cell in a different column

TODO: find some better examples for these as well

### strategy #1, do something if

This strategy really is the core of all advanced VBA development. It’s simple enough: “do something, if”. The endless possibilities come from the choices for “do something” and the things that could be checked in the “if”. There are a handful of common scenarios that are best covered by storing some utility code (e.g. convert to a number if numeric). Most of these two step solutions though are specific to the task at hand.

In this section, the goal is to show the general form of this strategy with a couple of examples.

TODO: add a couple examples of this

#### strategy #2, work through one Range and apply to another Range

This strategy comes up frequently when working through `Ranges` that are related somehow. The general idea is that you want to apply an action in one `Range` based on something about another `Range`. The simplest case of this is to move a value from `Range` to another. This simple case sometimes reduces to not much more than copying and pasting. Having said that, once you get past the simplest version of it, you will be doing something that copy and paste cannot handle.

TODO: add a couple examples of this

### things to change and check

This section will focus on the common properties that are checked and changed with these types of manipulations.

#### properties of the Range

The common properties of the `Range` to work with include:

- Value
- Text
- Formula
- Font
- Interior
- NumberFormat

TODO: add some examples of working with these

#### commonly used VBA functions

In addition to the properties of the `Range`, there are a handful of common VBA functions that come up when working with simple to moderate manipulations. These include:

- Split - split a string into an array based on a delimiter (the reverse of Join)
- Join - join an array into a string with a delimiter (this reverse of Split)
- Asc - determine the ASCII code for a character
- Chr - return a character for an ASCII code (the reverse of Asc)

- InStr - determine if a string is in another one (called Substring in other languages)
- Left, Mid, Right - grab parts of a string
- Trim - remove any whitespace from the start or end of a string
- Len - determine the length of a string
- UCase, LCase - used to force a string to upper or lower case
- UBound, LBound - determine the bounds of any array
- WorksheetFunction - get access to any Excel functions in VBA
- IsNumeric, IsEmpty - check if a number – TODO: add the others here
- CDbl, CLng, CBool, CDate - convert a value of one type to another – TODO: add any others
- Replace - replace one string in another
- Application.Index, Application.Match - these are the VBA versions of the Excel functions
- Application.Transpose - convert a 1D array from vertical to horizontal and back
- Is Nothing - check if a reference has been set
- TypeName - check the type of an object (useful if working with `Variant`)
- RGB - useful way to build colors from known red, green, and blue values
- Count - common way to get the size of a group, used often to resize an input/output or to check logic

TODO: search through bUTL for other common functions

### CategoricalColoring.md

```
Public Sub CategoricalColoring()



    '+Get User Input
    Dim targetRange As Range
    On Error GoTo errHandler
    Set targetRange = GetInputOrSelection("Select Range to Color")

    Dim coloredRange As Range
    Set coloredRange = GetInputOrSelection("Select Range with Colors")

    '+Do Magic
    Application.ScreenUpdating = False
    Dim targetCell As Range
    Dim foundRange As Variant

```

```
    For Each targetCell In targetRange
        foundRange = Application.Match(targetCell, coloredRange, 0)
        '+ Matches font style as well as interior color
        If IsNumeric(foundRange) Then
            targetCell.Font.FontStyle = coloredRange.Cells(foundRange).Font.
               FontStyle
            targetCell.Font.Color = coloredRange.Cells(foundRange).Font.Color
            '+Skip interior color if there is none
            If Not coloredRange.Cells(foundRange).Interior.ColorIndex =
               xlNone Then
                targetCell.Interior.Color = coloredRange.Cells(foundRange).
                   Interior.Color
            End If
        End If
    Next targetCell
    '+ If no fill, restore gridlines
    targetRange.Borders.LineStyle = xlNone
    Application.ScreenUpdating = True
    Exit Sub
errHandler:
    MsgBox "No Range Selected!"
    Application.ScreenUpdating = True
End Sub
```

**ColorForUnique.md**

```
Public Sub ColorForUnique()

    Dim dictKeysAndColors As New Scripting.Dictionary
    Dim dictColorsOnly As New Scripting.Dictionary

    Dim targetRange As Range

    On Error GoTo ColorForUnique_Error

    Set targetRange = GetInputOrSelection("Select column to color")
    Set targetRange = Intersect(targetRange, targetRange.Parent.UsedRange)

```

```
    'We can colorize the sorting column, or the entire row
    Dim shouldColorEntireRow As VbMsgBoxResult
    shouldColorEntireRow = MsgBox("Do you want to color the entire row?",
       vbYesNo)

    Application.ScreenUpdating = False

    Dim rowToColor As Range
    For Each rowToColor In targetRange.Rows

        'allow for a multi column key if intial range is multi-column
        'TODO: consider making this another prompt... might (?) want to color
            multi range based on single column key
        Dim keyString As String
        If rowToColor.Columns.Count > 1 Then
            keyString = Join(Application.Transpose(Application.Transpose(
               rowToColor.Value)), "||")
        Else
            keyString = rowToColor.Value
        End If

        'new value, need a color
        If Not dictKeysAndColors.Exists(keyString) Then
            Dim randomColor As Long
createNewColor:
            randomColor = RGB(Application.RandBetween(50, 255), _
                              Application.RandBetween(50, 255), Application.
                                 RandBetween(50, 255))
            If dictColorsOnly.Exists(randomColor) Then
                'ensure unique colors only
                GoTo createNewColor 'This is a sub-optimal way of performing
                   this error check and loop
            End If

            dictKeysAndColors.Add keyString, randomColor
        End If

        If shouldColorEntireRow = vbYes Then
```

```
            rowToColor.EntireRow.Interior.Color = dictKeysAndColors(keyString
                )
        Else
            rowToColor.Interior.Color = dictKeysAndColors(keyString)
        End If
    Next rowToColor

    Application.ScreenUpdating = True

    On Error GoTo 0
    Exit Sub

ColorForUnique_Error:
    MsgBox "Select a valid range or fewer than 65650 unique entries."

End Sub
```

**Colorize.md**

```
Public Sub Colorize()

    Dim targetRange As Range
    On Error GoTo errHandler
    Set targetRange = GetInputOrSelection("Select range to color")
    Dim lastRow As Long
    lastRow = targetRange.Rows.Count
    Dim interiorColor As Long
    interiorColor = RGB(200, 200, 200)

    Dim sameColorForLikeValues As VbMsgBoxResult
    sameColorForLikeValues = MsgBox("Do you want to keep duplicate values the
        same color?", vbYesNo)

    If sameColorForLikeValues = vbNo Then

        Dim i As Long
        For i = 1 To lastRow
            If i Mod 2 = 0 Then
```

```
                targetRange.Rows(i).Interior.Color = interiorColor
            Else: targetRange.Rows(i).Interior.ColorIndex = xlNone
            End If
        Next
    End If


    If sameColorForLikeValues = vbYes Then
        Dim flipFlag As Boolean
        For i = 2 To lastRow
            If targetRange.Cells(i, 1) <> targetRange.Cells(i - 1, 1) Then
               flipFlag = Not flipFlag
            If flipFlag Then
                targetRange.Rows(i).Interior.Color = interiorColor
            Else: targetRange.Rows(i).Interior.ColorIndex = xlNone
            End If
        Next
    End If
    Exit Sub
errHandler:
    MsgBox "No Range Selected!"
End Sub
```

**CombineCells.md**

```
Public Sub CombineCells()

    'collect all user data up front
    Dim inputRange As Range
    On Error GoTo errHandler
    Set inputRange = GetInputOrSelection("Select the range of cells to
       combine")

    Dim delimiter As String
    delimiter = Application.InputBox("Delimeter:")
    If delimiter = "" Or delimiter = "False" Then GoTo delimiterError

    Dim outputRange As Range
```

```
    Set outputRange = GetInputOrSelection("Select the output range")

    'Check the size of input and adjust output
    Dim numberOfColumns As Long
    numberOfColumns = inputRange.Columns.Count

    Dim numberOfRows As Long
    numberOfRows = inputRange.Rows.Count

    outputRange = outputRange.Resize(numberOfRows, 1)

    'Read input rows into a single string
    Dim outputString As String
    Dim i As Long
    For i = 1 To numberOfRows
        outputString = vbNullString
        Dim j As Long
        For j = 1 To numberOfColumns
            outputString = outputString & delimiter & inputRange(i, j)
        Next
        'Get rid of the first character (delimiter)
        outputString = Right(outputString, Len(outputString) - 1)
        'Print it!
        outputRange(i, 1) = outputString
    Next
    Exit Sub
delimiterError:
    MsgBox "No Delmiter Selected!"
    Exit Sub
errHandler:
    MsgBox "No Range Selected!"
End Sub
```

**ConvertToNumber.md**

```
Public Sub ConvertToNumber()

    Dim targetCell As Range
```

```
    Dim targetSelection As Range

    Set targetSelection = Selection

    Application.ScreenUpdating = False
    Application.Calculation = xlCalculationManual

    For Each targetCell In Intersect(targetSelection, ActiveSheet.UsedRange)
        If Not IsEmpty(targetCell.Value) And IsNumeric(targetCell.Value) Then
            targetCell.Value = CDbl(targetCell.Value)
    Next targetCell

    Application.ScreenUpdating = True
    Application.Calculation = xlCalculationAutomatic

End Sub
```

**CopyTranspose.md**

```
Public Sub CopyTranspose()

    'If user cancels a range input, we need to handle it when it occurs
    On Error GoTo errCancel
    Dim selectedRange As Range

    Set selectedRange = GetInputOrSelection("Select your range")

    Dim outputRange As Range
    'Need to handle the error of selecting more than one cell
    Set outputRange = GetInputOrSelection("Select the output corner")

    Application.ScreenUpdating = False
    Application.EnableEvents = False
    Application.Calculation = xlCalculationManual

    Dim startingCornerCell As Range
    Set startingCornerCell = selectedRange.Cells(1, 1)

```

```
    Dim startingCellRow As Long
    startingCellRow = startingCornerCell.Row
    Dim startingCellColumn As Long
    startingCellColumn = startingCornerCell.Column

    Dim outputRow As Long
    Dim outputColumn As Long
    outputRow = outputRange.Row
    outputColumn = outputRange.Column

    Dim targetCell As Range

    'We check for the intersection to ensure we don't overwrite any of the
       original data
    'There's probably a better way to do this than For Each
    For Each targetCell In selectedRange
        If Not Intersect(selectedRange, Cells(outputRow + targetCell.Column -
             startingCellColumn, outputColumn + targetCell.Row -
            startingCellRow)) Is Nothing Then
            MsgBox "Your destination intersects with your data"
            Exit Sub
        End If
    Next targetCell

    For Each targetCell In selectedRange
        ActiveSheet.Cells(outputRow + targetCell.Column - startingCellColumn,
             outputColumn + targetCell.Row - startingCellRow).Formula =
            targetCell.Formula
    Next targetCell

errCancel:
    Application.ScreenUpdating = True
    Application.EnableEvents = True
    Application.Calculation = xlCalculationAutomatic
    Application.Calculate
End Sub
```

### CreateConditionalsForFormatting.md

```
Public Sub CreateConditionalsForFormatting()

    On Error GoTo errHandler
    Dim inputRange As Range
    Set inputRange = GetInputOrSelection("Select the range of cells to
       convert")
    'add these in as powers of 3, starting at 1 = 10^0
    Const ARRAY_MARKERS As String = " ,k,M,B,T,Q"
    Dim arrMarkers As Variant
    arrMarkers = Split(ARRAY_MARKERS, ",")

    Dim i As Long
    For i = UBound(arrMarkers) To 0 Step -1

        With inputRange.FormatConditions.Add(xlCellValue, xlGreaterEqual, 10
           ^ (3 * i))
            .NumberFormat = "0.0" & Application.WorksheetFunction.Rept(",", i
               ) & " "" " & arrMarkers(i) & """"
        End With

    Next
    Exit Sub
errHandler:
    MsgBox "No Range Selected!"
End Sub
```

### ExtendArrayFormulaDown.md

```
Public Sub ExtendArrayFormulaDown()

    Dim startingRange As Range
    Dim targetArea As Range


    Application.ScreenUpdating = False
```

```
8
9       Set startingRange = Selection
10
11      For Each targetArea In startingRange.Areas
12
13          Dim targetCell As Range
14          For Each targetCell In targetArea.Cells
15
16              If targetCell.HasArray Then
17
18                  Dim formulaString As String
19                  formulaString = targetCell.FormulaArray
20
21                  Dim startOfArray As Range
22                  Dim endOfArray As Range
23
24                  Set startOfArray = targetCell.CurrentArray.Cells(1, 1)
25                  Set endOfArray = startOfArray.Offset(0, -1).End(xlDown).
                        Offset(0, 1)
26
27                  targetCell.CurrentArray.Formula = vbNullString
28
29                  Range(startOfArray, endOfArray).FormulaArray = formulaString
30
31              End If
32
33          Next targetCell
34      Next targetArea
35
36
37      'Find the range of the new array formula
38      'Save current formula and clear it out
39      'Apply the formula to the new range
40      Application.ScreenUpdating = True
41  End Sub
```

**MakeHyperlinks.md**

```
Public Sub MakeHyperlinks()

    '+Changed to inputbox
    On Error GoTo errHandler
    Dim targetRange As Range
    Set targetRange = GetInputOrSelection("Select the range of cells to
       convert to hyperlink")

    'TODO: choose a better variable name
    Dim targetCell As Range
    For Each targetCell In targetRange
        ActiveSheet.Hyperlinks.Add Anchor:=targetCell, Address:=targetCell
    Next targetCell
    Exit Sub
errHandler:
    MsgBox "No Range Selected!"
End Sub
```

**OutputColors.md**

```
Public Sub OutputColors()

    Const MINIMUM_INTEGER As Long = 1
    Const MAXIMUM_INTEGER As Long = 10
    Dim i As Long
    For i = MINIMUM_INTEGER To MAXIMUM_INTEGER
        ActiveCell.Offset(i).Interior.Color = Chart_GetColor(i)
    Next i

End Sub
```

**SelectedToValue.md**

```
Public Sub SelectedToValue()

```

```
    Dim targetRange As Range
    On Error GoTo errHandler
    Set targetRange = GetInputOrSelection("Select the formulas you'd like to
       convert to static values")

    Dim targetCell As Range
    Dim targetCellValue As String
    For Each targetCell In targetRange
        targetCellValue = targetCell.Value
        targetCell.Clear
        targetCell = targetCellValue
    Next targetCell
    Exit Sub
errHandler:
    MsgBox "No selection made!"
End Sub
```

**Selection_ColorWithHex.md**

```
Public Sub Selection_ColorWithHex()

    Dim targetCell As Range
    Dim targetRange As Range
    On Error GoTo errHandler
    Set targetRange = GetInputOrSelection("Select the range of cells to color
       ")
    For Each targetCell In targetRange
        targetCell.Interior.Color = RGB( _
                                    WorksheetFunction.Hex2Dec(Mid(targetCell.
                                       Value, 2, 2)), _
                                    WorksheetFunction.Hex2Dec(Mid(targetCell.
                                       Value, 4, 2)), _
                                    WorksheetFunction.Hex2Dec(Mid(targetCell.
                                       Value, 6, 2)))

    Next targetCell
    Exit Sub
errHandler:
```

```
    MsgBox "No selection made!"
End Sub
```

**SplitAndKeep.md**

```
Public Sub SplitAndKeep()

    On Error GoTo SplitAndKeep_Error

    Dim rangeToSplit As Range
    Set rangeToSplit = GetInputOrSelection("Select range to split")

    If rangeToSplit Is Nothing Then
        Exit Sub
    End If

    Dim delimiter As Variant
    delimiter = InputBox("What delimeter to split on?")
    'StrPtr is undocumented, perhaps add documentation or change function
    If StrPtr(delimiter) = 0 Then
        Exit Sub
    End If

    Dim itemToKeep As Variant
    'Perhaps inform user to input the sequence number of the item to keep
    itemToKeep = InputBox("Which item to keep? (This is 0-indexed)")

    If StrPtr(itemToKeep) = 0 Then
        Exit Sub
    End If

    Dim targetCell As Range
    For Each targetCell In Intersect(rangeToSplit, rangeToSplit.Parent.
       UsedRange)

        Dim delimitedCellParts As Variant
        delimitedCellParts = Split(targetCell, delimiter)

```

```
        If UBound(delimitedCellParts) >= itemToKeep Then
            targetCell.Value = delimitedCellParts(itemToKeep)
        End If

    Next targetCell

    On Error GoTo 0
    Exit Sub

SplitAndKeep_Error:
    MsgBox "Check that a valid Range is selected and that a number was
       entered for which item to keep."
End Sub
```

**SplitIntoColumns.md**

```
Public Sub SplitIntoColumns()

    Dim inputRange As Range

    Set inputRange = GetInputOrSelection("Select the range of cells to split"
       )

    Dim targetCell As Range

    Dim delimiter As String
    delimiter = Application.InputBox("What is the delimeter?", , ",",
       vbOKCancel)
    If delimiter = "" Or delimiter = "False" Then GoTo errHandler
    For Each targetCell In inputRange

        Dim targetCellParts As Variant
        targetCellParts = Split(targetCell, delimiter)

        Dim targetPart As Variant
        For Each targetPart In targetCellParts

            Set targetCell = targetCell.Offset(, 1)
```

```
            targetCell = targetPart

        Next targetPart

    Next targetCell
    Exit Sub
errHandler:
    MsgBox "No Delimiter Defined!"
End Sub
```

**SplitIntoRows.md**

```
Public Sub SplitIntoRows()

    Dim outputRange As Range

    Dim inputRange As Range
    Set inputRange = Selection

    Set outputRange = GetInputOrSelection("Select the output corner")

    Dim targetPart As Variant
    Dim offsetCounter As Long
    offsetCounter = 0
    Dim targetCell As Range

    For Each targetCell In inputRange.SpecialCells(xlCellTypeVisible)
        Dim targetParts As Variant
        targetParts = Split(targetCell, vbLf)

        For Each targetPart In targetParts
            outputRange.Offset(offsetCounter) = targetPart

            offsetCounter = offsetCounter + 1
        Next targetPart
    Next targetCell
End Sub
```

**TrimSelection.md**

```
Public Sub TrimSelection()

    Dim rangeToTrim As Range
    On Error GoTo errHandler
    Set rangeToTrim = GetInputOrSelection("Select the formulas you'd like to
       convert to static values")

    'disable calcs to speed up
    Application.ScreenUpdating = False
    Application.EnableEvents = False
    Application.Calculation = xlCalculationManual

    'force to only consider used range
    Set rangeToTrim = Intersect(rangeToTrim, rangeToTrim.Parent.UsedRange)

    Dim targetCell As Range
    For Each targetCell In rangeToTrim

        'only change if needed
        Dim temporaryTrimHolder As Variant
        temporaryTrimHolder = Trim(targetCell.Value)

        'added support for char 160
        'TODO add more characters to remove
        temporaryTrimHolder = Replace(temporaryTrimHolder, chr(160),
           vbNullString)

        If temporaryTrimHolder <> targetCell.Value Then targetCell.Value =
           temporaryTrimHolder

    Next targetCell

    Application.Calculation = xlCalculationAutomatic
    Application.EnableEvents = True
    Application.ScreenUpdating = True

    Exit Sub
```

```
35 errHandler:
36     MsgBox "No Delimiter Defined!"
37     Application.ScreenUpdating = False
38     Application.EnableEvents = False
39     Application.Calculation = xlCalculationManual
40 End Sub
```

## overview of charting

Charting will be a major chapter in this book since it's the focus of a lot of what I do. It also is a spot where a significant amount of time can be saved since the chart options are awful to mess with.

Some high level topics:

- Creating a chart
- Formatting a chart
- Manipulating the series on a chart
- Changing the layout of charts on a page (make grid code)
- Common patterns when working through charts (`ForEach` loops wherever possible)

### introduction to charting

Charting is the second most important aspect of automatic Excel behind manipulating `Ranges`. There is a bias when saying that because a lot of what I do after engineering calculations is chart the results. In particular, Excel can be used to great effect to chart time series of data. THe other reason charts are so amenable to VBA is that very often you are applying the same actions to the charts. In that sense, the VBA related ot charts is doing a lot of changing settings and formats so that the charts look the way you want. This ahs the immediate effect of making your charts look less like "they came from Excel" which si a common knock in some circles.

When working with `Charts`, there is a `Range` of difficulties depending on what you are trying to do. In some cases, working with an existing `chart` is much easier than creating a new one. In other instances, it can be much simpler to create a new chart rather, starting from a default, rather than change all the settings back. One other major difference between `Charts` nd `Ranges` is that working with charts is much more about knowing the object model than knowing how to program. The vast majority of your code

related to charts is simple iterating through objects to find the one property that you want to change. THis makes it easier to write chart VBA once you have the basics of `For Each` loops down. It also means that you need to spend some time getting comfortable with the object model.

There is one oddity related to Charts that is worth mentioning now. Charts can either be embedded as an object on a `Worksheet`, or they can be their own `Sheets`. I personally never use the latter case, but it is common enough that it needs to be on your mind when working with Charting code.

(I don't use the Chart as a Sheet model because I find that it is not necessary in terms of displaying data. In particular, you are at the mercy of your window size and cannot easily change the dimensions. Also, it complicates the VBA side of things to work in both formats all the time, so I just decided to always put my CHarts on Sheets. Your mileage may vary so I'll touch on both approaches in the code samples.)

#### a quick overview of the object model

- ChartObjects -> ChartObject - this derives from `Shape` and exists when the Chart is on a Worksheet
  - Chart
    * SeriesCollection -> Series
    * Axes -> Axis
    * ChartArea
    * PlotArea
- ActiveChart -> Chart - this works whether you have a Worksheet or Chart on a sheet
- Selection -> Variant - this one can be useful but is often not of the type that you want.

#### obtaining a reference to a Chart

When working with CHarts, the first task is typically to get a reference to an existing chart – unless you are creating a new chart. To obtain a reference to a chart, there are a handful of ways of doing it depending on what your spreadsheet contains and how it's structured.

THe main ways to do it are:

- Use the `ActiveChart` object
- Use the `Selection` object – this is highly depending on what is selected
- Use the `ChartObjects` object
  - If you know which chart you want, you can supply an index; this works great if there is only a single chart - `ChartObjects(1)`

  - If you want to do something to all charts, you can iterate this object
  - If you have named the chart (more on that later) you can supply the name as the index - `ChartObjects("SomeChart")`
- The `Workbook.Sheets` object if your charts are contained in their own sheets
  - Same as above, you can access via a numeric index, name, or iterate through all of them

**`ActiveChart`** `ActiveChart` is similar to the other `Active` objects in that it does about what you expect. The one difference is that the Chart actually has to be selected or have focus in order to be considered "active". This is similar but also different to something like `ActiveWorkbook` where having the workbook open makes it active.

Note that ActiveChart will work for a Chart that is contained on a Worksheet or also for one that is its own Sheet. If the latter case, then ActiveSheet and ActiveChart will refer to the same object. Side note: this technicality is why you will not get proper Intellisense when using ActiveSheet – that Sheet could technically be a Chart.

The nice thing about ActiveChart is that it gives you the Chart object which then gives you immediate access to the Chart related details you are like to want to change. The downside is that unless you have a single Chart that is already selected, ActiveChart has limited application when using VBA. Again, the goal is to avoid selecting objects in order to access them via VBA so ActiveChart has this limitation.

