Building Custom Functions

About User Defined Functions

Excel provides the user with a large collection of ready-made functions, more than enough to satisfy the average user. Many more can be added by installing the various Add-Ins that are available (for example the *Analysis Toolpak*).

Most calculations can be achieved with what is provided, but you might find yourself wishing that there was a function that did a particular job and you can't find anything suitable in the list. You need a UDF.

A UDF (User Defined Function) is simply a function that you create yourself with VBA. UDFs are often called "Custom Functions". A UDF can remain in a code module attached to a workbook, in which case it will always be available when that workbook is open. Alternatively you can create your own Excel Add-In containing one or more functions that you can install into Excel just like a commercial Add-In.

UDFs can be accessed by code modules too. Often UDFs are created by developers to work solely within the code of a VBA procedure and the user is never aware of their existence.

Writing Your First VBA Function in Excel

Like any function, the UDF can be as simple or as complex as you want. Let's start with an easy one...

A Function to Calculate the Area of a Rectangle

Yes, I know you could do this in your head! The concept is very simple so you can concentrate on the technique.

Suppose you need a function to calculate the area of a rectangle. You look through Excel's collection of functions, but there isn't one suitable. This is the calculation to be done:

AREA = LENGTH x WIDTH

Open a new workbook and then choose **Tools > Macro > Visual Basic Editor** (or keys: **ALT+F11**) to open the Visual Basic Editor. You will need a module in which to write your function so choose **Insert > Module** (*Fig. 1*).

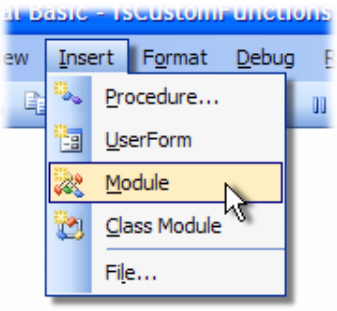

Fig. 1 Insert a module to hold the function code.

Into the empty module type: *Function Area* and press **[ENTER]**. The Visual Basic Editor completes the line for you and adds an End Function line as if you were creating a subroutine. So far it looks like this...

```
Function Area()

End Function
```

Place your cursor between the brackets after "Area". If you ever wondered what the brackets are for, you are about to find out! You need to specify the "arguments" that the function will take (an *argument* is a piece of information needed to do the calculation).

Type *Length as double, Width as double* and click in the empty line underneath. Note that as you type, a scroll box pops up listing all the things appropriate in the context of your typing (*Fig. 2*).

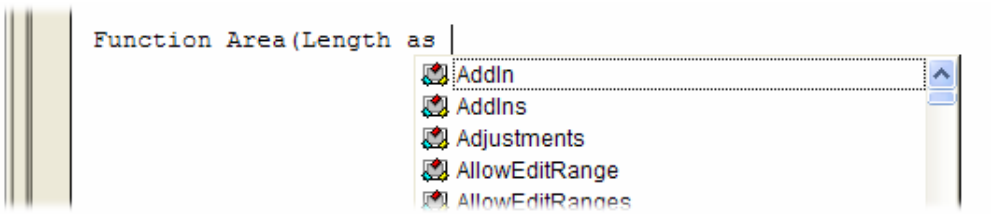

Fig. 2 The Visual Basic Editor suggests what to type.

This feature is called *Auto List Members*. If it doesn't appear either it is switched off (turn it on at **Tools > Options > Editor**) or you might have made a typing error earlier. It is a very useful check on your syntax. Find the item you need and double-click it to insert it into your code. You can ignore it and just type if you want. Your code now looks like this...

```
Function Area(Length As Double, Width As Double)

End Function
```

Declaring the data type of the arguments is not obligatory but makes sense. You could have typed *Length, Width* and left it as that, but warning Excel what data type to expect helps your code run more quickly and picks up errors in input. The *double* data type refers to number (which can be very large) and allows fractions.

Now for the calculation itself. In the empty line first press the **[TAB]** key to indent your code (making it easier to read) and type *Area = Length * Width.* Here's the completed code...

```
Function Area(Length As Double, Width As Double)
    Area = Length * Width
End Function
```

You will have noticed another of the Visual Basic Editor's help features pop up as you were typing. This is called *Auto Quick Info* (*Fig. 3*).

