Scripting fundamentals – saving your sanity

Writing code is no different from writing prose, each person develops their own style, and way of doing things, so the things that I'm suggesting below are really just pointers to make your life easier, and make your code more readable, and therefore more easily understood.

If you've done the 'scribble by function' tutorial then a lot of these ideas will be familiar, but this will try and explain them in a bit more detail.

Commenting

After you write your first line of code, you'll probably notice that it's not normal english as we are accustomed to seeing it.

```
int i = 5;
Print(“i is ” + i);
```

This doesn't tell us very much, and it certainly doesn't set up any sort of context for what it does. I can think 'This doesn't bother me, i know what it does, i just wrote it', but I have lost staring matches with gnats, and by the time I've finished my cup of coffee I've probably forgotten what it does. This is pretty bad, but not the end of the world, after all, i wrote it, so i should be able to decipher what it does, but it gets really bad when you are working with... (pause for dramatic effect)... other people! It took me a long time to realise that other people can't tell what I'm thinking (probably to their advantage most to the time) and you've probably come to the same conclusion. The way to get around that problem is to write comments in english (or whatever language you feel most comfortable with) that you can scan through and fin out what the program does before you actually launch into trying to read the code.

```
int i = 5;               //i is the number of cats in the room
Print(“i is ” + i);      //prints i to the console
```

You might feel like this is a total waste of time when you are writing it, but if you come back to your cat counting program after a few months, then if there are no comments then it will be almost impossible to read. Imagine you are a modern day Geppetto, and you make electrical toys. If you didn't make a note of what coloured wires did what, then when you came to fix a toy then it would look like a multi coloured pot noodle inside!

The other important thing to do with comments is to write an intro to your code. Even if it's just a simple function then a little header that explains what's going on will be really useful.

```
// this is a function to do really cool stuff      ← what the function does
// Ben Doherty 02 05 08 Version 1.0                ← who you are
// beware of the cracks in the pavement            ←know problems
```

you can put more stuff in the header, but just this basic stuff will be really useful to you/ others in the future.

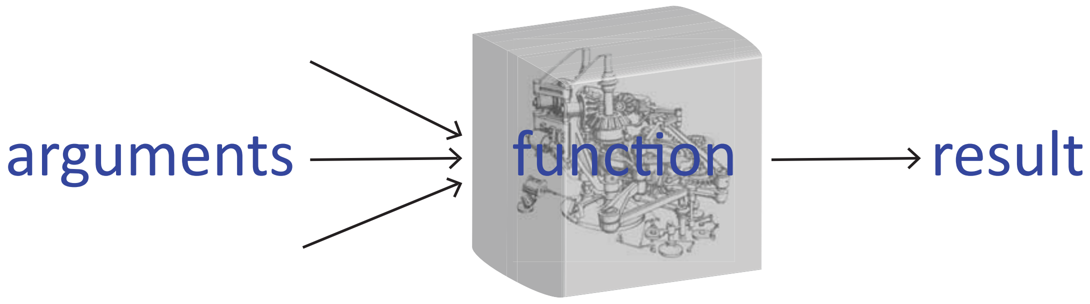
The black box

This isn't the sort of black box that aeroplanes have that reveal the last words of a pilot.

“I wonder if my altimeter is working, it shouldn't say 5 meters...
..my my that cow looks big...
..ouch!”

It's the idea that a function is a mysterious thing that you don't need to know how it works inside, just what it does as seen from the outside.

This means that as long as you don't change the inputs or outputs of a function then you can tinker with it's insides as much as you like and it'll still work fine in your model. The advantage of that is that if you mash together something that just about works to hit a deadline, then you can come back to it and make it elegant and bombproof later without changing any of the rest of the model.



The other implication of this is that because the function lives inside the black box, it can't see out, which means that you should make sure that information only passes in through the argument list and out through the return statement.

This seems a bit restrictive, and time consuming, but it will stop you getting your fingers burnt later, and it will improve the reusability of the code. (reusability means less work for you in the future, and less work in the future means more time to spend baking cakes and drinking tea)

An example of this is when we need to draw a load of lines that stick straight up. It's really tempting to call the baseCS directly from within the function. GC won't complain, because all top level features are available globally (see the scope tutorial) but if you decide that you want them all to tip over you'd need to go in and edit the function to make it happen rather than just editing the arguments list.

