

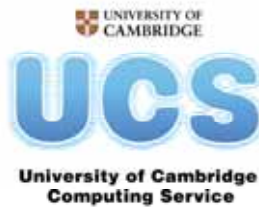
Simple Shell Scripting for Scientists

Day Two

Julian King

Bruce Beckles

University of Cambridge Computing Service



Introduction

- Who:
 - Julian King, Unix Support, UCS
 - Bruce Beckles, e-Science Specialist, UCS
- What:
 - Simple Shell Scripting for Scientists course, *Day Two*
 - Part of the **Scientific Computing** series of courses
- Contact (questions, etc):
 - scientific-computing@ucs.cam.ac.uk
- Health & Safety, etc:
 - Fire exits
- **Please switch off mobile phones!**

As this course is part of the Scientific Computing series of courses run by the Computing Service, all the examples that we use will be more relevant to scientific computing than to system administration, etc.

This does not mean that people who wish to learn shell scripting for system administration and other such tasks will get nothing from this course, as the techniques and underlying knowledge taught are applicable to shell scripts written for almost any purpose. However, such individuals should be aware that this course was not designed with them in mind.

We finish at:

17:00

The course officially finishes at 17.00, so don't expect to finish before then. If you need to leave before 17.00 you are free to do so, but don't expect us to have covered all today's material by then. How quickly we get through the material varies depending on the composition of the class, so whilst we may finish early you should not assume that we will. If you do have to leave early, please leave quietly.

If, and only if, you will not be attending *either* of the next two days of the course then ***please make sure that you fill in a green Course Review form*** and leave it at the front of the class for collection by the course giver.

What we don't cover

- Different types of shell:
 - We are using the **Bourne-Again SHell** (bash).
- Differences between versions of bash
- Very advanced shell scripting – try one of these courses instead:
 - “**Python: Introduction for Absolute Beginners**”
 - “**Python: Introduction for Programmers**”

bash is probably the most common shell on modern Unix/Linux systems – in fact, on most modern Linux distributions it will be the default shell (the shell users get if they don't specify a different one). Its home page on the WWW is at:

<http://www.gnu.org/software/bash/>

We will be using bash 4.1 in this course, but everything we do should work in bash 2.05 and later. Version 4, version 3 and version 2.05 (or 2.05a or 2.05b) are the versions of bash in most widespread use at present. Most recent Linux distributions will have one of these versions of bash as one of their standard packages. The latest version of bash (at the time of writing) is bash 4.2, which was released on 13 February, 2011.

For details of the “Python: Introduction for Absolute Beginners” course, see:

<http://www.training.cam.ac.uk/ucs/course/ucs-python>

For details of the “Python: Introduction for Programmers” course, see:

<http://www.training.cam.ac.uk/ucs/course/ucs-python4progs>

Outline of Course

1. Recap of day one
2. Shell functions

SHORT BREAK

3. Command substitution
4. The **mktemp** command
5. Handling data from standard input
 - Reading values from standard input
 - Pipelines
 - Loop constructs: **while**

SHORT BREAK

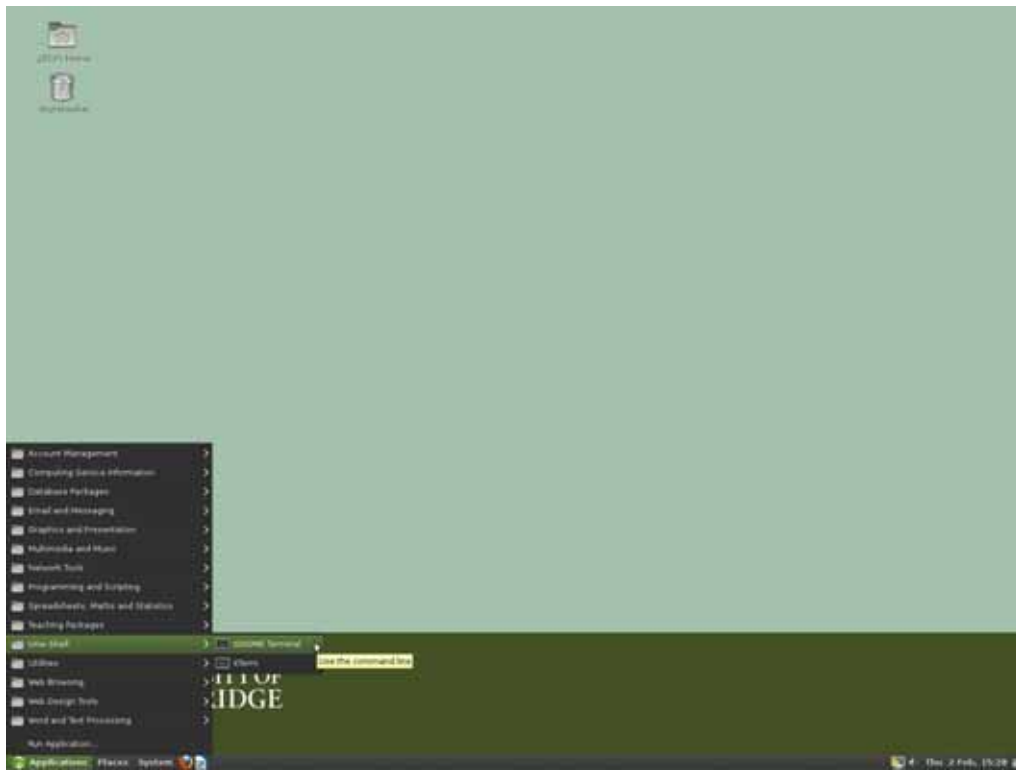
6. More **while** loops:
 - Shell arithmetic
 - Tests

