SumProduct

XLOOKUP

NEWSLETTER #111 - February 2022
www.sumproduct.com | www.sumproduct.com/thought

Are things XLOOK-ing up for 2022?

After last month's grumbles about **VLOOKUP** and **HLOOKUP**, we turn our attention to **XLOOKUP**, one of Excel 365's most popular new additions – a new addition for a new edition, if you like. And if you don't, I'm still going to make the quip anyway…

Power BI appears to be on holiday this month, but we have plenty to keep you occupied in its absence: there is another Beat the Boredom Challenge, plus our usual articles on Charts & Dashboards, Visual Basics, Power Pivot Principles and Power Query Pointers. We also see **I** to **I** on the A to Z of Excel Functions, with the most Important **F**unction in Excel and the Keyboard Shortcuts **SHIFT CTRL** to the end user too.

As always, happy reading and remember: stay safe, stay happy, stay healthy.

Liam Bastick, Managing Director, SumProduct

Looking Up Data Revisited

Last month, we received some positive feedback regarding my **VLOOKUP** tirade. However, I was taught if you are going to criticise, you should criticise constructively, so I thought, whilst it's a bit quieter here in SumProductLand, I'd revisit its natural successor: **XLOOKUP**.

Available in Excel for Microsoft 365, Excel for Microsoft 365 for Mac, Excel for the web and Excel 2021, **XLOOKUP** has the following syntax:

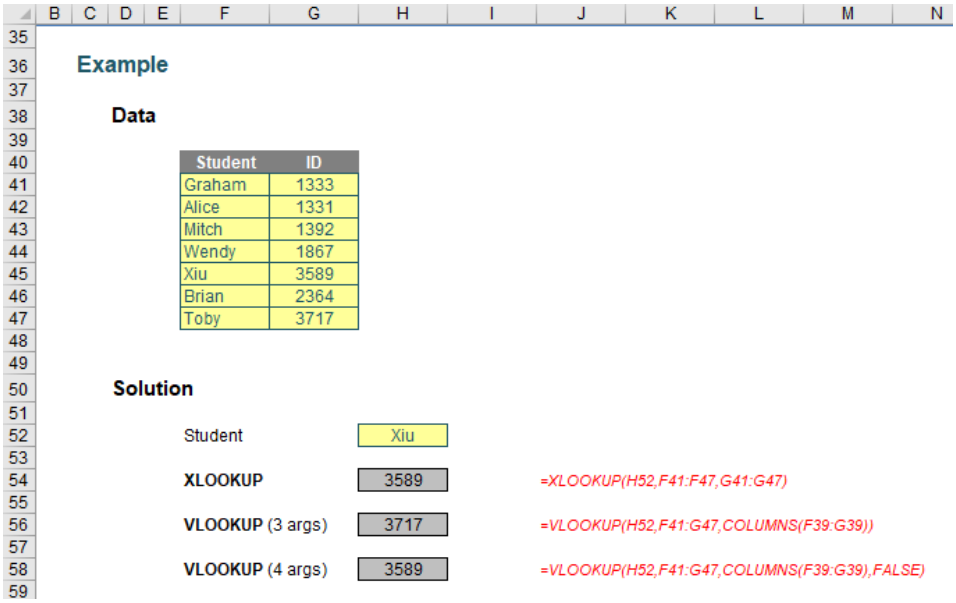
XLOOKUP(lookup_value, lookup_vector, results_array, [if_not_found], [match_mode], [search_mode])

This function seeks out a **lookup_value** in the **lookup_vector** and returns the corresponding value in the results_array. Similar to **RANDARRAY**, Microsoft again decided to make a change before pulling the pin and make both of these functions Generally Available. The current line of thinking is that there should be an error trap for when a value cannot be found. Having said that, most of the time you will only require the first three arguments:

- **lookup_value:** this is required and defines what value you want to look up
- **lookup_vector:** this reference is required and is the row or column of data you are referencing to look up **lookup_value**
- **results_array:** this is where the corresponding item is you wish to return and is also required (even if it is the same as lookup_vector). This does not have to be a vector (*i.e.* one row or one column of cells): it may be an array (with at least two rows and at least two columns of cells). The only stipulation is that the number of rows / columns must equal the number of rows / columns in the column / row vector – but more on that later
- **if_not_found:** this optional argument allows you to replace the usual return of *#N/A* with something more informative like an alternative formula, text or a value
- match_mode: this argument is optional. There are four choices:
  - **0:** exact match (default)
  - **-1:** exact match or else the largest value less than or equal to **lookup_value**
  - **1:** exact match or else smallest value greater than or equal to **lookup_value**
  - **2:** wildcard match. You should use the special character **?** to match any character and ***** to match any run of characters.

  What's impressive, though, is that for certain selections of the final argument (**search_mode**), you don't need to put your data in alphanumerical order! As far as I am aware, this is a first for Excel
- **search_mode:** this argument is also optional. There are again four choices:
  - **1:** search first to last (default)
  - **-1:** search last to first
  - **2:** what is known as a binary search, first to last (requires **lookup_vector** to be sorted). Just so you know, a binary search is a search algorithm that finds the position of a target value within a sorted array. A binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found
  - **-2:** another binary search, this time last to first (and again, this requires **lookup_vector** to be sorted).

Let's have a look at **XLOOKUP** versus last month's **VLOOKUP**:



Example

Data

Student	ID
Graham	1333
Alice	1331
Mitch	1392
Wendy	1867
Xiu	3589
Brian	2364
Toby	3717

Solution

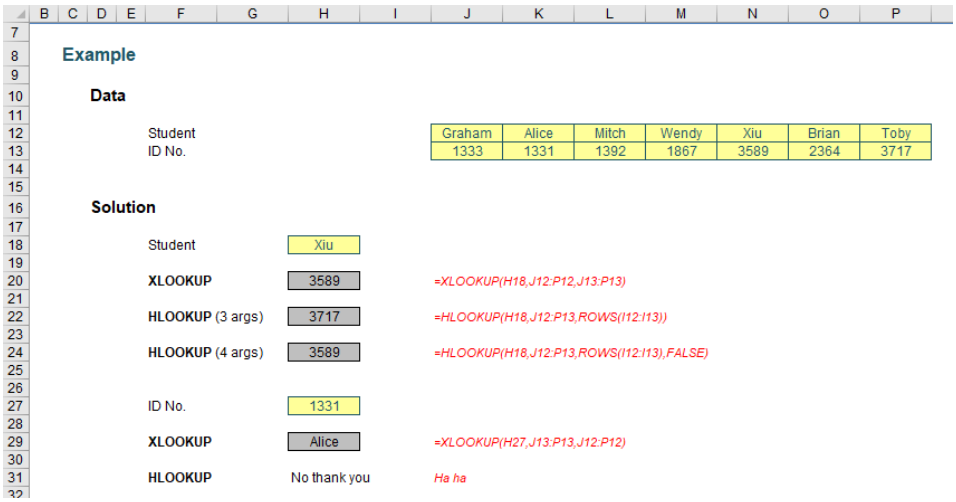
Student	Xiu	
XLOOKUP	3589	=XLOOKUP(H52,F41:F47,G41:G47)
VLOOKUP (3 args)	3717	=VLOOKUP(H52,F41:G47,COLUMNS(F39:G39))
VLOOKUP (4 args)	3589	=VLOOKUP(H52,F41:G47,COLUMNS(F39:G39),FALSE)

You can clearly see the **XLOOKUP** function is shorter:

=XLOOKUP(H52,F41:F47,G41:G47)

Only the first three arguments are needed, whereas **VLOOKUP** requires both a fourth argument, and, for full flexibility, the **COLUMNS** function as well. **XLOOKUP** will automatically update if rows / columns are inserted or deleted. It's just simpler.

HLOOKUP has similar issues:



Example

Data

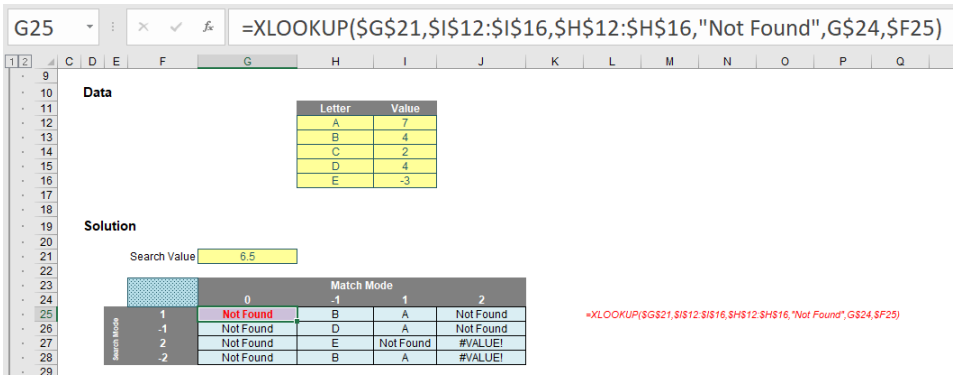
	Graham	Alice	Mitch	Wendy	Xiu	Brian	Toby
Student ID No.	1333	1331	1392	1867	3589	2364	3717

Solution

Student	Xiu	
XLOOKUP	3589	=XLOOKUP(H18,J12:P12,J13:P13)
HLOOKUP (3 args)	3717	=HLOOKUP(H18,J12:P13,ROWS(I12:I13))
HLOOKUP (4 args)	3589	=HLOOKUP(H18,J12:P13,ROWS(I12:I13),FALSE)
ID No.	1331	
XLOOKUP	Alice	=XLOOKUP(H27,J13:P13,J12:P12)
HLOOKUP	No thank you	*Ha ha*

Here, this highlights what happens if I try to deduce the student name from the Student ID. **HLOOKUP** cannot refer to earlier rows, just as **VLOOKUP** cannot consider columns to the left. Given any unused elements of the table are ignored also, it's just good news all round. Goodbye limitations, hello **XLOOKUP**.

Indeed, things get even more interesting when you start considering **XLOOKUP**'s final two arguments, namely **match_mode** and **search_mode**, *viz*.



Data

Letter	Value
A	7
B	4
C	2
D	4
E	-3

Solution

Search Value: 6.5

Search Mode \ Match Mode	0	-1	1	2
1	Not Found	B	A	Not Found
-1	Not Found	D	A	Not Found
2	Not Found	E	Not Found	#VALUE!
-2	Not Found	B	A	#VALUE!

=XLOOKUP($G$21,$I$12:$I$16,$H$12:$H$16,"Not Found",G$24,$F25)

Notice that I am searching the 'Value' column, which is neither sorted nor contains unique items. Do you see how the results have changed once more, depending upon **match_mode** and **search_mode**?

		Match Mode			
		0	-1	1	2
Search Mode	1	Not Found	B	A	Not Found
	-1	Not Found	D	A	Not Found
	2	Not Found	E	Not Found	#VALUE!
	-2	Not Found	B	A	#VALUE!

The **match_mode** zero (0) returns "Not Found" now instead of *#N/A* because there is no exact match and the formula has now stipulated what to do in such an instance.

When **match_mode** is -1, **XLOOKUP** seeks an exact match or else the largest value less than or equal to **lookup_value** (6.5). That would be 4 – but this occurs more than once (B and D both have a value of 4). **XLOOKUP** chooses depending upon whether it is searching top down (search_mode 1, where B will be identified first) or bottom up (**search_mode** -1, where D will be identified first). Note that with binary searches (with a **search_mode** of 2 or -2), the data needs to be sorted. It isn't – hence we have garbage answers that cannot be relied upon.

With **match_mode** 1, the result is clearer cut. Only one value is the smallest value greater than or equal to 6.5. That is 7, and is related to A. Again, binary search results should be ignored, although it is worth noting "Not Found" occurs when Excel identifies the lookup value has not been found.

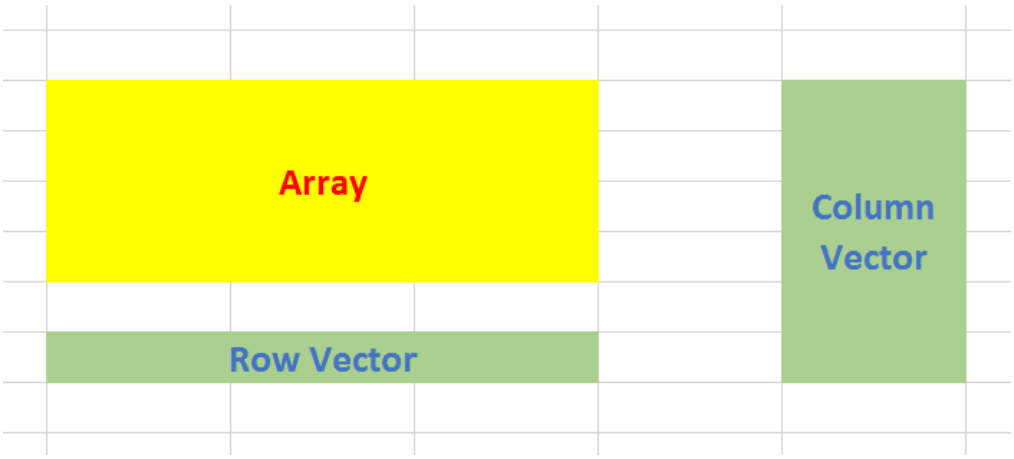
The **match_mode** 2 results are spurious. This is seeking wildcard matches, but there are no matches, hence "Not Found" instead of N/A for the only search_modes that may be seen as creditable (1 and -1). It's interesting to note a binary search causes errors which are not trapped by the new argument.

Clearly binary searches are higher maintenance. In the past, it was worth investing in them as they did return results more quickly. However, according to Microsoft, this is no longer the case: apparently, there is "… no significant benefit to using *(sic)* the binary search options…". If this is indeed the case, then I would strongly recommend not using them going forward with **XLOOKUP**.

Whilst **XLOOKUP** wins hands down against **HLOOKUP** and **VLOOKUP**, the same cannot necessarily be said for **LOOKUP**. You may recall **LOOKUP** has two forms: an array form and a vector form. As a reminder:

- an **array** is a collection of cells consisting of at least two rows and at least two columns
- a **vector** is a collection of cells across just one row (row vector) or down just one column (column vector).

The diagram should be self-explanatory:



The array form of **LOOKUP** looks in the first row or column of an array for the specified value and returns a value from the same position in the last row or column of the same array:

LOOKUP(lookup_value, array)

where:

- **lookup_value** is the value that **LOOKUP** searches for in an array. The **lookup_value** argument can be a number, text, a logical value, or a name or reference that refers to a value
- **array** is the range of cells that contains text, numbers, or logical values that you want to compare with **lookup_value**.

The array form of **LOOKUP** is very similar to the **HLOOKUP** and **VLOOKUP** functions. The difference is that **HLOOKUP** searches for the value of **lookup_value** in the first row, **VLOOKUP** searches in the first column, and **LOOKUP** searches according to the dimensions of array.

If **array** covers an area that is wider than it is tall (*i.e.* it has more columns than rows), **LOOKUP** searches for the value of **lookup_value** in the first row and returns the result from the last row. Otherwise, **LOOKUP** searches for the value of lookup_value in the first column and returns the result from the last column instead.