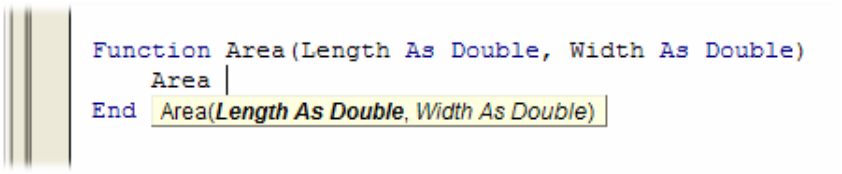

Fig. 3 The Visual Basic Editor displays Auto Quick Info.

It isn't relevant here. Its purpose is to help you write functions in VBA, by telling you what arguments are required.

You can test your function right away. Switch to the Excel window and enter figures for Length and Width in separate cells. In a third cell enter your function as if it were one of the built-in ones. In this example cell A1 contains the length (17) and cell B1 the width (6.5). In C1 I typed *=area(A1,B1)* and the new function calculated the area (110.5) (*Fig. 4*).

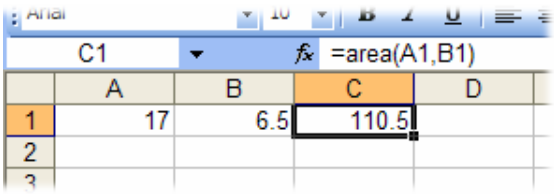

Fig. 4 Using the new Area function in Excel.

Sometimes, a function's arguments can be optional. In this example we could make the **Width** argument optional. Supposing the rectangle happens to be a square with Length and Width equal. To save the user having to enter two arguments we could let them enter just the Length and have the function use that value twice (i.e. multiply Length x Length). So the function knows when it can do this we must include an **IF Statement** to help it decide.

Change the code so that it looks like this...

```
Function Area(Length As Double, Optional Width As Variant)
    If IsMissing(Width) Then
        Area = Length * Length
    Else
        Area = Length * Width
    End If
End Function
```

Note that the data type for Width has been changed to *Variant* to allow for null values. The function now allows the user to enter just one argument e.g. *=area(A1)*. The IF Statement in the function checks to see if the Width argument has been supplied and calculates accordingly (*Fig. 5*).

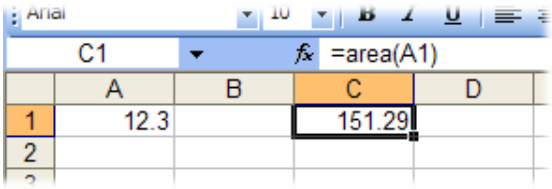

Fig. 5 Supplying a single argument to the modified function.

Now for a more practical example...

A Function to Calculate Fuel Consumption

I like to keep a check on my car's fuel consumption so when I buy fuel I make a note of the mileage and how much fuel it takes to fill the tank. Here in the UK fuel is sold in litres. The car's milometer (OK, so it's an odometer) records distance in miles. And because I'm too old and stupid to change, I only understand MPG (miles per gallon).

Now if you think that's all a bit sad, how about this. When I get home I open up Excel and enter the data into a worksheet that calculates the MPG for me and charts the car's performance.

The calculation is the number of miles the car has travelled since the last fill-up divided by the number of gallons of fuel used...

MPG = (MILES THIS FILL - MILES LAST FILL) / GALLONS OF FUEL

but because the fuel comes in litres and there are 4.546 litres in a gallon..

MPG = (MILES THIS FILL - MILES LAST FILL) / LITRES OF FUEL x 4.546

Here's how I wrote the function...

```
Function MPG(StartMiles As Integer, FinishMiles As Integer, Litres As Single)
    MPG = (FinishMiles - StartMiles) / Litres * 4.546
End Function
```

and here's how it looks on the worksheet (*Fig. 6*)...