Exercise

The course officially finishes at 17.00, but the intention is that the lectured part of the course will be finished by about 16.30 or soon after, and the remaining time is for you to attempt an exercise that will be provided. If you need to leave before 17.00 (or even before 16.30), please do so, but don't expect the course to have finished before then. If you do have to leave early, please leave quietly.

If, and only if, you will not be attending *either* of the next two days of the course then ***please make sure that you fill in a green Course Review form*** and leave it at the front of the class for collection by the course giver.

Start a shell



scientific-computing@ucs.cam.ac.uk

Simple Shell Scripting for Scientists: Day Two

6

Screenshot of newly started shell



Recap: Day One

- Simple shell scripts: linear lists of commands
- Simple use of shell variables and parameters
- Simple command line processing
- Output redirection
- **for** loops

Recap: What is a shell script?

- **Text** file containing commands understood by the shell
- Very **first** line is special:
`#!/bin/bash`
- File has its **executable** bit set
`chmod a+x`

Recall that the **chmod** command changes the permissions on a file. **chmod a+x** sets the executable bit on a file for all users on the system, i.e. it grants everyone permission to execute the file. (**Note** though, that all files in your home directory on the MCS Linux systems used in this course automatically have their executable bit set, so during this course you don't need to explicitly use the **chmod** command on such files.) Unix file permissions were covered in the "Unix: Introduction to the Command Line Interface" course, see:

<http://www.training.cam.ac.uk/ucs/course/ucs-unixintro1>

The notes from this course are available on-line at:

<http://www.ucs.cam.ac.uk/docs/course-notes/unix-courses/UnixCLI>

Recap: Very simple shell scripts

- Linear lists of commands
- Just the commands you'd type interactively put into a file
- Simplest shell scripts you'll write

Shell variables and parameters

Shell variables hold data, much like variables in a program:

```
$ VAR="My variable"
$ echo "${VAR}"
My variable
```

Shell parameters are special variables set by the shell:

- Positional parameter **0** holds the name of the shell script
- Positional parameter **1** holds the first argument passed to the script; positional parameter **2** holds the second argument passed to the script, etc
- Special parameter **@** expands to values of all positional parameters (starting from **1**)
- Special parameter **#** expands to the number of positional parameters (not including **0**)

We create shell variables by simply assigning them a value (as above for the shell variable **VAR**). We can access the value of a shell variable using the construct **\${VARIABLE}** where **VARIABLE** is the name of the shell variable. Note that there are *no* spaces between the name of the variable, the equal sign (=) and the variable's value in double quotes. *This is very important* as *whitespace* (spaces, tabs, etc) is significant in the names and values of shell variables.

Also note that although we can assign the value of one shell variable to another shell variable, e.g. **VAR1="\${VAR}"**, the two shell variables are in fact completely separate from each other, i.e. each shell variable can be changed independently of the other. Changing the value of one will not affect the other. So **VAR1** (in this example) is *not* a "pointer" to or an "alias" for **VAR**.

Shell parameters are special variables set by the shell. Many of them cannot be modified, or cannot be directly modified, by the user or by a shell script. Amongst the most important parameters are the *positional parameters* and the other shell parameters associated with them.

The positional parameters are set to the arguments that were given to the shell script when it was started, with the exception of positional parameter **0**, which is set to the name of the shell script. So, if `myscript.sh` is a shell script, and I ran it by typing:

```
./myscript.sh argon hydrogen mercury
```

```
then positional parameter 0 = ./myscript.sh
                        1 = argon
                        2 = hydrogen
                        3 = mercury
```

and all the other positional parameters are not set.

The special parameter **@** is set to the value of all the positional parameters, starting from the first parameter, passed to the shell script, each value being separated from the previous one by a space. You access the value of this parameter using the construct **\${@}**. If you access it in double quotes – as in **"\${@}"** – then the shell will treat each of the positional parameters as a separate *word* (which is what you normally want).

The special parameter **#** is set to the number of positional parameters *not counting positional parameter 0*. Thus it is set to the number of arguments passed to the shell script, i.e. the number of arguments on the command line when the shell script was run.