The alternative form is the vector form:

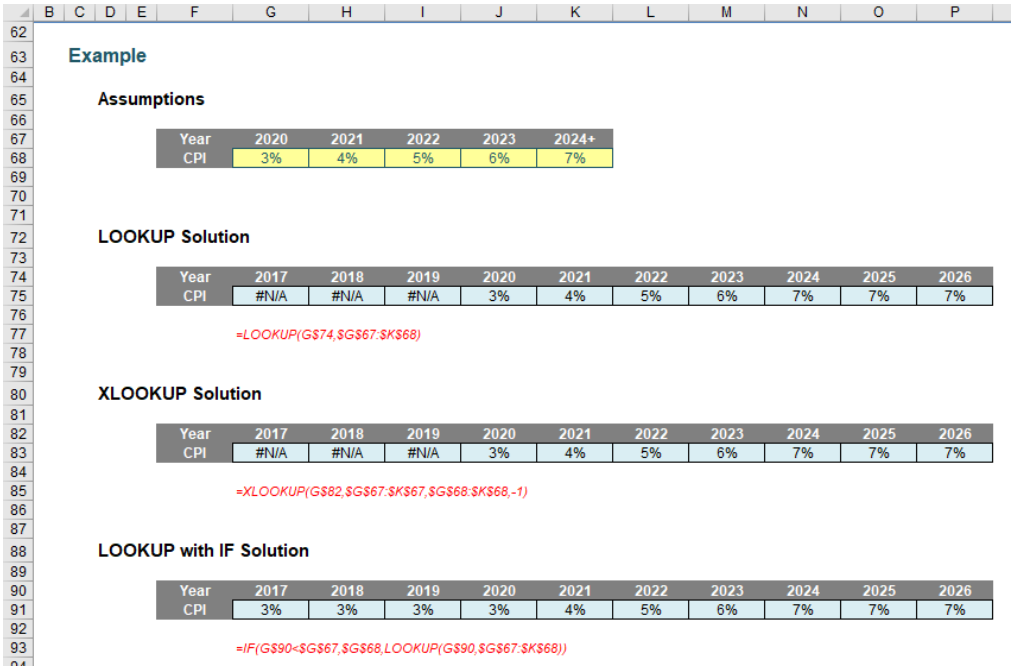
LOOKUP(lookup_value, lookup_vector, [result_vector])

The **LOOKUP** function vector form syntax has the following arguments:

- **lookup_value** is the value that **LOOKUP** searches for in the first vector
- **lookup_vector** is the range that contains only one row or one column
- **[result_vector]** is optional – if ignored, **lookup_vector** is used – this is the where the result will come from and must contain the same number of cells as the **lookup_vector**.

Like the default versions of **HLOOKUP** and **VLOOKUP**, **lookup_value** must be located in a range of ascending values.

Let me demonstrate with an example:



Example

Assumptions

Year	2020	2021	2022	2023	2024+
CPI	3%	4%	5%	6%	7%

LOOKUP Solution

Year	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026
CPI	#N/A	#N/A	#N/A	3%	4%	5%	6%	7%	7%	7%

=LOOKUP(G$74,$G$67:$K$68)

XLOOKUP Solution

Year	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026
CPI	#N/A	#N/A	#N/A	3%	4%	5%	6%	7%	7%	7%

=XLOOKUP(G$82,$G$67:$K$67,$G$68:$K$68,-1)

LOOKUP with IF Solution

Year	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026
CPI	3%	3%	3%	3%	4%	5%	6%	7%	7%	7%

=IF(G$90<$G$67,$G$68,LOOKUP(G$90,$G$67:$K$68))

LOOKUP is a great function to use with time series analysis / forecasting. Dates are in ascending order and the **LOOKUP** syntax is remarkably simple. As a modeller, I use it regularly when I am modelling many more forecast periods than I want assumption periods.

Here, you can see I carry assumptions only for 2020 until 2024 (the final value is 2024, just with a "+" in number formatting). The formula

=LOOKUP(G$74,$G$67:$K$68)

returns the corresponding value for the period that is either an exact match or else the largest value less than or equal to the **lookup_value**. **LOOKUP** uses the top row of the table for looking up its data and the final row for returning the corresponding value. Simple. As for **XLOOKUP**:

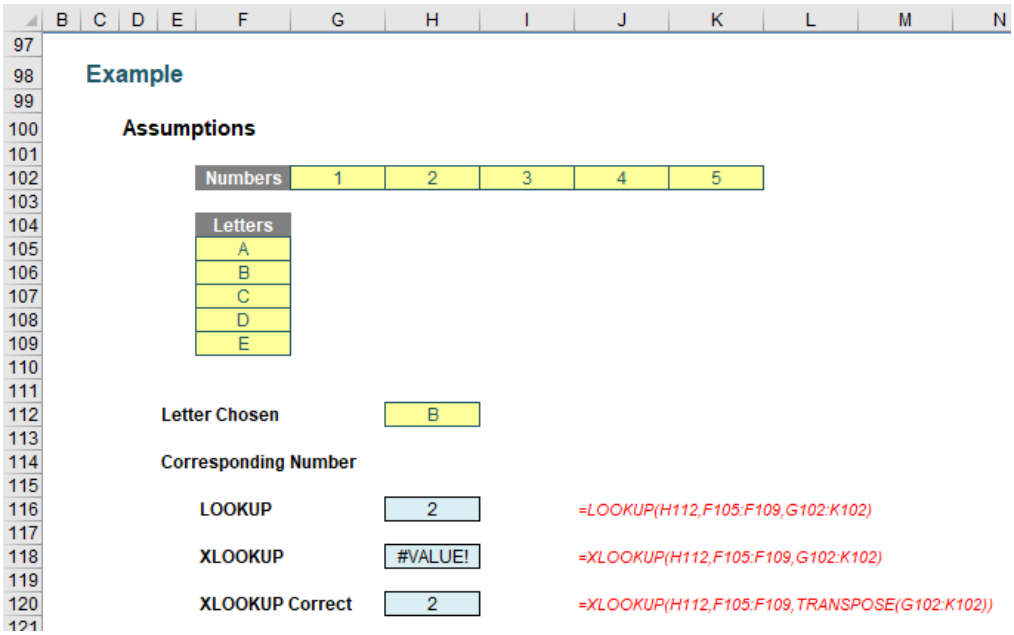
=XLOOKUP(G$82,$G$67:$K$67,$G$68:$K$68,-1)

This formula is longer and requires two additional arguments (**match_mode** -1 is required to mirror the behaviour of LOOKUP). Indeed, given that an **IF** statement is required to ensure no errors for earlier periods, *e.g.*

=IF(G$90<$G$67,$G$68,LOOKUP(G$90,$G$67:$K$68))

it may be argued that **LOOKUP** is a simpler function to use here than its counterpart.

This isn't the only time **LOOKUP** outperforms **XLOOKUP**:



Example

Assumptions

Numbers	1	2	3	4	5

Letters
A
B
C
D
E

Letter Chosen: B

Corresponding Number

LOOKUP: 2 =LOOKUP(H112,F105:F109,G102:K102)

XLOOKUP: #VALUE! =XLOOKUP(H112,F105:F109,G102:K102)

XLOOKUP Correct: 2 =XLOOKUP(H112,F105:F109,TRANSPOSE(G102:K102))

Here, we do see a limitation of **XLOOKUP**. Whilst the third argument of **XLOOKUP**, **results_array**, does not need to be a vector, it cannot be the transposition of the **lookup_vector**. You would have to transpose it using the **TRANSPOSE** function, for example. This makes **LOOKUP** much easier to use – compare:

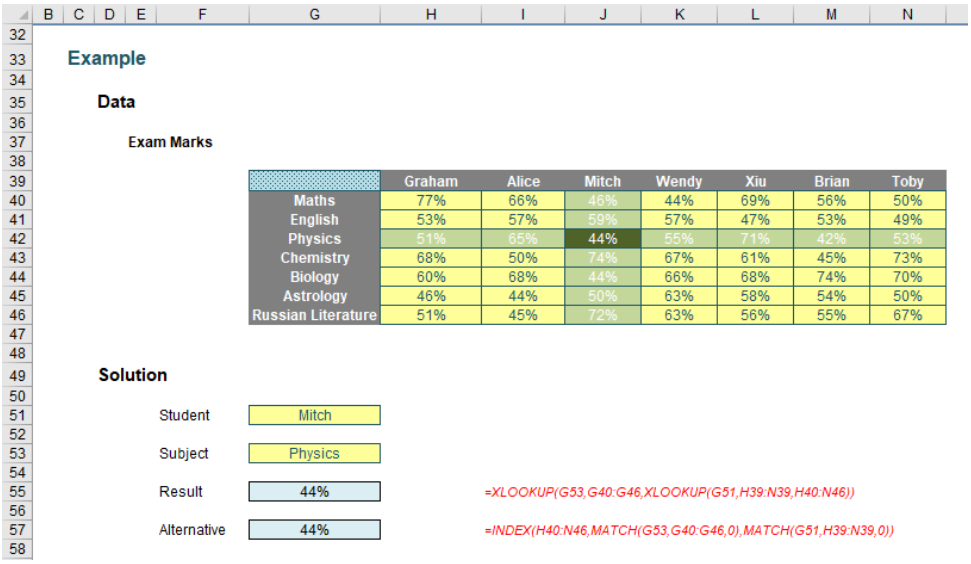
=LOOKUP(H112,F105:F109,G102:K102)

with

=XLOOKUP(H112,F105:F109,TRANSPOSE(G102:K102))

In this instance, **LOOKUP** wins.

XLOOKUP can be used to perform a two-way match, similar to **INDEX MATCH MATCH**:



Example

Data

Exam Marks

	Graham	Alice	Mitch	Wendy	Xiu	Brian	Toby
Maths	77%	66%	46%	44%	69%	56%	50%
English	53%	57%	59%	57%	47%	53%	49%
Physics	51%	65%	44%	55%	71%	42%	53%
Chemistry	68%	50%	74%	67%	61%	45%	73%
Biology	60%	68%	44%	66%	68%	74%	70%
Astrology	46%	44%	50%	63%	58%	54%	50%
Russian Literature	51%	45%	72%	63%	56%	55%	67%

Solution

Student	Mitch	
Subject	Physics	
Result	44%	=XLOOKUP(G53,G40:G46,XLOOKUP(G51,H39:N39,H40:N46))
Alternative	44%	=INDEX(H40:N46,MATCH(G53,G40:G46,0),MATCH(G51,H39:N39,0))

Many advanced users might use the formula

=INDEX(H40:N46,MATCH(G53,G40:G46,0),MATCH(G51,H39:N39,0))

where:

- **INDEX(array, row_number, [column_number])** returns a value or the reference to a value from within a table or range (list) citing the **row_number** and the **column_number**
- **MATCH(lookup_value, lookup_vector, [match_type])** returns the relative position of an item in an array that (approximately) matches a specified value. It's most commonly used with **match_type** zero (0), which requires an exact match.

Therefore, this formula finds the position in the row for the student and the position in the column of the subject. The intersection of these two provides the required result.

XLOOKUP does it differently:

=XLOOKUP(G53,G40:G46,XLOOKUP(G51,H39:N39,H40:N46))

Welcome to the wonderful world of the *nested* **XLOOKUP** function! Here, the internal formula

=XLOOKUP(G51,H39:N39,H40:N46)

demonstrates a key difference between this and your typical lookup function – the first argument is a cell, the second argument is a column vector and the third is an array – with, most importantly, the same number of rows as the **lookup_vector**. This means it returns a column vector of data, not a single value. This is great news in the brave new world of dynamic arrays.

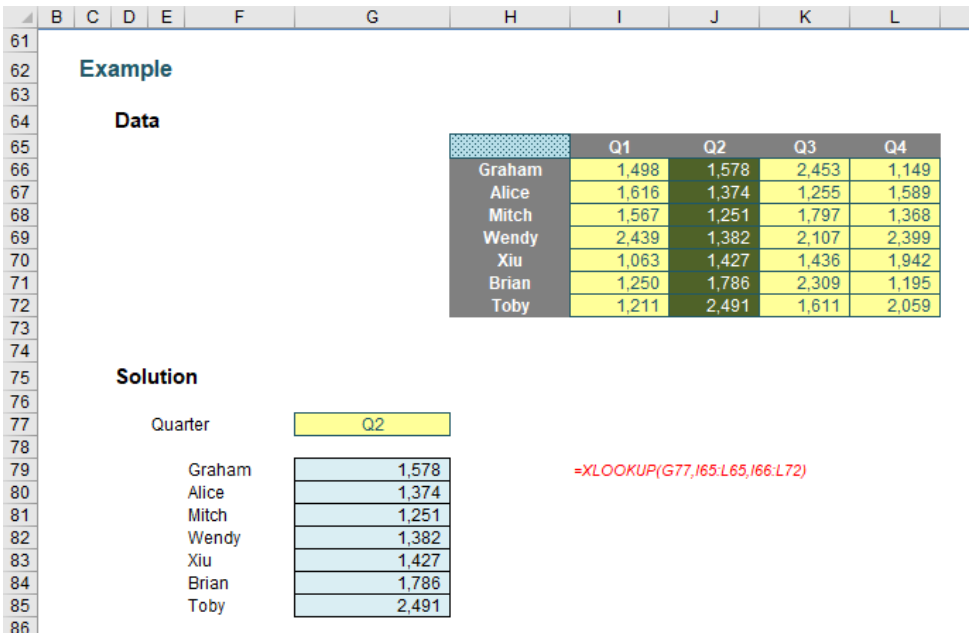
In essence, this means the formula resolves to

=XLOOKUP(G53,G40:G46,J40:J46)

as **J40:J46** is the resultant vector of **=XLOOKUP(G51,H39:N39,H40:N46)**. This is a really powerful – and virtually new – concept to get your head around, that admittedly **SUMPRODUCT** exploits too. Once you understand this, it's clear how this formula works and opens your eyes to the power of nested **XLOOKUP** functions.

I can't believe I am talking about the virtues of nested functions here! Let me change the subject quickly…

To show you how dynamic arrays can make the most of being able to create resultant vectors, consider the following example:



Example

Data

	Q1	Q2	Q3	Q4
Graham	1,498	1,578	2,453	1,149
Alice	1,616	1,374	1,255	1,589
Mitch	1,567	1,251	1,797	1,368
Wendy	2,439	1,382	2,107	2,399
Xiu	1,063	1,427	1,436	1,942
Brian	1,250	1,786	2,309	1,195
Toby	1,211	2,491	1,611	2,059

Solution

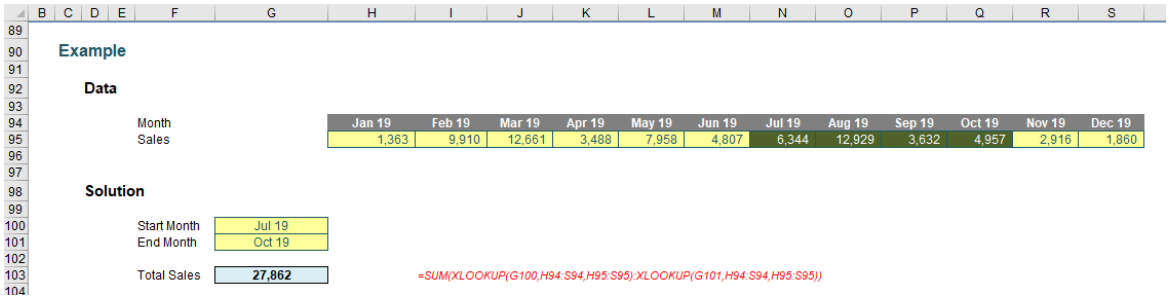
Quarter	Q2	
Graham	1,578	=XLOOKUP(G77,I65:L65,I66:L72)
Alice	1,374	
Mitch	1,251	
Wendy	1,382	
Xiu	1,427	
Brian	1,786	
Toby	2,491	

The formula

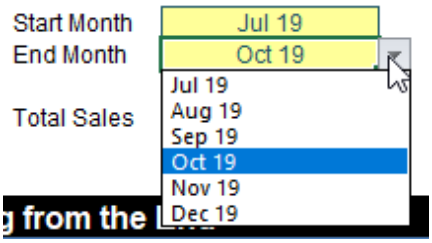
=XLOOKUP(G77,I65:L65,I66:L72)

again resolves to a vector – but this time is allowed to spill as a dynamic array. Obviously, this will only work in Office 365, but it's a very useful tool that might just make you think it's time to drop that perpetual licence.