**`Selection`** The Selection object is probably the greatest catch all for an object. It literally holds anything, and this means that using the object requires knowing what is selected, or checking vigorously before using the object. Technically, you also let your code error out if the wrong object is selected, and this works well at times. This works well because oyu are unlikely to be using Selection in a complicated workflow because, again, you should not be selecting objects to access them. This means that Selection is really limited to one-off and helper code where you can more tightly dictate that this code only works if you select a Chart. You should still add some error handling, but sometimes that step is skipped.

Since the Selection can hold anything, it's important to know what could be Selected. Related to charts, the following can all live in the Selection:

- ChartObjects
- Chart
- ChartArea
- PlotArea
- Legend

- ChartTitle
- Series

If you are writing VBA to work on Charts, you can technically require the user to select the correct part of the chart and always use `Selection`. You will quickly grow tired of having to remember which part of the Chart to select in order ot make the code work. To avoid this scenario, it is helpful to remember the object model and know how to work your way around a Chart.

My approach has always been to convert the Selection to a Collection of ChartObjects. I can then always iterate that resulting Collection to process the Charts. If only a single Chart was selected, the code works all the same. The downside to this approach is that a Chart as a Sheet cannot live inside a ChartObject. This is a large part of why I always put Charts on a Worksheet.

Below is the helper function I use in order to convert a possibly Chart containing selection into a Collection of ChartObjects. It works for all objects except for the Axis related ones.

TODO: consider improving this code if it is included as a de facto reference

```
Public Function Chart_GetObjectsFromObject(ByVal inputObject As Object) As
    Variant

    Dim chartObjectCollection As New Collection

    'NOTE that this function does not work well with Axis objects.  Excel
       does not return the correct Parent for them.

    Dim targetObject As Variant
    Dim inputObjectType As String
    inputObjectType = TypeName(inputObject)

    Select Case inputObjectType

        Case "DrawingObjects"
            'this means that multiple charts are selected
            For Each targetObject In inputObject
                If TypeName(targetObject) = "ChartObject" Then
                    'add it to the set
                    chartObjectCollection.Add targetObject
                End If
            Next targetObject
```

```
21
22          Case "Worksheet"
23              For Each targetObject In inputObject.ChartObjects
24                  chartObjectCollection.Add targetObject
25              Next targetObject
26
27          Case "Chart"
28              chartObjectCollection.Add inputObject.Parent
29
30          Case "ChartArea", "PlotArea", "Legend", "ChartTitle"
31              'parent is the chart, parent of that is the chart targetObject
32              chartObjectCollection.Add inputObject.Parent.Parent
33
34          Case "Series"
35              'need to go up three levels
36              chartObjectCollection.Add inputObject.Parent.Parent.Parent
37
38          Case "Axis", "Gridlines", "AxisTitle"
39              'these are the oddly unsupported objects
40              MsgBox "Axis/gridline selection not supported.  This is an Excel
                    bug.  Select another element on the chart(s)."
41
42          Case Else
43              MsgBox "Select a part of the chart(s), except an axis."
44
45      End Select
46
47      Set Chart_GetObjectsFromObject = chartObjectCollection
48  End Function
```

**ChartObjects** If you are working on a Worksheet, then that Worksheet will have the ChartObjects object. This object is great because it contains all of the Charts in their own collection (separate from any other Shapes or buttons). This ChartObjects collection contains object of type ChartObject. The ChartObject derives from Shape which means it contains all of the properties related to on-sheet position and size.

A typical workflow is included below since it is a pattern that shows up all the time in VBA code related to charts. At a high level the steps are:

- Use ActiveSheet or a Worksheet reference to access the ChartObjects
- Iterate through each ChartObject, storing a reference to the underlying Chart
- You then setup sections to work through the parts of the Chart you want
  - Iterate through the SeriesCollection
  - Iterate through the Axes
  - Touch the other top level properties including ChartTile, Legend, etc.

This workflow is quite powerful because it can quickly be wrapped with a loop to go through all Worksheets and even possible all Workbooks. It's also powerful because you can be quite comfortable learning this pattern and then adding in the parts that you actually want ot change. The only downside is that it can be quite tedious to type out all the loops every time, but there's not a good way around that other than to use the clipboard.

Another approach to using ChartObjects is to not iterate through all of them but instead to select a single ChartObject and work with it. There are two ways to do this:

- Use an integer index for the Chart – this is quite easy to do if there are only a few charts
- Name the chart and use that name

When using either of these approaches, it is quite helpful to show the `Selection Pane` window in Excel. This pane will pop out and tell you the order and the names of all the objects on the sheet (this includes comments, shapes, and Charts). From this pane, you can rearrange the charts into the order you want or rename them.

Although `For Each` loops are generally preferred when working with Charts, sometimes you simply know that you want to change one chart and an index just lets you do that. If you are in the habit of using loops however, you can easily do that with the helper code included above which stick a single chart into a Collection.

**Workbook.Sheets to get Chart references** The final approach to obtaining a Chart reference is to use the Sheets object. Aside from ActiveChart, this is the only way to deal with Charts that are their own Sheet. Again, you cna either use an index or a Name. Here, the Name is easily changed on the Sheet tab so it's much more common to use a Name when doing this. The other approach is to iterate through all the Sheets and pick off the ones that are Charts.

There are two key points when working with Charts as Sheets:

- You must use the Workbook.Sheets object to access them and not Workbook.Worksheets. The latter object contains only those Worksheets that are not Charts. The former contains both Charts and

Worksheets.

- It's possible that your Sheet is not actually a Chart. You should check the type of the object is you are going to iterate through all Worksheets. Also be aware that some sheets can be hidden which might lead to unexpected results.

TODO: is there a Charts object on Workbook?

### common objects/properties for a Chart

This section will focus on the common formatting changes that can be made to a Chart. the next section focuses on creating a Chart from scratch if you want to see that. These common changes will be grouped by the type that they affect, but this is not meant to be an exhaustive list. Instead, this is a list that will cover the objects nad functions that are actually used in regular code. There will be several other things that you will need to check the reference for (or record a macro), but this listing will get you started with the regular things.

To organize this section, we will focus on the different parts of a Chart in turn along with how to access the things you need. This section is meant to be a one stop shop for working on the common parts of a Chart. This will cover:

- ChartObject
  - Top, Left, Height, Width - control the location of a chart
- Chart
  - ChartType
  - Access the other objects and controls whether some things exist
    - HasLegend
    - HasTitle
- Legend
- Series – accessed the the Chart.SeriesCollection
  - ChartType
- Axis – accessed through Chart.Axes
  - Display the axis
  - Change the text
  - Change the min/max scale including automatic values
  - Change the number format of the axis
  - Change the format and display of the Gridlines

- Point – accessed through a Series
  - Change display of individual points
  - Control the DataLabels (HasLabel and then DataLabel)
- Trendline

TODO: go through bUTL and find other commonly appearing things

#### common changes to the ChartObject

The ChartObject is the main container for a Chart that is on a Worksheet. The common changes then are related to the position and size of the Chart on the Worksheet. The common properties to change here are:

- Top
- Left
- Height
- Width
- Placement (controls the move with cells option)

All of these are of type Double which means you can use decimal calculations to determine the size or position. In Excel, the 0,0 point is at the upper left hand corner (upper left of cell A1) and the Top and Left increase going to the right and down. If you are familiar with 0,0 being the center of the XY plane, then Excel will be a tad unfamiliar. Once you get used to it, you will realize that there is not really a better way to arrange the coordinate system since the spreadsheet can extend to the right and down nearly infinitely.

TODO: are there Bottom and Right properties too?

TODO: add a comment about Points vs. inches here and the function to convert them

The most common application of changing these properties is to either standardize the size of several charts or to arrange the charts in a grid (which standardizes the size and then position).

That code is included below:

TODO: clean up this code to only the required parts

```
Public Sub Chart_GridOfCharts( _
    Optional columnCount As Long = 3, _
    Optional chartWidth As Double = 400, _
    Optional chartHeight As Double = 300, _
    Optional offsetVertical As Double = 80, _
    Optional offsetHorizontal As Double = 40, _
```

```
    Optional shouldFillDownFirst As Boolean = False, _
    Optional shouldZoomOnGrid As Boolean = False)

    Dim targetObject As ChartObject

    Dim targetSheet As Worksheet
    Set targetSheet = ActiveSheet

    Application.ScreenUpdating = False

    Dim countOfCharts As Long
    countOfCharts = 0

    For Each targetObject In targetSheet.ChartObjects
        Dim left As Double, top As Double

        If shouldFillDownFirst Then
            left = (countOfCharts \ columnCount) * chartWidth +
               offsetHorizontal
            top = (countOfCharts Mod columnCount) * chartHeight +
               offsetVertical
        Else
            left = (countOfCharts Mod columnCount) * chartWidth +
               offsetHorizontal
            top = (countOfCharts \ columnCount) * chartHeight +
               offsetVertical
        End If

        targetObject.top = top
        targetObject.left = left
        targetObject.Width = chartWidth
        targetObject.Height = chartHeight

        countOfCharts = countOfCharts + 1

    Next targetObject

    'loop through columns to find how far to zoom
    'Cells.Left property returns a variant in points
```

```
42      If shouldZoomOnGrid Then
43          Dim columnToZoomTo As Long
44          columnToZoomTo = 1
45          Do While targetSheet.Cells(1, columnToZoomTo).left < columnCount *
                chartWidth
46              columnToZoomTo = columnToZoomTo + 1
47          Loop
48
49          targetSheet.Range("A:A", targetSheet.Cells(1, columnToZoomTo - 1).
                EntireColumn).Select
50          ActiveWindow.Zoom = True
51          targetSheet.Range("A1").Select
52      End If
53
54      Application.ScreenUpdating = True
55
56  End Sub
```

**common properties of the Chart**

The Chart object is mostly a container for the other more useful properties of the Chart, but there are a couple of common properties that live at this top level. Those include:

- The HasXXX: HasTitle, HasLegend (TODO: any others?) - control the display of these things
- ChartType
- Delete
- Copy (TODO: this on ChartObject also?)

TODO: find more of these

In addition to those properties, the Chart object provides access to other useful things via the common accessors:

- SeriesCollection
- Axes
- Legend
- ChartTitle
- ChartArea

- PlotArea

TODO: is this list complete?

#### common properties of the Series

One of the two most used Chart objects is the Series (other is the Axis). The Series ends up being powerful because it provides access to the data of the Chart along with the major formatting choices since the Series is the prominent feature of a Chart.

The common things to go after for a series are:

- Data related
  - Name
  - XValues
  - Values
  - Formula
- Formatting related
  - Format
    - Line
  - MarkerSize
  - MarkerStyle
  - MarkerForegroundColor, MarkerBackgroundColor

Also, from a Series you can access the following other objects:

- Points
- Trendlines

#### common properties of the Axis

The Axis is the second most common object to work with (behind the Series). This is largely because the Axis controls or provides access to a lot of the formatting related aspects of the CHart. The Axis also controls the scale of the Axis and in that regard, is a critical part of making or editing a Chart.

The first part of the Axis is accessing the correct one. This is slightly tricky the first time because the Axes are stored in the Chart.Axes object. THe real trick is that this object is indexed by the xlAxisType (TODO: check that) which can be xlCategory (for the x-axis) or xlValue/xlValue2 (for the y-axis, left and right).

Once you have an Axis object, you can set to work changing the common properties:

- Scale related
  - MinimumScale/MaximumScale
  - MinimumScaleIsAuto/MaximumScaleIsAuto
- Formatting related (most of these are accessors to a different object)
  - GridLines (Major/minor and the HasXXX)
  - Ticks (TODO: that right?)
  - HasTitle and AxisTitle

**Chart_Axis_AutoX.md**

```
Public Sub Chart_Axis_AutoX()

    Dim targetObject As ChartObject
    For Each targetObject In Chart_GetObjectsFromObject(Selection)
        Dim targetChart As Chart
        Set targetChart = targetObject.Chart

        Dim xAxis As Axis
        Set xAxis = targetChart.Axes(xlCategory)
        xAxis.MaximumScaleIsAuto = True
        xAxis.MinimumScaleIsAuto = True
        xAxis.MajorUnitIsAuto = True
        xAxis.MinorUnitIsAuto = True

    Next targetObject

End Sub
```

**Chart_Axis_AutoY.md**

```
Public Sub Chart_Axis_AutoY()

    Dim targetObject As ChartObject
    For Each targetObject In Chart_GetObjectsFromObject(Selection)
        Dim targetChart As Chart
        Set targetChart = targetObject.Chart

        Dim yAxis As Axis
```

```
        Set yAxis = targetChart.Axes(xlValue)
        yAxis.MaximumScaleIsAuto = True
        yAxis.MinimumScaleIsAuto = True
        yAxis.MajorUnitIsAuto = True
        yAxis.MinorUnitIsAuto = True

    Next targetObject

End Sub
```

**Chart_AxisTitleIsSeriesTitle.md**

```
Public Sub Chart_AxisTitleIsSeriesTitle()

    Dim targetObject As ChartObject
    Dim targetChart As Chart
    For Each targetObject In Chart_GetObjectsFromObject(Selection)
        Set targetChart = targetObject.Chart

        Dim butlSeries As bUTLChartSeries
        Dim targetSeries As series

        For Each targetSeries In targetChart.SeriesCollection
            Set butlSeries = New bUTLChartSeries
            butlSeries.UpdateFromChartSeries targetSeries

            targetChart.Axes(xlValue, targetSeries.AxisGroup).HasTitle = True
            targetChart.Axes(xlValue, targetSeries.AxisGroup).AxisTitle.Text
               = butlSeries.name

            '2015 11 11, adds the x-title assuming that the name is one cell
               above the data
            '2015 12 14, add a check to ensure that the XValue exists
            If Not butlSeries.XValues Is Nothing Then
                targetChart.Axes(xlCategory).HasTitle = True
                targetChart.Axes(xlCategory).AxisTitle.Text = butlSeries.
                   XValues.Cells(1, 1).Offset(-1).Value
            End If

        Next targetSeries
```

```
    Next targetObject
End Sub
```

**Chart_FitAxisToMaxAndMin.md**

```
Public Sub Chart_FitAxisToMaxAndMin(ByVal axisType As XlAxisType)

    Dim targetObject As ChartObject
    For Each targetObject In Chart_GetObjectsFromObject(Selection)
        '2015 11 09 moved first inside loop so that it works for multiple
           charts
        Dim isFirst As Boolean
        isFirst = True

        Dim targetChart As Chart
        Set targetChart = targetObject.Chart

        Dim targetSeries As series
        For Each targetSeries In targetChart.SeriesCollection

            Dim minSeriesValue As Double
            Dim maxSeriesValue As Double

            If axisType = xlCategory Then

                minSeriesValue = Application.Min(targetSeries.XValues)
                maxSeriesValue = Application.Max(targetSeries.XValues)

            ElseIf axisType = xlValue Then

                minSeriesValue = Application.Min(targetSeries.Values)
                maxSeriesValue = Application.Max(targetSeries.Values)

            End If

            Dim targetAxis As Axis
            Set targetAxis = targetChart.Axes(axisType)

            Dim isNewMax As Boolean, isNewMin As Boolean
            isNewMax = maxSeriesValue > targetAxis.MaximumScale
```

```
            isNewMin = minSeriesValue < targetAxis.MinimumScale

            If isFirst Or isNewMin Then targetAxis.MinimumScale =
               minSeriesValue
            If isFirst Or isNewMax Then targetAxis.MaximumScale =
               maxSeriesValue

            isFirst = False
        Next targetSeries
    Next targetObject

End Sub
```

**Chart_YAxisRangeWithAvgAndStdev.md**

```
Public Sub Chart_YAxisRangeWithAvgAndStdev()

    Dim numberOfStdDevs As Double

    numberOfStdDevs = CDbl(InputBox("How many standard deviations to include?
       "))

    Dim targetObject As ChartObject

    For Each targetObject In Chart_GetObjectsFromObject(Selection)

        Dim targetSeries As series
        Set targetSeries = targetObject.Chart.SeriesCollection(1)

        Dim avgSeriesValue As Double
        Dim stdSeriesValue As Double

        avgSeriesValue = WorksheetFunction.Average(targetSeries.Values)
        stdSeriesValue = WorksheetFunction.StDev(targetSeries.Values)

        targetObject.Chart.Axes(xlValue).MinimumScale = avgSeriesValue -
           stdSeriesValue * numberOfStdDevs
        targetObject.Chart.Axes(xlValue).MaximumScale = avgSeriesValue +
           stdSeriesValue * numberOfStdDevs

```

```
23      Next
24
25  End Sub
```

**common properties of the Legend**

The Legend is a simple affair compared to the others. There really only two things to do with it: remove it or move it. Both of these are simple enough:

- HasLegend (on the Chart)
- Delete
- Position

TODO: add an example of these in action

**common properties of a Point**

The Point represents the lowest level when it comes to how the data and formatting of a Chart is built. In general, you do not have to actively go editing Points. This is because you will typically edit the appearance of the Series and the Axes to get the Chart that you want. There are however times when you get down to the metal and edit the properties of the individual points. Before describing how to do this, it may help to give an example or two for why you want to get down to this level:

- Delete a data point without touching the Series
- Add a DataLabel to the point if the value is below some threshold (or if some other Range has a value)
- Hide a Point from one series because you want it to show up in another one

Of the tasks above, only one of them (the second) has to be accomplished via the Points. The others *could* be done via a different method, but you might find yourself in a spot where iterating some Points will save a ton of headache elsewhere. A cautionary note is that typically you should not be editing the properties of a Point; there is nearly always a better way to do these things. Part of the problem is that the settings you change will be quickly overwritten by changes in Excel or VBA. If you know you just need something done however, Points can be a quick way to make it happen.

TODO: look into ErrorBars here?

WHen thinking about working through the Points of a Series, consider the common properties you can change:

- HasLabel / DataLabel
- Value
- Formatting? (TODO: what are these)
- Hidden

TODO: finish this list

Note that in addition to the common properties, you can also change anything that can be changed from the normal Excel settings/properties window.

#### common properties of the TrendLine

The TrendLine is one of the lesser used properties, but it can be a real time saver when using VBA if you need to. The problem with the trendline normally is that you are required to work through a ton of menus to configure the properties. This is even more painful when you've got to do the same thing to multiple Series in a Chart or across multiple Charts. Similar to the other objects here, you can use VBA to quickly do the task that is otherwise a pain.

The most likely properties you'll use:

- Creating one off of a series
- Type
- Parameter

TODO: confirm these are correct

TODO: add an example showing how to add a Trendline for every Series

**Chart_AddTrendlineToSeriesAndColor.md**

```
Public Sub Chart_AddTrendlineToSeriesAndColor()

    Dim targetObject As ChartObject

    For Each targetObject In Chart_GetObjectsFromObject(Selection)
        Dim chartIndex As Long
        chartIndex = 1

        Dim targetSeries As series
        For Each targetSeries In targetObject.Chart.SeriesCollection

```

```
            Dim butlSeries As New bUTLChartSeries
            butlSeries.UpdateFromChartSeries targetSeries

            'clear out old ones
            Dim j As Long
            For j = 1 To targetSeries.Trendlines.Count
                targetSeries.Trendlines(j).Delete
            Next j

            targetSeries.MarkerBackgroundColor = Chart_GetColor(chartIndex)

            Dim newTrendline As Trendline
            Set newTrendline = targetSeries.Trendlines.Add()
            newTrendline.Type = xlLinear
            newTrendline.Border.Color = targetSeries.MarkerBackgroundColor

            '2015 11 06 test to avoid error without name
            '2015 12 07 dealing with multi-cell Names
            'TODO: handle if the name is not a range also
            If Not butlSeries.name Is Nothing Then
                newTrendline.name = butlSeries.name.Cells(1, 1).Value
            End If

            newTrendline.DisplayEquation = True
            newTrendline.DisplayRSquared = True
            newTrendline.DataLabel.Format.TextFrame2.TextRange.Font.Fill.
               ForeColor.RGB = Chart_GetColor(chartIndex)

            chartIndex = chartIndex + 1
        Next targetSeries

    Next targetObject
End Sub
```

### creating charts from scratch

The previous section discussed how to work with existing Charts. This section will focus on how to create those Charts from scratch if you are coming into a blank Worksheet or if you simply need to add a chart

to existing data. At the start, it's worth mentioning that creating Charts from scratch falls into one of two categories:

- Library/helper type code where you want to quickly create a Chart in a common way. This type of code works best in an addin and typically provides functionality that you wish Excel had from the start
- One-off code for a specific application. This involves creating a Chart with some sort of odd manipulation or formatting or other detail where automation saves time.

The two types of category will end up with code that looks similar, but the goals of the former category will be slightly different than the latter. Typically when making code for a one-off application, you can make more assumptions about how the data is structured and what sorts of actions need ot be taken. When working with helper code, you will spend more time asking for user input, and handling the different cases that might come up.

Another key point to make is that the type of work that is being done in a chart can vary as well. The splitting line here is whether the Chart creation is data heavy or formatting heavy (or possibly both). For a data heavy Chart, you will spend a lot of time collecting Ranges, creating Series, and possibly manipulating individual Points. For a formatting heavy chart, you will spend a lot of time iterating through the Series to apply formatting, label the Axes, set the number formats, and generally modify the Excel defaults. Both of these tasks are very time intensive if you are doing them without VBA, so both lend themselves to being automated if possible.

Excel provides two means of creating a Chart depending on how you want to handle things. Those two commands are:

- ChartObjects.Add
- TODO: what is the other method

I always prefer to use ChartObjects.Add because of it consistent application. The other approach tends to put you at the mercy of how Excel interprets your data and its layout.

TODO: add more detail here

The general process for creating a chart looks like this:

- Create a new ChartObject via ChartObjects.Add - store that reference
  - If you know where you want the Chart to go, you can use that information here
- Access the Chart of that object
- Change the properties of the Chart that you know – namely ChartType

- Access the SeriesCollection of the Chart and call NewSeries for each Series you want - store a reference to that Series
  - This is typically done inside a loop that is iterating through Ranges in some way
  - If you need to apply Series specific formatting, do that here
- Access the Axes collection and modify any specific parts of the Axes that you want
  - This may show up in the loop above if you want the Axis to draw information from the Series (maybe set the max to the max of the data?)

At this point, you will have a Chart with the Series you want along with the major formatting taken care of. Even better, this general framework lends itself nicely to adding new commands where needed. If you need to go after some of the finer details of the Chart, you can add those commands where the objects are being reference, or at the end of the code. The main thing to consider is whether you need to work inside loops (per Series) or if you can process the extra stuff at the end.

The other upside of this approach is that you can quickly wrap all of this code with another loop to create multiple Charts. You can then wrap that code with another loop to do it on multiple Worksheets, etc. When you write code that can cleanly live inside a loop, you make it easy to use the code elsewhere.

One other aspect of Charts that is somewhat unique is that you can typically reuse a lot of the code by creating new Subs. These can be called from the inside of a loop to create a chain of commands to process a Chart. This approach is highly effective if you work in an environment where the same or similar things need to be done. For example: you have a monthly report to create each month for multiple departments. Standardizing as much of that work into modules makes it easy to apply the code in multiple spots with minor changes. This is relevant to Charts because most of the work of Charts is changing the values of specific properties. There is typically far less logic that is unique to an application (like trying to build a Range based on the layout of data).