Shell parameters

- Positional parameters (`${0}`, `${1}`, etc)
- Value of all arguments passed: `${@}`
- Number of arguments: `${#}`

```
$ ~/examples/params.sh 0.5 62 38 hydrogen
```

```
This script is /home/y250/examples/params.sh
```

```
There are 4 command line arguments.
```

```
The first command line argument is: 0.5
```

```
The second command line argument is: 62
```

```
The third command line argument is: 38
```

```
Command line passed to this script: 0.5 62 38 hydrogen
```

In the `examples` subdirectory of your home directory there is a script called `params.sh`. If you run this script with some command line arguments it will illustrate how the positional parameters and related shell parameters work. Note that even if you type exactly the command line on the slide above your output will probably be different as the script will be in a different place for each user.

The positional parameter `0` is the name of the shell script (it is the name of the command that was given to execute the shell script).

The positional parameter `1` contains the first argument passed to the shell script, the positional parameter `2` contains the second argument passed and so on.

The special parameter `#` contains the number of arguments that have been passed to the shell script. The special parameter `@` contains all the arguments that have been passed to the shell script.

for

Execute some commands once for each value in a collection of values

```
for VARIABLE in <collection of values> ; do  
    <some commands>  
done
```

Examples:

```
myCOLOURS="red green blue"  
for zzVAR in ${myCOLOURS} ; do  
    echo "${zzVAR}"  
done  
  
for zzVAR in * ; do  
    ls -l "${zzVAR}"  
done
```

We can repeat a set of commands using a **for** loop. A **for** loop repeats a set of commands once for each value in a collection of values it has been given. We use a **for** loop like this:

```
for VARIABLE in <collection of values> ; do  
    <some commands>  
done
```

where **<collection of values>** is a set of one or more values (strings of characters). Each time the **for** loop is executed the shell variable **VARIABLE** is set to the next value in **<collection of values>**. The two most common ways of specifying this set of values is by putting them in a another shell variable and then using the **\${}** construct to get its value (note that this should *not* be in quotation marks), or by using a wildcard or file name glob (e.g. *****) to specify a collection of file names (*pathname expansion*). **<some commands>** is a list of one or more commands to be executed.

Note that you can put the **do** on a separate line, in which case you can omit the semi-colon (;):

```
for VARIABLE in <collection of values>  
do  
    <some commands>  
done
```

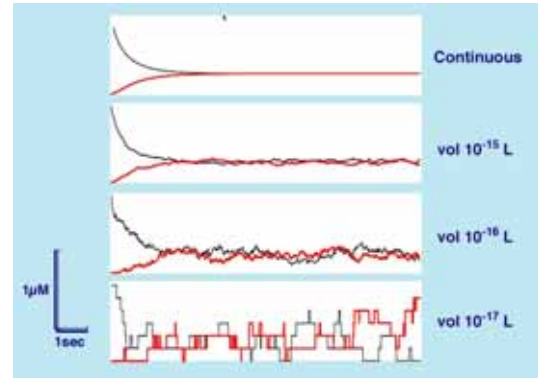
There are some examples of how to use it in the **for1.sh** and **for2.sh** scripts in the **examples** directory of your home directory. Note that a **for** loop can contain another **for** loop (the technical term for this is *nesting*).

Recap: What are we trying to do?



Scientific computing

i.e. shell scripts that do some **useful scientific work**, e.g. **repeatedly** running a simulation or analysis with **different** data



Recall the name of this course (“Simple Shell Scripting for Scientists”) and its purpose: to teach you, the scientist, how to write shell scripts that will be useful for your *scientific work*.

As mentioned on the previous day of the course, one of the most common (and best) uses of shell scripts is for automating repetitive tasks. Apart from the sheer tediousness of typing the same commands over and over again, this is exactly the sort of thing that human beings aren’t very good at: the very fact that the task is repetitive increases the likelihood we’ll make a mistake (and not even notice at the time). So it’s much better to write (once) – and test – a shell script to do it for us. Doing it via a shell script also makes it easy to **reproduce** and **record** what we’ve done, two very important aspects of any scientific endeavour.

So, the aim of this course is to equip you with the knowledge and skill you need to write shell scripts that will let you run some program (e.g. a simulation or data analysis program) over and over again with different input data and organise the output sensibly.

Sample program: **infect.py** (1)

```
$ ./infect.py 1.0 0.1 0.0005 500
```

SIR model (including births and deaths) with [event-driven] demographic stochasticity

```
Population size:          5.0000e+02
Model run time:          2.0e+00 years

Initial number of susceptible individuals:  5.000e+01
Initial number of infected individuals:     3.000e+00

Transmission rate (beta):          1.000000e+00
Recovery rate (gamma):             1.000000e-01
Per capita birth [and death] rate (mu): 5.000000e-04

Output file:                  infect.dat

Model took 1.506090e-02 seconds
```

The **infect.py** program is in your home directory. It is a program written specially for this course, but we'll be using it as an example program for pretty general tasks you might want to do with many different programs. Think of **infect.py** as just some program that takes some input on the command line and then produces some output (on the screen, or in one or more files, or both), e.g. a scientific simulation or data analysis program.

The **infect.py** program takes 4 numeric arguments on the command line: 3 positive floating-point numbers and 1 positive integer. It always writes its output to a file called `infect.dat` in the current working directory, and also writes some informational messages to the screen.