Once you start playing with the dynamic range side, you can start to get imaginative. For example:

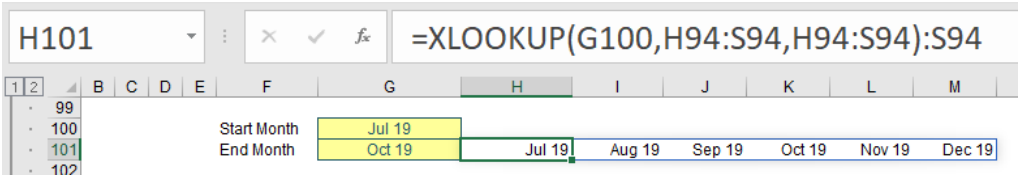


In this illustration, I want to calculate the sales between two periods:



This might seem like a simple drop-down list using data validation (**ALT + D + L**), but **XLOOKUP** has been used in determining the list to be used for the end months.

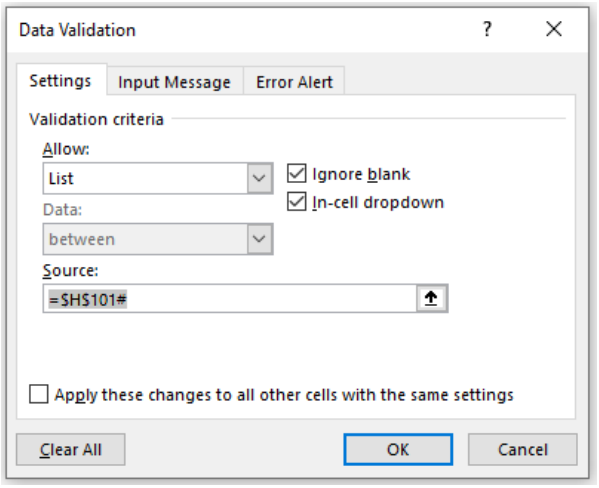
Let me explain. I have hidden the range of relevant dates in cell **H101** spilled across



XLOOKUP can return a reference, so the formula

=XLOOKUP(G100,H94:S94,H94:s94):S94

evaluates to the row vector **N94:S94** (since the start month is July). This spilled dynamic array formula is then referenced in the data validation:



(You may recall **$H$101#** means the spilled range starting in cell **H101**.) It should be noted that the formula **=XLOOKUP(G100,H94:S94,H94:s94):S94** may not be used directly in the 'Data Validation' dialog, but this is a neat trick to ensure you cannot select an end month before the start month (assuming you are a rational human being that selects the start before the end!).

The formula to sum the sales then is

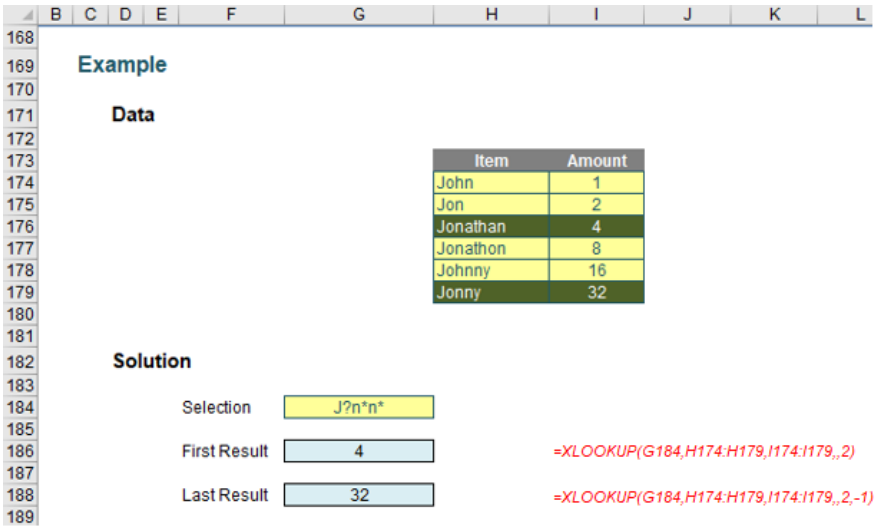
=SUM(XLOOKUP(G100,H94:S94,H95:S95):XLOOKUP(G101,H94:S94, H95:S95))

Again, this uses the fact **XLOOKUP** can return a reference, so this formula equates to

=SUM(N95:Q95)

Easy! Now I am combining two **XLOOKUP** formulae with a colon (:) to form a range. This joins other illustrious functions used this way such as **CHOOSE, IF, IFS, INDEX, INDIRECT, OFFSET, SINGLE (@), SWITCH** and **TEXT**. First nesting, now joining – what's next?

Seeking partial matches (sounds like an unfussy dating agency!) suddenly became a lot easier too. You can use wildcards if you want to – just set the **match_mode** to 2:



Here, I am searching for **J?n*n*** - which is fine as long as you know what the wildcard characters mean:

- **?** means "any character", but just one character. If you wanted to make space for two and only two characters you would use **??**
- ***** means "any number of characters' – including zero.

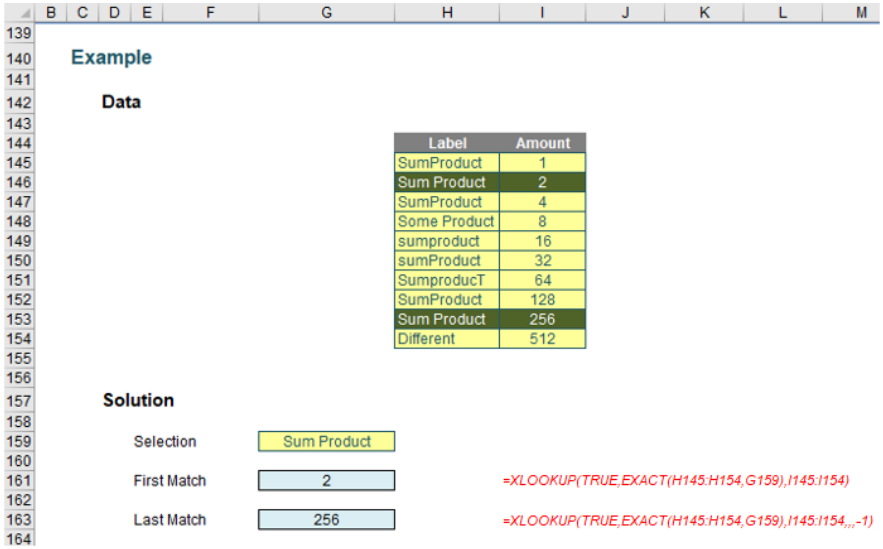
For example, **M?n*m*** would identify "Manmade", "minimum" and "Manikum" but would not accept "millennium". Here, our formulae

=XLOOKUP(G184,H174:H179,I174:I179,,2)

=XLOOKUP(G184,H174:H179,I174:I179,,2,-1)

would locate the first and last items that satisfied the condition **J?n*n*** (*i.e.* "Jonathan" and "Jonny" respectively).

But what if you wanted an exact match with case sensitivity? You just have to think a little but outside of the proverbial box:



Here, we use another feature of **XLOOKUP** – its ability to search a virtual vector, *i.e.* one that has been constructed in memory, rather than physically within the spreadsheet cells. Consider the formula

=XLOOKUP(TRUE,EXACT(H145:H154,G159),I145:I154)

Here, the interim calculation **=EXACT(H145:H154,G159)**, looks at the range **H145:H154** and deduces whether the cells are an exact match for the selection 'Sum Product' in cell **G159**. The **EXACT** function would evaluate as

{FALSE; TRUE; FALSE; FALSE; FALSE; FALSE; FALSE; FALSE; TRUE; FALSE}

Therefore, the formula coerces to

=XLOOKUP(TRUE,{FALSE; TRUE; FALSE; FALSE; FALSE; FALSE; FALSE; FALSE; TRUE; FALSE},I145:I154)

and then the formula becomes simple to understand.

Looking Up Data Revisited

*With many of us currently "working from home" / quarantined, there are only so Zoom / Teams calls and virtual parties you can make before you reach your (data) limit. Perhaps they should measure data allowance in blood pressure millimetres of mercury (mmHg). To try and keep our readers engaged, we will continue to reproduce some of our popular **Final Friday Fix** challenges from yesteryear in this and upcoming newsletters. One suggested solution may be found later in this newsletter. Here's this month's…*

If you weren't aware, it's possible to make specific characters bold in a cell, without emboldening others. The same applies to italics, underline, and so on. Therefore, you can **create** *lines* of text that look like this, **quite easily**.

The Challenge

This month's challenge is likely to need a VBA solution just for a change! Can you find a way to extract and output just the bold characters from a cell?

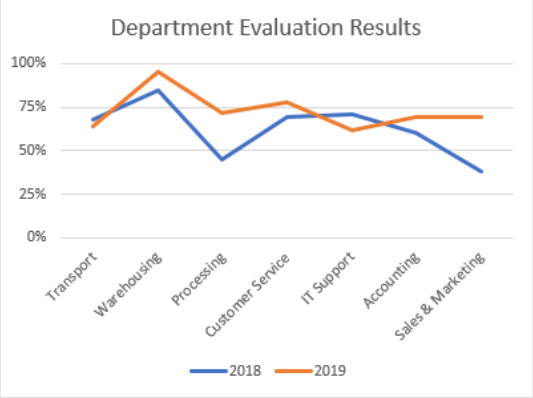
Sound easy? Try it. One solution just might be found later in this newsletter – but no reading ahead!

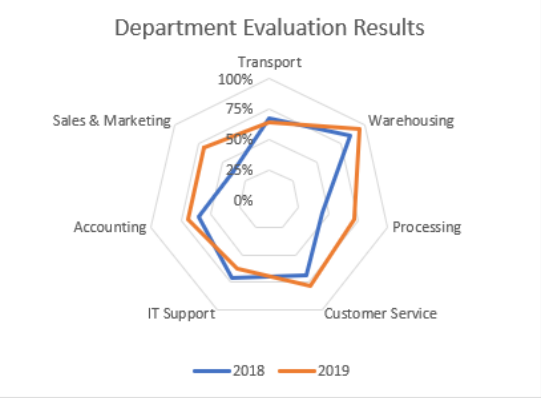
Charts and Dashboards

It's time to chart our progress with an introductory series into the world of creating charts and dashboards in Excel. This month, we look at Radar charts.

A Radar chart, also known as a Spider chart or a Web chart, shows movements in data relative to both a central point and to the other data points. Where a Line chart has a horizontal axis, the axis in a Radar chart is effectively wrapped around so that each category becomes like a spoke on a wheel. The length of each spoke which extends from the centre of the chart to the outermost point on the chart represents the vertical axis of a similar Line chart.

Let's imagine that a company management were asked to provide input about the performance of the company's departments at the end of the 2018 and 2019 financial years. This information was collated to produce a score for each department expressed as a percentage. The more satisfied management were with a particular department, the higher the percentage. This data could quite easily be used to produce a Line chart, Bar chart or Column chart, but using a Radar chart gives a different perspective on the results. Below is an example of how the same data would look on a Line chart versus a Radar chart:



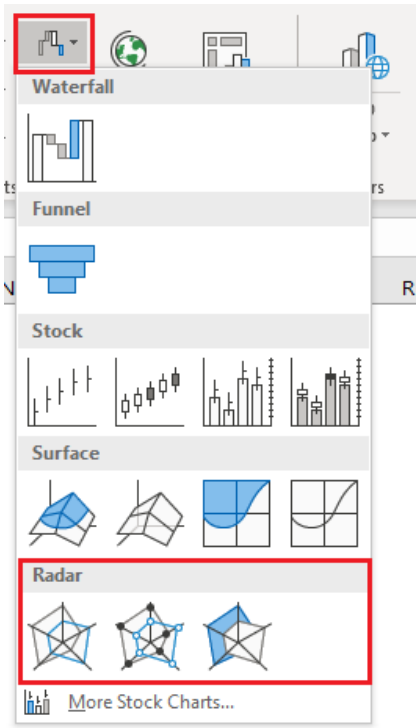


So how do we produce a Radar chart? First, prepare the data table. In the case of our example, we may list each department with their evaluation percentages for 2018 and 2019:

Department Evaluation Results		
Department	2018	2019
Transport	68%	64%
Warehousing	85%	95%
Processing	45%	72%
Customer Service	69%	78%
IT Support	71%	62%
Accounting	60%	69%
Sales & Marketing	38%	69%

Then, we highlight the data and the column headings (do not select the table heading) and go to the Insert tab on the Ribbon. The Radar chart is under the last small icon along the top of the Charts section. Alternatively, we may click on the 'Recommended Charts' icon or the small arrow in the bottom right of the Charts section, and then go to the 'All Charts' tab to locate the Radar Chart. There are just three variations for the Radar Chart:

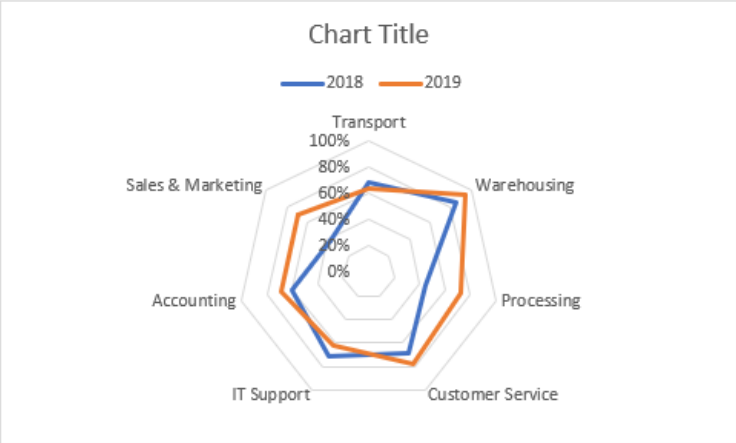
- the first plots the data series using lines only
- the second variation maps the data series with lines and markers
- the third chart option fills the area within the shape created by each data series.



Please be aware of the limitations of using the filled Radar chart. It is possible that the area covered by one data series on the Radar chart might overlap data points from another series, meaning you cannot see the data points underneath and therefore part of the area occupied by the second series. If you are using the filled Radar chart, it is highly recommended that you make the area partially transparent so you can see any data points and area underneath each data series. Transparency is found by selecting the data series, right click and choose 'Format Data Series', go to the Marker section and under Fill there is an option to set the Transparency.