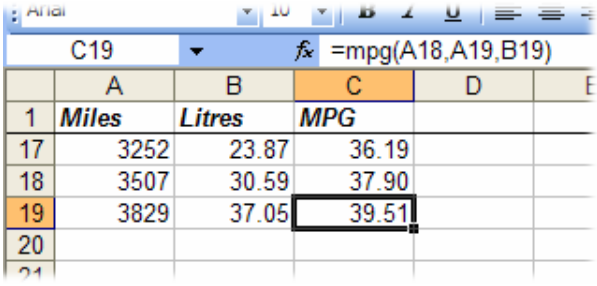

Fig. 6 Calculating fuel consumption with a custom function.

Not all functions perform mathematical calculations. Here's one that manipulates text...

Converting a Text String to a Date

It is not uncommon for data to be output from server databases as a text string in the format *YYYYMMDD*. So September 27 2006 would appear as 20060927. Although, having been told that

this piece of data represents a date, we can easily "translate" the string ourselves but Excel can't do this without our help.

You can do the calculation directly in Excel using the **Date** function. This function takes three arguments, each a number that represents part of the date: **Date(Year, Month, Day)**. You can use three of Excel's text functions to extract these numbers from the string. The **Left** function can be used to return the leftmost four characters of the string that represent the year. The **Mid** function can return the two characters of the month from the string, starting at the fifth character. The **Right** function provides the day by returning the rightmost two characters of the string. Here's how it looks in Excel (*Fig. 7*) …

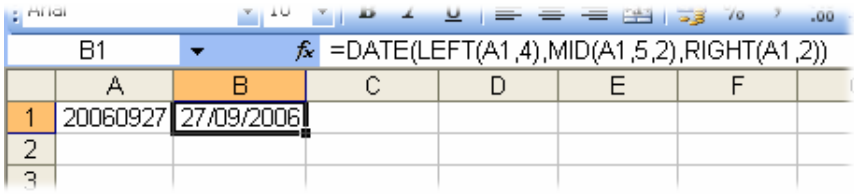

Fig. 7 Working from first principles in Excel

This calculation is easily transferred to VBA by changing Excel's **Date** function to the equivalent **DateSerial** function used in VBA, and inserting the argument name in the appropriate places. I've called it *TextDate* n the example below. My function is called *MakeDate*…

```
Function MakeDate(TextDate As String)
    MakeDate = DateSerial(Left(TextDate, 4), Mid(TextDate, 5, 2), Right(TextDate, 2))
    MakeDate = FormatDateTime(MakeDate, vbShortDate)
End Function
```

Since the **DateSerial** function returns the serial number if the resulting date an unformatted Excel cell will display this as a number (*Fig. 8*).

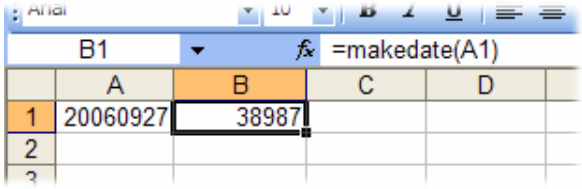

Fig. 8 In an unformatted cell the date serial number appears.

This is easily remedied by adding another line to the function code to apply a standard date format (*Fig. 9*)…

```
Function MakeDate(TextDate As String)
    MakeDate = DateSerial(Left(TextDate, 4), Mid(TextDate, 5, 2), Right(TextDate, 2))
    MakeDate = FormatDateTime(MakeDate, vbShortDate)
End Function
```

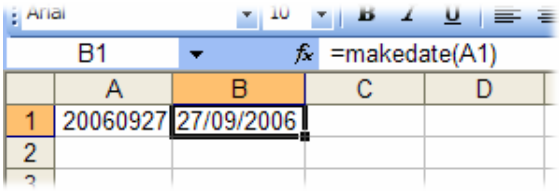

Fig. 9 The modified function returns a formatted date.

Using Your Custom Functions

If a workbook has a VBA code module attached to it that contains custom functions, those functions can be easily addressed within the same workbook as demonstrated in the examples above. You use the function name as if it were one of Excel's built-in functions.