Once you have this general framework mastered, you can quickly use it to make more charts.

TODO: add some examples of creating Charts

### specific charting examples

This section will focus on some specific applications of applying VBA to charts. The code here can be quickly reused for your own application. These examples include:

- Creating a grid of XY scatter plots (a scatter matrix) based on a block of data
- Creating a panel of time series, one chart per each value with a common x-axis

TODO: identify the examples to include here

**creating an XY scatter matrix**

**ChartCreateXYGrid.md**

```
Public Sub ChartCreateXYGrid()

    On Error GoTo ChartCreateXYGrid_Error

    DeleteAllCharts
    'VBA doesn't allow a constant to be defined using a function (rgb) so we
       use a local variable rather than
    'muddying it up with the calculated value of the rgb function
    Dim majorGridlineColor As Long
    majorGridlineColor = RGB(200, 200, 200)
    Dim minorGridlineColor As Long
    minorGridlineColor = RGB(220, 220, 220)

    Const CHART_HEIGHT As Long = 300
    Const CHART_WIDTH As Long = 400
    Const MARKER_SIZE As Long = 3
    'dataRange will contain the block of data with titles included
    Dim dataRange As Range
    Set dataRange = Application.InputBox("Select data with titles", Type:=8)

    Application.ScreenUpdating = False

    Dim rowIndex As Long, columnIndex As Long
    rowIndex = 0

    Dim xAxisDataRange As Range, yAxisDataRange As Range
    For Each yAxisDataRange In dataRange.Columns
        columnIndex = 0

        For Each xAxisDataRange In dataRange.Columns
            If rowIndex <> columnIndex Then
                Dim targetChart As Chart
                Set targetChart = ActiveSheet.ChartObjects.Add(columnIndex *
                   CHART_WIDTH, _
```

```
                                                        rowIndex *
                                                            CHART_HEIGHT
                                                             + 100, _
                                                        CHART_WIDTH,
                                                            CHART_HEIGHT
                                                            ).Chart

        Dim targetSeries As series
        Dim butlSeries As New bUTLChartSeries

        'offset allows for the title to be excluded
        Set butlSeries.XValues = Intersect(xAxisDataRange,
            xAxisDataRange.Offset(1))
        Set butlSeries.Values = Intersect(yAxisDataRange,
            yAxisDataRange.Offset(1))
        Set butlSeries.name = yAxisDataRange.Cells(1)
        butlSeries.ChartType = xlXYScatter

        Set targetSeries = butlSeries.AddSeriesToChart(targetChart)

        targetSeries.MarkerSize = MARKER_SIZE
        targetSeries.MarkerStyle = xlMarkerStyleCircle

        Dim targetAxis As Axis
        Set targetAxis = targetChart.Axes(xlCategory)
        targetAxis.HasTitle = True
        targetAxis.AxisTitle.Text = xAxisDataRange.Cells(1)
        targetAxis.MajorGridlines.Border.Color = majorGridlineColor
        targetAxis.MinorGridlines.Border.Color = minorGridlineColor

        Set targetAxis = targetChart.Axes(xlValue)
        targetAxis.HasTitle = True
        targetAxis.AxisTitle.Text = yAxisDataRange.Cells(1)
        targetAxis.MajorGridlines.Border.Color = majorGridlineColor
        targetAxis.MinorGridlines.Border.Color = minorGridlineColor

        targetChart.HasTitle = True
        targetChart.ChartTitle.Text = yAxisDataRange.Cells(1) & " vs.
            " & xAxisDataRange.Cells(1)
```

```
                'targetChart.ChartTitle.Characters.Font.Size = 8
                targetChart.Legend.Delete
            End If

            columnIndex = columnIndex + 1
        Next xAxisDataRange

        rowIndex = rowIndex + 1
    Next yAxisDataRange

    Application.ScreenUpdating = True

    dataRange.Cells(1, 1).Activate

    On Error GoTo 0
    Exit Sub

ChartCreateXYGrid_Error:

    MsgBox "Error " & Err.Number & " (" & Err.Description & _
           ") in procedure ChartCreateXYGrid of Module Chart_Format"
    MsgBox "This is most likely due to Range issues"

End Sub
```

**creating a panel of time series plots**

**Chart_TimeSeries.md**

```
Public Sub Chart_TimeSeries(ByVal rangeOfDates As Range, ByVal dataRange As
    Range, ByVal rangeOfTitles As Range)

    Application.ScreenUpdating = False
    Const MARKER_SIZE As Long = 3
    Dim majorGridlineColor As Long
    majorGridlineColor = RGB(200, 200, 200)
    Dim chartIndex As Long
    chartIndex = 1

    Dim titleRange As Range
```

```
    Dim targetColumn As Range

    For Each titleRange In rangeOfTitles

        Dim targetObject As ChartObject
        Set targetObject = ActiveSheet.ChartObjects.Add(chartIndex * 300, 0,
            300, 300)

        Dim targetChart As Chart
        Set targetChart = targetObject.Chart
        targetChart.ChartType = xlXYScatterLines
        targetChart.HasTitle = True
        targetChart.Legend.Delete

        Dim targetAxis As Axis
        Set targetAxis = targetChart.Axes(xlValue)
        targetAxis.MajorGridlines.Border.Color = majorGridlineColor

        Dim targetSeries As series
        Dim butlSeries As New bUTLChartSeries

        Set butlSeries.XValues = rangeOfDates
        Set butlSeries.Values = dataRange.Columns(chartIndex)
        Set butlSeries.name = titleRange

        Set targetSeries = butlSeries.AddSeriesToChart(targetChart)

        targetSeries.MarkerSize = MARKER_SIZE
        targetSeries.MarkerStyle = xlMarkerStyleCircle

        chartIndex = chartIndex + 1

    Next titleRange

    Application.ScreenUpdating = True
End Sub
```

**applying common formatting to all Charts**

**ChartDefaultFormat.md**

```
Public Sub ChartDefaultFormat()

    Const MARKER_SIZE As Long = 3
    Dim majorGridlineColor As Long
    majorGridlineColor = RGB(242, 242, 242)
    Const TITLE_FONT_SIZE As Long = 12
    Const SERIES_LINE_WEIGHT As Single = 1.5

    Dim targetObject As ChartObject

    For Each targetObject In Chart_GetObjectsFromObject(Selection)
        Dim targetChart As Chart

        Set targetChart = targetObject.Chart

        Dim targetSeries As series
        For Each targetSeries In targetChart.SeriesCollection

            targetSeries.MarkerSize = MARKER_SIZE
            targetSeries.MarkerStyle = xlMarkerStyleCircle

            If targetSeries.ChartType = xlXYScatterLines Then targetSeries.
               Format.Line.Weight = SERIES_LINE_WEIGHT

            targetSeries.MarkerForegroundColorIndex = xlColorIndexNone
            targetSeries.MarkerBackgroundColorIndex = xlColorIndexAutomatic

        Next targetSeries


        targetChart.HasLegend = True
        targetChart.Legend.Position = xlLegendPositionBottom

        Dim targetAxis As Axis
        Set targetAxis = targetChart.Axes(xlValue)

        targetAxis.MajorGridlines.Border.Color = majorGridlineColor
```

```
        targetAxis.Crosses = xlAxisCrossesMinimum

        Set targetAxis = targetChart.Axes(xlCategory)

        targetAxis.HasMajorGridlines = True

        targetAxis.MajorGridlines.Border.Color = majorGridlineColor

        If targetChart.HasTitle Then
            targetChart.ChartTitle.Characters.Font.Size = TITLE_FONT_SIZE
            targetChart.ChartTitle.Characters.Font.Bold = True
        End If

        Set targetAxis = targetChart.Axes(xlCategory)

    Next targetObject

End Sub
```

**Chart_AddTitles.md**

```
Public Sub Chart_AddTitles()

    Dim targetObject As ChartObject
    Const X_AXIS_TITLE As String = "x axis"
    Const Y_AXIS_TITLE As String = "y axis"
    Const SECOND_Y_AXIS_TITLE As String = "2nd y axis"
    Const CHART_TITLE As String = "chart"

    For Each targetObject In Chart_GetObjectsFromObject(Selection)
        With targetObject.Chart
            If Not .Axes(xlCategory).HasTitle Then
                .Axes(xlCategory).HasTitle = True
                .Axes(xlCategory).AxisTitle.Text = X_AXIS_TITLE
            End If

            If Not .Axes(xlValue, xlPrimary).HasTitle Then
                .Axes(xlValue).HasTitle = True
```

```
                .Axes(xlValue).AxisTitle.Text = Y_AXIS_TITLE
            End If

            '2015 12 14, add support for 2nd y axis
            If .Axes.Count = 3 Then
                If Not .Axes(xlValue, xlSecondary).HasTitle Then
                    .Axes(xlValue, xlSecondary).HasTitle = True
                    .Axes(xlValue, xlSecondary).AxisTitle.Text =
                       SECOND_Y_AXIS_TITLE
                End If
            End If

            If Not .HasTitle Then
                .HasTitle = True
                .ChartTitle.Text = CHART_TITLE
            End If
        End With
    Next targetObject

End Sub
```

**Chart_ApplyFormattingToSelected.md**

```
Public Sub Chart_ApplyFormattingToSelected()

    Dim targetObject As ChartObject
    Const MARKER_SIZE As Long = 5

    For Each targetObject In Chart_GetObjectsFromObject(Selection)

        Dim targetSeries As series

        For Each targetSeries In targetObject.Chart.SeriesCollection
            targetSeries.MarkerSize = MARKER_SIZE
        Next targetSeries
    Next targetObject

End Sub
```

### Chart_ApplyTrendColors.md

```
Public Sub Chart_ApplyTrendColors()

    Dim targetObject As ChartObject
    For Each targetObject In Chart_GetObjectsFromObject(Selection)

        Dim targetSeries As series
        For Each targetSeries In targetObject.Chart.SeriesCollection

            Dim butlSeries As New bUTLChartSeries
            butlSeries.UpdateFromChartSeries targetSeries

            targetSeries.MarkerForegroundColorIndex = xlColorIndexNone
            targetSeries.MarkerBackgroundColor = Chart_GetColor(butlSeries.
                SeriesNumber)

            targetSeries.Format.Line.ForeColor.RGB = targetSeries.
                MarkerBackgroundColor

        Next targetSeries
    Next targetObject
End Sub
```

### Chart_CreateChartWithSeriesForEachColumn.md

```
Public Sub Chart_CreateChartWithSeriesForEachColumn()
    'will create a chart that includes a series with no x value for each
       column

    Dim dataRange As Range
    Set dataRange = GetInputOrSelection("Select chart data")

    'create a chart
```

```
    Dim targetObject As ChartObject
    Set targetObject = ActiveSheet.ChartObjects.Add(0, 0, 300, 300)

    targetObject.Chart.ChartType = xlXYScatter

    Dim targetColumn As Range
    For Each targetColumn In dataRange.Columns

        Dim chartDataRange As Range
        Set chartDataRange = RangeEnd(targetColumn.Cells(1, 1), xlDown)

        Dim butlSeries As New bUTLChartSeries
        Set butlSeries.Values = chartDataRange

        butlSeries.AddSeriesToChart targetObject.Chart
    Next targetColumn

End Sub
```

**Chart_CreateDataLabels.md**

```
Public Sub Chart_CreateDataLabels()

    Dim targetObject As ChartObject
    On Error GoTo Chart_CreateDataLabels_Error

    For Each targetObject In Chart_GetObjectsFromObject(Selection)

        Dim targetSeries As series
        For Each targetSeries In targetObject.Chart.SeriesCollection

            Dim dataPoint As Point
            Set dataPoint = targetSeries.Points(2)

            dataPoint.HasDataLabel = False
            dataPoint.DataLabel.Position = xlLabelPositionRight
            dataPoint.DataLabel.ShowSeriesName = True
            dataPoint.DataLabel.ShowValue = False
```

```
            dataPoint.DataLabel.ShowCategoryName = False
            dataPoint.DataLabel.ShowLegendKey = True

        Next targetSeries
    Next targetObject

    On Error GoTo 0
    Exit Sub

Chart_CreateDataLabels_Error:

    MsgBox "Error " & Err.Number & " (" & Err.Description & ") in procedure
       Chart_CreateDataLabels of Module Chart_Format"

End Sub
```

**Chart_ExtendSeriesToRanges.md**

```
Public Sub Chart_ExtendSeriesToRanges()

    Dim targetObject As ChartObject

    For Each targetObject In Chart_GetObjectsFromObject(Selection)

        Dim targetSeries As series

        'get each series
        For Each targetSeries In targetObject.Chart.SeriesCollection

            'create the bUTL obj and manipulate series ranges
            Dim butlSeries As New bUTLChartSeries
            butlSeries.UpdateFromChartSeries targetSeries

            If Not butlSeries.XValues Is Nothing Then
                targetSeries.XValues = RangeEnd(butlSeries.XValues.Cells(1),
                   xlDown)
            End If
```

```
            targetSeries.Values = RangeEnd(butlSeries.Values.Cells(1), xlDown
                )

        Next targetSeries
    Next targetObject
End Sub
```

### Chart_GoToXRange.md

```
Public Sub Chart_GoToXRange()


    If TypeName(Selection) = "Series" Then
        Dim b As New bUTLChartSeries
        b.UpdateFromChartSeries Selection

        b.XValues.Parent.Activate
        b.XValues.Activate
    Else
        MsgBox "Select a series in order to use this."
    End If

End Sub
```

### Chart_SortSeriesByName.md

```
Public Sub Chart_SortSeriesByName()
    'this will sort series by names
    Dim targetObject As ChartObject
    For Each targetObject In Chart_GetObjectsFromObject(Selection)

        'uses a simple bubble sort but it works... shouldn't have 1000 series
            anyways
        Dim firstChartIndex As Long
        Dim secondChartIndex As Long
        For firstChartIndex = 1 To targetObject.Chart.SeriesCollection.Count
```

```
            For secondChartIndex = (firstChartIndex + 1) To targetObject.
                Chart.SeriesCollection.Count

                Dim butlSeries1 As New bUTLChartSeries
                Dim butlSeries2 As New bUTLChartSeries

                butlSeries1.UpdateFromChartSeries targetObject.Chart.
                    SeriesCollection(firstChartIndex)
                butlSeries2.UpdateFromChartSeries targetObject.Chart.
                    SeriesCollection(secondChartIndex)

                If butlSeries1.name.Value > butlSeries2.name.Value Then
                    Dim indexSeriesSwap As Long
                    indexSeriesSwap = butlSeries2.SeriesNumber
                    butlSeries2.SeriesNumber = butlSeries1.SeriesNumber
                    butlSeries1.SeriesNumber = indexSeriesSwap
                    butlSeries2.UpdateSeriesWithNewValues
                    butlSeries1.UpdateSeriesWithNewValues
                End If

            Next secondChartIndex
        Next firstChartIndex
    Next targetObject
End Sub
```

**ChartFlipXYValues.md**

```
Public Sub ChartFlipXYValues()

    Dim targetObject As ChartObject
    Dim targetChart As Chart
    For Each targetObject In Chart_GetObjectsFromObject(Selection)
        Set targetChart = targetObject.Chart

        Dim butlSeriesies As New Collection
        Dim butlSeries As bUTLChartSeries

        Dim targetSeries As series
```

```
        For Each targetSeries In targetChart.SeriesCollection
            Set butlSeries = New bUTLChartSeries
            butlSeries.UpdateFromChartSeries targetSeries

            Dim dummyRange As Range

            Set dummyRange = butlSeries.Values
            Set butlSeries.Values = butlSeries.XValues
            Set butlSeries.XValues = dummyRange

            'need to change the series name also
            'assume that title is same offset
            'code blocked for now
            If False And Not butlSeries.name Is Nothing Then
                Dim rowsOffset As Long, columnsOffset As Long
                rowsOffset = butlSeries.name.Row - butlSeries.XValues.Cells
                    (1, 1).Row
                columnsOffset = butlSeries.name.Column - butlSeries.XValues.
                    Cells(1, 1).Column

                Set butlSeries.name = butlSeries.Values.Cells(1, 1).Offset(
                    rowsOffset, columnsOffset)
            End If

            butlSeries.UpdateSeriesWithNewValues

        Next targetSeries

        ''need to flip axis labels if they exist
        ''three cases: X only, Y only, X and Y

        If targetChart.Axes(xlCategory).HasTitle And Not targetChart.Axes(
            xlValue).HasTitle Then

            targetChart.Axes(xlValue).HasTitle = True
            targetChart.Axes(xlValue).AxisTitle.Text = targetChart.Axes(
                xlCategory).AxisTitle.Text
            targetChart.Axes(xlCategory).HasTitle = False

```

```
        ElseIf Not targetChart.Axes(xlCategory).HasTitle And targetChart.Axes
            (xlValue).HasTitle Then
            targetChart.Axes(xlCategory).HasTitle = True
            targetChart.Axes(xlCategory).AxisTitle.Text = targetChart.Axes(
                xlValue).AxisTitle.Text
            targetChart.Axes(xlValue).HasTitle = False

        ElseIf targetChart.Axes(xlCategory).HasTitle And targetChart.Axes(
            xlValue).HasTitle Then
            Dim swapText As String

            swapText = targetChart.Axes(xlCategory).AxisTitle.Text

            targetChart.Axes(xlCategory).AxisTitle.Text = targetChart.Axes(
                xlValue).AxisTitle.Text
            targetChart.Axes(xlValue).AxisTitle.Text = swapText

        End If

        Set butlSeriesies = Nothing

    Next targetObject

End Sub
```

**ChartMergeSeries.md**

```
Public Sub ChartMergeSeries()

    Dim targetObject As ChartObject
    Dim targetChart As Chart
    Dim firstChart As Chart

    Dim isFirstChart As Boolean
    isFirstChart = True

    Application.ScreenUpdating = False

```

```
    For Each targetObject In Chart_GetObjectsFromObject(Selection)

        Set targetChart = targetObject.Chart
        If isFirstChart Then
            Set firstChart = targetChart
            isFirstChart = False
        Else
            Dim targetSeries As series
            For Each targetSeries In targetChart.SeriesCollection

                Dim newChartSeries As series
                Dim butlSeries As New bUTLChartSeries

                butlSeries.UpdateFromChartSeries targetSeries
                Set newChartSeries = butlSeries.AddSeriesToChart(firstChart)

                newChartSeries.MarkerSize = targetSeries.MarkerSize
                newChartSeries.MarkerStyle = targetSeries.MarkerStyle

                targetSeries.Delete

            Next targetSeries

            targetObject.Delete

        End If
    Next targetObject

    Application.ScreenUpdating = True

End Sub
```

**ChartSplitSeries.md**

```
Public Sub ChartSplitSeries()

    Dim targetObject As ChartObject
    Dim targetChart As Chart
```

```
5
6       Dim targetSeries As series
7       For Each targetObject In Chart_GetObjectsFromObject(Selection)
8
9           For Each targetSeries In targetObject.Chart.SeriesCollection
10
11              Dim newChartObject As ChartObject
12              Set newChartObject = ActiveSheet.ChartObjects.Add(0, 0, 300, 300)
13
14              Dim newChartSeries As series
15              Dim butlSeries As New bUTLChartSeries
16
17              butlSeries.UpdateFromChartSeries targetSeries
18              Set newChartSeries = butlSeries.AddSeriesToChart(newChartObject.
                    Chart)
19
20              newChartSeries.MarkerSize = targetSeries.MarkerSize
21              newChartSeries.MarkerStyle = targetSeries.MarkerStyle
22
23              targetSeries.Delete
24
25          Next targetSeries
26
27
28          targetObject.Delete
29
30      Next targetObject
31  End Sub
```

**DeleteAllCharts.md**

```
1   Public Sub DeleteAllCharts()
2
3       If MsgBox("Delete all charts?", vbYesNo) = vbYes Then
4           Application.ScreenUpdating = False
5
6           Dim chartObjectIndex As Long
7           For chartObjectIndex = ActiveSheet.ChartObjects.Count To 1 Step -1
```

```
            ActiveSheet.ChartObjects(chartObjectIndex).Delete

        Next chartObjectIndex

        Application.ScreenUpdating = True

    End If
End Sub
```

**RemoveZeroValueDataLabel.md**

```
Public Sub RemoveZeroValueDataLabel()

    'uses the ActiveChart, be sure a chart is selected
    Dim targetChart As Chart
    Set targetChart = ActiveChart

    Dim targetSeries As series
    For Each targetSeries In targetChart.SeriesCollection

        Dim seriesValues As Variant
        seriesValues = targetSeries.Values

        'include this line if you want to reestablish labels before deleting
        targetSeries.ApplyDataLabels xlDataLabelsShowLabel, , , , True, False
            , False, False, False

        'loop through values and delete 0-value labels
        Dim pointIndex As Long
        For pointIndex = LBound(seriesValues) To UBound(seriesValues)
            If seriesValues(pointIndex) = 0 Then
                With targetSeries.Points(pointIndex)
                    If .HasDataLabel Then .DataLabel.Delete
                End With
            End If
        Next pointIndex
    Next targetSeries
```

```
End Sub
```

**UpdateFromChartSeries.md**

```
Public Sub UpdateFromChartSeries(targetSeries As series)


    'this will work for the simple case where all items are references
    Const FIND_STRING As String = "SERIES("
    Const COMMA As String = ","
    Const CLOSE_BRACKET As String = ")"

    Set series = targetSeries

    Dim targetForm As Variant

    '=SERIES("Y",Sheet1!$C$8:$C$13,Sheet1!$D$8:$D$13,1)

    'pull in the formula
    targetForm = targetSeries.Formula

    'uppercase to remove match errors
    targetForm = UCase(targetForm)

    'remove the front of the formula
    targetForm = Replace(targetForm, FIND_STRING, vbNullString)

    'find the first foundPosition
    Dim foundPosition As Long
    foundPosition = InStr(targetForm, COMMA)

    If foundPosition > 1 Then
        'need to catch an error here if a text name is used instead of a
           valid range
        On Error Resume Next
        Set Me.name = Range(left(targetForm, foundPosition - 1))
        If Err <> 0 Then pName = left(targetForm, foundPosition - 1)
        On Error GoTo 0
```

```
    End If

    'pull out the title from that
    targetForm = Mid(targetForm, foundPosition + 1)

    foundPosition = InStr(targetForm, COMMA)

    If foundPosition > 1 Then Set Me.XValues = Range(left(targetForm,
        foundPosition - 1))

    targetForm = Mid(targetForm, foundPosition + 1)

    foundPosition = InStr(targetForm, COMMA)
    Set Me.Values = Range(left(targetForm, foundPosition - 1))
    targetForm = Mid(targetForm, foundPosition + 1)

    foundPosition = InStr(targetForm, CLOSE_BRACKET)
    Me.SeriesNumber = left(targetForm, foundPosition - 1)

    Me.ChartType = targetSeries.ChartType
End Sub
```

## The Worksheet object

### introduction to the Worksheet object

This chapter will focus on the aspects of the Worksheet that appear commonly in VBA code. This chapter is a little shorter than others because in general, the Worksheet is a conduit to more useful things. There is very little that takes place within the Worksheet object that is not just a pass through to the more interesting details (e.g. Range or Chart). Having said that, there are a handful of areas that are relevant to the Worksheet and not accessible anywhere else. Those specific areas include:

- Creating and managing Worksheets – this sounds obvious but managing the references to Worksheets becomes a major issue when working with large, complicated workflows
- Print layout, printing, and exporting
- Locking and setting passwords on Worksheets

- Managing the properties of the Worksheet itself including Name, tab color, etc.