Also, it is important to note that by joining the data points with lines, it can be interpreted that these data points may relate to each other, but this may not be correct. With our example for instance, while the data points representing the evaluation rating for each department are joined together by lines, the score for one department has no bearing or relationship to the score of the adjacent departments on the chart.

Using the department performance result data, our Radar Chart initially looks like the following:

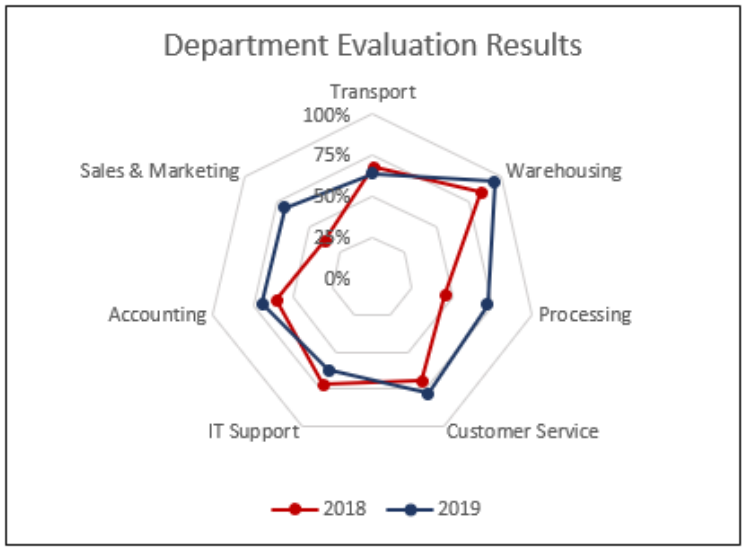


There are some changes we need to make:

- to move the legend below the chart, select the legend, right-click and choose 'Format Legend', then specify that you require the legend position to be at the bottom of the chart
- we'd also like to see the 50% line on the graph. To achieve this, select the axis labels (by clicking on one of the 0% to 100% labels), right-click and choose 'Format Axis', then under the 'Axis Options', change the Major setting under Units to 0.25 instead of 0.2. This will set the chart units to be 0%, 25%, 50%, 75% and 100%
- also, still within 'Format Axis', under the 'Fill & Line' area (the bucket icon), we may add spoke lines by changing the lines to be solid and assigning a colour to them

- The “rings” in the chart are the equivalent of the horizontal gridlines of the Line chart. To change the formatting for these gridlines, simply click on one of the “rings”, right-click and choose ‘Format Gridlines’, and proceed to change the colour, width, type, *etc*.

Once we have applied all the formatting, the final chart looks similar to the following:

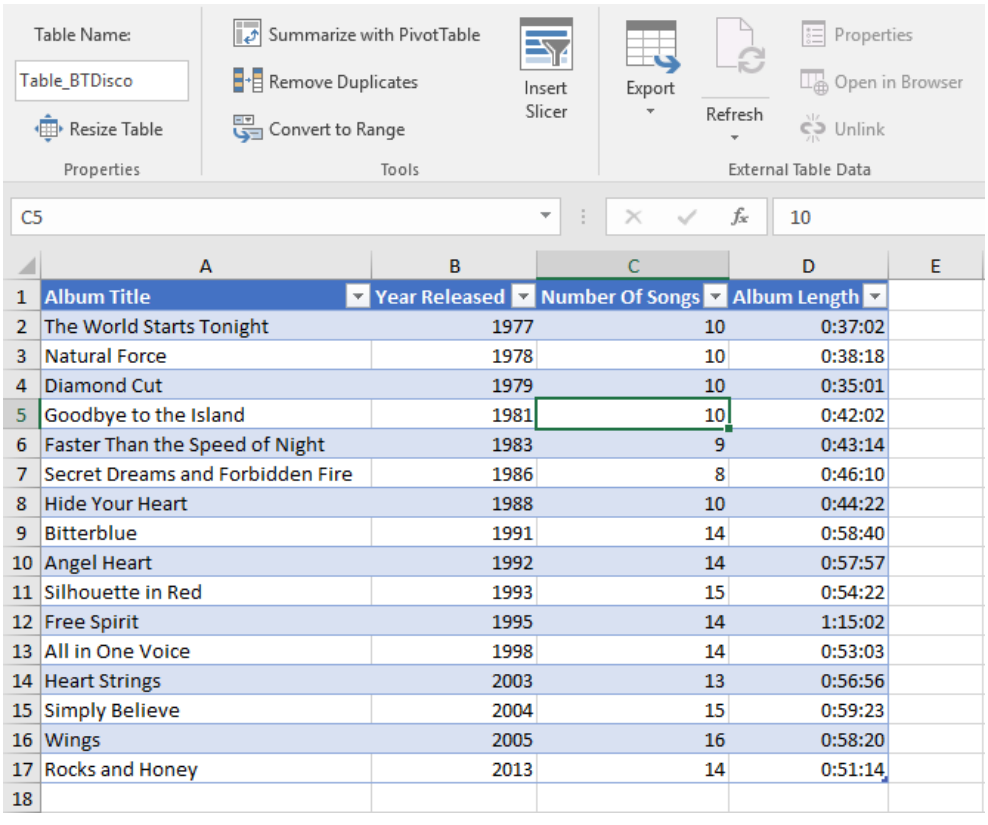


More next month…

Visual Basics

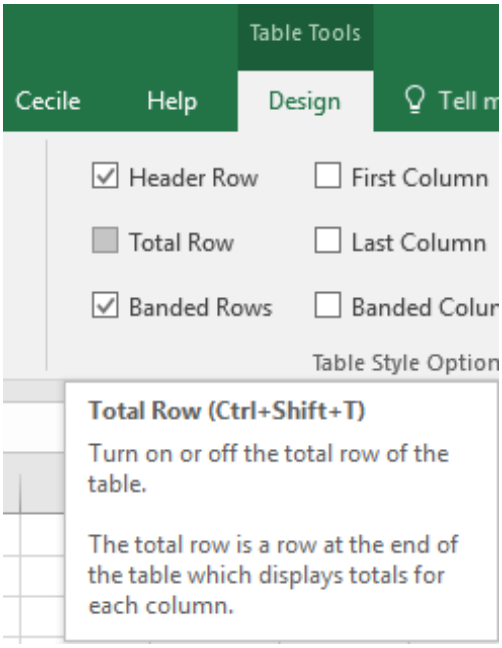
We thought we’d run an elementary series going through the rudiments of Visual Basic for Applications (VBA) as a springboard for newer users. This month, we continue looking at using ListObjects to manipulate Tables within an Excel workbook in VBA, this time featuring the Totals Row.

Sometimes, tables don’t have totals rows. Let’s consider the following table:



	A	B	C	D	E
1	Album Title	Year Released	Number Of Songs	Album Length	
2	The World Starts Tonight	1977	10	0:37:02	
3	Natural Force	1978	10	0:38:18	
4	Diamond Cut	1979	10	0:35:01	
5	Goodbye to the Island	1981	10	0:42:02	
6	Faster Than the Speed of Night	1983	9	0:43:14	
7	Secret Dreams and Forbidden Fire	1986	8	0:46:10	
8	Hide Your Heart	1988	10	0:44:22	
9	Bitterblue	1991	14	0:58:40	
10	Angel Heart	1992	14	0:57:57	
11	Silhouette in Red	1993	15	0:54:22	
12	Free Spirit	1995	14	1:15:02	
13	All in One Voice	1998	14	0:53:03	
14	Heart Strings	2003	13	0:56:56	
15	Simply Believe	2004	15	0:59:23	
16	Wings	2005	16	0:58:20	
17	Rocks and Honey	2013	14	0:51:14	
18					

The Totals Row is easily found in the Table Menu here (or **CTRL + SHIFT + T** for you keyboard shortcut enthusiasts):



However, how can we do this in VBA? It is simply the ShowTotals property of the *ListObject*. This is a Boolean setting; if it is TRUE then the Total Row is displayed (and you may switch it off by setting it to FALSE).

```
(General)
Option Explicit

Sub ShowTotals()

    Dim MyTable As ListObject
    Set MyTable = Range("Table_BTDisco").ListObject

    MyTable.ShowTotals = True

End Sub
```

Then the Totals Row appears:

	A	B	C	D	E
1	Album Title	Year Released	Number Of Songs	Album Length	
2	The World Starts Tonight	1977	10	0:37:02	
3	Natural Force	1978	10	0:38:18	
4	Diamond Cut	1979	10	0:35:01	
5	Goodbye to the Island	1981	10	0:42:02	
6	Faster Than the Speed of Night	1983	9	0:43:14	
7	Secret Dreams and Forbidden Fire	1986	8	0:46:10	
8	Hide Your Heart	1988	10	0:44:22	
9	Bitterblue	1991	14	0:58:40	
10	Angel Heart	1992	14	0:57:57	
11	Silhouette in Red	1993	15	0:54:22	
12	Free Spirit	1995	14	1:15:02	
13	All in One Voice	1998	14	0:53:03	
14	Heart Strings	2003	13	0:56:56	
15	Simply Believe	2004	15	0:59:23	
16	Wings	2005	16	0:58:20	
17	Rocks and Honey	2013	14	0:51:14	
18	**Total**			**16**	
19					

Notice how it has put a formula in the last column which is the default setting of showing the totals row:

=SUBTOTAL(103,[Album Length])

Why? Excel makes a rough judgment about which of the **SUBTOTAL** functions it would like to use and in this case has chosen 103 – COUNT. Sometimes it doesn't use the right one. To edit the Totals Row, you could very easily edit it by using the *TotalsRowRange* property of *ListObject*. Let's delete the word "Total" in the row.

```
(General)
Option Explicit

Sub TotalsRowDelete()

    Dim MyTable As ListObject
    Set MyTable = Range("Table_BTDisco").ListObject

    MyTable.TotalsRowRange.Cells(1, 1).Clear

End Sub
```

It then results in the following (as expected):

Album Title	Year Released	Number Of Songs	Album Length
The World Starts Tonight	1977	10	0:37:02
Natural Force	1978	10	0:38:18
Diamond Cut	1979	10	0:35:01
Goodbye to the Island	1981	10	0:42:02
Faster Than the Speed of Night	1983	9	0:43:14
Secret Dreams and Forbidden Fire	1986	8	0:46:10
Hide Your Heart	1988	10	0:44:22
Bitterblue	1991	14	0:58:40
Angel Heart	1992	14	0:57:57
Silhouette in Red	1993	15	0:54:22
Free Spirit	1995	14	1:15:02
All in One Voice	1998	14	0:53:03
Heart Strings	2003	13	0:56:56
Simply Believe	2004	15	0:59:23
Wings	2005	16	0:58:20
Rocks and Honey	2013	14	0:51:14
			16

You might wish to populate the Totals Row with calculations. This is done using the *ListColumns* method of ListObject. Although *ListColumns* hasn't been covered in detail in previous articles, it's straightforward. Columns in a table may be referred to by the *ListColumns* property by using the index or by the header. Then, we use the *TotalsCalculation* method to change the calculation in the row. The following calculations may be used:

Subtotal Number	Excel Function	Function	VBA Syntax
101	AVERAGE	Average	xlTotalsCalculationAverage
102	COUNTA	Count Numbers	xlTotalsCalculationCountNums
103	COUNT	Count	xlTotalsCalculationCount
104	MAX	Max	xlTotalsCalculationMax
105	MIN	Min	xlTotalsCalculationMin
106	PRODUCT	Product	
107	STDEV.S/STDEV	Standard Deviation Sample	xlTotalsCalculationStdDev
108	STDEV.P	Standard Deviation Population	
109	SUM	Sum	xlTotalsCalculationSum
110	VAR	Variance	xlTotalsCalculationVar

Using this table:

```
(General)                                                    TotalsRowFormula
Option Explicit

Sub TotalsRowFormula()

    Dim MyTable As ListObject
    Set MyTable = Range("Table_BTDisco").ListObject

    'Referring to column by index number
    MyTable.ListColumns(2).TotalsCalculation = xlTotalsCalculationMin

    'Referring to column by index number
    MyTable.ListColumns("Number of Songs").TotalsCalculation = xlTotalsCalculationAverage

End Sub
```

Album Title	Year Released	Number Of Songs	Album Length
The World Starts Tonight	1977	10	0:37:02
Natural Force	1978	10	0:38:18
Diamond Cut	1979	10	0:35:01
Goodbye to the Island	1981	10	0:42:02
Faster Than the Speed of Night	1983	9	0:43:14
Secret Dreams and Forbidden Fire	1986	8	0:46:10
Hide Your Heart	1988	10	0:44:22
Bitterblue	1991	14	0:58:40
Angel Heart	1992	14	0:57:57
Silhouette in Red	1993	15	0:54:22
Free Spirit	1995	14	1:15:02
All in One Voice	1998	14	0:53:03
Heart Strings	2003	13	0:56:56
Simply Believe	2004	15	0:59:23
Wings	2005	16	0:58:20
Rocks and Honey	2013	14	0:51:14
	1977	**12.25**	**16**

Notice that **SUBTOTAL** functions 106 and 108 are not available in the VBA Syntax. However, there are two further *TotalsCalculations* that are available: *xlTotalsCalculationNone* which is identical to clearing the cell and *xlTotalsCalculationCustom*, which doesn't appear to do much at all.

```
(General)                                                    TotalsRowFormula
Option Explicit

Sub TotalsRowFormula()

    Dim MyTable As ListObject
    Set MyTable = Range("Table_BTDisco").ListObject

    'Referring to column by index number
    MyTable.ListColumns(2).TotalsCalculation = xlTotalsCalculationNone

    'Referring to column by index number
    MyTable.ListColumns("Number of Songs").TotalsCalculation = xlTotalsCalculationCustom

End Sub
```

C18 =0

	A	B	C	D	E
1	Album Title	Year Released	Number Of Songs	Album Length	
2	The World Starts Tonight	1977	10	0:37:02	
3	Natural Force	1978	10	0:38:18	
4	Diamond Cut	1979	10	0:35:01	
5	Goodbye to the Island	1981	10	0:42:02	
6	Faster Than the Speed of Night	1983	9	0:43:14	
7	Secret Dreams and Forbidden Fire	1986	8	0:46:10	
8	Hide Your Heart	1988	10	0:44:22	
9	Bitterblue	1991	14	0:58:40	
10	Angel Heart	1992	14	0:57:57	
11	Silhouette in Red	1993	15	0:54:22	
12	Free Spirit	1995	14	1:15:02	
13	All in One Voice	1998	14	0:53:03	
14	Heart Strings	2003	13	0:56:56	
15	Simply Believe	2004	15	0:59:23	
16	Wings	2005	16	0:58:20	
17	Rocks and Honey	2013	14	0:51:14	
18			**0**	**16**	
19					

xlTotalsCalculationCustom just puts a **=0** for the formula, which is not very helpful. However, what if you wanted to calculate the average song length? Let's use an array formula using the **AVERAGE** function as follows:

{=AVERAGE(Table_BTDisco[Album Length]/Table_BTDisco[Number Of Songs])}

So how could this be achieved?