The **infect.py** program is not as well behaved as we might like (which, sadly, is also typical of many programs you will run). The particular way that **infect.py** is not well behaved is this: every time it runs it creates a file called `running` in the current directory, and it will not run if this file is already there (because it thinks that means it is already running). Unfortunately, it doesn't remove this file when it has finished running, so we have to do it manually if we want to run it multiple times in the same directory.

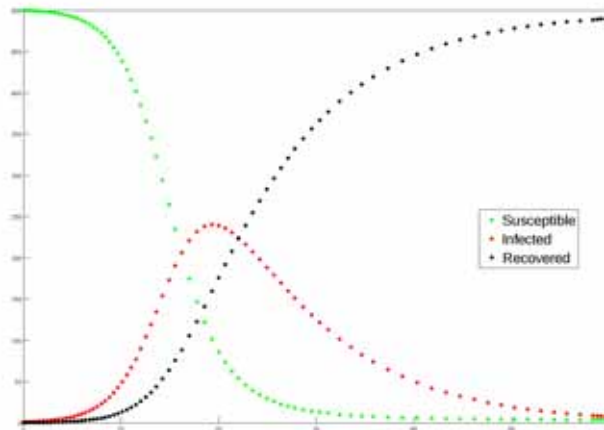
Sample program: `infect.py` (2)

Simulation of the spread of infection in a closed population (a variant of the SIR model used in epidemiology)

green = Susceptible

red = Infected

black = Recovered



The `infect.py` program uses a variant of the SIR model to simulate the spread of an infection in a closed (i.e. no one enters or leaves) population. The SIR model is what is called a *compartmental* model in epidemiology. In such models the population is divided into a series of mutually exclusive categories (“compartments”). In the SIR model the categories are “**S**usceptible” (those susceptible to infection who have not yet caught it), “**I**nfected” (those who have the infection) and “**R**ecovered” (those who have recovered from the infection or who have natural immunity). (It is assumed that once you have recovered from the infection you can't be re-infected.)

The SIR model is a simple model that works well for modelling the spread of infectious diseases such as measles, mumps and rubella.

In the variant of the model used by the `infect.py` program, births and deaths have been added to the model (these are called “demographic events”), and the model is stochastic (i.e. events occur randomly according to how probable they are). However, so that it is easier to compare parameter sets, the default behaviour of the program is to use a fixed sequence of pseudo-random numbers, which means that given the same set of input parameters it should produce the same output.

The `infect.py` program writes its output to a file of numbers rather than producing graphical output. As we saw at the end of the previous afternoon of the course, we can then use that file to produce graphs of its output.

Exercise from Day One

We have a directory that contains the output of several runs of the **infect.py** program in separate files. We have a file of commands that will turn the output into a graph (using **gnuplot**). We want to write a shell script that turns the output from each run into a graph.

We are specifically using the **gnuplot** program and the output of the **infect.py** program we met on the previous day of the course. (**gnuplot** is a program that creates graphs, histograms, etc from numeric data.) Think of this task as basically: I have some data sets and I want to process them all in the same way. My processing might produce graphical output, as here, or it might produce more data in some other format.

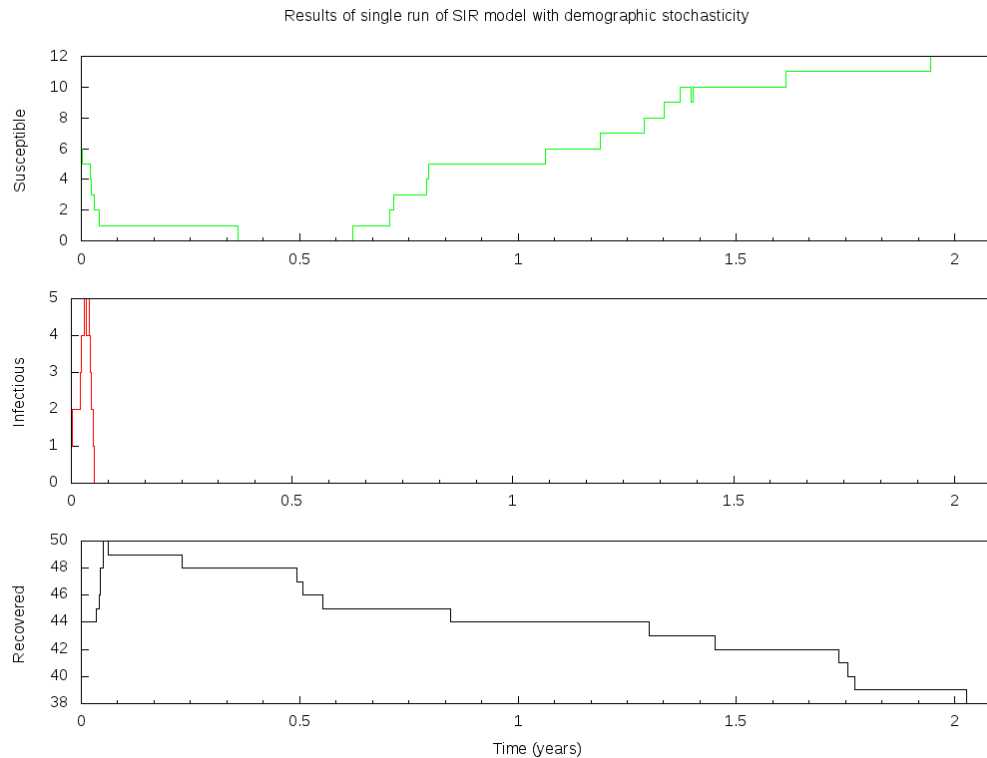
If you haven't met **gnuplot** before, you may wish to look at its WWW page:

<http://www.gnuplot.info/>

If you think you might want to use the **gnuplot** program for creating your own graphs, then you may find the "Introduction to Gnuplot" course of interest – the course notes are on-line at:

<http://www.ucs.cam.ac.uk/docs/course-notes/unix-courses/earlier/Gnuplot/>

Output of gnuplot



scientific-computing@ucs.cam.ac.uk

Simple Shell Scripting for Scientists: Day Two

18

If you want to get an idea of what we're trying to do, you can try the following:

```
$ cd
```

```
$ scripts/multi-run.sh 50
```

```
$ cp gnuplot/infect.gplt .
```

```
$ cp infect-50.dat infect.dat
```

```
$ ls infect.png
```

```
/bin/ls: infect.png: No such file or directory
```

```
$ gnuplot infect.gplt
```

```
$ rm infect.dat
```

```
$ ls infect.png
```

```
infect.png
```

```
$ eog infect.png &
```

Note that the output of “`ls infect.png`” may look slightly different – in particular, the colours may be slightly different shades (assuming you are reading these notes in colour).

Details of exercise

What we want to do is, **for** each output file:

1. Rename (or copy) the output file we want to process to `infect.dat`

```
$ mv infect-50.dat infect.dat
```

2. Run `gnuplot` with the `infect.gplt` file

```
$ gnuplot infect.gplt
```

3. Rename (or delete if you copied the original output file) `infect.dat`

```
$ mv infect.dat infect-50.dat
```

4. Rename `infect.png`

```
$ mv infect.png infect-50.dat.png
```

The exercise set at the end of the previous day of the course was to create a shell script that does the above task. Basically, for each of the `.dat` files produced by the `multi-run.sh` script, the shell script should run **gnuplot** on it to create a graph (which will be stored as a `.png` file). The `infect.gplt` file provided will only work if the `.dat` file is called `infect.dat` and is in the current directory. Also, **gnuplot** should not be allowed to overwrite each `.png` file, so the shell script must rename each `.png` file after **gnuplot** has created it.

multi-gnuplot1.sh

```
#!/bin/bash

# Run gnuplot program once for each output file
for zzFILES in infect-*.dat ; do

    # Rename output file to infect.dat
    mv "${zzFILES}" infect.dat

    # Run gnuplot
    gnuplot infect.gplt

    # Rename infect.dat to original name
    mv infect.dat "${zzFILES}"

    # Rename infect.png
    mv infect.png "${zzFILES}.png"
done
```

So here's one solution to that exercise. This file (`multi-gnuplot1.sh`) is in the `gnuplot` directory.

It takes each file whose name is of the form `infect-<something>.dat` (where the `<something>` can be any set of characters that can appear in a filename) in turn and renames it to `infect.dat`, runs **gnuplot**, then renames the file back to its original name, and renames the `infect.png` file to `infect-<something>.dat.png`.

multi-gnuplot2.sh

```
#!/bin/bash

# Run gnuplot program once for each output file
for zzFILES in infect-*.dat
do
    # Copy output file to infect.dat
    cp -f "${zzFILES}" infect.dat

    # Run gnuplot
    gnuplot infect.gplt

    # Delete infect.dat file
    rm -f infect.dat

    # Rename infect.png
    mv infect.png "${zzFILES}.png"
done
```

...and here's another solution. This file (`multi-gnuplot2.sh`) is in the `gnuplot` directory.

It takes each file whose name is of the form `infect-<something>.dat` (where the `<something>` can be any set of characters that can appear in a filename) in turn and copies it to `infect.dat`, runs **gnuplot**, then deletes the copy, and renames the `infect.png` file to `infect-<something>.dat.png`.

These two shell scripts are functionally equivalent – you can use whichever you like and the results will be identical.

Note that one purely cosmetic difference between them is that one has the **do** keyword on the same line as the **for** keyword (with a semi-colon (;) before the `do`) whilst the other has the **do** keyword on a separate line (and no semi-colon). Some people feel that it makes scripts more readable to put the **do** on a separate line.