TODO: any other Worksheet things?

Of the topics listed above, the most important area is actually creating and managing the Worksheets in a complicated workflow. This is closely related to working with Ranges since presumably you create the Worksheet to put data into or something else into it. Managing the references to Worksheets can be a big deal and determining how best to access or select a given Worksheet can be important. In addition to getting references, there are a handful of times where you actually need to Activate a Worksheet. Knowing when this is and is not required is important.

TODO: when do you have to Activate?

## creating and managing Worksheets

This section will focus on how to create a Worksheet and get a reference to new Worksheets. In addition to that, it will discuss managing Worksheets, including rearranging and deleting them.

### references to Worksheets

The process for working with Worksheets is the same as all the other Excel Object Model objects: obtain a reference to the object and access it properties. For the Worksheet, there are a handful of ways to obtain a reference to a Worksheet. Those include:

- ActiveSheet
- Worksheets(index) or Sheets(index) global objects
- Workbook.Sheets(index) or Workbook.Worksheets(index) with a workbook object
- VBA references (Sheet1, Sheet2)
- Store the reference after creating a new sheet
- Iterating through Worksheets and picking with some criteria
- Copy a Worksheet and then search for the result (see notes below; TODO: add notes)

The basic dividing line of the methods above is when you want to access the Worksheet and what you potentially know about it. The simplest approach is when you want some code to run on the ActiveSheet because you can just ask for it. Technically, you can avoid most refereces to the ActiveSheet use the unqualified global references, but this can be lead to errors later. The business of obtaining a reference to a Worksheet using the other means typically only comes up when you are working with multiple Worksheets. This is quite common to do.

Once you start workign wtih multiple Worksheets, there are a couple of common things you may want to do:

- Apply the same action to multiple sheets
- Process some data on one sheet based on the data on another sheet
- Move data from one sheet to another
- Move a chart or other object from one sheet to another
- Create a throwaway Worksheet with some information about the rest of your Workbook (e.g. output all the sheet names)

In some of those instances, you are working with multiple sheets becuase you want to do something (e.g. print layout or formatting) to multiple sheets. In others, you are working on multiple sheets because you know in advance that some task will use data from multiple Worksheets. For the former case, you are likely to throw you code into a loop across all Worksheets and then use some logic to determine whether or not to apply the action. In the other case, you will likely use a sheet name or index to directly access the sheet you want.

It is worth mentioning that every Workbook has built in dedicated referneces to the Worksheets which can be used. These exist as a part of the Object Model. By default they are called `Sheet1`, `Sheet2`, etc. These objects are always available and provide a direct refernece to the worksheet. They can be quite helpful if you rename them from the default names. A couple of important items about these objects:

- They only exist as objects in the current Workbook. That is, if you want to access a Worksheet in another Workbook, this approach will not work. You can technically add a reference to the other Workbook, but I don't recommend doing that.
- Their naming is independent of the actual sheet name displayed in Excel. This cna be incredibly confusing for a new developer (especially if they are not using `Option Explicit`).
- It is very difficult to use these objects to perform some action to multiple Worksheets.

For what it's worth, I've never used the objects directly. I find myself using the sheet name directly when needed. This leads to issues with the name being changed, but at some point searching for the string in code is easier than trying to rename the object in the VBE sidebar. All of the references will break either way.

#### creating a Worksheet

Aside from referencing an existing Worksheet often times the core task of some automtion is to create a new Worksheet. There are a number of reasons you might want to do this:

- A blank sheet is a great starting part for storing some intermediate or final result. It is nearly gauranteed to be the same every time you call for oen which is much better than putting new data in an existing sheet.
- You need a blank sheet for the output of some process that is run over a number of items (each analysis gets a new sheet).
- Copying an existing Worksheet and then applying some transformation to the result.
- You created a new Workbook. This adds an extra step but leaves you with the same result as a new sheet alone (unless it was created from a template).

From my own expereince, I find that creating a new Worksheet is an absolutely critical task. Very often the goal of usign VBA is to automate some task over a range of inputs or possible outputs. This often means that the outut for a given command may need to be produced several times. In this case, I regularly create new Worksheets instead of managing the multiple sets of data in one sheet.

In other cases, you may use a temporary intermediate new Worksheet to provide a dumping place for some calculations or other work. This is a much safer approach than to use the existing Worksheet for temporary efforts. Unless you are certain of the contents of an existing Worksheet, there is little reason to avoid creating a new one.

It's worht noting that Excel is quite performant even with a large number of Worksheets. This is especially true if the Worksheets are not linked or related via calculations. My strongest advice on this front is to liberally create new Worksheets and deal with the aftermath later. If you are building a complicated workflow, sometimes the best output is one that is useful but completely disposable. This means that the output is impressive but due to the speed of the automation there is little reason to save or otherwise consume the resulting file. When this is the case, there is no penalty for disorganized Worksheets if the intended product is still there. Let Excel deal with the references and Ranges etc. while you deal with maintainign the rereferences in VBA.

Having said all of that, creating a Worksheet is incredibly simple `Workbook.Sheets.Add()`. That Function will return the Worksheet object which is a refernece to the new sheet. The new sheet will have a default name. THe parameters to `Add` control the location of the new sheet with respect to others. It is very, very unlikely that you will create a new Worksheet and not immediately want the sheet refernece. That is, you will probably always call `Add` with a preceding `Set` to save the reference. This reference can be as good as gold in an automated workflow since an empty Worksheet is a very powerful starting (and possibly daunting) point.

If you need a copy of an existing Worksheet instead of a blank one, the command is quite simple: `Worksheet.Copy()`. This will create a Copy with parameters for lcoation (TODO: is that true?). The major

downside of using `Copy` is that it will NOT return a reference to the newly created Worksheet. This is a real travesty because it means you then have to turn around and do some work to find the newly create Worksheet. My preferred approach is to Copy the Worksheet to the first or last location in the Sheet order and then find it there. Once found, you can move the Worksheet to a desired location and then use the reference.

#### removing a Worksheet

If you need to delete a Worksheet, it is a simple command again: `Worksheet.Delete`. The one downside to this command is that it will fire off a warning prompt if the Worksheet contained any data or was otherwise not "blank". This warning box will stall the execution of your VBA until it is addressed. This is a major issue for any serious workflow since your users will have to constantly click "Yes" to delete the Worksheet but they may also have no idea what they are deleting. To avoid this issue, you will nearly ALWAYS wrap the `Delete` command with the comannds to disbale and then reenable the alerts. The typical code looks like:

```
Application.DisplayAlerts = False
Worksheet.Delete
Application.DisplayAlerts = True
```

When doing this dance, be absolutely certain that you reenable the alerts. Excel will not do it for you. You may benefit from creating a new helper Sub which contains the above code as a `DeleteSheet` command to avoid constantly adding those alerts.

TODO: add a note about when to create a new Worksheet vs. a new Workbook and the pros/cons there (maybe put this in the workflow section of book)

#### rearranging Worksheets

To rearrange the Worksheets, the command is simple: `Worksheet.Move(Before, After)`. The parametrs there will indicate the sheet ot place it before or after. The real task here is determining whcih sheet to refernece there, but finding that reference is the same task that is described up at the top of the seciton.

**AscendSheets.md** TODO: move the AscendSheets code elsewhere or delete (not helpful here)

```
Public Sub AscendSheets()

    Application.ScreenUpdating = False
    Dim targetWorkbook As Workbook
    Set targetWorkbook = ActiveWorkbook

    Dim countOfSheets As Long
    countOfSheets = targetWorkbook.Sheets.Count

    Dim i As Long
    Dim j As Long

    With targetWorkbook
        For j = 1 To countOfSheets
            For i = 1 To countOfSheets - 1
                If UCase(.Sheets(i).name) > UCase(.Sheets(i + 1).name) Then .
                    Sheets(i).Move after:=.Sheets(i + 1)
            Next i
        Next j
    End With

    Application.ScreenUpdating = True
End Sub
```

### properties and methods on the Worksheet

This section will focus on the specific properties and functions that exist for a Worksheet.

Some of the useful properties of a Worksheet include:

- Name
- Move
- Copy
- Protect/Unprotect
- Range, Cells, Rows, Columns, UsedRange
- The accessors which will give you a Collection of other objects
  - ChartObjects

    - Charts (TODO: that right?)
    - Shapes
    - ListObjects
    - PivotTables
    - Hyperlinks
    - Comments?
- TODO: add others

TODO: determine how ot explain these and which to include

#### LockAllSheets.md

TODO: clean up this code

```
Public Sub LockAllSheets()

    Dim userPassword As Variant
    userPassword = Application.InputBox("Password to lock")

    If Not userPassword Then
        MsgBox "Cancelled."
    Else
        Application.ScreenUpdating = False

        'Changed to ActiveWorkbook so if add-in is not installed, it will
           target the active book rather than the xlam
        Dim targetSheet As Worksheet
        For Each targetSheet In ActiveWorkbook.Sheets
            On Error Resume Next
            targetSheet.Protect (userPassword)
        Next

        Application.ScreenUpdating = True
    End If

End Sub
```

**OutputSheets.md**

TODO: clean up this code

```
Public Sub OutputSheets()

    Dim outputSheet As Worksheet
    Set outputSheet = Worksheets.Add(Before:=Worksheets(1))
    outputSheet.Activate

    Dim outputRange As Range
    Set outputRange = outputSheet.Range("B2")

    Dim targetRow As Long
    targetRow = 0

    Dim targetSheet As Worksheet
    For Each targetSheet In Worksheets

        If targetSheet.name <> outputSheet.name Then

            targetSheet.Hyperlinks.Add _
                outputRange.Offset(targetRow), "", _
                "'" & targetSheet.name & "'!A1", , _
                targetSheet.name
            targetRow = targetRow + 1

        End If
    Next targetSheet

End Sub
```

**UnlockAllSheets.md**

TODO: clean up this code

```
Public Sub UnlockAllSheets()

```

```
    Dim userPassword As Variant
    userPassword = Application.InputBox("Password to unlock")

    Dim errorCount As Long
    errorCount = 0

    If Not userPassword Then
        MsgBox "Cancelled."
    Else
        Application.ScreenUpdating = False
        'Changed to ActiveWorkbook so if add-in is not installed, it will
           target the active book rather than the xlam
        Dim targetSheet As Worksheet
        For Each targetSheet In ActiveWorkbook.Sheets
            'Let's keep track of the errors to inform the user
            If Err.Number <> 0 Then errorCount = errorCount + 1
            Err.Clear
            On Error Resume Next
            targetSheet.Unprotect (userPassword)

        Next targetSheet
        If Err.Number <> 0 Then errorCount = errorCount + 1
        Application.ScreenUpdating = True
    End If
    If errorCount <> 0 Then
        MsgBox (errorCount & " sheets could not be unlocked due to bad
           password.")
    End If
End Sub
```

### print layout and exporting

THis section will focus on the print and export related details of a Worksheet. In particular, it will focus on the details that are typically accessed through the Page Layout menu. This is one of the unique aspects of Worksheets because they are the holder of the print/export information. The main details related to this are:

- Print area

- Page layout – this is a very large object with a lot of properties to be set
- Exporting and printing

The details in this section can be a real time saver because one of the more tedious aspects of working with Excel is ensuring that your reports/graphs/data will print or export correctly. Being able to control these properties with VBA makes it possible to quickly apply the same formatting to a large number of Worksheets without having to click nine million times.

When editing the Page Layout, you can change nearly everything. The one thing ot be aware of is related to printers. There are a number of settings in the Worksheet that are internally tied to the defualt (or active) printer. This shows up if you are attmepting to set the page size specifically. IF you always use the same printer or have coworkers who use the smae printers, you amy not notice these pissues. It becomes a serious probelm when you ar etrying ot make code work for multiple differnet printers that support or dientify page sizes differently.

The best way to see what is savialable for page settings is to record a macro and change one thing. Excel is a bit agressive at including all possible settings that coudl ahve hcanged. This is very nice if you want to grab some setings nad work them into your code.

There are a couple of other items to describe so you know what they are:

- Using `Zoom` and `FitToPages` to set the number of pages that the output will be included in (TODO: review)
- TODO: add others

Also, be aware that hcanging the print settings is a per Worksheet change. This amy be ovvious since hte peroperies are off the Worksheet, but it is easy to forget this. The nice thing however s that you can just iterate your WOrksheets and apply the same settings to all of them. THis is oen of the greatest time savers comapred to changing properties in Excel (TODO: can these be changed with multi selecteion?).

#### Rand_common print settings

TODO: clean up this code

```
1  Sub Rand_CommonPrintSettings()
2
3      Application.ScreenUpdating = False
4      Dim sht As Worksheet
5
```

```
    For Each sht In Sheets
        sht.PageSetup.PrintArea = ""
        sht.ResetAllPageBreaks
        sht.PageSetup.PrintArea = ""

        With sht.PageSetup
            .LeftHeader = ""
            .CenterHeader = ""
            .RightHeader = ""
            .LeftFooter = ""
            .CenterFooter = ""
            .RightFooter = ""
            .LeftMargin = Application.InchesToPoints(0.75)
            .RightMargin = Application.InchesToPoints(0.75)
            .TopMargin = Application.InchesToPoints(1)
            .BottomMargin = Application.InchesToPoints(1)
            .HeaderMargin = Application.InchesToPoints(0.5)
            .FooterMargin = Application.InchesToPoints(0.5)
            .PrintHeadings = False
            .PrintGridlines = False
            .PrintComments = xlPrintNoComments
            .PrintQuality = 600
            .CenterHorizontally = False
            .CenterVertically = False
            .Orientation = xlLandscape
            .Draft = False
            .PaperSize = xlPaperLetter
            .FirstPageNumber = xlAutomatic
            .Order = xlDownThenOver
            .BlackAndWhite = False
            .Zoom = False
            .FitToPagesWide = 1
            .FitToPagesTall = False
            .PrintErrors = xlPrintErrorsDisplayed
            .OddAndEvenPagesHeaderFooter = False
            .DifferentFirstPageHeaderFooter = False
            .ScaleWithDocHeaderFooter = True
            .AlignMarginsHeaderFooter = False
            .EvenPage.LeftHeader.Text = ""
```

```
            .EvenPage.CenterHeader.Text = ""
            .EvenPage.RightHeader.Text = ""
            .EvenPage.LeftFooter.Text = ""
            .EvenPage.CenterFooter.Text = ""
            .EvenPage.RightFooter.Text = ""
            .FirstPage.LeftHeader.Text = ""
            .FirstPage.CenterHeader.Text = ""
            .FirstPage.RightHeader.Text = ""
            .FirstPage.LeftFooter.Text = ""
            .FirstPage.CenterFooter.Text = ""
            .FirstPage.RightFooter.Text = ""
            .PrintTitleRows = ""
            .PrintTitleColumns = ""
        End With
    Next sht

    Application.ScreenUpdating = True
End Sub
```

## The Workbook object

### introduction to the Workbook

This chapter will focus on the Workbook object. There are only a handful of things that are done at the Workbook level, but those few items are quite important. The main things to be done with a Workbook are:

- Create, open, close, and save Workbooks
- Manage references to Workbooks
- Change certain properties of the Workbook (TODO: are there any?)
- Access certain properties of the Workbook (e.g. Path)

This will be a quick chapter since the items below are fairly straight forward. Having said that, this chapter will be quite relevant if you are working through a complex workflow that involves creating temporary or new Workbooks to store data or analysis. Being able to create and work with Workbooks will give you the confidence to fire up a new Workbook for a one-off analysis instead of polluting the existing Workbook with a one-off Worksheet or other data dump.

### understanding the Workbook Object Model

The Workbook is an object that serves two main purposes:

- Provide a foothold to a number of other more useful functions (e.g. Sheets)
- Provide a reference to the underlying data within a spreadsheet while working through a workflow

For the first point, the goal of the Workbook is to actually use its properties to do some task. For the latter point, the Workbook is simply a container that holds data which is necessary to interact with while creating a workflow. To be honest, there is very little of use within the Workbook object that is not a refernece to some other object. Typically, the main tasks to actually be done with the Workbook are to Open, Save, and Close them. That is, you move away from the Workbook object as quickly as you can because you just need a reference.

### working with Workbook references

There are a couple of ways to obtain a refernece to a Workbook that are useful:

- ActiveWorkbook - refers to the Workbook that has focus
- ThisWorkbook - refers to the Workbook which contains the code that is executing
- Workbooks.Open() - will open a Workbook and return a reference
- Workbooks(index) - will grab a refernece to the currently opened Workbook
- Workbooks.Add() - will create a new blank Workbook or a Workbook according to a supplied template

I find that all of those approaches are used equally across my code. The one exception might be ThisWorkbook which I typically avoid. In reality, I should probably use it more becasue I find myself going to some length to maintain a reference to a Workbook while opening or creating Workbooks.

For Workbooks, the biggest thing to be aware of that there are a number of unqualified references that exist within VBA that are a part of the ActiveWorkbook. Those include:

- Worksheets and Sheets
- Names?

These unqualified referecnes can really bite you when you are expecting it. The problem with unqalified references is taht they work great initially, before the workflow becomes complex. They will then silently fail later when you start creating new Workbooks and otherwise changing the focus or active Workbook. The problem is that nearly all of the unqualified references apply to the ActiveWorkbook. Working with Workbooks is the one task that will often change the focus of Excel regardless of how you create things.

### useful properties of the Workbook

Although I have railed against the Workbook object, there are a handful of things that it can do:

- Reference `Names` which contains all of the global named ranges
- Others?
- Charts?

#### Worksheets vs. Sheets

WHen working with Worksheets, there are a pair of objects which will provide access to the underlying Sheets. They are different in how they handle Charts which are visible as a Worksheet. The rule is: Sheets will return the Charts, whereas Worksheets will only return the list of objects which are actually Worksheets. If you do not use Charts as Worksheets, then you will never notice a difference between these two objects. The one thing you will notice is that the ActiveWorksheet will not be of type Worksheet which means that you can never get Intellisense on one of the most useful objects.

## The Application object

### introduction to the Application

This chapter will focus on the Application object. There are a number of significant details of Excel and its interaction with VBA that are controlled form the Application. In particular, the Application is responsible for providing access to calculation related properties and user display preferences.

The major features of the Application object include:

- Controlling the calculation
- Controlling events and other visual effects
- Controlling the StatusBar and providing feedback during a macro

TODO: any other items for this list?

The Application can do a number of other things related to Excel settings which will not be covered here.

## Controlling calculations

When you are creating macro workflows, there are a number of tools at your disposal to contorl calcualtion flow. Before describing those tools, it's worth stepping back adn discussing why you might watn to control the calculation flow. There are a couple of common reason:

- Performance. Your code will run faster if you control the calculation process. This mainly involves disabling automatic calculation at key points.
- Accuracy. For some types of calculatiojns, you need to tightly contorl the calculation flow for accuracy. This is often the case if you are building a spreadsheet that does some form of recurison or self reference.
- Usability. There are some situations where you are interacting with calcualtions and need to prevent the normal behavior. The most common is when you add Workbook events like `Change`.
- Profiling. If you are buidling a code profiler (i.e. a tool that tracks execution time of your code) you must control calculations in order to get the tracking right.

We'll get back to the applciations, but it's also worht hittin gthe high points on how you can control the calculatiojn. THe main knobs:

- Disable application wide
- Disable for a Worksheet
- Manually calcualte a Range, Worksheet, or Applicaiton

THe types of changes you will make are fiarly tightly couple to the applciations above. In general, for perfomrance nad usability reasons, you will be disable calculations. For accuracy or profiling applcaitions, you will manually walking the calculation through.

### Disabling calculations

The most common approach to controllign calculatiojns is to simply disable them. To "dsiable" the calculations, is really to set the CalculationMode to Manual. It does not actually disable calcualtions, but instead it prevents the automatic calcualtion updates from firing like normal. The spreadsheet still maintains its normal model of calculations; they just don't run. This is an incredibly common approach to speeding up the perofmrance of VBA code. The performance boost results form the fact that when VBA code executes, it is very tihglty coupled to the normal Excel operaitons that take place. When you use VBA to set a `.Value` equal to some new value, it is functioanlyl equivlabet to manually entering the value. Behind the scenes, Excel will fire off the normal Change events and update the dependent cells. THis can

become a bottleneck because VBA is able to rapidly fire off `.Value` changes. So rapidly, that processing all of the associated stuff can become a limitation. It is more or less gauranteed that you will run into this issue once you start writing VBA code. It is so common, that you will likely memorize the fix:

TODO: check this code

```
Application.CalculationMode = xlManual
Application.ScreenUpdating = False
Applicaiton.EnabledEvents = False

Applicaiton.EnabledEvents = True
Application.ScreenUpdating = True
Application.CalculationMode = xlAutomatic
```

Why does this code make everything faster? Well, it disables the slowest steps of Excel keeping track of your spreadsheet: visual updates, calcualtion updates, and other events. Turning all of those off will dramtically remove the bottlenecks to your code. What's the downside? Well, all of that stuff exists for a reaosn and it's possible you need it to kepe funcitojning for smoe VBA operaitons. The non-calculatojn optiojsn ar ecovered in subsequent chapters, so we'll focus on the claculatiojn part now.

What happens when you disbale calcualtiojns? THis is the key concept to understand to make sure your spreadsheets do not break when you go looking for perfomrance. So what changes?

- Dependent cells are not updated. The "chain" is processed to its end on every update. Note, updates are sent downstream. Not all cells are updated, unless your Workbook contains a VOLATILE function.
- Charts and other functional graphics do not update. Internally, they don't change at all. It's not just a matter of the visuals being hiddne, they are not calcualted.
- Less important items:
  - Conditional formatting will not update.

So why might those things matter? The biggest reason is that if your VBA code depends on the state of the spreadsheets, then you are likely depending on calcualtions at some point. This menas that you need ot split you rcode into semgents where you are not worried about cell values and those wehre you are. An example:

You are building a tool to process data from a CSV file. You have been told that you should delete data that is in the 0th to 10th percentile of a cost column. Unforateunyl, the data needs to be preprocessed in order to create an accurate cost column. Your CSV file contains a mess of extra text and ohter issues when need to be removed. Your workflow then is:

1. Import the CSV data
2. Preprocess the cost column to clean up the mess
3. Remove the rows below the 10th percentile.

You do a quick test and have no problem importing the CSV data. You've gone ahead and worked out the preprocessing logic... only took a couple calls to `Split` and `Trim`. YOu also went ahead and added a new colujmn to compute the `PERCENTILE` based on the now cleaned result. This is looking great on your 100 row test data set. Your set your application loose on the 90,000 "real" data and quickly find that it will not complete within 10 minutes. What's going on here? THe most likely provlem is that your new PERCENTILE column is being reculated every time a preprocessed data cell is being added back to the spreadsheet. Your processing code looks liek:

```
For Each rngCell in rngData
    rngCell.Value = CleanUpThisMess(rngCell)
Next
```

If `rngData` contains 90,000 cells, then your update code will call for at least 90,000 full Worksheet recalculations. Even worse, your PERCENTILE fomrula requires the entire colujmn of data and so all cells have to update every time. `90,000 x 90,000` quickly becomes a problem.

