Basic Shell Scripting.

Stephen A. Wells (saw42@bath.ac.uk), Feb 2018.

**Scripting** is a method by which common, repetitive computing tasks can be automated. This allows the computer to carry out a set of similar or predictable steps based on the user's instructions.

Typography: I will use *italics* to denote important pieces of jargon or terminology, `monospace` to indicate commands that could be typed, and

```
$ monospace with background
```

to represent lines typed at the command prompt.

The first basic concept is the *shell*. This is simply the system by which the computer accepts your commands such as `ls` and understands what you mean. There are a few different shells in common use, but on most scientific computing systems you will meet the *bash shell*. BASH is an acronym for "Bourne Again SHell", as it is derived from the earlier Bourne shell. Our scripts are simple enough that they should work in any shell. We can identify our shell:

```
$ echo $SHELL

/bin/bash
```

Here I have asked for the *value* of a *variable* called `SHELL` to be echoed on screen. The result is `/bin/bash`, which is the bash shell program. This illustrates a fundamental point: the value of a variable called `varname` is obtained by writing `$varname`. `SHELL` is a built-in variable, but we can also make our own.

When we are working on the command line, we can give several commands in succession by separating them using a semicolon, thus:

```
$ pwd; ls
```

will give you the name of your current directory followed by a list of all its contents (incidentally, it shouldn't matter whether there is a space before the semicolon). If you are not working in a directory with some files in it, go to one which does, please.

Loops: The shell also includes special command constructs to carry out *loops*. This allows us to, for example, loop over a series of files, carrying out the same action on each file. Two very common loops we will meet are the *for loop* and the *while loop*. We can experiment with both using harmless commands like `echo` and `ls` which will not change any files. The basic structures of these loops are

```
for … ; do ( something ) ; done
```

and

```
while … ; do ( something else ) ; done.
```

This will make more sense shortly.

Firstly, since we know `ls` returns a list of files, we might try something like this:

```
$for file in ls ; do ( echo $file ) ; done

ls
```

Hmm. That did **not** do what we wanted. It did, however, do **exactly what we told it to do**! Let's investigate a little:

```
$for file in 1 b Nineteen ; do ( echo $file ) ; done

1
b
Nineteen
```

Now it should be clearer what the loop is actually doing; it's echoing every member of the list we give it. By the way, the word "file" is (a) meaningless to the shell, it's just a label, and (b) doesn't have to refer to an actual file. To confirm this:

```
$for antelope in 1 b Nineteen ; do ( echo $antelope ) ; done

1
b
Nineteen
```

In general I recommend your variable names should either be very clear and meaningful to you, or as silly as possible, so long as they are memorable and you can reliably type them correctly.

Clearly, to work on every file in a directory, we want to run a loop over the *output* of the `ls` command. Try this:

```
$ for file in $(ls) ; do ( echo $file ) ; done

cubic-a-list-1
eri.bonding
eri-00.cif
```

…

Success! In older scripts you may also see this done using backticks, `` `ls` ``. This should work, but the `$(…)` method is recommended. This is called *command substitution*.

Another approach, which is particularly useful when we want to work on only **some** of the files in a directory, is to first prepare the list we want in a text file, and then use a *while loop* which reads from the file. For example:

```
$ ls eri* > files-eri
$ while read line ; do ( echo $line ) ; done < files-eri

eri.bonding
eri-00.cif
```

```
eriflex.inp
```
…

The first line listed all files whose names started with “eri”, and directed the output (with the > symbol) into a newly created file, `files-eri`. The while loop then reads the names out of `files-eri` using the < symbol, and echoes each one. Do something similar in your own filespace, choosing a subset of your own files.

Of course, the word `line` has no special meaning; also, we can do more within the loop. For example:

```
$ while read zebra ; do ( echo "Oh, precious file, your name is "; 
echo $zebra ) ; done < files-eri

Oh, precious file, your name is
eri.bonding
Oh, precious file, your name is
eri-00.cif
```
…

or, using the `echo –n` version (no newline):

```
$ while read zebra ; do ( echo -n "Oh, precious file, your name 
is "; echo $zebra ) ; done < files-eri

Oh, precious file, your name is eri.bonding
Oh, precious file, your name is eri-00.cif
```
…

Find out about options like `echo –n` using `man echo` (which gives you the `man`ual!) or by Googling. Note that the command is written all on one line, do not hit `Enter` until the very end. Clearly, once we start trying to do anything complicated, the commands will become longer than can reasonably be typed on a single line. This brings us to the next stage…

Shell scripts. A *script* is nothing but a series of commands written in a text file (using your editor of choice; try `nano`), so that you can prepare them in advance and then have the shell execute them in sequence. I have taken the previous loop and converted it to script form in a text file that I have called `zebrascript.sh`, which we can examine:

```
$ cat zebrascript.sh

#!/bin/bash

while read zebra
do
  #this line is a comment and is ignored
  echo -n "Oh precious file, your name is "

  #blank lines, as above, also ignored
  echo $zebra
```

```
done < files-eri
```

Notice the use of the `cat` command (it's short for "concatenate", by the way) to dump the contents of a file to screen. Traditionally shell scripts end with `.sh` but the computer doesn't care. The specification of the shell on the first line is not essential for simple scripts like this. Within the script, we now have line ends instead of semicolons. The hashmark `#` allows us to place comments inside the script; this is increasingly useful as the script becomes longer, and commenting is **strongly encouraged**. The indentation of the lines inside the loop is purely for clarity.

Now, how can we actually run our script? Simple:

```
$ bash zebrascript.sh

Oh precious file, your name is eri.bonding
Oh precious file, your name is eri-00.cif
```

…

One more refinement: as written, this script will only ever read from the file `files-eri`, because that filename is explicitly written ("hard-coded") in the script. It would be much more useful and flexible if we could apply the commands in a script to any file we wanted. The shell includes a method to do this using some special built-in variables. Here's a modified version of the script to demonstrate:

```
$ cat zebrascript.sh

#!/bin/bash

echo "My own name is " $0
echo "I am reading from file " $1

while read zebra
do
  #this line is a comment and is ignored
  echo -n "Oh precious file, your name is "

  #blank lines, as above, also ignored
  echo $zebra
done < $1
```

which can be run like this:

```
$ bash zebrascript.sh files-eri

My own name is  zebrascript.sh
I am reading from file  files-eri
Oh precious file, your name is eri.bonding
Oh precious file, your name is eri-00.cif
```

…

So `$0` is the name of the script itself and `$1` is the name that follows – the first *argument* of the script. Multiple arguments can be obtained using `$2`, `$3`…

A script can include any command that you could use yourself on the command line – which means that, during the operation of a script, it is possible to create or remove files and directories, to navigate from one directory to another, and to change the contents of files. Obviously, you should be very, very sure of what a script is going to do before you actually run it!

In the next guide, we will obtain a list of MOF structures and see how we might use scripting to set up simulations on each of a selected set of structures.