You can also find the functions listed in the Insert Function tool (also called the *Function Wizard*). Use the tool to insert a function in the normal way (**Insert > Function**). Scroll down the list of

function categories to find **User Defined** and select it (*Fig. 10*) to see a list of available user defined functions (*Fig. 11*)...

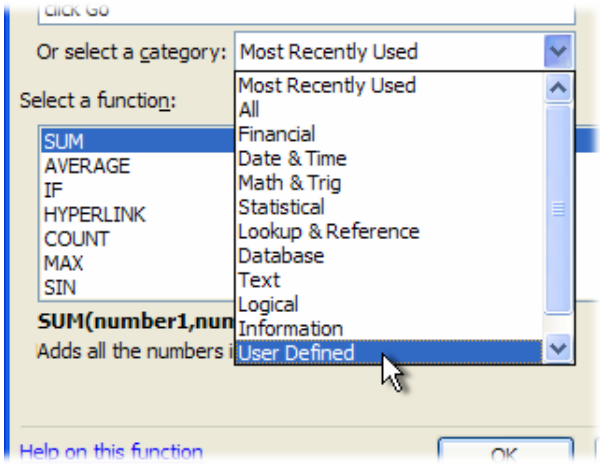

Fig. 10 Custom functions are listed in the User Defined section of the Insert Function tool.

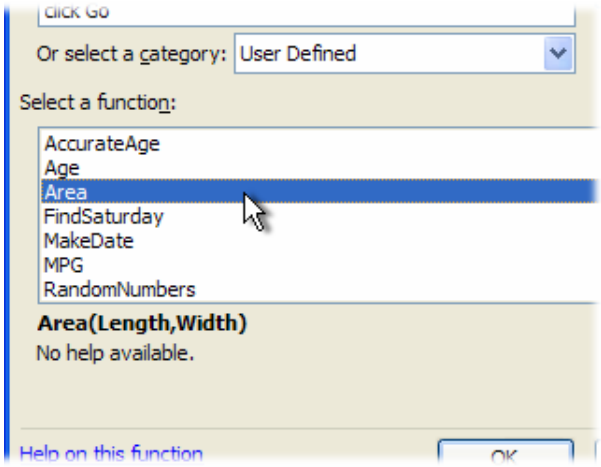

Fig. 11 Select a function in the usual way.

Adding a Description of the Function

You can see that the user defined functions lack any description other than the unhelpful "*No help available*" message, but you can add a short description. Here's how to do it…

Make sure you are in the workbook that contains the functions. Go to **Tools > Macro > Macros**. You won't see your functions listed here but Excel knows about them. In the **Macro Name** box at the top of the dialog, type the name of the function, then click the dialog's **Options** button. If the button is greyed out either you have misspelled the function name, or you are in the wrong workbook, or it doesn't exist! This opens another dialog into which you can enter a short description of the function. Click **OK** to save the description and (here's the confusing bit) click **Cancel** to close the Macro dialog box. Remember to **Save** the workbook containing the function. Next time you go to the Function Wizard your UDF will have a description (*Fig. 12*). The description lives in the workbook containing the function so it will be displayed even if the workbook is opened on another computer.

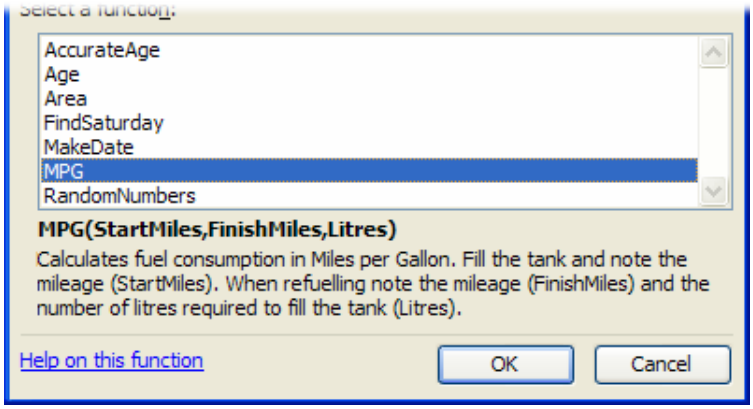

Fig. 12 A description of the function appears in the Insert Function dialog.