So, why is the PERCENTILE function updating after eveyr change? Do we really care what hte intermediate values are? No.

This is why you want to have control of the calcualtion. in this case, you kknow that the processing code is not affected by the value of the PERCENTILE column. We only need the static data available in order to complete the processing. The fix here is to turn calcuaitliosn to manual during the processing step so that you do not incur 90,000 extra recalculations.

Once the processing is done, what do we do with the calculatiojn mode? Well, that depends on how we do the deletion. There are a couple of optionsm:

- Turn on an `AutoFilter` and do a FILTER-DELETE to remove all the rows in one shot.
- Iterate through the rows, one by one, and remove those which are in the 10th or lower percentils

Looks like either will work, but hwo does calculatiojn mode affect things? Well, if you go with the latter option, you will find that your PERCENTILES will update after each deletion. This is not hte behavior you intended. You somehow want to remember the PERCENTILE value before oyu started the deletions. The solution then is to contorl the calculation mode again. Here, we are controllign things for **accuracy**. Our deletiojn appraihc will not work if we allow cells to update as we go.

Pro tip: if you are deleting cells, you should pretty much never go a row, column, or cell at a time. Instead you should build a `Range` of cells to be deleted using `Union` and delete them in one shot using `Delete`. This approach is called a `UNION-DELETE` and avoids all of the issues described above. It's also the fatest approach since it does a single deletion.

### Controlling events and visuals

THe previous seciton focused on controlling calcualtiojns, generally for the sake of perofmrnace. When seeking better perfomrance, there are two otehr changes that are commonly made. They rely on disbale the screen updating also disabling events. The former is a pretty tame change and is a no brainer if you want perofmrance. There are very few downsides to disabling the screen. Disabling events can give a big boost in perfomrance also, but tehre are a couple more risks involved. In addition to perofmrance, there are otehr times where you need to disable events in order for your code to wokr.

The most common code for perfomrance is repated here for clarity:

```
Application.CalculationMode = xlManual
Application.ScreenUpdating = False
Applicaiton.EnabledEvents = False

Applicaiton.EnabledEvents = True
Application.ScreenUpdating = True
Application.CalculationMode = xlAutomatic
```

What does that code do? Again, it forces calculatiojn to manual mode, the screen to not update, and events to not fire.

#### ScreenUpdating

Screen updating is one of those things that seems fairly silly if you are coming from another programmign environemtn. In the vast majority of other programmign settings, it is very uncommon for all of your changes to produce an immdaite visual reuslt. ON the one hand, this is doccifult and on the other, it is awful for performance. Exel takes the opposite approahc since it is a user focused GUI that offers a scripting environemtn for uaotmatiojn. The default in Excel is that all of your comands will trigger hte normal render and refersh that woudl ahve occured had you manually made the change. IN practice, this can often prpduce a very cool effect where the cimouter is quite litterally doing all of the work for the user.

once oyu get over the appeal of this, you will quikcly be left with teh question of: how much ƒaster will my code be if I do not process all of these visual updates? The answer: much faster.

There are very few risks to disabling the screen. Teh biggest risk is that you forget to enabled it again (usally because of an error) and hten your usre will get odd behavior. In a lot of case, Excel will actually enable the screen anyways so teh actual risk here is minimal.

The onyl real reason to leave the screen active is in teh case where your automation can ataully be usefully reviewed by the user. In that case, you may consider leaving the scren active so taht the user can “wathc” what is happening. Some users like to see what hte code is doing.

#### EnableEvents

The more aggressive ooptiojn in this chapter is to disable events form firing. This has the ffect of inmprpving perfomrance bcause your code will be able to skip a number of potentially lsow steps. The downsdie f this approach ist hat someitmes you need events to fire in order to achieve a deisred result. This is espeically the case if you wrote the event handlers.

For events, there is one extra consderiong for when yu might disable them. If you possibly making changes to the Workbook or Workseeht in an event, you will liekly need to disable events whiel you make your changes. THe reaosn for this is to prevent an endless loop of your event handler processing the change you jut made. This is only relevant for a handful of event tpyes (Selection and Change are common) but it happens to be the case that this is a problem on teh most commonly used event handlers.

TODO: add an example of a full event handler that disables event hadnling

### Controlling the StatusBar

Did you know you can control the STatusBar? Did you know that the area at the bototm of you rscreen is called the Status Bar? This is an area that can be used to provide feedback to the user. It can be quite helpful for a long runnign calculation where you intentioanlly disable all of the normla feedback tha thte user receives (screen udpates, events, etc.). If you have done that, be aware that oyu cna still provider an updatin gmessage to the user.

```
1 Applicaiton.StatusBar = "Some message"
```

This functionality is best used when you have a mesurable way of prvoding progress feedback. This is commonyl one when you are looping through a large list of item sand processing each one in turn.

Depending on how quickly you can process a single item, you may chooes ot update the StatusBar to provider the progress to the user. Biggest issue to be aware of is that you can overload the STatusBar and create a sitation where Excel is slow processing all of your STatusBra updates. If this is your problem, you can usually rememdy this with a quick modulo function to only update the status every 10th iteration or similar.

TODO: add the general purpose status tracking code

## overview of adv processing

Advanced processing should include some recipe type sections that go through the more advanced aspects of working up VBA code.

This could focus on:

- Speed improvements and how to do it (disable screen, events, calculation) and how to undo it
- Working with arrays of values instead of outputting a cell at a time
- Cranking through an entire automated workflow without user interaction: creating new workbooks, worksheets, charts, formulas and then outputting it all to PDF
- Focus on the interplay of manual steps and code (sometimes you have to run part of the code to see what to do next; other times you can sit down and type the whole thing out)
- Cleaning up macro recorder code (some discussion about what works well/doesn't)
- How to avoid `Select` and why
- Using `DoEvents` to wait a set amount of time
- Using `Application.OnWait` (?) to do some thing at a regular time
- Parsing through existing formulas or values and manipulating with confidence
- Reading and writing to external files
- Working with the file system to do some processing
- Running through a folder or batch of files and doing something with each one
- Structuring code in a way that the different pieces can be called on their own
- Going through a workflow that involves using other office products
- Strategy for identifying cells using Styles and working through them; effectively a tag feature

The long list of sections here says that maybe there is enough code to put together a couple of "case study" type things that break down the development of an entire workflow. This could related to charting/processing or some other thing.

TODO: consider going through StackOverflow answers to see what the most common slightly advanced topics are that come up

### some thoughts on creating a workflow

If you are sitting down to create an advanced workflow, there are a handful of things to consider. The list that follows is not complete nor is it meant to include items that are always relevant. The problem with these lists is that with a general programming environmetn like Excel, it's impossible to describe everything to consider. HAving said that, I have built tons of these workflows and can comment on a handful of things that nearly always come up. The first item to touch on is the general structure/outline of a VBA workflow. This breakdown seems to always hold true.

Your VBA workflow will contain steps or sub steps that roughly be described as:

- Inputs
- Intermediate results
- Outputs

If your workflow is advnaced enoguh to include a number of sub steps built from other steps, then you are likely to find that this breakdown applies within and across levels of your workflow. That is, the outputs of one step may very well be the inputs to another step. The intermediate result from one action will be the input for another.

When thinking in terms of these categories, there is a useful distinction to make that is somewhat unqiue to Excel programming: do your inputs and outputs exist in the Excel spreadsheet or only in the VBA code? This distinction is meaningful because it helps you think about how much of your workflow is the automation of otherwise human tasks (which could still be done by a human) vs. steps that are purely programmatic and could not be replicated by a human. Where this distinction is most likely to show up is when you are deciding where and how to perform a calcualtion. In theory, all of the Excel spreadhseet could be done in VBA via the `WorksheetFunction` object. Doing everything in VBA defeeats a large part of the benefit that comes from programmign with VBA. It's easy to lose sight of this when you see a clean code-only solution to a problem, but realize that the greatest benefit to programmign alongside Excel is that you have a powerful, human readable scratch pad that lives alongisde your VBA.

As a comment, I have seen incredibly complicated workflows that involved detailed calculations of arrays taht were donely exclusively in VBA. The math was fine adn the results were generally useful. The problem was that there was no way to spot check a givne result without debugging code. This makes it nearly

impossibel for someone without VBA expereicne to validate your work. It also provides you job security, but ideally you'd gain security by other menas.

A better marraige of VBA and Excel is to utilize Excel for all of the tasks it's great at: caluclation, visual outputs, charting, page layouts and printing, and also the deep data oriented features (sorting, filtering, etc). Where VBA comes in handy, is wiring together all of these items into a coherent package that runs more efficiently than anything that a human alone could do. The best workflows typically take a very simply underlying spreadsheet and apply to a large number of items. In this way, you are able to spot check a singel result, verify the formulas, and investigate an interesting result. You are also free to just hit go and have 10,000+ reulsts streamed into a table for consumption. IF you find yourself looking for all sorts of tricks to avoid using the underlying Excel model for your programming, I'd strongly encourage to just switch to a fully programmatic language that does not have teh Excel UI. You will save yourself a ton of headache. If you are only aware of VBA and lookign to push teh envelope in terms of perofmrance, then that's an OK place to be. Just realize that there are better alternatives to Excel for high performance computing.

**inputs**

Back to the overall structure, there are inputs, outputs, and intermediate results. Depending on what you are doing, some of these aspects may just exist on/within the spreadsheet and be easy to overlook as an input or output. It's not until you wire up a more complicated workflow that you are forced to recognize the different pieces in a spreadsheet for waht they are. On the input front, there are a handful of items that should trigger your thought of "input":

- A file that contains some data to be processed, filtered, etc.
- A couple of columns in a spreadsheet that need to be processed adn then charted.
- 15 scattered cells that meet some criteria within a block of data
- THe contents of the clipbaord from anotehr program
- The formatting of a couple of cells

All of those items could be used as the input to a VBA workflow. Some of these items are odd to think about if you are coming frmo a notehr programmign environemtn. WHat does it mean for the formatting of a cell to be an input? Well Excel provides you with a rich Object Model full of metadata about all of the various cells of data. That metadata can be as useful as actualy strcutured data if there is a structure to it. I've seen it countless times where someone has methodically bolded all fo the cells of intereste in a block of data. That bold format is as good as some field called `Important = True` which could then be

processed in anotehr language. Instead of that flag, you just check `Range.Format.Bold = True`. This of course relies on an implicit assumption about how the data is structured, but this is common in the Excel/VBA world.

Excel also has a very strong UI which makes it possible to immediately solicit user input in a way that is not easily replciated coming from other languages. Where this shows up most frequently is when you start using the `ActiveCell`, `ActiveWorkbook`, `Selection` and other objects whcih are dependent on user input. In a lot of other languages you have to spend a ton of time pointing the program to teh correct file, or rows, or columns, or other items to process. In Excel, you leverage the fact that most people know how to select or activate items they want, and you can use that user input as an actual input to your VBA. This beocmes quite powerful wehn you are building utlilty code that may be used across multiple workbooks. This becomes much harder in other languages where the idea of a "open file" is far less well defined. You certainly cannot query the selected cells in an R data table.

#### outputs

THe next item to hit are the otuputs of a workflow. Very often, the outputs are obviosu because you had some task to complete with VBA, and the outputs are simply the results of that task. Where things become more complicated is when you string together steps and the output of one becomes the input for the next. When that happens, you often have to decided what intermediate format is best for hte transfer. You may or may not settle on a format that is easily human consumable. There are tradeoffs here that will be discussed later. The output of a workflow can be a number of things:

- A string, number, cell, row, column, or table of data that was processed by the VBA
- A chart
- A collection of shapes
- A worksheet that includes any of the items above
- A workbook that includes a number of constructued worksheets
- A change to the fomratting of a number of cells
- A change to the properties of a Range, WOrksheet or Workbook
- A new text file written to disk
- Some result output to the Clipboard
- Pages of physical paper if your VBA prints
- Some change to the filesystem or disk
- Some other program opened or run with specific parameters

This is a shortened list since the possibilities here are closer to endless. The idea however is that you cna effect a large amount of change from VBA and so your possible outputs can be quite numerous. A tpyicaly workflow will accumulate a large number of these outoputs indviudally and will then produce some final product which highlights some of those outputs.

#### intermediate results

When discussign intermediate results, it is generally best to limit your thoughts to whatever will live oonly in VBA. In that sense, teh question of intermediate results is: what programming constructs can exist without the user ever seeing them? Sometimes you need to determine the unique items in a list to do some processing. Do you generate that list of unique items in Excel somewhere? Or, do you determine the unique items using VBA and then outptu some result which may or may not include the full list of unique items. If you are doing the former, Excel provides a nice `RemoveDuplicates` function which will replicate the `Data->Remove Duplicates` functiojnality. This works great if you want the user to see the final list of values. YOu can also use a `Dictiojnary` in VBA to only store the unique vluaes from a list. In this sense, the `Dictionry` represents an intermediate value that may not be shown to the user. You will make this decision several times before you realize that you are decideing whether or not somethign should exist in VBA only. Often times, the decision does not matter, but for certain workflows it can make a huge difference.

An example is a multi step process where you might want the user to verify the calcualtions so far and correct any errors. This cna technically be done with VBA or Excel, but it is much easier to ask a user to verify an Excel spreadsheet than to debug the code and check `Locals`. If you need to do this verification step, then it mkaes a lot of sense to use an intermediate result that dumps back into Excel. In this sense, you've taken an intermediate reuslt and converted it to an output. That output may or may not be modified by teh user and it then becomes the input for the next step.

#### putting it all together

Having given a snapshot of hte options for inputs and outputs, it's worht commenting generally on how they all fit togehter. Your goal shoudl be to build a workflow that consists of steps that can all be described invidually and possinly run on their own. Your task is then generating these indidiual steps and determining how to wire them together. The most common approach to buildng these workflows is that oyu start with some single task and then the scope expands as the analysis expands. You can build the ultimate workhorse of a workflow initially, or you can adapt your code to the task as the task comes into view.

Depending on where you're starting and the defition at the start, you will determine how complicated to make things at the start.

It is very common to start with a single, straight-through workflow adn then build it out into Modules as the work expands. IN this way, you are constantly reevaluating the inputs nad output sof your program ot build the smaller blocks which need these defintions. In my experience, nearly all VBA workflows will take shape in this process eventually. It's quite rate to build a complicate workflow once and for all. Generally you start simple and end up with a full featured application at the end.

## overview of events

chapter will focus on using events to interact with the user and also to drive more functional spreadsheets

the major events to focus on are the `Workbook` events, including:

- `SelectionChanged` which can be used to track when the user clicks on something (do this if clicking in this row)
- `Changed` which allows for watching cells and doing specific things if the change was somewhere specific
  - using the Intersect technique to determine if the change was in an area of interest
- disabling events while making changed during events
- Application.OnWait event to trigger something to take place at a given interval

other ways to interact with events include via class modules with the WithEvents designation. These can be used to associate an object with an event and then wire up the code separate from the original macro code. This section might be useful for charting events if I ever get that code put together

Other areas where events take place is via the Ribbon and also via different controls that can live on the sheet. It would be good to discuss these as well.

### common events

When talking about events, there are a couple of high level details to touch on:

- Where the events occur? That is, which object owns the event and how do you hook into it?
- When the does event occur?
- What are you allowed or not allowed to do while responding to the event?

For a spreadsheets, the events tend to occur within the objects of interest: Worksheet, Workbook, others (TODO: is that right?).

THe most common events are assoicated with the Workbook and Worksheet. If you want to tie into those events, you can typically just add a new handler using the VBE. Thsi process is actually fairly straightforward. The task bcomes more difficult when you want to tie into an event but you are not certain which object will fire the event, or you want to trakc an event that takes place outside of your code.

THe main considering when working wtih event handling code is that you need to be sensitive to teh fact that you can enter an endless loop if you accidentally trigger the same event as the one you are respondng to. This is suprprisingly easy to do if you are tied into the `Changed` or `Selection_Chnaged` events which trigger quite frequtnly.

#### callbacks

ONe important point here is that all events are hanlded via callbacks. That is, you will create a Sub with a specific name and a speciifc signature which VBA then uses when the event occurs. This callback is important because it includes the information that you will need to discern what happened with the evnet. Each type of event can include its own speciifc parmaeters and your code can respond to them accoridnhly. This is important because if a cell was Changed, you will want different infomraiton than when a Worksheet was activated. In general, VBA is good about providing you usefu lparmaeters so that your events can properly determine what took place. Despite the good parmaeters, you will very likely need to include If/Then code to determine if your event needs further processing.

### specific events

For actual event handling code, it makes most sense to take a look at the specific events taht can occur and show some techniques for handling them.

#### Worksheet

THe Worksheet hsa a number of events which are commonly used. These include:

- Changed
- SelectionChanged
- Activate

These events roughly correspond to their name and are easy enough to handle. The idea wiht these is that you ahve a specifc cworksheet that you want to monitor for a speciifc event. In that case, you add the event using the VBE and then add the handling code.

The most common approaches for using these events is to trakc what the user is doing and then provide some adidtioanl functionality based on their actions. There are a number of reasons that you might want to respond to their input:

- Advanced usability where you allow the act of sleecting a cell or cells to determine that some macro should run. You could imagine on a certain sheet that sleectng a new cell may mean “please load more data baout this row” and the VBA responds accordingly.
- Validatjon fo user input. It is common to watch what hte user is changing and then determine if that change is allowed or not based on specific rules.
- Starting a new action with some user input. I have previously used editing a cell to trigger a goal seek on that cell. THis was quite nice because the VBA would undo my edit and then goal seek teh previous cell to a new value based on its formula. This provided a very slick means of trigger goal seek without having to collect furhter user input.
- Refershing some display. It’s possible you set calcualtiojns to manual and hten force a reclacualte each itme the Worksheet comes into focus.

For teh Worksheet, the typical flow is that you will create events at the WOrksheet level only if oyu know tha tyou will only wnat the code for a single Worksheet (or you are willing to duplicate it across Worksheets). If you want to have the same code run for *all* worksheets, you should look at the Workbook events which provide better views of the entire Workbook.

For some examples here, I will show you how to do the goal seek business along with a separate event which watches user Selection adn then processes the cells accordingly.

TODO: add the code and description for the Goal Seek event

TODO: add the code nad desc for some event which activates on Sleect

#### Workbooks

For Workbooks, you have a lot of the same events as for a WOrksheet. These events take the same parameters (TODO: is that right?) but allow you to watch for htat event across all Worksheets in a Workbook. Dpeending on what you are watching for, this either mkaes perfect sense or is a real burden wiht false events that are not interesting. You will hav eot determine the proper scope for your events dpeending on waht you need them to do. There are not fast rules here. The summar of possilby events then includes:

- Changed
- SelectionCHange
- Activated
- Opened
- BeforeSave
- Other Save events
- Closing

TODO: add some callback parametsrs above (and verify names)

TODO: review other events to see what may be useful

These events are similar to WOrksheets except that they give you additioanl hooks that only make sense for a Workbook, specifically realted to Saving, Opening, adn Closing. There events can be quite useful if you want to do some amount of processing befor ethe file is saved. One common approach I have taken is to delete extraneous data from a workflow spreadhseet to reduce the size and save time for a file. This can be used to great effect if your processing spreadhseet is generally pretty lean wihtout the data it processes. You could alos use this to delete a Chart that is large and having a big impact on file size. Once the file is opened, you can then use VBA to recreate the chart.

IN some cases, these events are used for that type of exmaple where it seems like a lot of work to save some amount of hassle. Oftentimes, this is the case. You can spend a lot of time with event code to make it do exactyl waht you rwant. Sometimes for a user focused spreadsheet, however, this is the level of detial tha tis requried to ensure that eveyrhting will work everytime for everyone.

TODO: add an example and desc for a data removal VBA

#### Application

There are also a couple of events taht exist at the Application level. These include:

- OnWait
- TODO: any others?

Application.OnWait can be used to trigger an event at some point in the future. This can then be used to trigger a block of code which runs at an interval by having the triggered code start a new event in the future. In this ways, you can use VBA to start a timer which executes every so often.

TODO: add the OnWait code for a timer

TODO: find another examples

### common patterns

THere are a number of patterns that are very common with Events. These patterns typically exist to avoid causing a problem or to avoid extra work where possible. Most VBA is not performance critical, but it is possible for an event to be called hundreds of times for a given chucnk of code. Since this is true, you can start to have an immeidate impact on perofmrance if your event hanlding code includes a number of unnecessary steps. As a side note, this is a good reminder that when trying to speed up code, you will nearly always do better to add `Application.EnableEvents = False` before your performance critical code; this assumes that your VBA does not rely on events firign to function properly.

#### Intersect

The first is the `Intersect` technique to determine if a Range that was affected by an event was a Range of interest. With this approach, you define a Range which includes your “interesting” cells. You then do a `If Not Intersect(rngEvent, rngTarget)Is Nothign` to see if the interesection of the callback Range adn the desired Range overalp. If they overlap, yhen you typically execute some code. This allows you to quickly filter out Ranges which have hcanged but are not reelvenat ot ahtever code you need to run.

TODO: add a code sample here

#### Application.EnableEvents = FAlse

One of the biggest gotchas with Events is that oyu can quickly and accidentally create an endless loop of Event code running if your event handler is able to retirgger the original event. This is quite common if oyu are looking at the Selection and then change the selected cell. The same can happen if you are using an event to watch for a change and then you rrespond wiht additiojnal cahnges. Both of these accidents are so common, that you should seriously consider alwyas disbaling events in your handler. It is quite rare that you will need an otehr event to fire following your own processing.

The main thing ot remember here is that you really need ot enable events again. Excel will not do this for you. You can create odd situatiojns if you have an error in your code that goes unchecked. This sitautiojn can mean that events are disabled. For really sensitive, user focused code, you should add a proper error handler adn enable events following that.

To handle this event, the code is quite simple:

```
1 Sub EventHandler()
```

```
    'disable events
    Application.EnableEvents = False

    '' do some stuff

    'reenable events
    Application.EnableEvents = True
End Sub
```

### more advanced events

This section will focus on using events in more advanced settings. In particular, the focus here will be on using Class Modules to allow for events to be attached to arbitrary objects that are not necssarily known at compile time. This is an advanced approach that is typically not required. Where it may be hepful is if you are building library code that needs to work in a range of settings. It may also be needed if you are trying to attach events to a WOrksheet that will not exist until some other VBA has been run. In this case, you will attach to the same events as above, but you will add the event after the Worksheet has been created. It's owrth noting for that specific exampl etha thte Workbook can handle a large number of events on teh WOrksheet and will work for Worksheets that were created later.

TODO: add a section explaining how to use WithEvents

TODO: add some examples of attaching evnents to a new Worksheet.