The *TotalsRowRange* could be used as above, but ListColumns also has a method *Total*, which allows access to the Totals Range for that particular column. Let's use *ArrayFormula* to put the formula in the **Album Length** column and change the number format to show minutes and seconds.

```
(General)                                                    Tota
Option Explicit

Sub TotalsRowFormula()

    Dim MyTable As ListObject
    Set MyTable = Range("Table_BTDisco").ListObject

    MyTable.ListColumns("Album Length").Total.FormulaArray = _
        "=AVERAGE(Table_BTDisco[Album Length]/Table_BTDisco[Number Of Songs])"
    MyTable.ListColumns("Album Length").Total.NumberFormat = "mm:ss"

End Sub
```

	A	B	C	D
1	Album Title	Year Released	Number Of Songs	Album Length
2	The World Starts Tonight	1977	10	0:37:02
3	Natural Force	1978	10	0:38:18
4	Diamond Cut	1979	10	0:35:01
5	Goodbye to the Island	1981	10	0:42:02
6	Faster Than the Speed of Night	1983	9	0:43:14
7	Secret Dreams and Forbidden Fire	1986	8	0:46:10
8	Hide Your Heart	1988	10	0:44:22
9	Bitterblue	1991	14	0:58:40
10	Angel Heart	1992	14	0:57:57
11	Silhouette in Red	1993	15	0:54:22
12	Free Spirit	1995	14	1:15:02
13	All in One Voice	1998	14	0:53:03
14	Heart Strings	2003	13	0:56:56
15	Simply Believe	2004	15	0:59:23
16	Wings	2005	16	0:58:20
17	Rocks and Honey	2013	14	0:51:14
18			**0**	**04:11**

More next time.

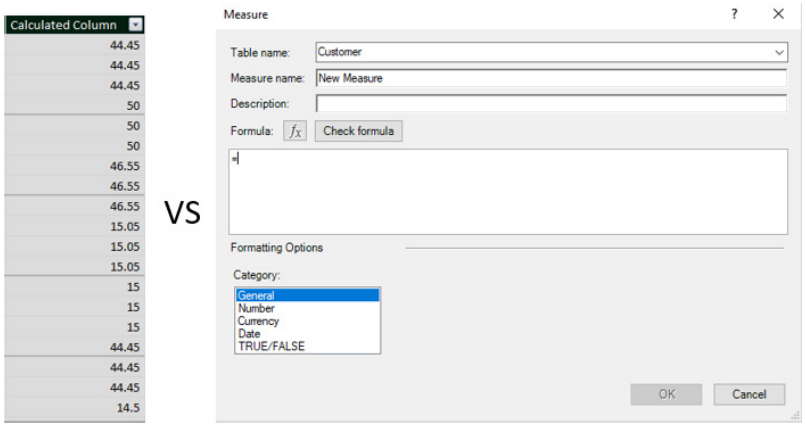
Power Pivot Principles

We continue our series on the Excel COM add-in, Power Pivot. This month, we revisit calculated columns in Power Pivot.

Calculated columns perform a calculation for every individual row in a given table, whereas a measure is only calculated for the filtered, aggregated cells that are used in a PivotTable or a PivotChart; because of this, the formula in a calculated column can be more resource intensive than a formula used in a measure.

For instance, a calculated column in a table with a million rows will always have to calculate one million results. A PivotTable will generally have filters and slicers culling the reporting table to much less than one million rows. Furthermore, any measure is only calculated for the subset of data in each cell in the PivotTable.

Also, note that if a formula in a calculated column has dependencies on object references, such as other columns and other expressions, the calculated column at the end of the dependency cannot be evaluated until all of the other columns have been evaluated. Updating data will cause the entire dependency chain to refresh. This may slow down the responsiveness of the model if there are too many dependencies built into the model.



Keeping these points in mind, unless it is absolutely necessary we should stick to creating measures rather than calculated columns when we can. If multiple calculated columns are needed, we recommend the following:

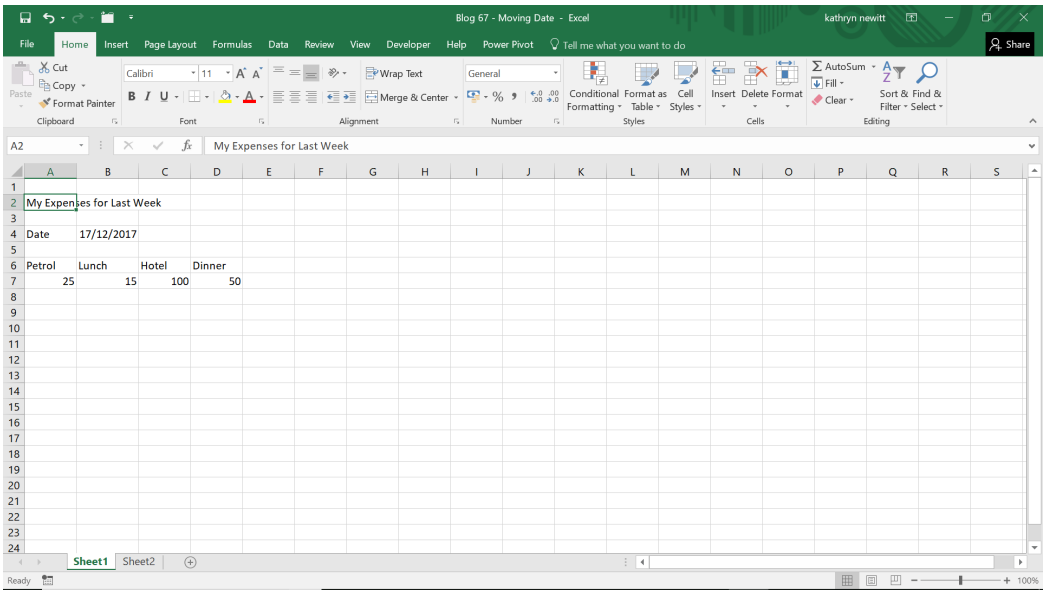
- step out any formula that contain multiple dependencies, with results saved to columns so that we are able to validate the results and evaluate any impacts on performance
- a little more controversially, if updating data with numerous calculated columns with interdependencies, you might wish to consider setting the (re)calculation mode temporarily to manual. Remember to switch the mode back to automatic after updating though.

More *Power Pivot Principles* next month.

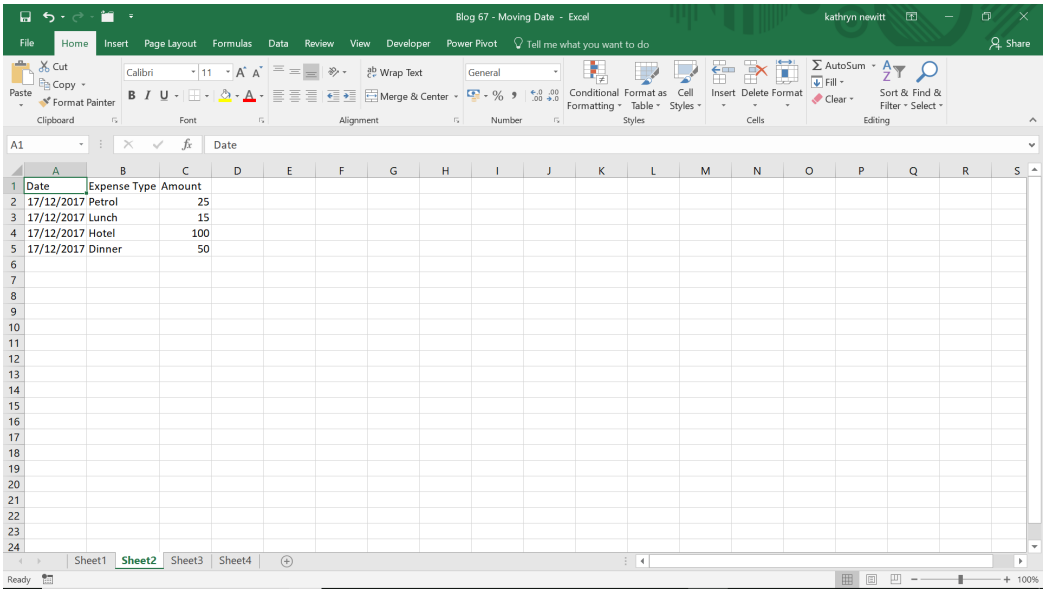
Power Query Pointers

Each month we'll reproduce one of our articles on Power Query (Excel 2010 and 2013) / Get & Transform (Office 365, Excel 2016 and 2019) from www.sumproduct.com/blog. If you wish to read more in the meantime, simply check out our Blog section each Wednesday. This month, look at how to transform extracted data into a useful table.

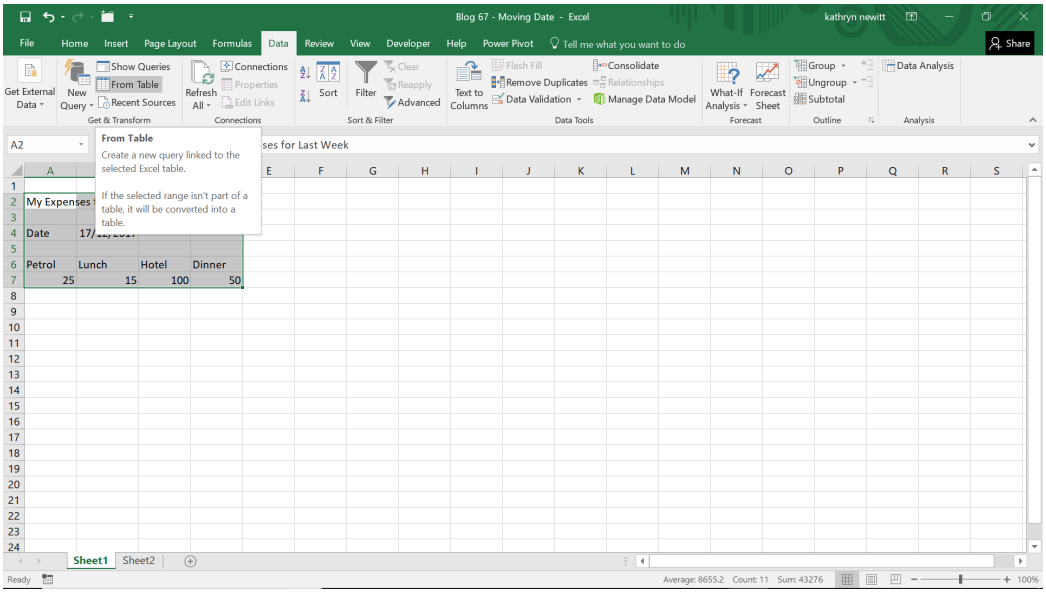
Regular readers will be familiar with our fictional salespeople and their tendency to supply data in the wrong format. Let's meet John.



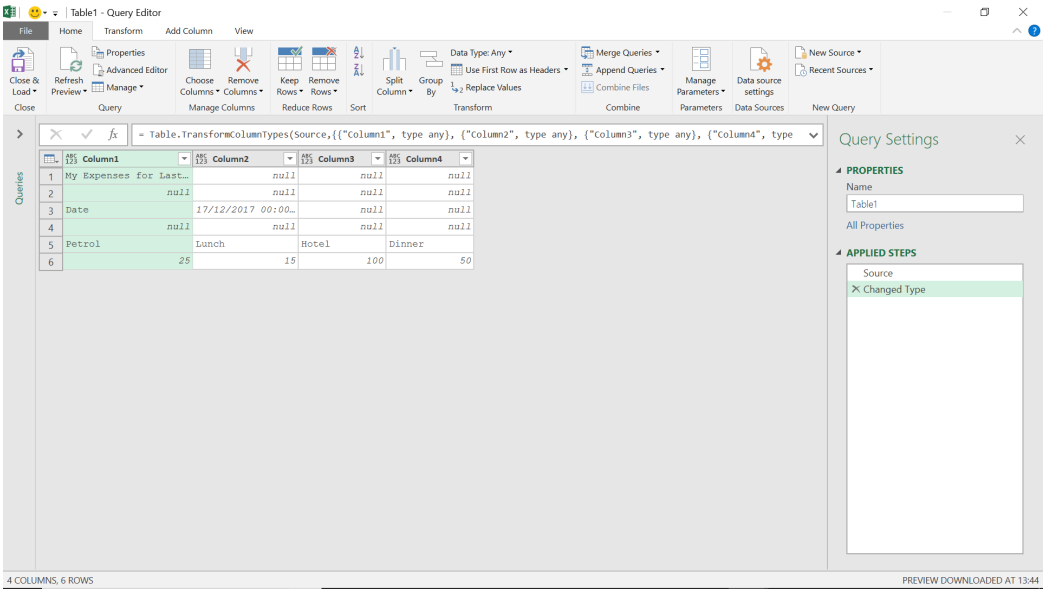
Whilst John has supplied his expenses, the format we would like to see them in is something like this:



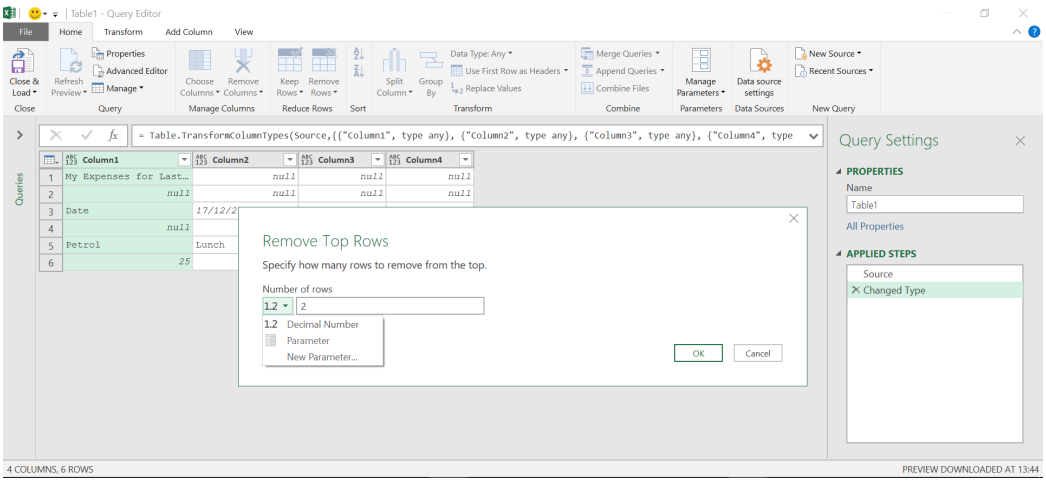
To start the process, let's extract John's data into Power Query:



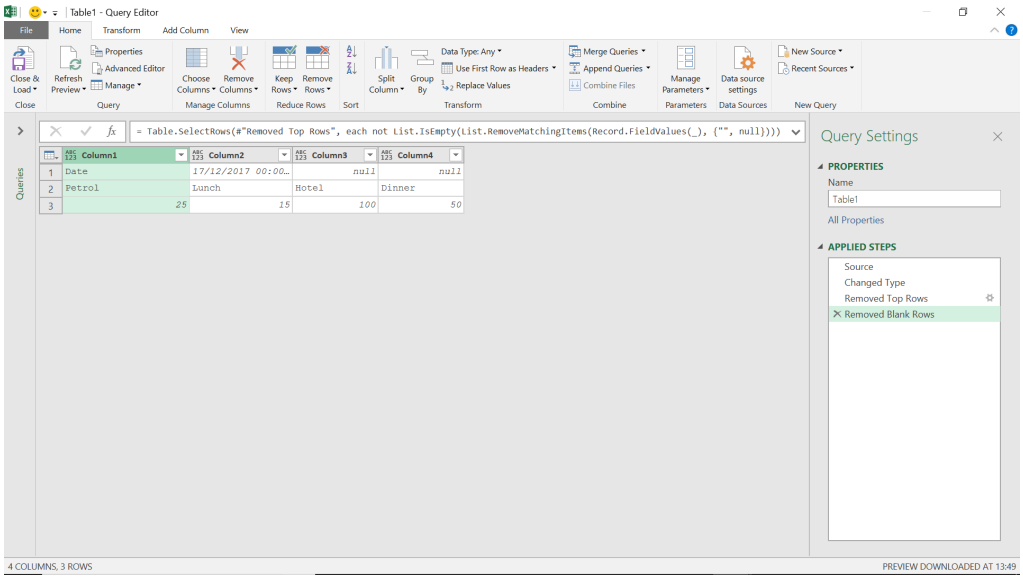
We may select the data and use 'From Table' on the 'Get and Transform' section of the 'Data' tab. Our data will be converted to a Table as part of the process.



