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 to 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 to 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 can 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 you 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 is 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 to 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 to 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 to 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 can 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 to 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 you 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 power, but, in general, you able to do some very powerful stuff in the same interface that you normally 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 complicate areas where you will want to write a UDF include:

- Using Range information from cells not related to the parameters
- Accessing the FileSystem

Related to the Range, you have full access to all of the Workbooks and Worksheets that are available 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 formatting of a cell (e.g. return the background color) or the display value (i.e. `Range.Text`). These UDFs can be great for either long term usage or for a quick throwaway to get information into the spreadsheet. When doing the latter, there is essentially no difference between using a UDF and running 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 from 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 differentiator.
- A Sub can run without embedding itself into a spreadsheet. This is key if you need to save the spreadsheet with your information 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 the same as debugging normal code except you need to understand when your code will be called and hence, what you may be debugging. The simplest way to think about debugging a UDF is with an empty spreadsheet. In this example, once you type your UDF into the spreadsheet, Excel will execute the code and you can debug it via a breakpoint. This is simple.

For a larger spreadsheet however, you are very likely to use your UDF more than once while only having a problem with a specific instance of it. Let's say your UDF does some fancy statistics btu cannot handle certain types of inputs. 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 the Caller's address is a specific cell.

The first example is easy enough to understand and si the typical approach for debugging 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 switch over to manual calculation to avoid all the other cells calculating. 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 error. You can add a temporary statement at the top to check for the Caller address and then set a breakpoint inside there. once it's hit, you know you are debugging the right call and can then step through the code. You can do the same approach to check for the incoming value or really anything 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 have n aidea of which conditions may cause the problem.

TODO: how are runtime errors handled here? any way to get them thrown with a prompt.

ConcatArr.md

```
Public Function ConcatArr(rngCells As Variant, strDelim As String) As String
    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

    ConcatRange = Join(arrValues, strDelim)
End Function
```

RandLetters.md

```
Public Function RandLetters(ByVal letterCount As Long) As String

    Dim letterIndex As Long

    Dim letters() As String
    ReDim letters(1 To letterCount)

    For letterIndex = 1 To letterCount
```

```
        letters(letterIndex) = chr(Int(Rnd() * 26 + 65))
    Next

    RandLetters = Join(letters(), "")

End Function
```