### SetUpKeyboardHooksForSelection.md

```
Public Sub SetUpKeyboardHooksForSelection()



    'SHIFT =    +
    'CTRL =     ^
    'ALT =      %

    'set up the keys for the selection mover
    Application.OnKey "^%{RIGHT}", "SelectionOffsetRight"
    Application.OnKey "^%{LEFT}", "SelectionOffsetLeft"
    Application.OnKey "^%{UP}", "SelectionOffsetUp"
```

```
    Application.OnKey "^%{DOWN}", "SelectionOffsetDown"

    'set up the keys for the indent level
    Application.OnKey "+^%{RIGHT}", "Formatting_IncreaseIndentLevel"
    Application.OnKey "+^%{LEFT}", "Formatting_DecreaseIndentLevel"

End Sub
```

## overview of user forms and input

Some sections to include here:

- using the Application.InputBox to get different types of input (especially Ranges)
- creating a UserForm and wiring up events
- interacting with the spreadsheet from a UserForm
- a quick summary of the different pieces, when they're useful and specific/common properties

at the end of the day, there is not much too special about forms other than knowing what the different pieces are

this should be a relatively quick chapter since most of the reference material is elsewhere

### introduction to UserForms

This chapter will focus on how to use UserForms to create interface that allow the user to interact with your VBA code. UserForms can be used from anything to simple text inputs to very complicated forms. There is really no limit to what you can do with UserForms, but at some point you will hit the limit of what you want to do inside the VBE. Some folks will push the limit and develop fully featured programs in Excel. I'd highly recommend you not do that and instead use UserForms to augment a good usage of VBA with useful interactivity.

When considering whether or not to use UserForms, there are a handful of pros and cons to using them.

Pros to using UserForms

- Provide a form to display/edit user input independent of Excel
- Provide for much better interaction with the user via "normal" programming events click, keyboard input, etc.

- Allow your program to collect several pieces of information before completing an action, especially if some real time process of the information is useful
- Provide a form that can “sit” on top of Excel and provide helper functionality or application specific functionality

Cons to using UserForms

There are a number of alternatives to creating a UserForm that are worth considering before committing to UserForms for a specific application. Those alternatives and the cons against UserForms include:

- Editing the code of a UserForm can be a bit of a nuisance because you have to flip between design and code views
- The InputBox provides a simple way to collect a number of different input types without needing to create your own form. For simple inputs (including Ranges), the InputBox is typically simpler and more consistent across applications
- Some types of inputs (including lists) can be easier to manage in a non-VBA context by using default Excel features. For example, you do not need a ListBox on a UserForm if you can put a Table somewhere in the spreadsheet.
- If you are trying to use some form of version control on your source code, UserForms are very difficult to manage and version.
- If you want to provide buttons to perform an action, the Ribbon can be much more robust. Of course editing the Ribbon can be a different pain, and I’ve gone the other way on this point before.

Once you’ve decided you want to use a UserForm to provide for user input/interaction, there are a number of areas that are important. Those areas from start to finish are:

- Adding a UserForm to an existing Workbook or addin
- Using a Sub to make a UserForm show up using a keyboard shortcut or button or some other means
- Adding controls to UserForm and using those inputs to drive VBA
- Working with a UserForm to process input and update the UserForm
- Using the non-control events to provide interactivity

### creating a UserForm

Creating a UserForm is a simple process: open the VBE and then right click to Insert -> UserForm. This will give you a default UserForm that is blank, has a default name, and is a default size. If you’re lucky, this will also open the UserForm for editing and show you the toolbox which provides controls to edit. Once the form is created, there are a couple of things which you should do immediately, before you forget:

- Change the name of the form to something more useful than UserForm1
- Change the `Caption` on the form to something better than UserForm
- Consider changing the `ShowModal` property if you know you do not want a modal dialog

Once those items are done (or decided against) you can start adding controls to the USerForm and changing its size.

TODO: add pictures of the steps

### making that USerForm show up

Once your UserForm is created, there are a couple of ways of showing it on screen:

- Run any code from the VBE that is contained within the form. This will show the form.
- Create an instance of the form somewhere and show it

For those two methods, the latter is really the only one that will work for user applicatiojns or other “real” uses. If you are simply testing or doing things for yourself, then hitting F5 in the VBE may not be a large ask.

For the former, see the code below for an example of how to show the form.

```
DIm frm as UserForm
Set frm = New UserForm

frm.Show
```

### adding controls to a UserForm and wiring them up

Once your UserForm is created and the defaults are changed, your next task is to do somethign useful with the form. To that end, you will quickly need to add controls to the form and then wire thsoe forms to useful actions. Adding the contorls to the form is a straight forward process: show the Toolbox and then drag the items onto the form. Once you have dragged out a button, text box, and possibly a ListBox, you can simply copy and paste the previous items adn avoid the Toolbox all around. There are a handful of controls that are not in the Toolbox by default. My strong advice here is to not use those controls if you are deploying this addin. INeveitably, some user will not have the OCX file or whatever is requried to make it work. Just pass. Having said that, you may need to add a Date picker or possibly a RefEdit which are not included here (TODO: is that true about RefEdit?).

Dragging a control onto the form is fairly easy compared to the actual task of making a control do somethign useful. Below is a quick primer on how the idffernet contorls work, which properties are important, and will give you a guide for accomplishing 90% of waht can be done with forms.

#### CommandButton

The CommnadButton or simply button is one of the most common controls to use. Its use is simple, known to everyone, and easy enough to program against. A button does one thing: get clicked. The event you want to know about is the `_Clicked` event. Fortunately, the VBE will atuomtallica create and wire up this event if you double click the button on the Desinger versiojn of the form. This makes it dead simple to create the button code that you want: just double click the button.

Note that the default event will be created with the current name of the control. To avoid this, you need to change the name of the button before you create the event. Be aware that VBA and the VBE are not that smart with respect to naming things nad wiring up changes. If you change the name of the button after you create the event, your event will not work. You should not chagne teh button name after creating the vent (or plan to recreate it).

Other properties of the button that might be used:

- Value - will change the text that is dispalyed (TODO: is this right?)
- Enabled - will change wheteher the button can be pressed and will change the visuals. Useful when you want to show that an option could be possible but is not currently allowed or enabeld
- Formatting and other visuals - you may change this on the proeprty editor but it is far less common to modify the fomratting once you are running. It can be done but is not common.

That's it; buttons are simple.

#### CheckBox and Radio

The CheckBox and Radio are cousins (or siblings?) of each other and will be dealt with at once. They allow for a Boolean selection of an option. For the Checkbox, you are allowed to indicate the on/off state of a given button. For a Radio, you are allowed to indicate the on/off state for a singel option *within a group of options*. The main thing to note about the Radio is that by selecting one item, you will uneslect the others. In this way, the uses of these two controls maps naturally to the tasks you are likely to see.

Aside from the Name, the main items to deal with are:

- Clicked event - just double click ot get htis one
- Value - note you get this by defualt usign the name, but it will include a Boolean of the selected state
- Enabled - can be used to disable the control

That's about it. You can change the formatting and other stuff, but these items typically exist to get an input and get to the real work. They are very common when you are providing options to the user or otherwise want to direct downstream If/Switch statements.

Beware that the Click event may be changed multiple times depending on how it was triggered (TODO: is that right?).

#### TextBox

The TextBox is another simple one: it provides a means for the user to provide some text input. They work great for a range of things including input and output, although input is more typical. The idea is simple, the user provides a string and you use it somewhere. The properties to know:

- Value - this gets or sets teh value that is displayed
- Enabled - can be used to disable the control (TODO: same as readonly?)

In terms of events, the main one to watch for is the KeyPress (TODO: or changed?). The idea is simple, if oyu want to track the input of hte user, you tag along for that event and can respond to their key presses. The common uses of htis are:

- Close a form or clear an input when ESC is pressed
- Do some action when ENTER is pressed
- Provide some form of vlaidation or checking as the user types to either modify their input (e.g. ignore dashes) or otherwise update the UI based on tehir input.

TODO: add some addl onctent here baout the event and its callback/parameters

That's it.

#### ListBox

The ListBox is one control that has a number of options and a means of using it that are less obviosu than the other controls. It's a shame really that the ListBox is so unintuitive in VBA beucase it is qutie powerful and other programming langugaegs have handled htis better. THe idea behidn a ListBox is that it provides a list of items whose use can vary according to what you wnat. Some common applications include:

- Allow the user to select from one or multiple options in a list
- Provide some output to the user (and possinly then use that output as the input for next step)

THe input/output decision here is somewhat critical because the thigns that will annoy you about the ListBox break on this point. If you are collecting input, then really you have to also deal with output because at the end of the day, you ahve to put somethign in teh ListBox in order for a user to select it. ONce you've handled the output stuff, then determinign whcih items have been sleected by the user is straightforward enough. Therefore, covering the output part is a good starting point.

To put items into teh ListBox, you need to modify the List collection on the object. There are two ways to do this:

- Directly, via the List boject
- Indirectly, usign the `AddItem` command

Either way you go, you ahve a couple of decisions after adding the item: what text do you want displayed for the item and do you want multiple columns? If you are dealing with a singel column, then you can simply add the text in the call for an addition and that's all. IF you are working with columns, then you will need to do two things:

- Set up the columns (using the editor or via commands) (TODO: add pictures or code here)
- Call the commnad to set the fields usign the row and column nujmber (TODO: add some code)

ALthough I have descibed a simple process here, oftentimes, you will deal with something that is more complicated. THe issue comes when you want to maintain some reference to an obiject but you are reuqired to use a string for display purposes. This means that you need some menas of maintaining that refernece back to the object. There are options for dealing with this:

- Rely on teh index of the objects matching (and not changing) and simply use the row index
- Create a Dictionary that stores the link between the string and the object
- Use some other object or Colleciton that can reference the object back to teh string
- Serialize teh object into the ListBox value (if multiple fields, join with a | or similar)

Each of those approaches has its pros and cons, btu the main idea is that you are often forced to deal with somethign that is typically much easier in other languages. My general appproach is to rely on row index if I know that changes are not possible. This is common for a lot of code isnce yout ar elikely to contorl both side. If that is not ideal, then you can typucally find some way to store a referenc ebetween the display value and the object usign a Dictionary.

Once you hav eth einfomration in the ListBox, you can simply iterate the `Items` by index nad chekc the `Selected(index)` property to see if the item is sleected. Note that if you do not allow multiple selection,

then you can also use the `SelectediNdex` property (TODO: is that right?).

TODO: add some code here to demonstrate iterate through a ListBox

Although this section has the most text, the ListBox is not always a pain to deal with. Typically they are much better than the alternatives (like using the Excel spreadsheet somehow) but require that you remember some boilerplate for accessing and changing items.

#### Other Controls

There are a couple of other controls that you may see that are summarized here:

- Label: these don't do much other htan provide some fixed text when could be changed later (I rarely ever do it)
- RefEdit: this contorl tehcnically allows you to select a Range from Excel. They are quite buggy. Depending on you main goal, you may od much better to use `Application.InputBox(Type:=8)` to access a Range.
- Tabs: these can be heplful for organizing a complicated workflow. You will find yourself wanting to change the active tab and possibly limit access to later tbas.
- Wells?, whatever it's called, there allow you to Grouo controls. These may be requried for a Radio to work like you want (if you have mutliple sets of Radios on a singel form).
-

### doing actions on a UserForm

Section will focus on performing commands on a UserForm without leaving the UserForm. This looks mostly like normal forms programming.

Programming on a UserForm is one area of VBA that is largely indepednet of Excel. Yes, you are able ot access the Object Model from a form, but most of the programming on a form is simple related to the form. This is also the one area of VBA where if you have done it before in another language, your expereince will transfer nearly 1:1. Forms programming is largely all about building a good UI that meets your needs. You can do a lot with VBA in terms of events and other sorts of dyanmic programming, but most of the itme you just make the form and get on with the real work.

Some of the ocmmont things you will want ot od when programming on a form include:

- Creating event handlers
- Parsing and respoding to user input
- Populating data in teh UserForm from the Object Model
- Accessing Properties of controls to change or use

**Event Handlers**

Event Handlers are at the core of User Forms and making them useful. To be clear, your Form will do nothing without events. You could it to display static content from the designer mode, but it will do nothing useful. To make your Form become useful, you add Controls to it and then add Events to those Controls. Event Handlers are the glue (or wires) that take the actions perofmred on Controls adn direct them somewhere useful. Evnets control everything from Clicking, Loading, Typing and everything else. Each COntrol has a unique set ofo events depending on what it can do, but in general, there's a bit of overalp between different controls.

To add an event handler, there are a couple of options:

- Double click on the Control in Design Mode, and oyu will get the defualt event handler created
- Go to the code view, and select the Control and then Event you want from the drop downs (TODO: add image)
- Type the Event handler based on teh named of the Control adn the event you want

If you know the defualt events, then option 1 is as good as teh toehrs. IF you want to see a list of events beffore creating one, then you will go with optojn 2. You will pretty much never type the event handler out by handler unless you are copying it from somewhere else.

ONce you have created the ahndler, you simply add hte code that oyu want to fire in teh event. One good tip here is to use the event handler to call other Subs. It's a good habit to not put logic or other execution based code into Even thNadlers. The reason for this is that you may want to perfomr the same action from multipl events. Putting the code in a handler makes it idfficult to resue the code becasue som ehadnlers have parameters nad other details that make it hard to arbitrary call them. Of course, I regualrly put code into event Hnadlers, but at least I know I shoudl avoid it. I am constantly reminded of why to avoid it when I ahve to extract code from one event to put into a SUb to call from another event.

One important note about Event handlers is that the hadnler can have some number of parmaeters that are included in teh handler signature. These parameters ar etpyically used to pass along infomraiton related to the event. For ecample the key press event contains the key code of the key that was pressed.

The Click event however has no paramteers. The presence of apramters is easy to check when teh VBE creates the handler for an event since it will give the parameters.

TODO: given an example of using Handlers?

TODO: include a blurb about the Initialize event (if it was not addressed ealrier)

#### Processing User Input

User Input on a User Form is one of the most critical aspects ofm aking them. IT is less common to use a Form purely for output of information (although that is done). tpyically, you use a Form to provie input in a format that is easier to use than teh default Excel itnerafce. There are a ahdnful of Controls which are viewed as collectors of user input. You can then process tha tinput in an Evne tHandler or in other code which accesses hte properties of the Control. Those common controls are:

- TextBox: Works great when you wnat to control a single value from teh user. You can then parse the string into a number or wahtever else oyu need
- ListBox.: Works great for allowing the user to select form a list from still beign able to see multiple items in teh list. Also uspports multple seleciton
- COmboBox: Same as ListBox but the contorl collapses to a single line when you are not selecting items. Does not allow for multiple selection.
- CheckBox or RadioButton: Allow the user to make a sleection between choices while seeing the choiecs
- Button: Really allows a user to unput a single click
- RefEdit: Not recommended but it allows you to select a Range from the Spreadsheet.
- TODO: any others (number bumper?)

For each of those Controls, you have a number of evnets which can be used to process the input as it comes in, or you can process the Properties of the Control once other code is running. One common pattern is to allow the user to input data into a number of TextBoxes, hit a button to run some action, and ten process all of hta tinput in one step after hte button press. Anotehr way to do the same thing would be to process and vlaidate the input as it comes in, providign an error message if bad data was input.

For most of the Controls given above, you will find a `Value` property which gives either the Text of the Contorl or the selected state. The one exception to this is the ListBox which requries a little more owrk to get the Selection. For the ListBox, you need to ierate the items adn check if hte `Selected(index)` property of the ListBox is `True`.

TODO: add an example of usign Vaule

Once you have the user input, it will typically be a `String` or a `Boolean`. To do somethign with these niputs, you will need to parse them into the deisred types if not a string. The most ocmmon transformation is to parse a number from teh string. This is done with `CInt` or `CDbl` which will *C*onvert a String into a Integer or Double. You will get an erorr if the string was not parseable. If you do not need a number, there are a couple of other "C" functions:

- CBool
- CDate
- CErr
- TODO: add others, and descriptions

#### Accessing the Excel Object Model

From a UserForm, you have full access to the Excel Object Model. Thsi can be very handy if you are trying to access informaiton from teh USerForm to determine what infomraiton to show in teh Form. It can also be helpful if oyu want to make changes to teh underlying spreadsheet from a USerForm without leaving the form. Both of those options are very common and very easy to do with UserForms. In general, any code that can run without a USerForm present can be run with a USerForm. There are some limiations when it comes to teh user's ability ot Select items with a Fomr visible, but you are not limited in calling teh same commands from VBA (TODO: is that right?). Teh exception ehre is that if the form is `ShowModal = False` then the user is able to make selections while the fomr is bisible.

There is no real limit to waht you can do from a SuerForm. A couple of examples to give you a feel:

- present a list of all open Workbooks so that they user can sleect which one that want to process
- Create a form that can process all of the slected CHarts.
- Present a ListBox with teh unique values from all of the AutoFilters that are active. Allow the user to selectively remove or chagne those filters without having to use the normal drop downs.

#### Accessing control Properties

THe final piece of Forms programmign is somewhat meta: allow the UserFOrm code ot change the USerFOrm. There are a couple of obvious reaosns you might want to do this:

- Change the position of the USerFOrm (center on start)
- Enable or disable a buttn or other control based ons ome input. YOu can extend this to making things vivsible or not as well.

- Change teh text, format, or other visual detail of a Control based on some other state or user input.

TODO: add the code for cnetering a UserForm.

IN addition tot hose simple concners, you also have hte ability ot danmically create controsl on demand. This makes it possible to add/remove controls ot the USerForm as needed. This can be helpful if oyu want to create Control based on some proeprt yof the Worksheet but where you may not know how many times to do it in advance. For example. maybe you want to prvoide a LIstBOx with unque values for each column that was slected. IN advnace, you may not know the column coujtn so you need to create ListBoxes on demand. This can be done with UserForm programming.

TODO: example fo create a Control from scrathc

## overview of UDFs

This chapter could focus on a couple aspects of UDFs. High level topics:

- Using them to return simple info that is hard to get otherwise (e.g. Range.Formula)
- Using them to hide complicated logic that could be done in a formula but would be a mess
- Using them to do things that are not possible otherwise

UDFs are a great way to extend Excel with some common features

Could include some examples of where this has been done in bUTL:

- String processing is much easier with UDFs instead of formulas (concatenation)
- Doing logic that might otherwise require an array formula
- UDFs are a great way to simplify formulas for conditional formatting
- UDFs are a great addition to a personal addin where the functionality is available without copying/changing formulas

Some technical points to hit:

- THe pitfalls of using Ranges outside of the ones referred to
- Making a function Volatile and what that means

### introduction to user defined functions (UDFs)

This chapter will focus on using VBA to create user defined functions (UDFs). This area of VBA is so-named because it allows you to add functions that are callable from the spreadsheet. Once you're familiar with

VBA, you'll recognize that there is no difference between a normal VBA Function and a UDF. The only difference is that a Functions "becomes" a UDF once it is called from the spreadsheet. Having said that, UDFs are still incredibly powerful and can be an incredible time saver when working with a spreadsheet. The power of UDFs is that there are very few limitations to what you can do inside a UDF. This means that you can do complicated tasks from a single function call in Excel. Contrast this with the mess you get when doing complicated things with normal Excel functions.

This chapter will hit the major topics related to UDFs including:

- Debugging them
- Working with variable types, especially parameters but including outputs
- Limitations of UDFs – what you cannot do
- Limitations of UDFs in addins – must have the addin
- Different applications of UDFs
  - Simple things - string functions, etc.
  - Complicated things
  - Duplicating Excel functionality in a simpler package
- Understanding volatility
- Understanding Ranges and how they relate to your function being called
- Hiding a VBA function from UDFs, and using the Option Private Module
- Building more powerful UDFs with ExcelDna

## getting started with UDFs

This section will focus on how to get started with UDFs. This will be a crude overview of VBA Functions and then a discussion of getting them to execute inside the Excel spreadsheet.

### a primer on VBA Functions

Check the start of this book for a proper review of VBA Functions. The key points when using a function to execute as a UDF are:

TODO: link to section

- Function needs to be declared as Public
- Function needs to have a return type that can be processed in a cell (has a Value)
- Function needs to return something

- Function needs to be created in a code Module (not in a Worksheet or Workbook object)

Once you've met these criteria, you will be off and running. Typically a UDF will not work for one of those three reasons above. In particular, I regularly forget to declare the function Public and put it into a module. It's typically easier to remember to set the return type, but it is possible to forget ot actually return something from the function

The best indicator of whether or not these steps have been followed is to type your UDF into a spreadsheet and see if it is recognized. Excel does a very good job of identifying valid functions and offering them in the autocomplete.

Tip: Sometimes it is difficult ot remember the parameters that a UDF takes. You can either use the function input helper (TODO: add details about that) ro you can use the shortcut CTRl+SHIFT+A which will populate the names of the parameters into the UDF. Note that these are unlikely to be valid inputs to the function, so you will actually need to update the parameters. If you use descriptive names for the parameters (which you should!), this is a very helpful shortcut.

TODO: add an example of a very simple UDF here

### some simple UDFs

This section will focus on the "simple" UDFs. It may sound silly, but there are a handful of surprisingly useful UDFs that are just a single line of code. In general, these UDFs are used to return some information about the spreadsheet that you'd prefer Excel simply have a function for. In later versions of Excel, some of these gaps have been filled (e.g. obtaining the formula for a cell) but sometimes these gaps still remain. In addition to one-liners, there are a large number of simple UDFs that exist to replace a more complicated Excel formula. These types of UDFs can be much easier to read and debug/test than a complicated array formula for example. The final group of UDFs that comes up frequently is string processing. Excel provides good functions for manipulating strings, but these cna be a complete pain without the use of helper columns. A simple UDF can hold a variable which eliminates a lot of the need for helper columns via traditional formulas.

Before committing whole hog to UDFs being the best way to do things in Excel, it's important to remember that there are downsides to UDFs. The most important is that if you want the UDF to live with the workbook (and not in an adding) then you are required to save the workbook as macro enabled. This can be a deterrent to using them in certain environments. THe other thing to remember is that UDFs can often be a crutch for not actually learning how to get the most out of Excel functions. It can be easy and tempting (especially for an experienced programmer) to start blasting through a spreadsheet with UDFs instead of

learning how to do something “the Excel way”. Depending on your work setting and who else will see your workbooks/code, this may be a bigger issue for some people.

#### Common reasons for using a UDF

There are a number of consistent spots where I will use a UDF instead of fighting the Excel formulas. Thee typically fall into a couple of categories:

Excel formulas can be quite complicated/repetitive if need to store a variable

Certain valuable pieces of information about the cell or a Range are not available via functions

Some things are just much easier to do with VBA than with Excel

#### examples of simple UDFs

TODO: add some examples here of different types

### limitations of UDfs

This section will focus on the aspects of UDFs where you are limited. There are couple of key things to remember here:

- A UDF is not allowed to change the Workbook, Worksheet, or a Range – no side effects are allowed
- A UDF will only update if the cells it refers to change
- You can mark a UDF as Volatile, but this may create other problems (namely speed)
- UDFs are allowed to use global variables but you can wreck this process by having errors while they execute
- UDFs inside an addin can pollute a spreadsheet that might be used by someone without that addin
- You can debug a UDF but not by using the Evaluate Formula option that might be familiar to more people

#### no side effects

The biggest temptation of a UDF is one of the few things that is not allowed – you are not allowed to have a side effect from a UDF. This generally comes up when you want to change something about the Range that

the UDF is referring to or being called from. You think: “I’d just love to color this cell red if the UDF detects some state while executing”. This thought comes up because it’d be nice to have the UDF update when called and even better if oyu can avoid dealing with conditional formatting. Alas, this is not allowed. The UDF must execute without making a change to the spreadsheet. This generally makes sense if you think about how Excel goes about calculating the spreadsheet. It makes a map of how cells are related and then proceeds to calculate the values in an order where each cell that depends on another is calculated in the precise order that is required. This process allows Excel to complete as fast as possible, without errors, and while using as many CPU cores as possible. If your UDF is able to change the spreadsheet after Excel has determine the order of calculations, then it becomes impossible to ensure that the spreadsheet is still correct. Because of this, Excel does not allow sde effects from a UDF.