However, whether you put the **do** on the same line as the **for** (and use the semi-colon) or put it on a different line is entirely a matter of style and personal preference – well, you want some outlet for your individuality, don't you? 😊

multi-gnuplot3.sh

```
#!/bin/bash

# Run gnuplot program once for each output file
for zzFILES in infect-*.dat ; do

    # Create symbolic link called infect.dat to output file
    ln -s -f "${zzFILES}" infect.dat

    # Run gnuplot
    gnuplot infect.gplt

    # Delete infect.dat symbolic link
    rm -f infect.dat

    # Rename infect.png
    mv infect.png "${zzFILES}.png"
done
```

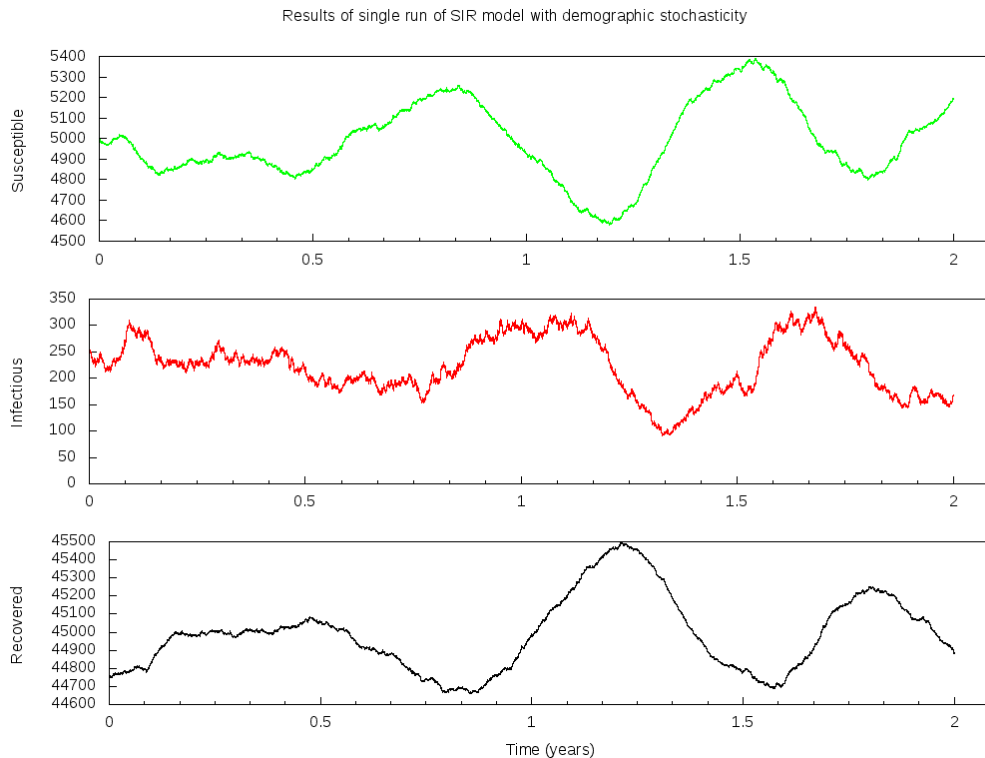
...and here's yet another solution. This file (`multi-gnuplot3.sh`) is also in the `gnuplot` directory.

It takes each file whose name is of the form `infect-<something>.dat` (where the `<something>` can be any set of characters that can appear in a filename) in turn and creates a *symbolic link* to it called `infect.dat`, runs **gnuplot**, then deletes the symbolic link (*not* the original file), and renames the `infect.png` file to `infect-<something>.dat.png`.

This shell script is functionally equivalent to the previous two – you can use whichever you like and the results will be identical.

There is, though, one way in which this script is better than the previous two. Since it only creates a symbolic link to each file in turn rather than making a copy of the file (like `multi-gnuplot2.sh`), it uses considerably less disk space (symbolic links take up almost no space on disk), which can be an issue if the files you are processing are large. Also, since it does not rename the original file (like `multi-gnuplot1.sh`), if it is interrupted part way through its execution you don't need to worry about potentially "losing" any output files. If `multi-gnuplot1.sh` was interrupted *after* it had renamed a file to `infect.dat` but *before* it had a chance to rename it back, then, unless the person running it realised this had happened and dealt with it, the `infect.dat` file would be **deleted** next time the script was run(!).

Sample output: infect - 50000 . dat . png



scientific-computing@ucs.cam.ac.uk

Simple Shell Scripting for Scientists: Day Two

23

You can try out one of these scripts if you want. First, create some output files for the script to process:

```
$ cd
$ rm -f *.dat stdout-* logfile
$ scripts/multi-run.sh 50 100 500 1000 3000 5000 10000 50000
```

Now, make sure that the `infect.gplt` file is in your current directory:

```
$ cp gnuplot/infect.gplt .
```

Now run one of the scripts, either `multi-gnuplot1.sh` or `multi-gnuplot2.sh` or `multi-gnuplot3.sh`, it doesn't matter which:

```
$ gnuplot/multi-gnuplot1.sh
```

Now do an `ls` to see what files have been created, and then try viewing some of them:

```
$ eog infect-50000.dat.png &
```

Your solutions to this exercise (you did do it, didn't you?) should have been similar to the ones presented here. If they weren't, or if you had problems with the exercise, please let the course giver or demonstrator know.