This doesn't sound too bad, but if you have loads of variables called globally then you are in for a big disaster sooner or later.
The other big disadvantage of calling things globally rather than through the arguments list is that the symbolic graph doesn't update, so you can't visualise the logic of your model. Which is about as cool as wearing 80's neon spandex tights to a Metalica concert.

Indenting

I'm not really a tidy freak, anyone who knows me can tell you that, but nobody wants to have to go to more effort than necessary to read code. Keeping it tidy makes it much easier to read.

In normal writing there are devices for making the text meaningful without actually reading it. You can roughly work out what's going on by looking at the paragraphs and the headings and seeing the structure of the document. (screw up your eyes and look at this page.)

It's no different with code. The main thing that comes in handy is indenting.

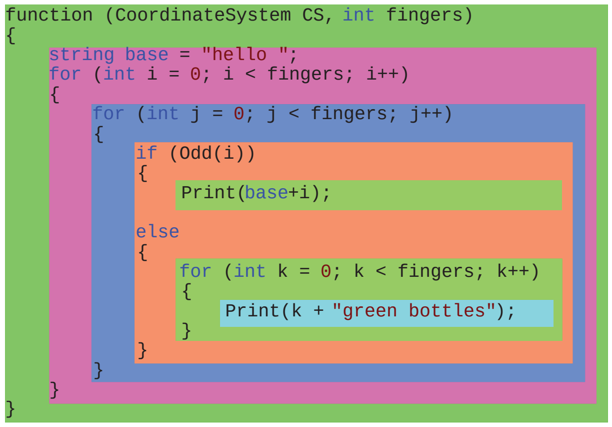
In general, whenever you pass the opening brace of a function, loop, conditional statement etc. you should move inwards by one tab stop (roughly 4 spaces).

As the pair of braces defines a structural thing (in terms of meaning, not making things stand up), it makes sense that everything inside that structural block is equally important. If you keep nesting the brace sets then the contents keeps moving to the right and therefore you can see the structure of the program without ever even reading the code.
This makes holding the whole program in your head much simpler and avoids unnecessary faffing about.

```
function (CoordinateSystem CS, int fingers)
{
    string base = "hello ";
    for (int i = 0; i < fingers; i++)
    {
        for (int j = 0; j < fingers; j++)
        {
            if (Odd(i))
            {
                Print(base+i);

            else
            {
                for (int k = 0; k < fingers; k++)
                {
                    Print(k + "green bottles");
                }
            }
        }
    }
}
```

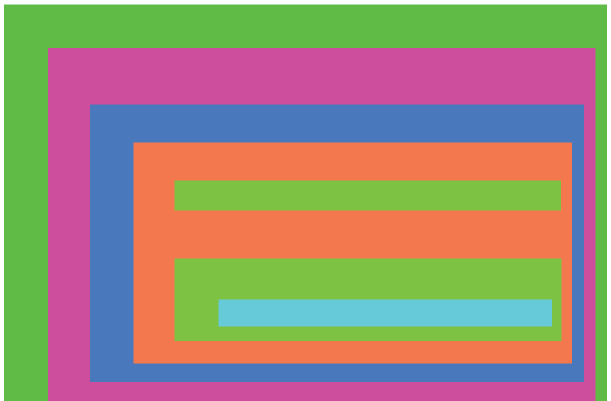
With this example function, there are a series of indents that correspond with structural elements of the code. The function doesn't do anything particularly useful, but it does have lots of nesting!



```
function (CoordinateSystem CS, int fingers)
{
    string base = "hello ";
    for (int i = 0; i < fingers; i++)
    {
        for (int j = 0; j < fingers; j++)
        {
            if (Odd(i))
            {
                Print(base+i);

            else
            {
                for (int k = 0; k < fingers; k++)
                {
                    Print(k + "green bottles");
                }
            }
        }
    }
}
```

If we draw a box around each structural element, then we can see how they nest.



eventually you'll get to the point where you can see roughly what's going on in a program without having to look at the individual words.