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 'Selected, 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 the 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 just act immediately on the Range as you are iterating through it. This approach is dead simple and is used in abundance throughout good workflows. As long as there is some meaningful logic which can be applied to identify whether or not a subset of a Range is interesting, you can use this technique. Some common logical steps that are checked:

- 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 you just need to be told that this is an acceptable way of doing things. You do not always need to use `Find` to search for a cell that contains 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

    If secondDirection = -1 Then
        Set RangeEnd = Range(rangeBegin, rangeBegin.End(firstDirection))
    Else
        Set RangeEnd = Range(rangeBegin, rangeBegin.End(firstDirection).End(
            secondDirection))
    End If
End Function
```

RangeEnd_Boundary.md

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

    If secondDirection = -1 Then
        Set RangeEnd_Boundary = Intersect(Range(rangeBegin, rangeBegin.End(
            firstDirection)), rangeBegin.CurrentRegion)
    Else
        Set RangeEnd_Boundary = Intersect(Range(rangeBegin, rangeBegin.End(
            firstDirection).End(secondDirection)), rangeBegin.CurrentRegion)
    End If
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 to iterate through a Range or what you are searching for. The idea is that you know your starting Range contains some pieces that you would like to iterate 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 Columns will each return a new utterable 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 you want to use these, 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()`
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 to 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 to 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 can 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 different 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 referenced.

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

TODO: add a picture of set operations

Some common examples of where these functions come up:

- Intersect is used with Events and other usability tasks to determine if a given or selected Cell is within a target Range
- Interacted is very useful with Offset and Resize to grab a new Range that contains a subset of data of the original Range without having to worry about creating a new Range that includes cells not previously included. IN this sense, Intersect only allows a Range to get smaller.
- Union 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 calculations if you change the `Value` all at once. This sam technique can b used to build a Range to delete

TODO: add Union-Delete example

TODO: add Intersect example to remove headers

TODO: add Intersect technique for Events and Selection 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 scratch. The key then is knowing when these properties exist and how to 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 point 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 ranges in the Workbook. You cannot iterate them through `Range` because you want to know what they are!

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

- Iterate the `Names` with no knowledge 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 can 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 option 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 Range based on a formula which included the previous one. This can be a critical step to improving the performance of array formulas that previously 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 helpful to create them from time to time from VBA. This can be a helpful way of storing a complicated Range that your VBA created without having to select the cells and hope you can type the name correctly.

Using `Application.InputBox(, Type:=8)` One very useful technique for obtaining 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 the Application alone. IF you use the other version, then you cannot supply the Type
- You have to know that Type:=8 allows for a Range selection

Once you have two those things down (because you read this book!) then you are able to ask the 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 to process the Selection (by default) or to allow the user to select something different. This is a very clean solution to snivel defaults while also allowing the user to do something different once they read your initial prompt. It is also dead simple to upgrade your current `Set rng = Range()` to `Set rng = Application.InputBox("Select a cell", Type:=8)` instead. For utility type 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 Function here form bUTL GetOrSelect…

Using `Application.Index` The `=INDEX` formula is the most potent formula in Excel. Its counterpart 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 minimal change to formulas. Once you have the work converted over, you can then set about addin the 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 the problem for me is that

I have never had a problem using some other method than Find. I also generally find myself somewhat confused by the parameters and the general execution 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 Range 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 formula (e.g. take `A1` and surround it with an `ABS(A1)`)
- You want to make all of the cells in a specific formula a specific color (like a permanent version of hitting `F2`)

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

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 information

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

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

- Use a Regular Expression keyed in to Range options

- Use your knowledge of the possible formulas to extract the relevant parts with string functions

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 to accidentally add a blank or neighboring column to your Range. Knowing this, you can then take whatever steps are necessary to "remove" bad sections from our Range. This is most commonly used to remove a header row from the 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 with 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 one except for the first row.

TODO: add an image of how this works

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

AutoFilter and then SpecialCells This approach is straight forward and mirrors a common operation in non VBA Excel. You use an AutoFilter to filter out specific cells and then you can select only 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 columns. Typically Excel will not apply formatting to hidden cells and will also not fill a formula through them (assuming you used the Fill command).

In VBA, things are often more difficult because you are working with the underlying Range independent 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 criteria. when using the AutoFilter, the most common criterion to use is that of visibility. You can call `Range.SpecialCells(xlCellTypeVisible)` to obtain a new Range which only contains visible cells.

If you have ever written a loop which does a `If rng.Hidden = True Then...` then you will be grateful to know that Excel VBA provides this feature automatically. SpecialCells really is one of the most powerful ways to access Ranges in an intuitive fashion 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 information about one sheet to build a Range on another or several others. THe problem with Ranges however is that they are not allowed to span multiple Worksheets. This means that if you want to apply some action to 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 using code and not a direct address. To get around 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 a nice way to replicate the functionality of Excel where you can select multiple Worksheets with CTRL or SHIFT and then apply some action to all of them. The really 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 because the normal multi edit feature do the exact same steps to all spreadsheets whereas you may want to use `End` or something 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
```