Shell functions

```
$ cd
$ cat hello-function.sh
#!/bin/bash
function hello()
{
    # This is a shell function.
    echo "Hello."
    echo "I am function ${FUNCNAME}."
}

$ ./hello-function.sh
$
```

Shell functions are similar to functions in most high-level programming languages. Essentially they are “mini-shell scripts” (or bits of shell scripts) that are invoked (*called*) by the main shell script to perform one or more tasks. When called they can be passed arguments (parameters), as we will see later, and when they are finished they return control to the statement in the shell script immediately after they were called.

To define a function, you just write the following at the start of the function:

```
function function_name()
{
where function_name is the name of the function. Then, after the last line of the
function you put a line with just a closing curly brace (}) on it:
}
```

Note that *unlike* function definitions in most high level languages you don't list what parameters (arguments) the function takes. This is not so surprising when you remember that shell functions are like “mini-shell scripts” – you don't explicitly define what arguments a shell script takes either.

Like functions in a high-level programming language, defining a shell function doesn't actually make the shell script do anything – the function has to be called by another part of the shell script before it will actually *do* anything.

FUNCNAME is a special shell variable (introduced in version 2.04 of bash) that the shell sets within a function to the name of that function. When not within a function, the variable is unset.

Calling a shell function

```
$ gedit hello-function.sh &
#!/bin/bash
function hello()
{
    # This is a shell function.
    echo "Hello."
    echo "I am function ${FUNCNAME}."
}
```

hello

```
$ ./hello-function.sh
Hello.
I am function hello.
$
```

Start your favourite editor (or gedit if you don't have a preference) and modify the file `hello-function.sh` in your home directory as shown above. *Make sure you save the file after you've modified it or your changes won't take effect.*

Now try running the shell script again:

```
$ ./hello-function.sh
Hello.
I am function hello.
$
```

This time it actually does something – the function **hello** is called and does what we would expect.

You call a shell function by just giving its name (just as you would with any of the standard Unix commands (or shell builtin commands) that we've met). Note that you don't put brackets after the name of the function when you call it. You only do that when you first define the function. That's one of the ways that the shell figures out that you are trying to define a shell function.

Shell function arguments (1)

```
$ gedit hello-function.sh &
#!/bin/bash
function hello()
{
    # This is a shell function.
    echo "Hello, ${1}"
    echo "I am function ${FUNCNAME}."
}

hello

$ ./hello-function.sh Dave
Hello,
I am function hello.
$
```

Modify the file `hello-function.sh` in your home directory as shown above. *Make sure you save the file after you've modified it or your changes won't take effect.*

Recall that the positional parameter **1** (whose value is accessed using the construct `${1}`) contains the value of the first argument passed to the shell script (or is unset if no arguments are passed). So what would we expect the above shell script to do? Surely, it will print out "Hello, *<whatever argument we gave it>*"?

(For the pedants amongst you: *<whatever argument we gave it>* means whatever argument we passed the shell script on the command line when we invoked it – "Dave" in the above example.)

Apparently not. Maybe something's wrong with our shell script? Maybe positional parameter **1** isn't being set correctly? Let's try some debugging and see.

Shell function arguments (2)

```
$ gedit hello-function.sh &
#!/bin/bash
function hello()
{
    # This is a shell function.
    echo "Hello, ${1}"
    echo "I am function ${FUNCNAME}."
}
```

```
echo "First argument: ${1}"
```

```
hello
```

```
$ ./hello-function.sh Dave
```

```
First argument: Dave
```

```
Hello,
```

```
I am function hello.
```

Modify the file `hello-function.sh` in your home directory as shown above. *Make sure you save the file after you've modified it or your changes won't take effect.*

This is a simple but useful debugging trick for shell scripts. When something isn't working right, make the shell script print out the values of all the shell variables, environment variables or shell parameters that you are interested in *just before* the point where you think it is going wrong.

In this case, what this shows us is that positional parameter **1** is being set correctly. So that's not the problem.

The problem is that *within* a function the positional parameters (from **1** onward, **0** doesn't change) are set to the arguments that the function was given when it was called. (Similarly, within a function the special parameters `@` and `#` are set to all the arguments passed to the function, and the number of arguments passed to the function, respectively.) Since we called the function **hello** without any arguments, while the function **hello** is executing positional parameter **1** is unset, and so when we try to print its value, nothing is printed.

The way you call a shell function with arguments is to list those arguments immediately after the name of the shell function, e.g. in our script:

```
hello Dave
```

would call the function **hello** with one argument: "Dave".

Shell function arguments (3)

```
$ gedit hello-function.sh &
#!/bin/bash
function hello()
{
    # This is a shell function.
    echo "Hello, ${1}"
    echo "I am function ${FUNCNAME}."
}

echo "First argument: ${1}"
hello Hal

$ ./hello-function.sh Dave
First argument: Dave
Hello, Hal
I am function hello.
```

Modify the file `hello-function.sh` in your home directory as shown above. *Make sure you save the file after you've modified it or your changes won't take effect.*

So, if we call our function with an argument (in this case the argument is "Hal"), then the value of the positional parameter **1** is indeed set to that argument within the function.