The other aspect of this limitation that comes up often enough in practice ist hat you cannot use a Worksheet function that modifies the spreadsheet even if you intend to undo that function. For example: I have attempted to use the AutoFilter inside a UDF in order to determine how many times some condition showed up in a table. This is not allowed even though I intended to undo the AutoFilter before returning from my UDF. This limitation also applies to Copy/Paste and other common functions.

**when does a UDF update**

The next limitation to consider is that a UDF will only update when the Ranges it refers to are changed. This is related ot the dependency tree described above. Excel will only call your UDF if one of the cells that it directly depends on it is updated. This is important because you have access to the entire Workbook inside a UDF so you can create a situation where your UDF *should* update something, but it doesn’t because it does not know that it should have been updated. This si discussed later, but the quick way around this limit is to mark your UDF as Volatile. See the warnings later related to this.

A common example of when this sort of issue pops up is when you are using a reference to a Range inside the UDF that is computed only inside the UDF. For example, you want to do some statistics for a single Range that are dependent on a larger Range of data. YOu can write a UDF that takes the single cell as a parameter but then compute the larger Range inside the UDF without having to refer to it. Maybe that larger Range is a mess via normal Excel so you’ve skipped that step. Well, be aware that your UDF will only calculate for the even cell if the cell it refers to changes. This means that the larger group may change – and invalidate your current result – but if the single cell stays the same, then your UDF will not update that cell.

This same issue pops up if you are using properties of the Range that are not a part of the calculation model for Excel. That is, there are some changes which will not trigger a recalculation from Excel. These

are typically related to using the formatting of a cell in a UDF. A very common example is returning the Range.Text from a UDF so that you can get the value exactly as it is displayed in the spreadsheet. If you change the format of the cell, you are not guaranteed ot have the UDF called updating your UDF value.

**using Application.Volatile**

Mentioned above, there is one surefire way to ensure that your UDF will be called whenever there is a change anywhere on the spreadsheet: mark the function as Volatile. This is done by calling Application.Volatile somewhere in your UDF. TODO: is this right? Once you have made this call, your UDF will be called anytime a calculation is done. THis also means that anything that depends on your cell will be recalculated every time. There is a huge upside to using Volatile UDFs in certain instance: you are guaranteed that they represent the correct value. THe downside is that your UDF is being called constantly which means that if it is slow, your entire spreadsheet will be slow. If your UDF is littered across 10,000 cells, it will be run 10,000 times even if only a single cell changed. It is easy to underestimated how much this can slow down a Workbook. Having said that, sometimes speed is not a factor and you just want things to eb correct.

There are other functions (INDIRECT and OFFSET are the main ones) in Excel that are volatile, so it is not some awful thing ot do necessarily. You should mark something as Volatile however only as a last resort or possibly as a first resort if you're just punching something out.

To avoid using Volatile, you may be able to have your UDF take an additional parameter to ensure that it is on the calculation chain of all the cells it depends on. Note: you don't actually have to use the parameters for anything, but if they appear in the UDF call, it will force Excel's calculation tree. Continuing with the statistic example from above, if you know that all of the data that could change is in columns B and C, you can simple send B:C in as a parameter to the UDF. This ensures that a change in those columns will force the UDF to call. You can then continue to compute the Range using your more complicated logic. This is somewhat wasteful and means you have extra parameters which don't do anything, but it can be a cleaner (and faster) solution than using Volatile.

**beware of global variables**

VBA allows you to declare a variable outside of any Sub or Function definition. These are typically called global variables because they can be accessed from any code. This means that you can create some variables in a Sub and then use them in subsequent UDF calls. A good example is loading up a database of information and then using that information inside the UDF. This cna be nice because then you do not

have to load the data every time you call the UDF. I've used this effectively when doing unit conversions with UDFs.

The downside to this approach is that it seems to be relatively easy to corrupt those global variables if you have errors while the UDF runs. I've had it happen where that loaded database becomes corrupted somehow and then all of the dependent cells start to fail when their UDF is called. This type of error can be quite difficult to track down because it may not be obvious why the variable was corrupted.

**beware of UDFs in addins**

A personal addin is a great way to organize helper code without constantly created macro enabled files to use the code. For Subs this works great because there is no lasting trace that a SUb was run, at least in terms of code in the file. For a UDF however, your UDF call will be a part of the spreadsheet. THis does not force the spreadsheet to be a macro enabled one – which is great – but it does mean that anyone using the spreadsheet needs access to the UDF code. This creates a problem when you get comfortable using UDFs in an addin but then save the workbook with them in there. You have effectively "polluted" the workbook with addin UDF names which may or may not be available to others. This is fine if the addin truly is critical to the workbook, but it can create a mess for others if you're using UDFs for your own help and make a spreadsheet that others cannot use.

The solution to this problem is to simply save the UDF as a Module in the spreadsheet, but this requires you to save the Workbook as macro enabled.

A rule I like to follow is simply: if I know that a UDF is required for the spreadsheet and that UDF is currently in an addin, I force myself to move the code into the Workbook and save as macro enabled. This can be a pain, but it's all too common that a Workbook is saved with a UDF from an addin, that addin changes or becomes unavailable, and now your Workbook is broken. It's best to avoid this scenario especially if you work with others who are not macro savvy.

**debugging UDFs is different**

Most folks are familiar with the "Evaluate Function" feature of Excel which will help you walk through a function's evaluation in the order that Excel evaluates things. This can be incredibly helpful for array formulas where it's not always obvious the order Excel will do things in. Your UDF will also be evaluated in that feature, but it will not step through the logic of your UDF. This might seem obvious, but it's worth

mentioning. IF you want to debug the logic of your addin, you need to set a breakpoint and actually debug the code. See the later section on this for the details.

TODO: add link to that section

### managing the parameters and types of UDFs

This section will focus on a topic that is quite nuanced but can have a large impact on how reusable your UDF code is. The focus here is on how to specify the type of the parameters and possibly the return of the UDF.

The reason things get tricky is that Excel is able to feed a wide range of object types to a UDF depending on how it was called. The common types to see are:

- Range
- Array/Variant
- Double/Number
- String
- Date
- Error

The most common ways to call a UDF are

- Use a Range reference UDF(A1:B2)
- Use the result of some other operation UDF(5*A2). This can result in different object
  - Array formula gives an array
  - Math might give a number
  - String formulas will give a string
  - IF or CHOOSE might allow for multiple options depending on the result

Given this wide range of choices, it's important to consider how you intend for you UDF to be called and what types of inputs you want to be able to handle. You can choose ot be as loose or as restrictive as you want on the parameter type, but this will have an impact on usage. If you go the loose route, you can call everything a Variant, but then you lose the utility of Intellisense as you are programming. If you go the strict route, you gain Intellisense, but might make your UDF fail on a simple case that it should be able to process.

As an example, let's say you've written a UDF that simple squares the number that it is fed. If you specify the parameter of this as a Range, your code will work fine with usages like UDF(A1), etc., but it will fail if

someone sends in the result of math UDF(5*A1). This is odd because assuming that A1 is a number, there is no reason that you cannot square the result of that. Instead however, you will get an error that the result of that math (which is a Double) cannot be converted to a Range and your code will error out. For a simple example like this, it makes the most sense to declare the parameter as a Variant and just rely on the Value being correct.

TODO: add code for that example

Things are fixed simple in that case, but it quickly becomes an issue when you want to handle different types of input. Maybe you are making a function that will concatenate an array of strings together. What happens when you only get a single string as a String instead of an Array containing Strings? Most likely, your code will fail in this instance, unless you've built int eh proper checks on the type. In this case, you will likely need to take a parameter of Variant and then do the checking to see how to handle it.

TODO: add an example of string concat code that works

The most common spot to see this sort of issue is when deciding whether to deal with a type of Range or Variant (to handle an array). It is nice to work directly with Ranges and avoid the Variant, but this will make your code weak against someone who wants to use an array formula to call your UDF. It typically does no take much work to process an Array, but it helps to design things from th start like that.

TODO: add before example of UDF using Range

TODO: add after example of that UDF using a Variant/Array instead of the Range

#### a note on return types

THe same thing can happen on the return side of the equation, but it is typically less of a problem. The main issues on the return side are returning arrays and dealing with Strings. If you want your UDF to work as an array formula, you can simply return an array and it will work. If that array is only a single cell, then it will look the same as a non-array formula.

Another issue is when working with Strings. If you return a string from a UDF, it will be formatted as Text instead of General. TODO: is that true? THis can have intended consequences as Excel tends to treat Text differently when it is then sent to other functions. THe most common example is that a number stored as text will not be available for normal math operations.

You can avoid this by returning Variant but it can become an issue when yuo want a Function to work as a UDF and as a normal VBA Function. You might have a good reason to use a specific return type on the VBA side of things, but then Excel may not handle that the way you want (if using a String). Or, going the other

way, you may have a UDF that works great because Excel can treat a single entry array as a single cell, but that becomes complicated when you call the UDF from another VBA location and then have to deal with a single number versus an array.

### complicated UDFS

One of the great advantages of UDFs is that you give you full access to all of VBA while still executing within the spreadsheet. There are some limits to this pwoer, but, in general, you able to do some very powerful stuff in teh same interface that you nromally do a `SUM`. To take advantage of this power, you need to be aware that these things are possible and then consider taking a shot at it.

Some of the more complciated areas where you will want to write a UDF include:

- Using Range information from cells not related to the parmaeters
- Accessing the FileSystem

Related to the Range, you have full access to all of the Workbooks and Worksheets that are avialable in VBA. THis means that you can combine a large amount of data in VBA and then output it to a UDF return. Where this becomes useful if when you want to look at the metadata of a Range of Worksheet. Until Excel 2013, most of this information was simply not available without VBA. Post 2013, you are able to use the `CELL` function (TODO: is this right). Some of the more useful things here are to use the fomratting of a cell (e.g. retunr the background color) or the display value (i.e. `Range.Text`). These UDFs can be great for eitehr long term usage or for a quick throwaway to get infomraiton into the spreadsheet. When doing the latter, there is essentially no difference between using a UDF and runnign through the cells using a normal Sub. The main reason you might use a UDF is if the cells you want to target are not easy to identify in VBA.

Other possible UDFs allow you to access the file system and possibly return information frmo there. ONe example would be to return the size of a file in KB given a file name. Really, you could go get any information you want. Again, this type of UDF can be easily done as a UDF or just as a Sub that runs through a Range as an input.

When considering whether or not to use a UDF or a Sub, consider the following:

- A UDF will update automatically when the parameters to it change (or always if marked as Volatile). This is the key differnentiator.
- A Sub can run without embedding itself into a spreadsheet. This is key if you need to save the spreadsheet with your informaiton without a link to your code. This is a moot point if your UDF lives in an XLSM file but starts to matter for an addin. You can also do a copy/paste values if you want to remove the UDF.

TODO: consider adding more here or refining this section

### debugging UDFs

Debugging a UDF is really teh same as debuggin normal code except you need to understand when your code will be called and hence, what you may be deugging. The simplest way to think about deugging a UDF is with an empty spreadsheet. In this example, once you type your UDF into the spreadhseet, Excel will execute the code and you can debug it via a breakppint. This is simple.

For a larger spreadsheet however, you are very likely to use your UDF more than once while only haivng a problem with a specific instance of it. Let's say your UDF does some fancy statistics btu cannot handle certin types of intputs. You can see that your code is throwing an error with a `#VALUE!` output. If you add a breakpoint to the UDF, then you risk having to debug a large number of successful calls before your bad one happens.

There are a couple of approaches to deal with this:

- Edit a formula for the cell you want with a breakpoint set in the debugger. Excel will execute that "new" formula first which will be the one of interest.
- Right a quick If statement to check if the teh Caller's address is a specific cell.

The first example is easy enough to understand and si the typical approach for debuggin a UDF. It's a bit of a pain because your breakpoint will stay in place and may be hit several times later. To get around this, you can siwtch over to manual calculation to avoid all the other cells calcualting. TODO: is that right?

The second approach works well when you have a UDF in several place but where only one of them is causing an erorr. You can add a temporary statement at the top to check for the Calller address and then set a breakpoint inside there. ocne it's hit, you know you are debugging the right call and can then step through teh code. You can do the same approach to check for the incoming value or really anythign else that is unique to the problematic cell. The nice thing ehre is that if you can figure out what statement to use for the breakpoint, you will ahve n aidea of which condiditons may cause the problem.

TODO: how are runtime errors handled here? any way to get them thrown with a prompt.

### ConcatArr.md

```
1 Public Function ConcatArr(rngCells As Variant, strDelim As String) As String
2     Dim cellCount As Long
```

```
    cellCount = UBound(rngCells, 1)

    Dim arrValues As Variant
    ReDim arrValues(1 To cellCount)

    Dim index As Long
    index = 1

    Dim rngCell As Variant
    For Each rngCell In rngCells
        arrValues(index) = rngCell

        index = index + 1
    Next

    ConcatArr = Join(arrValues, strDelim)
End Function
```

**ConcatRange.md**

```
Public Function ConcatRange(rngCells As Range, strDelim As String) As String
    Dim cellCount As Long

    cellCount = rngCells.CountLarge

    Dim arrValues As Variant
    ReDim arrValues(1 To cellCount)

    Dim index As Long
    index = 1

    Dim rngCell As Range
    For Each rngCell In rngCells
        arrValues(index) = rngCell

        index = index + 1
    Next
```

```
18
19      ConcatRange = Join(arrValues, strDelim)
20  End Function
```

**RandLetters.md**

```
1   Public Function RandLetters(ByVal letterCount As Long) As String
2
3       Dim letterIndex As Long
4
5       Dim letters() As String
6       ReDim letters(1 To letterCount)
7
8       For letterIndex = 1 To letterCount
9           letters(letterIndex) = chr(Int(Rnd() * 26 + 65))
10      Next
11
12      RandLetters = Join(letters(), "")
13
14  End Function
```

## overview of building an addin

The addin chapter can be a real difference maker. Making an addin is a great way to bring together a large number of related features that would be a pain to keep together anyways.

Some topics to hit:

- The advantages of always having an addin available to save new code into
- How to actually create an addin (the actual steps of saving it)
- Working with the Ribbon
- Adding keyboard shortcuts to the addin (via events)
- Managing the source code for an addin?

In addition to the VBA addin, it would be good to discuss the other options for extend Excel.

Include a section about Excel DNA and creating more powerful addins over there?

## introduction to creating an addin

This chapter will focus on creating an addin for Excel using VBA. There are other ways to create an addin but using VBA is simple because it can be done entirely from Excel and the Visual Basic Editor. The main distinction between an addin and other VBA code is that an addin is meant to be available to all open Workbooks without having to put the code inside a Workbook. This can be a very nice thing to have if you are regularly do the same or similar operations across different Workbooks. The alternative to an addin is often to maintain a library of code that you regularly export/import into macro enabled files as needed. This can create a mess as you change code in one file but not in another. The alternative also typically requires you to put the code inside a the Workbook and make it macro enabled. For certain applications, this is a non-starter. The one other alterative to a true addin is to create a Workbook that contains the code you want, and then you can open that file and execute the code in the context of whatever other files are open. This works, and creating an addin can be viewed as the logical conclusion of this approach. More than the logical conclusion, this is actually the first step for creating an addin.

When considering whether or not to create a proper addin with your code, consider the following:

- An addin provides a nice package for helper code and UDFs that might be used in multiple places
- An addin has easy access to the Ribbon and can create its own Ribbon tab
- An addin can be put in a central location and used as a repository of code for an organization (works best if the file is read-only)

Item 1 in the list above is typically enough of a reason to consider creating an addin. A common example of an addin is as a personal repository of VBA code. This typically replaces the use of the Personal Workbook, which I have never found to work well.

When considering a personal addin, one of the biggest upsides is that you can always open the VBE and have immediate access to your library of code. This makes it easy to make edits and save the new addin. Immediately, your updated code is available for future use in all your Workbooks.

There are a couple of downsides related to addins:

- UDFs from an addin require that anyone opening the spreadsheet has the addin loaded
- For code in a single Workbook, it is often easier to simply use a macro enabled Workbook and save the code directly there
- Some folks are highly resistant to “installing an addin” but will happily open a XLSM file. These are equivalent in the case of opening an addin, but the hesitation still exists.

Point 2 above is worth expanding on. Sometimes it’s tempting to add code to an existing addin that make sense only in the context of a single file. This works well if you and everyone else have the addin. This

starts to become a nuisance when you are constantly going through your addin to find code that should have been place in a Workbook to start. The cleaner way to store code that may be useful later is to place a copy of it in a personal addin. This ensures that the original code is always available in the Workbook and that future updates to the code don't break the original application.

## creating an addin

Creating an addin is a relatively simple process. You start with a normal XLSM macro enbaled file. From there, you save it as the add-in type (XLAM). That's it.

If you want to get more complicated, there is a property in teh VBE that can be toggled to change the addin status. (TODO: add picture of that). You would only need to change that flag if for some reason you wanted to save somethign back to a normal XLSM workbook without chanigng the extension.

There is one additional process that can be done to change how the addin is created is that is if you are modifyign the Ribbon for your addin. To do that, you will need to manaully edit hte XLAM file and change a file within it to add Ribbon support. You can do this manually or you can use a tool to help you out. Check the later section for details on that process.

## specific aspects to addin development

Depending on the addin that you are creating, you may expect for it to have a handful of features available. In general, those tpyes of features include keyboard shortcuts, special forms or user prompts, and possibly automatic feautres that fire depending on the user's action or the state of the workbook or Application.

### Keyboard Shortcuts

The simplest thing to do is to add keybaord shortcuts to your addin. There are two ways to do that:

- Open up the Macros form on the Developer tab. You can then hit "options" for a given Sub and assign a keyabord shortcut (TODO: add picture of htis)
- That approach can sometimes be a pain to edit later, so you can also add code to your addin ot add the shortcut.

The latter approach is nice because you can easily change the shortcut or the calling method. For addins, I will nearly always take the latter approach since it is much easier to deal with alter. For XLSM workbooks, I will do the former since it is easier to change from a workbook.

If you want to add the keyboard shortcut using code, use the code below. Ideally, you would put this in a Workbook_Open event that is called when the workbook opens. You can also use this approach to add/remove shortcuts depending on user input.

```
Public Sub SetUpKeyboardHooksForSelection()


    'SHIFT =    +
    'CTRL =     ^
    'ALT =      %

    'set up the keys for the selection mover
    Application.OnKey "^%{RIGHT}", "SelectionOffsetRight"
    Application.OnKey "^%{LEFT}", "SelectionOffsetLeft"
    Application.OnKey "^%{UP}", "SelectionOffsetUp"
    Application.OnKey "^%{DOWN}", "SelectionOffsetDown"

    'set up the keys for the indent level
    Application.OnKey "+^%{RIGHT}", "Formatting_IncreaseIndentLevel"
    Application.OnKey "+^%{LEFT}", "Formatting_DecreaseIndentLevel"

End Sub
```

#### USer Forms

One of the nice features of an addin are adding custom forms to provide the user with a better experience. Creating a UserForm in VBA is dead simple, and this is the best bang for your buck in terms of creating a profressional looking product. The simplest of forms with the simplest of features can save the end user hours and hours of time (I've seen it happen).

The nice thing here is that creating a UserForm in an addin is not any different than creating them normally. You simply create the UserForm. The only extra step is that you need to manage how/when the form is created and what infomration it has access to. Tpyically this is done by adding a buttong or using a keyboard shortcut. The only other issue is that oyu need to be aware of which WOrkbook or Worksheet is active when opening a UserForm if you are using ActiveSheet or ActiveWorkbook for anything. In general, inside an addin, you need to be careful with this comamnds since it is not always obvous that the ActiveXXX is the one you want ot access.

### Helpful COmmands

There are a couple of commands that exist outside of addins that become far more useful inside the addin. They are included below for reference:

- `ThisWorkbook` refers to the workbook that contains the code being executed. This is the surefire way to refer to the XLAM file that is running isntead of the ActiveWorkbook. IN general, your addin will never be the ActiveWorkbook. This becomes relevant if your addin workbook contains sheets of data that may need to be acesssed during runtime. You woud use THisWorkbook to refer to those sheet.
- TODO: add any other commands that are addin specific

### Other functionality

THe other functiojnality that you can add is related to Events. You have great power when it comes to listening to evnests and tirggering various actions. THe real difficulty is deciding what is an appropraite use of that power. Namely, when will you create an experiecne that benefits the user versus creating a very confusing workbook that is prone to breaking?

Before diving into what events can do, it's worht nting that potential downfalls of using them:

- They can be quite finicky somtimes. That is, using events adds a layer of complexity that tends to just complicate Excel adn VBA. I don't have a technical explanation, but there seem to be a number of bugs that creep out of the dark once you start really using events.
- Your user can disbale events at will and it can be quite difficult to determine when that was doen. THis is done with `Application.EnableEvents = False`.
- Events are triggered all teh time for all sorts of reasons. If you are doing a lot of checking in Events, you will dramatically slow down the workbook.

With all of those warngins, there is ntohign wrong with using Events. They generally do waht you want and can be quite powerful. I add the caveatas only because I have seen them ruin an otherwise working workbook. That cmoplexity gets amped up a level when your Event code is inside an addin instead of the main workbook.

To really make the most of Events, you are going to need to use Class Modules. The reason is that your Events need to "latch on" to the host workbooks or worksheets, and teh only way to do that is by using Class Modules. Normally, outside of an addin, you can simply open up the relevant VBA object (Workbook or Worksheet) and add the event code there. For an addin, you cannot add that code outside of the addin

so you are in a bind. How then can you hook onto the Event? Foretunatley, VBA makes thsi possible iwth teh `With Events` command inside of a Class MOdule.

TODO: provide a concreate example of usign this code

## UI features for addins, Ribbon, toolbars, UserForms

There are a number of UIs that can be provided for an addin. The most common involve using the Ribbon or providing UserForms (typically accessible from a keyboard shortcut). Those two approaches will disucssed in detail. It is also worth mentioning that if you are going to support Excel 2007 and before, that the Ribbon did not exist back then. For those prior versions of Excel, the interfaces were built using toolbars and menu items. That's before my day, so I'll just say that if you use that code today, it will show up in the Ribbon. If you need to support those versions of Excel, you would do well to find a different book.

### the Ribbon

When using the Ribbon, there are a couple of items to consider:

- Do you want your UI to show up in an existing tab or on your own?
- How interactive do you want your UI to be? This can range from simple buttons that trigger actions to text boxes and other more interactive features taht are able to detect user input and respond accordingly.
- How do you prefer to edit the file? How fast do you wnat your developmetn cycle to be with respect to the UI?

For the first point, this is a simple prefernece. For a given addin it may make sense to simply put the buttons and other access on an existing tab (Developer and Data are popular!) and provide that level of access. For an addin that has a dedicated purposed indepednet of other Excel features, it starts to make sense to add your own tab exlusively for your addin. THis is good for helping your users find your features. It can also be more consistent in terms of keyboard hsortucts. If you are going to modoify an exisitn tab, be abolustley certain that you verify that the keyboard shortcuts work as expected. There is nothign worse than having an addin break the ALT+A+R+Y shortcut which is supposed to reapply an autofilter. It is not fun when that shortcut becomes ALT+A+R+Y2. Seriously?