Although it is easy to create a description for a custom function, adding individual argument descriptions is not so easy and requires additional programming to register and unregister the information when the host workbook is opened or closed. The process will not be described here.

Like macros, user defined functions can be used in any other workbook as long as the workbook containing them is open. However it is not good practice to do this. Entering the function in a different workbook is not simple. You have to add its host workbook's name to the function name. This isn't difficult if you rely on the Insert Function tool, but it is clumsy to write out manually (*Fig. 13*).

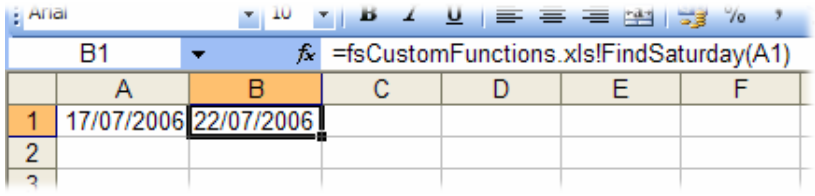

Fig. 13 The function name is prefixed by the name of its host workbook.

The Insert Function tool shows the full names and addresses of any user defined functions in other open workbooks (*Fig. 14*)...

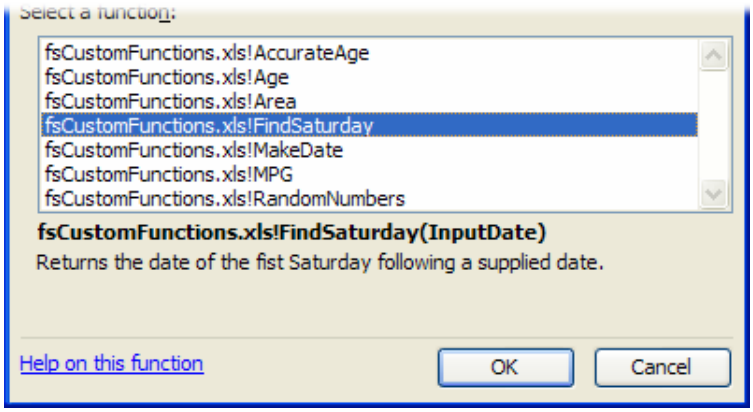

Fig. 14 Functions in another workbook are prefixed by its name.

If you open the workbook in which you used the function at a time when the workbook containing the function is closed you might have a problem. In earlier versions of Excel you will see an error message (*#NAME?*) in the cell in which you used the function. Excel has forgotten about it! Open the function's host workbook, recalculate (press the **[F9]** key) and all is fine again. In later versions of Excel you will see a message telling you there is a link to another workbook. You will prompted to open the linked workbook and you will have to do this for your function to work.

Fortunately there is a better way. If you want to write User Defined Functions for use in more than one workbook you can store them in your *Personal Macro Workbook* (*Personal.xls*) which opens and is hidden each time you open Excel, but the best method is to create an Excel **Add-In**.

Notes and Summary

Custom Functions (or UDFs – User Defined Functions) use VBA code to create a function that can be used in VBA macro programming or on an Excel worksheet where a suitable built-in function is not available.

Specify the required arguments and their data types. Any appropriate VBA coding can be used to arrive at the required calculation as long as the result is finally returned using a statement equivalent to <*FunctionName*> = <*Value*> where *FunctionName* is the name you gave the function and <*Value*> is a statement representing the final result of the calculation.

When naming functions and their arguments try to avoid using the names of existing functions or other "reserved words" (i.e. words used in the VBA language).

If Custom Functions are to be used in more than one workbook it is more convenient to store them in *Personal.xls* or in an Excel Add-In. An Add-In is better if you plan to distribute the functions to other users.

In these examples I have not specified a data type for the value returned by the function e.g.

```
Function MakeDate(TextDate As String) As Date
```

I find that because Excel interprets the data type itself the additional statement is not necessary and if used sometimes causes the data type to be interpreted wrongly.

If your custom function does not automatically update in use when the values in the cells it uses changes, add the statement `Application.Volatile` as the first line of the function's code.