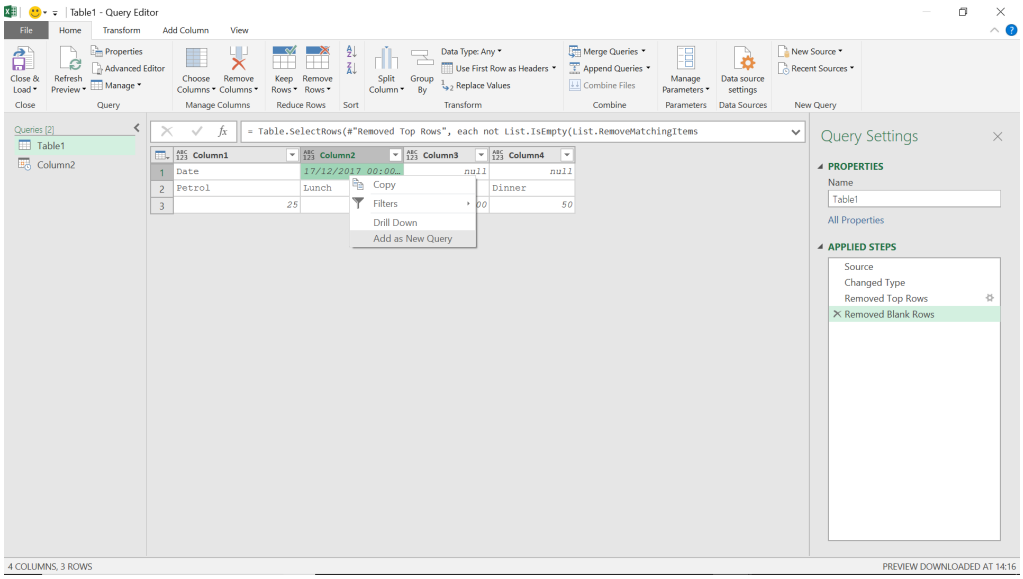
The first two rows are not useful, so our first step is to remove them using the 'Remove Rows' option in the 'Reduce Rows' section.



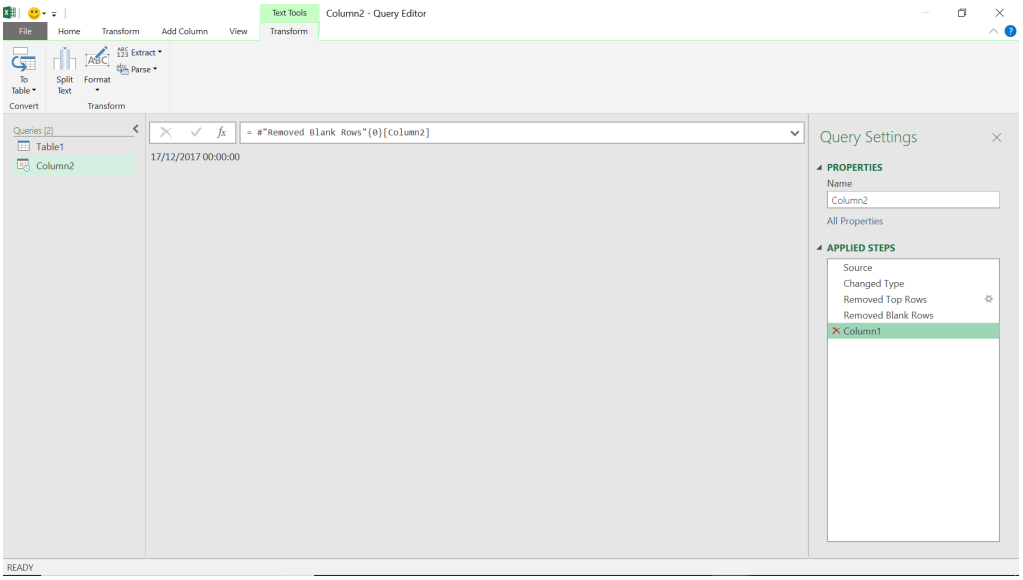
We could remove them based upon a parameter, but we just want to get rid of the first two rows so let's choose the 'Decimal Number' option. We also remove the row of *null* values beneath our 'Date' row by removing blank rows.



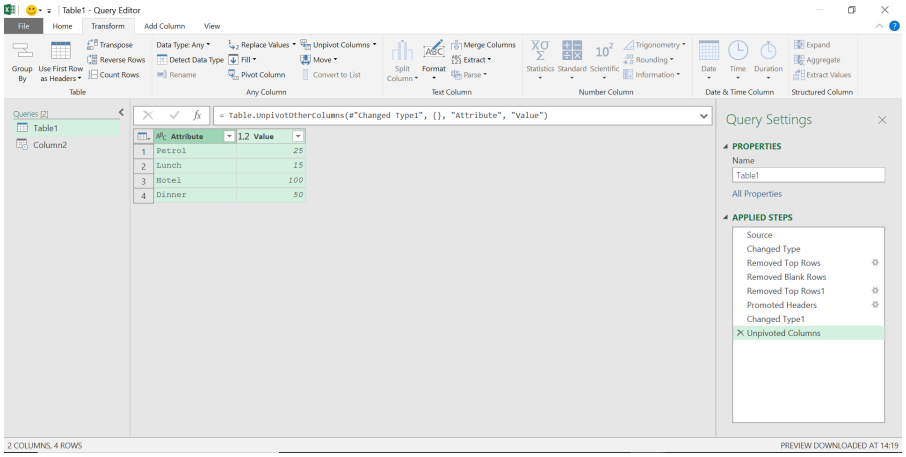
We want to create a column from the **Date** cell. The first step to achieving this is to right-click on the **Date** cell and use the option to 'Add as New Query'. This creates a new query in the queries panel on the left of the screen.



The new query, automatically called '**Column2**', includes our earlier source steps and the value in the top row of **Column2** – the date.



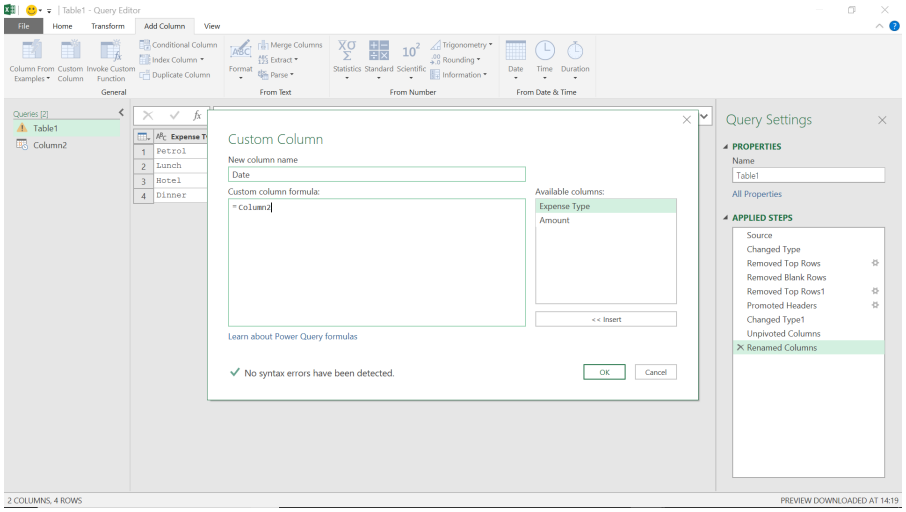
Having created this query, we need to make sure the next steps we add are to the **'Table1'** query. Now we may transform the rest of the data to appear in the format that we would like.



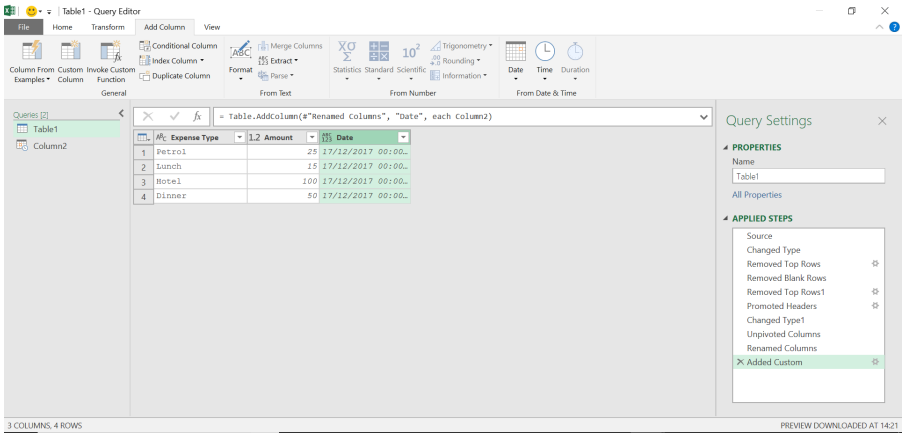
The steps we have taken are:

1. Removed Top Rows1: having moved the date to a separate query, we could remove the 'Date' row
2. Promoted Headers: since we wanted to just keep my expense types and values, we promoted the expense to headers to get rid of the generic **Column1** *etc.* (the 'Changed Type1' step was an automated Power Query step)
3. Unpivoted Columns" we didn't want to keep our expense types as headers; we ultimately want to store them in a column under the heading 'Expense Type', so we unpivoted to get the data as it is shown *(above)*.

Now all that remains is to rename our columns and add a **Date** column. To do this, we will add a custom column.



Having referenced the other query (which is easy to check as I can see it in the left-hand pane), we click 'OK'.



The date appears as a new column.

More next month!

Power BI Updates

January always seems to be a relatively quiet month down at Power BI HQ. As at the time of writing, no new features have been announced – so do watch out for our next newsletter which will probably have double the fun if history were to repeat itself.

More next month, we're sure!!

The A to Z of Excel Functions: IF

So what's the most **I**mportant **F**unction in Excel? Did you realise that's what **IF** is an abbreviation for? Not surprising as I just made it up. However, there is some truth in the jest. The syntax for **IF** demonstrates just how useful this function is for financial modelling:

=IF(logical_test, [value_if_TRUE], [value_if_FALSE])

This function has three arguments:

- **logical_test:** this is the "decider", that is, a test that results in a value of either TRUE or FALSE. Strictly speaking, the **logical_test** tests whether something is TRUE; if not, it is FALSE
- **value_if_TRUE:** what to do if the **logical_test** is TRUE. Note that you do not put square brackets around this argument. This is just the Excel syntax for saying sometimes this argument is optional. If this argument is indeed omitted, this argument will have a default value of TRUE
- **value_if_FALSE:** what to do if the **logical_test** is FALSE (strictly speaking, not TRUE). If this argument is left blank, this argument will have a default value of FALSE.

This function is actually more efficient than it may look at first glance. Whilst the **logical_test** is always evaluated, only one of the remaining two arguments is computed, depending upon whether the **logical_test** is TRUE or FALSE.

Care should be taken with logical tests as this is the source of many, many errors in spreadsheets. Logical tests assess the criterion/criteria stipulated, no more no less. It assumes a binary universe: **X** and **NOT(X)**. This isn't always how our minds think, as I will explain with an exaggerated example.

Intrepid explorer Ivor Challenge is lost in the jungle and needs to find shelter for the night as a rainstorm beckons. Immediately ahead is a clearing with two caves. He writes a formula to determine which cave to sleep in:

=IF(Cave 1 has a bear, sleep in Cave 2, sleep in Cave 1).

The **logical_test** is to check whether Cave 1 contains a bear. As it turns out, it doesn't so he sleeps in there and is mauled to death by the lioness who was in there.

Next day, his wife, Cher Challenge, goes searching for him, gets tired and comes across the same caves and uses the same formula to determine which cave to sleep in:

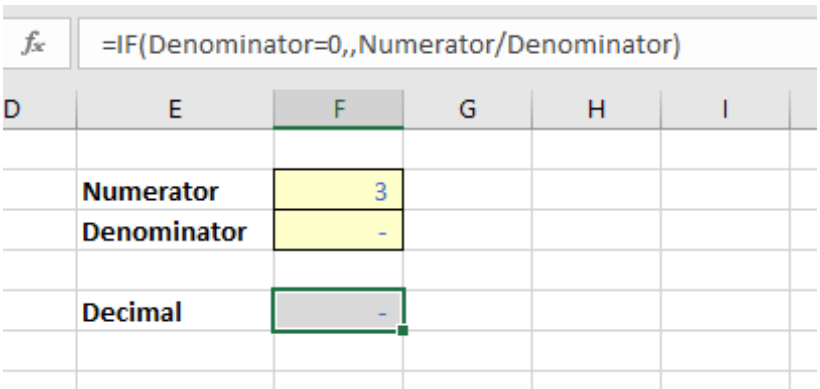
=IF(Cave 1 has a bear, sleep in Cave 2, sleep in Cave 1).

The **logical_test** is to check whether Cave 1 contains a bear. As it turns out, this time there is (together with some human bones) and so she sleeps in Cave 2 and is eaten by the other bear.

When using **IF** formulas, you need to train yourself to think logically like a computer. Common sense does not apply. Consider the logic function **NOT(expression)**, which is everything that is not equivalent to the **expression**. The opposite of a boy is "not a boy": "girl" is incorrect.

Take care with inequalities in particular. The opposite of **x is greater than y** is either **x is less than or equal to y**, or **NOT(x is greater than y)**. This is a common error and it has caused embarrassing mistakes time and time again in business.

Returning to the **IF** function, let's consider an example:



In this example, the intention is to evaluate the quotient **Numerator / Denominator**. However, if the **Denominator** is either blank or zero, this will result in an *#DIV/0!* error. Excel has several errors that it cannot evaluate, such as *#REF!, #NULL, #N/A, #Brown, #Pipe.* OK, so one or two of these I may have made up, but prima facie errors should be avoided in Excel as they detract from the key results and cause the user to doubt the overall model integrity. Worse, in some instances these errors may contribute to Excel crashing and/or corrupting.

This is where **IF** comes in. In my example above,

=IF(Denominator=0,,Numerator/Denominator)

tests whether the **Denominator** is zero. This is the conditional formula. If so, the value is unspecified (blank) and will consequently return a value of zero in Excel; otherwise, the quotient is calculated as intended.

This type of conditional formula is known as creating an **error trap**. Errors are "trapped" and the 'harmless' value of zero is returned instead. You could put "n.a" or "This is an error" as the **value_if_TRUE**, but you get the picture.