For the second point, you will need to consdier the average user and their expectations. Keep in mind that the default Excel Ribbon includes a number of locations where user input is collected nad used beond a simple button. This includes things like some number input (font size, page layout) and other drop downs.

There is a willingness for Excel users to ues these features where it make sense. For what it's worht, from teh Excel VBA point of view, it is much simpler to not try nad colect user input. This can be done (TODO: add examples), but the effort here is typically not worht the user expereince. If you choose to go this route, I would highly recommend using drop downs and other inputs that provide some automtica filterng of user input. Trying to validate user input off the Ribbon is a pain adn does not provide a good expereince. Having siad that, if oyu are deisnging for power users, you can build a very slick interface in the Ribbon that is unmathced.

The final point gets down to the nitty gritty of actually editing the Ribbon. The problem is that the Ribbon is defined in a file inside your XLAM file and is not eidtable from any part of the VBE or Exce interface. This means that it is a real pain to edit the Ribbon definition in the same way that oyu cna eidt the otehr VBA code. I have typically taken the approahc of usign a button on teh Ribbon to launch a form that exists in VBA. That form cna then be edited without having to touch the Rbbon definiont. This sounds tribial, but it can make a huge differnece if you are designing an addin that has a lot of possible interactivity; it is very difficult to edit the Ribbon in real time. Having said that, there is one addin that amkes this process much more managebale. It is from Andy Pope (TODO: add link) and works great for building out the interface. Even using that addin, it is a pian to add the callback necessary to tie the Ribbon to VBA. Don't be dissuaded from creating a nice Ribbon UI, but realize that it takes time and effort and attention to detail to properly detail out the Ribbon UI aspects.

**editing the Ribbon** Note that the Ribbon is defined in an XML file inside your XLAM file. Remeber that an XLAM file is simply a ZIP file of a bunch of different folders. By default, the Ribbon deifnition is not included and you must add the fodler and file. To do this, simple create the `customUi` folder and then create a XXX XML file inside there. This file will define the speciif cchanges you are making to the RIbbon.

**callabacks** ONce you hav ethe Ribbon XML set up, you will be defining callbacks that need to actualyl exist in the VBA code. I always like to create a `Ribbon` module in the XLAM file which is solely repsondble for callbacks. This is nice for larger addins with a large number of callbacks becuase it provides a isngle place. It also avoids debugging errors later when you accidentally put some critical code into a callback adn forgot to check that out.

The callbacks take an odd signature. I always use Andy Pope's addin or copy a previous one. I have very seldom used the parmaeters in the callback for accessing the Ribbon information. My approach has always been to avoid extra interactivity with teh ribbon. I have done it before, and it works, but the problem is that it is just not intutive to do it that way. It is much eaiser to add a keyboard shortuct which shows a

USerForm than to attempt to get the user to focus on teh Ribbon (using the keyboard or mouse) and then provide the best info.

#### UserForms

If you are going to use UI features within your addin, you are gonig to use UserForms. They provide the cleanest and easiest interface for teh vast majority of automation and other tasks. THe one exception to using a UserForm is when you can get by with a simple `InputBox`. You should alwa sprefer the INputBox because the procedure for calling them and obtainign a value is dead simple. Also, teh InputBox is the best way to ask the user for a Range input via seleciton. You cna tehcnically use the `RefEdit` control, but hta tcontorl is very sensitive when it works.

If you are building a USerFOrm, there is very little tha tis different from a normla USerForm. The only thing ot be aware of are the logistics of creating, showign, adn hiding the USeForm. I have previously tried to keep an instance of a given form live in order to use the previous values. This has worked well inside a single Workbook btu seems to be very finicky when working across multiple Workbooks. The procedure here is very simpel:

```
DIm frm as UserForm
Set frm = New UserForm

fmr.Show
```

The code above is all that is required to create a new instance of a form and show it to the user. From there, the code is the same as before: you simply create the form and call the various SUbs you want. One thign which is hepful is ot hide the form when you are done. This is done with the `Unload Me` command.

One other item to be aware of is that the default UserForm is set to `ShowModal = True` whcih applies the modal property. A “modal” dialog is one who steals focus from any other elements and must be dealt with before you can go back to your previously focusbale elements. This is often good for certian workflows wher eyou do not want the user to change the underylinyg spreadsheet while you collect their input. THere are other instances however where it makes sense to allow the user ot change the active Workbook, Worksheet, or Selection and then inteact with you rform. To allow for this behavior, set `ShowModal = False`. THs will allow your form to exit even when teh user clicks off and interacrs with teh spreadsheet again. This is a real game changer when you are workign with code that operates on teh current selection. You are then able to leave your form up whil ethe user changes the sleection. From there, they are able to call the cod

ethey want on teh objects they want. I have used this technique to great effect when workign with Charts: allow the user to select their charts and then hit a button.

## overview of utility code

This chapter might be a dumping ground for useful code that is worth being able to reference. Some things to include here:

- Code to work through the selection on a Chart
- The RangeEnd function to quick get the end of a Range
- Some string processing code?
- The code to work with split values
- Code to convert a 2D array of values to 1D
- GetOrCreateWorksheet which gives you a valid object regardless of what existed
- CreateNextSheet which increments a name as needed
- Creating a Chart based on the XValues, Values, and Name (and order).
- CopyResize command which is used to replicate Copy/PasteValues without using the Clipboard

### ColorInputs.md

```
Public Sub ColorInputs()

    Dim targetCell As Range
    Const FIRST_COLOR_ACCENT As String = "msoThemeColorAccent1"
    Const SECOND_COLOR_ACCENT As String = "msoThemeColorAccent2"
    'This is finding cells that aren't blank, but the description says it
       should be cells with no values..
    For Each targetCell In Selection
        If targetCell.Value <> "" Then
            If targetCell.HasFormula Then
                targetCell.Interior.ThemeColor = FIRST_COLOR_ACCENT
            Else
                targetCell.Interior.ThemeColor = SECOND_COLOR_ACCENT
            End If
        End If
    Next targetCell
```

```
End Sub
```

**CombineAllSheetsData.md**

```
Public Sub CombineAllSheetsData()

    'create the new wkbk and sheet
    Dim targetWorkbook As Workbook
    Dim sourceWorkbook As Workbook

    Set sourceWorkbook = ActiveWorkbook
    Set targetWorkbook = Workbooks.Add

    Dim targetWorksheet As Worksheet
    Set targetWorksheet = targetWorkbook.Sheets.Add

    Dim isFirst As Boolean
    isFirst = True

    Dim targetRow As Long
    targetRow = 1

    Dim sourceWorksheet As Worksheet
    For Each sourceWorksheet In sourceWorkbook.Sheets
        If sourceWorksheet.name <> targetWorksheet.name Then

            sourceWorksheet.Unprotect

            'get the headers squared up
            If isFirst Then
                'copy over all headers
                sourceWorksheet.Rows(1).Copy targetWorksheet.Range("A1")
                isFirst = False

            Else
                'search for missing columns
                Dim headerRow As Range
```

```
                For Each headerRow In Intersect(sourceWorksheet.Rows(1),
                    sourceWorksheet.UsedRange)

                    'check if it exists
                    Dim matchingHeader As Variant
                    matchingHeader = Application.Match(headerRow,
                        targetWorksheet.Rows(1), 0)

                    'if not, add to header row
                    If IsError(matchingHeader) Then targetWorksheet.Range("A1
                        ").End(xlToRight).Offset(, 1) = headerRow
                Next headerRow
            End If

            'find the PnPID column for combo
            Dim pIDColumn As Long
            pIDColumn = Application.Match("PnPID", targetWorksheet.Rows(1),
                0)

            'find the PnPID column for data
            Dim pIDData As Long
            pIDData = Application.Match("PnPID", sourceWorksheet.Rows(1), 0)

            'add the data, row by row
            Dim targetCell As Range
            For Each targetCell In sourceWorksheet.UsedRange.SpecialCells(
                xlCellTypeConstants)
                If targetCell.Row > 1 Then

                    'check if the PnPID exists in the combo sheet
                    Dim sourceRow As Variant
                    sourceRow = Application.Match( _
                                sourceWorksheet.Cells(targetCell.Row, pIDData)
                                    , _
                                targetWorksheet.Columns(pIDColumn), _
                                0)

                    'add new row if it did not exist and id number
                    If IsError(sourceRow) Then
```

```
                        sourceRow = targetWorksheet.Columns(pIDColumn).Cells(
                            targetWorksheet.Rows.Count, 1).End(xlUp).Offset(1)
                            .Row
                        targetWorksheet.Cells(sourceRow, pIDColumn) =
                            sourceWorksheet.Cells(targetCell.Row, pIDData)
                    End If

                    'get column
                    Dim columnNumber As Long
                    columnNumber = Application.Match(sourceWorksheet.Cells(1,
                        targetCell.Column), targetWorksheet.Rows(1), 0)

                    'update combo data
                    targetWorksheet.Cells(sourceRow, columnNumber) =
                        targetCell

                End If
            Next targetCell
        End If
    Next sourceWorksheet
End Sub
```

**ConvertSelectionToCsv.md**

```
Public Sub ConvertSelectionToCsv()

    Dim sourceRange As Range
    Set sourceRange = GetInputOrSelection("Choose range for converting to CSV
        ")

    If sourceRange Is Nothing Then Exit Sub

    Dim outputString As String

    Dim dataRow As Range
    For Each dataRow In sourceRange.Rows

        Dim dataArray As Variant
```

```
        dataArray = Application.Transpose(Application.Transpose(dataRow.Rows.
            Value2))

        'TODO:  improve this to use another Join instead of string concats
        outputString = outputString & Join(dataArray, ",") & vbCrLf

    Next dataRow

    Dim myClipboard As MSForms.DataObject
    Set myClipboard = New MSForms.DataObject

    myClipboard.SetText outputString
    myClipboard.PutInClipboard

End Sub
```

**CopyCellAddress.md**

```
Public Sub CopyCellAddress()


    'TODO: this need to get a button or a keyboard shortcut for easy use
    Dim myClipboard As MSForms.DataObject
    Set myClipboard = New MSForms.DataObject

    Dim sourceRange As Range
    Set sourceRange = Selection

    myClipboard.SetText sourceRange.Address(True, True, xlA1, True)
    myClipboard.PutInClipboard
End Sub
```

**CutPasteTranspose.md**

```
Public Sub CutPasteTranspose()

```

```
3
4       '########Still Needs to address Issue#23##############
5       On Error GoTo errHandler
6       Dim sourceRange As Range
7       'TODO #Should use new inputbox function
8       Set sourceRange = Selection
9
10      Dim outputRange As Range
11      Set outputRange = Application.InputBox("Select output corner", Type:=8)
12
13      Application.ScreenUpdating = False
14      Application.EnableEvents = False
15      Application.Calculation = xlCalculationManual
16
17      Dim topLeftCell As Range
18      Set topLeftCell = sourceRange.Cells(1, 1)
19
20      Dim topRow As Long
21      topRow = topLeftCell.Row
22      Dim leftColumn As Long
23      leftColumn = topLeftCell.Column
24
25      Dim outputRow As Long
26      Dim outputColumn As Long
27      outputRow = outputRange.Row
28      outputColumn = outputRange.Column
29
30      outputRange.Activate
31
32      'Check to not overwrite
33      Dim targetCell As Range
34      For Each targetCell In sourceRange
35          If Not Intersect(sourceRange, Cells(outputRow + targetCell.Column -
                leftColumn, outputColumn + targetCell.Row - topRow)) Is Nothing
                Then
36              MsgBox ("Your destination intersects with your data. Exiting.")
37              GoTo errHandler
38          End If
39      Next
```

```
40
41      'this can be better
42      For Each targetCell In sourceRange
43          targetCell.Cut
44          ActiveSheet.Cells(outputRow + targetCell.Column - leftColumn,
                outputColumn + targetCell.Row - topRow).Activate
45          ActiveSheet.Paste
46      Next targetCell
47
48  errHandler:
49      Application.CutCopyMode = False
50      Application.ScreenUpdating = True
51      Application.EnableEvents = True
52      Application.Calculation = xlCalculationAutomatic
53      Application.Calculate
54
55  End Sub
```

**FillValueDown.md**

```
1   Public Sub FillValueDown()
2
3       Dim inputRange As Range
4       Set inputRange = GetInputOrSelection("Select range for waterfall")
5
6       If inputRange Is Nothing Then Exit Sub
7
8       Dim targetCell As Range
9       For Each targetCell In Intersect(inputRange.SpecialCells(xlCellTypeBlanks
            ), inputRange.Parent.UsedRange)
10          targetCell = targetCell.End(xlUp)
11      Next targetCell
12
13  End Sub
```

**ForceRecalc.md**

```
Public Sub ForceRecalc()

    Application.CalculateFullRebuild

End Sub
```

**GenerateRandomData.md**

```
Public Sub GenerateRandomData()

    Const NUMBER_OF_ROWS As Long = 10
    Const NUMBER_OF_COLUMNS As Long = 3 '0 index
    Const DEFAULT_COLUMN_WIDTH As Long = 15

    'Since we only work with offset, targetcell can be a constant, but range
       constants are awkward
    Dim targetCell As Range
    Set targetCell = Range("B2")

    Dim i As Long

    For i = 0 To NUMBER_OF_COLUMNS
        targetCell.Offset(, i) = chr(65 + i)

        With targetCell.Offset(1, i).Resize(NUMBER_OF_ROWS)
            Select Case i
            Case 0
                .Formula = "=TODAY()+ROW()"
            Case Else
                .Formula = "=RANDBETWEEN(1,100)"
            End Select

            .Value = .Value
        End With
    Next i

    ActiveSheet.UsedRange.Columns.ColumnWidth = DEFAULT_COLUMN_WIDTH
```

```
29
30 End Sub
```

### OpenContainingFolder.md

```
1  Public Sub OpenContainingFolder()
2
3      Dim targetWorkbook As Workbook
4      Set targetWorkbook = ActiveWorkbook
5
6      If targetWorkbook.path <> "" Then
7          targetWorkbook.FollowHyperlink targetWorkbook.path
8      Else
9          MsgBox "Open file is not in a folder yet."
10     End If
11
12 End Sub
```

### PivotSetAllFields.md

```
1  Public Sub PivotSetAllFields()
2
3      Dim targetTable As PivotTable
4      Dim targetSheet As Worksheet
5
6      Set targetSheet = ActiveSheet
7
8      'this information is a bit unclear to me
9      MsgBox "This defaults to the average for every Pivot table on the sheet.
           Edit code for other result."
10     On Error Resume Next
11     For Each targetTable In targetSheet.PivotTables
12         Dim targetField As PivotField
13         For Each targetField In targetTable.DataFields
14             targetField.Function = xlAverage
15         Next targetField
```

```
    Next targetTable

End Sub
```

**SeriesSplit.md**

```
Public Sub SeriesSplit()

    On Error GoTo ErrorNoSelection

    Dim selectedRange As Range
    Set selectedRange = Application.InputBox("Select category range with
       heading", Type:=8)
    Set selectedRange = Intersect(selectedRange, selectedRange.Parent.
       UsedRange).SpecialCells(xlCellTypeVisible, xlLogical + xlNumbers +
       xlTextValues)

    Dim valueRange As Range
    Set valueRange = Application.InputBox("Select values range with heading",
        Type:=8)
    Set valueRange = Intersect(valueRange, valueRange.Parent.UsedRange)

    On Error GoTo 0

    'determine default value
    Dim defaultString As Variant
    defaultString = InputBox("Enter the default value", , "#N/A")
    'strptr is undocumented
    'detect cancel and exit
    If StrPtr(defaultString) = 0 Then
        Exit Sub
    End If

    Dim dictCategories As New Dictionary

    Dim categoryRange As Range
    For Each categoryRange In selectedRange
        'skip the header row
```

```
        If categoryRange.Address <> selectedRange.Cells(1).Address Then
            dictCategories(categoryRange.Value) = 1
    Next categoryRange

    valueRange.EntireColumn.Offset(, 1).Resize(, dictCategories.Count).Insert
    'head the columns with the values

    Dim valueCollection As Variant
    Dim counter As Long
    counter = 1
    For Each valueCollection In dictCategories
        valueRange.Cells(1).Offset(, counter) = valueCollection
        counter = counter + 1
    Next valueCollection

    'put the formula in for each column
    '=IF(RC13=R1C,RC16,#N/A)
    Dim formulaHolder As Variant
    formulaHolder = "=IF(RC" & selectedRange.Column & " =R" & _
                 valueRange.Cells(1).Row & "C,RC" & valueRange.Column & "," &
                     defaultString & ")"

    Dim formulaRange As Range
    Set formulaRange = valueRange.Offset(1, 1).Resize(valueRange.Rows.Count -
        1, dictCategories.Count)
    formulaRange.FormulaR1C1 = formulaHolder
    formulaRange.EntireColumn.AutoFit

    Exit Sub

ErrorNoSelection:
    'TODO: consider removing this prompt
    MsgBox "No selection made.  Exiting.", , "No selection"

End Sub
```

**SeriesSplitIntoBins.md**

```
Public Sub SeriesSplitIntoBins()

    Const LESS_THAN_EQUAL_TO_GENERAL As String = "<= General"
    Const GREATER_THAN_GENERAL As String = "> General"
    On Error GoTo ErrorNoSelection

    Dim selectedRange As Range
    Set selectedRange = Application.InputBox("Select category range with
       heading", Type:=8)
    Set selectedRange = Intersect(selectedRange, selectedRange.Parent.
       UsedRange) _
                                .SpecialCells(xlCellTypeVisible, xlLogical +
                                    _
                                 xlNumbers + xlTextValues)

    Dim valueRange As Range
    Set valueRange = Application.InputBox("Select values range with heading",
         Type:=8)
    Set valueRange = Intersect(valueRange, valueRange.Parent.UsedRange)

    ''need to prompt for max/min/bins
    Dim maximumValue As Double, minimumValue As Double, binValue As Long

    minimumValue = Application.InputBox("Minimum value.", "Min", _
                                        WorksheetFunction.Min(selectedRange),
                                            Type:=1)

    maximumValue = Application.InputBox("Maximum value.", "Max", _
                                        WorksheetFunction.Max(selectedRange),
                                            Type:=1)

    binValue = Application.InputBox("Number of groups.", "Bins", _
                                    WorksheetFunction.RoundDown(Math.Sqr(
                                       WorksheetFunction.Count(selectedRange)
                                       ), _
                                    0), Type:=1)

    On Error GoTo 0
```

```
31
32      'determine default value
33      Dim defaultString As Variant
34      defaultString = Application.InputBox("Enter the default value", "Default
            ", "#N/A")
35
36      'detect cancel and exit
37      If StrPtr(defaultString) = 0 Then Exit Sub
38
39      ''TODO prompt for output location
40
41      valueRange.EntireColumn.Offset(, 1).Resize(, binValue + 2).Insert
42      'head the columns with the values
43
44      ''TODO add a For loop to go through the bins
45
46      Dim targetBin As Long
47      For targetBin = 0 To binValue
48          valueRange.Cells(1).Offset(, targetBin + 1) = minimumValue + (
                maximumValue - _
49                                                          minimumValue) *
                                                              targetBin / binValue
50      Next
51
52      'add the last item
53      valueRange.Cells(1).Offset(, binValue + 2).FormulaR1C1 = "=RC[-1]"
54
55      'FIRST =IF($D2 <=V$1,$U2,#N/A)
56      '=IF(RC4 <=R1C,RC21,#N/A)
57
58      'MID =IF(AND($D2 <=W$1, $D2>V$1),$U2,#N/A)  '''W current, then left
59      '=IF(AND(RC4 <=R1C, RC4>R1C[-1]),RC21,#N/A)
60
61      'LAST =IF($D2>AA$1,$U2,#N/A)
62      '=IF(RC4>R1C[-1],RC21,#N/A)
63
64      ''TODO add number format to display header correctly (helps with charts)
65
66      'put the formula in for each column
```

```
    '=IF(RC13=R1C,RC16,#N/A)
    Dim formulaHolder As Variant
    formulaHolder = "=IF(AND(RC" & selectedRange.Column & " <=R" & _
                    valueRange.Cells(1).Row & "C," & "RC" & selectedRange.
                        Column & ">R" & _
                    valueRange.Cells(1).Row & "C[-1]" & ")" & ",RC" &
                        valueRange.Column & "," & _
                    defaultString & ")"

    Dim firstFormula As Variant
    firstFormula = "=IF(AND(RC" & selectedRange.Column & " <=R" & _
                    valueRange.Cells(1).Row & "C)" & ",RC" & valueRange.
                        Column & "," & defaultString _
                    & ")"

    Dim lastFormula As Variant
    lastFormula = "=IF(AND(RC" & selectedRange.Column & " >R" & _
                    valueRange.Cells(1).Row & "C)" & ",RC" & valueRange.
                        Column & "," & defaultString _
                    & ")"

    Dim formulaRange As Range
    Set formulaRange = valueRange.Offset(1, 1).Resize(valueRange.Rows.Count -
        1, binValue + 2)
    formulaRange.FormulaR1C1 = formulaHolder

    'override with first/last
    formulaRange.Columns(1).FormulaR1C1 = firstFormula
    formulaRange.Columns(formulaRange.Columns.Count).FormulaR1C1 =
       lastFormula

    formulaRange.EntireColumn.AutoFit

    'set the number formats

    formulaRange.Offset(-1).Rows(1).Resize(1, binValue + 1).NumberFormat =
       LESS_THAN_EQUAL_TO_GENERAL
    formulaRange.Offset(-1).Rows(1).Offset(, binValue + 1).NumberFormat =
       GREATER_THAN_GENERAL
```

```
98
99      Exit Sub
100
101 ErrorNoSelection:
102     'TODO: consider removing this prompt
103     MsgBox "No selection made.  Exiting.", , "No selection"
104
105 End Sub
```

**Sheet_DeleteHiddenRows.md**

```
1  Public Sub Sheet_DeleteHiddenRows()
2      'These rows are unrecoverable
3      Dim shouldDeleteHiddenRows As VbMsgBoxResult
4      shouldDeleteHiddenRows = MsgBox("This will permanently delete hidden rows
           . They cannot be recovered. Are you sure?", vbYesNo)
5
6      If Not shouldDeleteHiddenRows = vbYes Then Exit Sub
7
8      Application.ScreenUpdating = False
9
10     'collect a range to delete at end, using UNION-DELETE
11     Dim rangeToDelete As Range
12
13     Dim counter As Long
14     counter = 0
15     With ActiveSheet
16         Dim rowIndex As Long
17         For rowIndex = .UsedRange.Rows.Count To 1 Step -1
18             If .Rows(rowIndex).Hidden Then
19                 If rangeToDelete Is Nothing Then
20                     Set rangeToDelete = .Rows(rowIndex)
21                 Else
22                     Set rangeToDelete = Union(rangeToDelete, .Rows(rowIndex))
23                 End If
24                 counter = counter + 1
25             End If
26         Next rowIndex
```

```
    End With

    rangeToDelete.Delete

    Application.ScreenUpdating = True

    MsgBox (counter & " rows were deleted")
End Sub
```

### UnhideAllRowsAndColumns.md

```
Public Sub UnhideAllRowsAndColumns()

    ActiveSheet.Cells.EntireRow.Hidden = False
    ActiveSheet.Cells.EntireColumn.Hidden = False

End Sub
```