It is my preference not to put a zero in for the **value_if_TRUE**: personally, I think a formula looks clearer this way, but inexperienced end users may not understand the formula and you should consider your audience when deciding to put what may appear to be an unnecessary zero in a formula. The aim is to keep it simple **for the end user**.

An **IF** statement is often used to make a decision in the model:

=IF(Decision_Criterion=TRUE, Do_it, Don't_Do_It)

This automates a model and aids management in decision making and what-if analysis. **IF** is clearly a very powerful tool when used correctly.

The A to Z of Excel Functions: IFERROR

IFERROR first came into being back in Excel 2007. It was something users had asked Microsoft for, for a very long time. But let me go back in time first and explain why.

At the time of writing, there are 12 **IS** functions, *i.e.* functions that give rise to a TRUE or FALSE value depending upon whether a certain condition is met:

1. **ISBLANK(Reference):** checks whether the **Reference** is to an empty cell
2. **ISERR(Value):** checks whether the **Value** is an error (*e.g. #REF!, #DIV/0!, #NULL!*). This check specifically excludes *#N/A*
3. **ISERROR(Value):** checks whether the Value is an error (*e.g. #REF!, #DIV/0!, #NULL!*). This is probably the most commonly used of these functions in financial modelling
4. **ISEVEN(Number):** checks to see if the **Number** is even
5. **ISFORMULA(Reference):** checks to see whether the **Reference** is to a cell containing a formula
6. **ISLOGICAL(Value):** checks to see whether the **Value** is a logical (TRUE or FALSE) value
7. **ISNA(Value):** checks to see whether the **Value** is *#N/A*. This gives us the rather crude identity **ISERR + ISNA = ISERROR**
8. **ISNONTEXT(Value):** checks whether the **Value** is not text (*N.B.* blank cells are not text)
9. **ISNUMBER(Value):** checks whether the **Value** is a number
10. **ISODD(Number):** checks to see if the **Number** is odd. Personally, I find the number 46 very odd, but Excel doesn't
11. **ISREF(Value):** checks whether the **Value** is a reference
12. **ISTEXT(Value):** checks whether the **Value** is text.

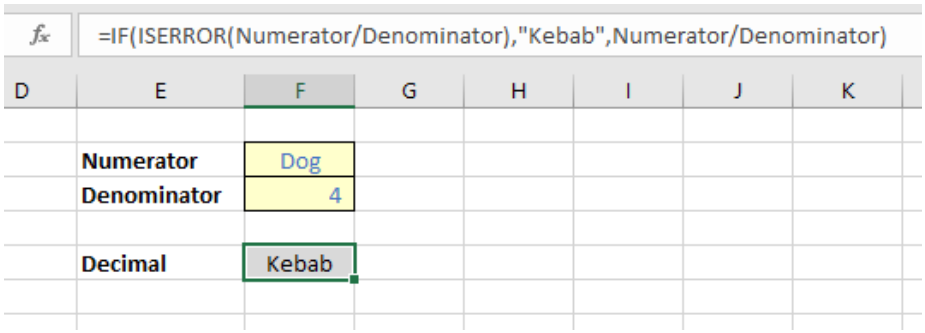
You get the idea. As mentioned previously, sometimes you need to trap errors that may originate from a formula that is correct most of the time. Where possible, you should be specific with regard to what you are checking, *e.g.*

=IF(Denominator=0, Error_Trap, Numerator / Denominator)

In this example, I am checking to see whether the Denominator is zero. I could use this formula instead:

=IF(ISERROR(Numerator / Denominator), Error_Trap, Numerator / Denominator)

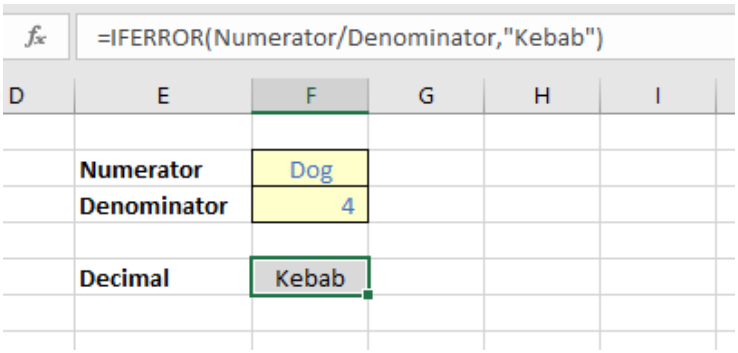
The difference here is that this will check for anything that may give rise to an error:



Do you see the problem here? I have to put the same formula in twice. If that is a long formula, then the calculation becomes doubly long. This is where **IFERROR** comes in; it halves the length of the calculation but still achieves the same effect

=IFERROR(Calculation, Error_Trap)

Essentially, this formula is the bastard lovechild of **IF** and **ISERROR**. It checks to see whether the **Calculation** will give rise to a *prima facie* error. If it does, it will return **Error_Trap**; otherwise, it will perform the said **Calculation**, *e.g.*



You shouldn't just sprinkle **IFERROR** throughout your models like your formulae are confetti. Used unwisely, **IFERROR** can disguise the fact that your formula isn't working correctly and that modifications to the logic may be required. Try to use it sparingly.

Sometimes you have to use **IF** and **ISERROR** in combination anyway:

=IF(ISERROR(Calculation), Error_Trap, Different_Calculation)

In this example, the formula is checking to see whether a particular **Calculation** gives rise to an error. If it does, the Error_Trap will be referenced in the usual way, but if not a **Different_Calculation** (not the **Calculation** used for the test) will be computed.

These two methodologies should be mastered. You will create more robust and flexible models once your error become a thing of the past. Not just the model – but your own expertise – will become more trusted in your organisation if users never encounter *prima facie* errors in your model.

The A to Z of Excel Functions: IFNA

The **IFNA** function returns the value you specify if a formula returns the *#N/A* error value; otherwise it returns the result of the formula.

IFNA has the following syntax:

IFNA(value, value_if_NA)

- **value:** this is required and represents the argument that is checked for the #N/A error
- **value_if_NA:** this is also required. This is the value to return if the formula (**value**) evaluates to the *#N/A* error value.

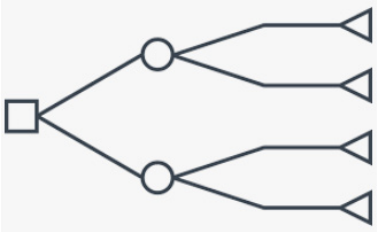
It should be noted that:

- if value or **value_if_NA** is an empty cell, **IFNA** treats the argument(s) as an empty string value ("")
- if **value** is an array formula, **IFNA** returns an array of results for each cell in the range specified in **value**.

Please see our example below.

	A	B	C
1	Student	Mark	
2	Anna	68%	
3	Boris	92%	
4	Charlie	59%	
5	Dee	77%	
6			
7	Student to Look Up		
8	Mark		
9			
10	Formula	Description	Result
11	=IFNA(VLOOKUP(A8,A2:B5,2,FALSE),"Not located.")	Looks up name "Mark" (cell A8) in the range A2:A5, and determines this name is not in the list. This would normally give rise to an #N/A error, but the error trap, "Not located.", is returned instead	Not located.
12			

The A to Z of Excel Functions: IFS



As a model developer and reviewer, I must confess I remain unconvinced about this one. If you have ever used a formula with nested **IF** statements beginning with

=IF(IF(IF…

then maybe this next function is for you – however, if you have ever written Excel formulas like this, then maybe Excel isn't for you! There are usually better ways of writing the formula using other functions.

Office 365 and Excel 2019 in all its variants has the relatively new function **IFS**. The syntax for **IFS** is as follows:

IFS(logical_test1, value_if_true1, [logical_test2, value_if_true2], [logical_test3, value_if_true3],…)

where:

- **logical_test1** is a logical condition that evaluates to TRUE or FALSE
- **value_if_true1** is the result to be returned if logical_test1 evaluates to TRUE. This may be empty
- **logical_test2** (and onwards) are further conditions that evaluate to TRUE or FALSE also
- **value_if_true2** (and onwards) are the respective results to be returned if the corresponding **logical_test** evaluates to TRUE. Any or all may be empty.

Since functions are limited to 254 arguments (sometimes known as parameters), the **IFS** function can contain 127 pairs of conditions and results.

One thing to note is that **IFS** is not quite the same as **IF**. With the **IF** statement, the third argument corresponds to what do if the **logical_test** is not TRUE (that is, it is an ELSE condition). **IFS** does not have an inherent ELSE condition, but it can be easily generated. All you have to do is make the final logical_test equal to a condition which is always true such as TRUE or 1=1 *(say)*.

Other issues to consider:

- whilst the **value_if_true** may be empty, it must not be omitted. Having an odd number of arguments in an **IFS** statement would give rise to the "You've entered too few arguments for this function" error message
- if a **logical_test** is not actually a logical test (for example, it evaluates to something other than TRUE or FALSE, the function returns an *#VALUE!* error. Numbers still appear to work though: any number than zero evaluates as TRUE and zero is considered to be FALSE
- if no TRUE conditions are found, this function returns the *#N/A* error.

To show how it works, consider the following example:

Becoming a Qualified Excel 2019 Guru

Criteria

Criteria	Yes / No	Grade
Already qualified?	No	3 Star
Work for Microsoft?		2 Star
Passed exam?	Yes	1 Star
Studying?		Student

Grade Achieved **1 Star**

Here, would-be gurus are graded based on evaluation criteria in the table, applied in a particular order:

=IFS(H13="Yes",I13,H14="Yes",I14,H15="Yes",I15,H16="Yes",I16,TRUE,"Not a Guru")

I think it's safe that although it is reasonably straightforward to follow, it is entirely reasonable to say it's not the prettiest, most elegant formula ever put to Excel paper. In particular, do pay heed to the final **logical_test**: TRUE. This ensures we have an ELSE condition as discussed above.

To be fair, one similar solution using previous Excel functions isn't any better:

=IF(H13="Yes",I13,IF(H14="Yes",I14,IF(H15="Yes",I15,IF(H16="Yes",I16,"Not a Guru")))).

You may note I am not supplying multiple examples of **IFS** formulae. This is because wherever possible you should try and replace the logic with a simpler, more accessible, logic for end users. For instance, sometimes the logic of an elongated **IF** or **IFS** formula may be condensed to

=IF(Multiple Conditions = TRUE, Do Something, Do Something Else).

In this situation, there is a function in Excel that can help.

My old English teacher said you should never start or finish a sentence with the word "and". **AND** is one of several Excel logic functions (others include **NOT** [already mentioned earlier, which takes the logical opposite of an expression] and **OR**). It returns TRUE if all of its arguments evaluate to TRUE; it returns FALSE if one or more arguments evaluate to FALSE.

One common use for the **AND** function is to expand the usefulness of other functions that perform logical tests. For example, the **IF** function performs a logical test and then returns one value if the test evaluates to TRUE and another value if the test evaluates to FALSE. By using the **AND** function as the **logical_test** argument of the **IF** function, you can test many different conditions instead of just one.

For example, imagine you are in New York on a Monday. Consider the expression

=AND(condition1, condition2, condition3)

where:

- **condition1** is the condition, "today is Monday"
- **condition2** is the condition, "you are in New York" *and*
- **condition3** is the condition, "this author is the best looking guy you have ever seen".

This would clearly be FALSE as not everywhere in the world it would be Monday (that is, **condition1** would be breached)…

As alluded to above, the syntax for **AND** is as follows:

AND(logical1, [logical2], …)

where:

- **logical1:** the first condition that you want to test that can evaluate to either TRUE or FALSE
- **logical2:** additional conditions that you want to test that can evaluate to either TRUE or FALSE, up to a maximum of 255 conditions. **logical2** is optional and is not needed in the syntax.

It should be noted that:

- the arguments must evaluate to logical values, such as TRUE or FALSE, or the arguments must be arrays or references that contain logical values
- if an array or reference argument contains text or empty cells, those values are ignored
- if the specified range contains no logical values, the **AND** function returns the *#VALUE!* error value.

To highlight how **AND** works:

	A	B	C
1	Condition 1	TRUE	
2	Condition 2	FALSE	
3	Condition 3	TRUE	
4			
5			
6	**Description**	**Results**	**Formula**
7	All arguments are true	FALSE	*=AND(B1:B3)*
8	The first and last arguments are true	TRUE	*=AND(B1,B3)*
9	At least one argument is false	TRUE	*=NOT(AND(B1:B3))*
10			

For a more practical example, consider the following summary data table:

	A	B	C	D	E	F
1	**Staff ID**	**Works in Sales?**	**Sales Made**	**Threshold**	**Bonus %**	**Bonus Paid**
2	ID 2017	yes	$ 9,069	$ 5,000	2.50%	$ 227
3	ID 3102	yes	$ 6,285	$ 8,000	2.00%	$ -
4	ID 3148	no	$ 8,458			$ -
5	ID 3321	yes	$ 5,635	$ 3,000	3.00%	$ 169
6	ID 3817	no	$ 19,973			$ -
7	ID 5298	no	$ 7,986			$ -
8	ID 6774	yes	$ 1,571	$ 5,000	2.50%	$ -
9	ID 8563	no	$ 16,124			$ -
10						
11				*=IF(AND(B2="yes",C2-D2>=0),C2*E2,)*		

Here, we have a list of staff in column **A**, with identification of those who work in Sales (that is, eligible for a bonus) in column **B**. Details of the sales made, the threshold for getting a bonus, and what rate it is paid are detailed in columns **C**, **D**, and **E** respectively. The formula in cell **F2**:

=IF(AND(B2="yes",C2-D2>=0),C2*E2,)

denotes the Bonus Paid and is conditional on them working in Sales (**B2="yes"**) and that the sales made were at or above the required threshold (**C2-D2>=0**). If both conditions are TRUE, then a bonus (**C2*E2**) is paid accordingly (putting nothing after the final comma is the equivalent of saying "else zero"). This is a prime example of IF and working together – and more often than not, these formulas are much easier to read than their **IF(IF(IF** or **IFS** counterparts.

The other logic function not yet mentioned, **OR**, is similar to **AND**, but only requires one condition to be TRUE. Similar to **AND**, the **OR** function may be used to expand the usefulness of other functions that perform logical tests. For example, the **IF** function performs a logical test and then returns one value if the test evaluates to TRUE and another value if the test evaluates to FALSE. By using the **OR** function as the **logical_test** argument of the **IF** function, you can test many different conditions instead of just one.

For example, imagine you are in London on a Tuesday. Consider the expression

=OR(condition1, condition2, condition3)

where:

- **condition1** is the condition, "today is Tuesday"
- **condition2** is the condition, "you are in London" or
- **condition3** is the condition, "the Earth is flat".

This would clearly be TRUE as you are definitely in London (that is, **condition2** holds).

The syntax for **OR** is as follows:

OR(logical1, [logical2], …)

where:

- **logical1:** the first condition that you want to test that can evaluate to either TRUE or FALSE
- **logical2:** additional conditions that you want to test that can evaluate to either TRUE or FALSE, up to a maximum of 255 conditions. **logical2** is optional and is not needed in the syntax.

It should be noted that:

- the arguments must evaluate to logical values, such as TRUE or FALSE, or the arguments must be arrays or references that contain logical values
- if an array or reference argument contains text or empty cells, those values are ignored
- if the specified range contains no logical values, the **OR** function returns the *#VALUE!* error value.

In summary, **OR** works as follows:

	A	B	C
1	Condition 1	TRUE	
2	Condition 2	FALSE	
3	Condition 3	FALSE	
4			
5			
6	**Description**	**Results**	**Formula**
7	At least one argument is true	TRUE	*=OR(B1:B3)*
8	All arguments are false	FALSE	*=NOT(OR(B1:B3))*
9	At least one argument is false	TRUE	*=NOT(AND(B1:B3))*
10			

For a more practical example, consider the following summary data table:

	A	B	C	D	E	F
1	**Staff ID**	**Works in Sales?**	**Sales Made**	**Threshold**	**Bonus %**	**Bonus Paid**
2	ID 2017	yes	$ 9,069	$ 5,000	2.50%	**$ 227**
3	ID 3102	yes	$ 6,285	$ 8,000	2.00%	$ -
4	ID 3148	no	$ 8,458			**$ 85**
5	ID 3321	yes	$ 5,635	$ 3,000	3.00%	**$ 169**
6	ID 3817	no	$ 19,973			**$ 200**
7	ID 5298	no	$ 7,986			**$ 80**
8	ID 6774	yes	$ 1,571	$ 5,000	2.50%	$ -
9	ID 8563	no	$ 16,124			**$ 161**
10						
11	***Non-Sales Staff***					
12						
13		Threshold	$ 5,635			
14						
15		Bonus %	1.00%			
16						
17						
18	*=IF(OR(AND(B2="yes",C2-D2>=0),AND(B2<>"yes",C2-$C$13>=0)),C2*IF(B2="yes",E2,$C$15),)*					

Now there is a complex formula:

=IF(OR(AND(B2="yes",C2-D2>=0),AND(B2<>"yes",C2-$C$13>=0)),C2*IF(B2="yes",E2,$C$15),)

It isn't quite as bad as it first seems. This is based on the **AND** case study from earlier, but it also allows for Non-Sales staff to participate in the bonus scheme too. The **logical_test** in the primary **IF** statement,

OR(AND(B2="yes",C2-D2>=0),AND(B2<>"yes",C2-$C$13>=0))

Is essentially **OR(condition1, condition2)**. The first condition is as before for Sales staff, whereas the second,

AND(B2<>"yes",C2-$C$13>=0)

checks whether Non-Sales staff have exceeded the Non-Sales Staff threshold (cell **C13**). Do you see that the check for Non-Sales staff is given by **B2<>"yes"** (**B2** is not equal to **"yes"**) rather than B2="no"? This takes me back to my earlier point about ensuring you develop your **logical_test** correctly. It's a subtle point, but will ensure all staff are considered (rather than excluding staff where no entry has been made in column **B**).

The other **IF** statement,

IF(B2="yes",E2,$C$15)

simply ensures the correct bonus rate is applied to the sales figure.

To summarise so far, sometimes your **logical_test** might consist of multiple criteria:

=IF(condition1=TRUE,IF(condition2=TRUE,IF(condition3=TRUE,formula,),),)

Here, this formula only gives a value of 1 if all three conditions are true. This nested IF statement may be avoided using the logical function **AND(Condition1,Condition2,…)** which is only TRUE if and only if all dependent arguments are TRUE,

=IF(AND(condition1,condition2,condition3),formula,)

which is actually easier to read. A similar example may be constructed for OR also. However, even using these logic functions, formulas may become – or simply look – complex quite quickly. There is an alternative: ***flags***. In its most common form, flags are evaluated as

=(condition=TRUE)*1

condition=TRUE will give rise to a value of either TRUE or FALSE. The brackets will ensure this is **condition** is evaluated first; multiplying by 1 will provide an end result of zero (if FALSE, as FALSE*1 = 0) or one (if TRUE, TRUE*1 = 1). I know some modellers prefer TRUEs and FALSEs everywhere, but I think 1's and 0's are easier to read (when there are lots of them) and more importantly, easier to sum when you need to know how many issues there are.

Flags make it easier to follow the tested conditions. Consider the following:

D9 =PRODUCT(D4:D7)

	B	D	E	F	G	H	I	J	K	L	M	O
2	Counter	1	2	3	4	5	6	7	8	9	10	
4	Divisible by 3	-	-	1	-	-	1	-	-	1	-	=(MOD(Counter,3)=0)*1
5	Greater than 4	-	-	-	-	1	1	1	1	1	1	=(Counter>4)*1
6	Less than or equal to 9	1	1	1	1	1	1	1	1	1	-	=(Counter<=9)*1
7	Is not 6	1	1	1	1	1	-	1	1	1	1	=(Counter<>6)*1
9	Product	-	-	-	-	-	-	-	-	1	-	=PRODUCT(D4:D7)

In this illustration, you might not understand what the **MOD** function does, but hopefully, you can follow each of the flags in rows 4 to 7 without being an Excel guru. Row 9, the product, simply multiplies all of the flags together (using the **PRODUCT** function allows you to add additional conditions / rows easily). This effectively produces a sophisticated **AND** flag, where all of the formulas are mercifully short. If I wanted the flag to be a 1 as long as one of the above conditions is TRUE (that is, I wish to construct an **OR** equivalent), that is easy too:

D9 =MAX(D4:D7)

	B	D	E	F	G	H	I	J	K	L	M	O
2	Counter	1	2	3	4	5	6	7	8	9	10	
4	Divisible by 3	-	-	1	-	-	1	-	-	1	-	=(MOD(Counter,3)=0)*1
5	Greater than 4	-	-	-	-	1	1	1	1	1	1	=(Counter>4)*1
6	Less than or equal to 9	1	1	1	1	1	1	1	1	1	-	=(Counter<=9)*1
7	Is not 6	1	1	1	1	1	-	1	1	1	1	=(Counter<>6)*1
9	MAX	1	1	1	1	1	1	1	1	1	1	=MAX(D4:D7)

Flags frequently make models more transparent and this example provides a great learning point. Often, we mistakenly believe that condensing a model into fewer cells makes it more efficient and easier follow. On the contrary, it is usually better to step out a calculation. If it can be followed on a piece of paper (without access to the formula bar), then more people will follow it. If more can follow the model logic, errors will be more easily spotted. When this occurs, a model becomes trusted and therefore is of more value in decision-making.

Ne careful though. Sometimes you just <u>can't</u> use flags. Consider the following instance:

=(Numerator/Denominator)*(Denominator<>0)

	E	F
	Numerator	3
	Denominator	-
	Decimal	#DIV/0!

Here, the flag does not trap the division by zero error. This is because this formula evaluates to

=#DIV/0! x 0

which equals *#DIV/0!* If you need to trap an error, you must use an **IF** function.

More Excel Functions next month.

Beat the Boredom Suggested Solution

Earlier, we asked if you could find a way to extract and output just the **bold** characters from the cell. How did you go? We couldn't find a function that would help us at all, so we created our own! And not a **LAMBDA** in sight…

Suggested Solution

If you've been following our VBA blog series and don't have much experience with VBA at all, then this might go a bit over your head. However, I'll try to explain as we go along:

```
Public Function udfExtractNonBold(Data As Range) As String

Dim index As Long
Dim returnValue As Variant

returnValue = ""
With Data
   If VarType(Data.Value2) = vbString Then
      For index = 1 To .Characters.Count
         If Not .Characters(index, 1).Font.Bold Then
            returnValue = returnValue & .Characters(index, 1).Text
         End If
      Next
   Else
      If Not Data.Font.Bold Then
         returnValue = Data.Value
      End If
   End If
End With

udfExtractNonBold = returnValue

End Function
```

Let's consider what it's doing by breaking it into steps:

- firstly, this produces a function that we're going to call **udfExtractNonBold**, and it takes a cell range as its input (*e.g.* **=udfExtractNonBold(A1)**)
- for the cell range input, initially it will check to see whether it's actually a text string. If it's a number, it will check to see if the whole thing is bold
- if it's a text string, then it will run through each character, one at a time. It will then check if that character is bold, and if so, add it to a new string that we're storing (the thing called **'returnValue'**)
- finally, it will output the **returnValue**: if there's no bold in the text string, it will give us a blank result. If there is bold text, it will give us only the text that was made bold, and nothing else in the cell.

Until next time.

Upcoming SumProduct Training Courses - COVID-19 update

Due to the COVID-19 pandemic that is currently spreading around the globe, we are suspending our in-person courses until further notice. However, to accommodate the new working-from-home dynamic, we are switching our public and in-house courses to an online delivery stream, presented via Microsoft Teams, with a live presenter running through the same course material, downloadable workbooks to complete the hands-on exercises during the training session, and a recording of the sessions for your use within 1 month for you to refer back to in the event of technical difficulties. To assist with the pacing and flow of the course, we will also have a moderator who will help answer questions during the course.

If you're still not sure how this will work, please contact us at training@sumproduct.com and we'll be happy to walk you through the process.

Location	Course	Date	Date	Duration	Duration
Online (Australia)	Power Pivot, Power Query and Power BI	16 - 18 Feb 2022	09:00-17:00 AEDT	(-1 day) 22:00-06:00 GMT	3 Days
Online (Australia)	Excel Tips and Tricks	23 Feb 2022	09:00-17:00 AEDT	(-1 day) 22:00-06:00 GMT	1 Day
Online (Australia)	Financial Modelling	24 - 25 Feb 2022	09:00-17:00 AEDT	(-1 day) 22:00-06:00 GMT	2 Days
Online (Australia)	Excel Tips and Tricks	11 Apr 2022	09:00-17:00 AEST	(-1 day) 23:00-07:00 GMT	1 Day
Online (Australia)	Financial Modelling	12 - 13 Apr 2022	09:00-17:00 AEST	(-1 day) 23:00-07:00 GMT	2 Days
Online (Australia)	Power Pivot, Power Query and Power BI	10 - 12 May 2022	09:00-17:00 AEST	(-1 day) 23:00-07:00 GMT	3 Days
Online (Australia)	Excel Tips and Tricks	17 May 2022	09:00-17:00 AEST	(-1 day) 23:00-07:00 GMT	1 Day
Online (Australia)	Financial Modelling	18 - 19 May 2022	09:00-17:00 AEST	(-1 day) 23:00-07:00 GMT	2 Days

Location	Course	Date	Date	Duration	Duration
Online (Australia)	Power Pivot, Power Query and Power BI	19 - 21 Jul 2022	09:00-17:00 AEST	(-1 day) 23:00-07:00 GMT	3 Days
Online (Australia)	Excel Tips and Tricks	26 Jul 2022	09:00-17:00 AEST	(-1 day) 23:00-07:00 GMT	1 Day
Online (Australia)	Financial Modelling	27 - 28 Jul 2022	09:00-17:00 AEST	(-1 day) 23:00-07:00 GMT	2 Days
Online (Australia)	Excel Tips and Tricks	29 Aug 2022	09:00-17:00 AEST	(-1 day) 23:00-07:00 GMT	1 Day
Online (Australia)	Financial Modelling	30 - 31 Aug 2022	09:00-17:00 AEST	(-1 day) 23:00-07:00 GMT	2 Days
Online (Australia)	Power Pivot, Power Query and Power BI	28 -30 Sep 2022	09:00-17:00 AEST	(-1 day) 23:00-07:00 GMT	3 Days
Online (Australia)	Excel Tips and Tricks	5 Oct 2022	09:00-17:00 AEDT	(-1 day) 22:00-06:00 GMT	1 Day
Online (Australia)	Financial Modelling	6 - 7 Oct 2022	09:00-17:00 AEDT	(-1 day) 22:00-06:00 GMT	2 Days
Online (Australia)	Power Pivot, Power Query and Power BI	9 - 11 Nov 2022	09:00-17:00 AEDT	(-1 day) 22:00-06:00 GMT	3 Days
Online (Australia)	Excel Tips and Tricks	16 Nov 2022	09:00-17:00 AEDT	(-1 day) 22:00-06:00 GMT	1 Day
Online (Australia)	Financial Modelling	17 - 18 Nov 2022	09:00-17:00 AEDT	(-1 day) 22:00-06:00 GMT	2 Days
Online (Australia)	Power Pivot, Power Query and Power BI	7 - 9 Dec 2022	09:00-17:00 AEDT	(-1 day) 22:00-06:00 GMT	3 Days
Online (Australia)	Excel Tips and Tricks	14 Dec 2022	09:00-17:00 AEDT	(-1 day) 22:00-06:00 GMT	1 Day
Online (Australia)	Financial Modelling	15 - 16 Dec 2022	09:00-17:00 AEDT	(-1 day) 22:00-06:00 GMT	2 Days

Key Strokes

Each newsletter, we'd like to introduce you to useful keystrokes you may or may not be aware of. This month we'd **SHIFT CTRL** of the numbers:

Keystroke	What it does
CTRL + SHIFT 0	Show column
CTRL + SHIFT 1	Fixed decimal and comma format
CTRL + SHIFT 2	Time (AM/PM) format
CTRL + SHIFT 3	Date format
CTRL + SHIFT 4	Currency format
CTRL + SHIFT 5	Percentage format
CTRL + SHIFT 6	Exponential format
CTRL + SHIFT 7	Outline border
CTRL + SHIFT 8	Select current region
CTRL + SHIFT 9	Unhide row

There are *c.*550 keyboard shortcuts in Excel. For a comprehensive list, please download our Excel file at www.sumproduct.com/thought/keyboard-shortcuts. Also, check out our new daily **Excel Tip of the Day** feature on the www.sumproduct.com homepage.

Our Services

We have undertaken a vast array of assignments over the years, including:

- **Business planning**
- **Building three-way integrated financial statement projections**
- **Independent expert reviews**
- **Key driver analysis**
- **Model reviews / audits for internal and external purposes**
- **M&A work**
- **Model scoping**
- **Power BI, Power Query & Power Pivot**
- **Project finance**
- **Real options analysis**
- **Refinancing / restructuring**
- **Strategic modelling**
- **Valuations**
- **Working capital management**

If you require modelling assistance of any kind, please do not hesitate to contact us at contact@sumproduct.com.

Link to Others

These newsletters are not intended to be closely guarded secrets. Please feel free to forward this newsletter to anyone you think might be interested in converting to "the SumProduct way".

If you have received a forwarded newsletter and would like to receive future editions automatically, please subscribe by completing our newsletter registration process found at the foot of any www.sumproduct.com web page.

Any Questions?

If you have any tips, comments or queries for future newsletters, we'd be delighted to hear from you. Please drop us a line at newsletter@sumproduct.com.

Training

SumProduct offers a wide range of training courses, aimed at finance professionals and budding Excel experts. Courses include Excel Tricks & Tips, Financial Modelling 101, Introduction to Forecasting and M&A Modelling.

Check out our more popular courses in our training brochure:

Drop us a line at training@sumproduct.com for a copy of the brochure or download it directly from www.sumproduct.com/training.

Sydney Address: SumProduct Pty Ltd, Suite 803, Level 8, 276 Pitt Street, Sydney NSW 2000
New York Address: SumProduct Pty Ltd, 48 Wall Street, New York, NY, USA 10005
London Address: SumProduct Pty Ltd, Office 7, 3537 Ludgate Hill, London, EC4M 7JN, UK
Melbourne Address: SumProduct Pty Ltd, Ground Floor, 470 St Kilda Road, Melbourne, VIC 3004
Registered Address: SumProduct Pty Ltd, Level 14, 440 Collins Street, Melbourne, VIC 3000

contact@sumproduct.com
www.sumproduct.com
+61 3 9020 2071