So, if we want to our function to have the same first argument as the shell script itself, then we need to call the function with the first argument with which the shell script was invoked.

You can probably guess how we do this...

Shell function arguments (4)

```
$ gedit hello-function.sh &
#!/bin/bash
function hello()
{
    # This is a shell function.
    echo "Hello, ${1}"
    echo "I am function ${FUNCNAME}."
}

#echo "First argument: ${1}"
hello "${1}"

$ ./hello-function.sh Dave
Hello, Dave
I am function hello.
```

Modify the file `hello-function.sh` in your home directory as shown above. *Make sure you save the file after you've modified it or your changes won't take effect.*

Note that now we think we've cracked it, we can get rid of our debugging effort. We could delete that line, but, if we were wrong, we'd only have to put it back in again as we tried to figure it out. So it is easier to just comment it out by inserting a hash character (`#`) at the start of the line – recall that the shell treats everything after a hash at the start of a line as a comment.

But as you probably guessed – it does indeed work the way we want. Positional parameter **1** holds the first argument that was given on the command line to the shell script, so if we want to pass that argument to the hello function, we just put:

```
hello "${1}"
```

in our shell script, and voilà!

Why use shell functions?

- Allow us to structure our shell script:
 - Functions ↔ sub-tasks
- Easier to write small parts of the shell script at a time:
 - So we can write the easy bits first!
- Easier to test individual parts of the shell script
- Repetitive sequences of commands only appear in one place:
 - Less typing! Fewer typos!
 - Easy to make changes
 - Easy to fix errors
- Can re-use functions in different shell scripts

If you're familiar with computer programming, you'll probably have already come across the concept of functions in whatever programming languages you are familiar with. The advantages of using shell functions are basically the same as the advantages of using functions in a programming language, as you can probably tell from the slide above.

Script as a series of functions

```
function start()
{
    ...
}
function do_something()
{
    ...
}
function end()
{
    ...
}
function main()
{
    ...
}

main "${@}"
```

If you've implemented your shell script entirely as shell functions, there is a really nice trick you can use when something goes wrong and you need to debug your script, or if you want to re-use some of those functions in another script. As you've implemented the script entirely as a series of functions, you have to call one of those functions to start the script actually doing anything. For the purposes of this discussion, let's call that function **main**. So your script looks something like that shown on the slide above. (You can see an example of a script like this in the `examples` directory in the file `function-script.sh`.)

By commenting out the call to the **main** function, you now have a shell script that does *nothing* except define some functions. You can now easily call the function(s) you want to debug/use from another shell script using the **source** shell builtin command (as we'll see on the optional final day of this course). This makes debugging *much* easier than it otherwise might be, even of really long and complex scripts.

Improving `multi-run.sh`

- Depends on `run-once.sh`
- Location of `run-once.sh` hard-coded into script:
 - If we move `run-once.sh`, script breaks until it is updated

```
#!/bin/bash

# Parameters that stay the same each run
myFIXED_PARAMS="1.0 0.1 0.0005"

# Run infect.py program once for each argument
# Note: *no* quotes around ${myFIXED_PARAMS}
#       or they'll be interpreted as one argument!
for zzARGS in "${@}" ; do
    "${HOME}/scripts/run-once.sh" ${myFIXED_PARAMS} "${zzARGS}"
done
```

On the previous day of this course we met the scripts `multi-run.sh` and `run-once.sh`. Together, these scripts gave us a nice way of running a program several times with different parameter sets. However, they are not as versatile as we might hope. `run-once.sh` requires that the program it runs (**`infect.py`**) be in the current directory. Since, in this example, **`infect.py`** is a special program for us (imagine it were your program that you had written from scratch), that's not such a bad limitation, since we quite probably would have a working copy of the program in the directory where we were going to store its output.

`multi-run.sh`, on the other hand, depends on the `run-once.sh` script, and has the location of that script *hard-coded* into it. If we move the `run-once.sh` script for some reason, then `multi-run.sh` will immediately stop working. Wouldn't it be nice if we could somehow avoid this problem, but still keep the functionality of the two scripts somewhat separate?

One of way of doing exactly that would be to incorporate `run-once.sh` into `multi-run.sh` as a shell function. That should be quite easy. We define a function in `multi-run.sh` that does *exactly* the same thing as the `run-once.sh` script, and in our for loop, instead of calling the `run-once.sh` script, we call our function.

So, let's do that and see what happens.

First exercise

Add `run-once.sh` to `multi-run.sh`
as a *shell function*:

```
#!/bin/bash

function run_program()
{
# This function runs the infect.py program
    What goes here?
}

# Parameters that stay the same each run
myFIXED_PARAMS="1.0 0.1 0.0005"

# Run infect.py program once for each argument
# Note: *no* quotes around ${myFIXED_PARAMS}
#       or they'll be interpreted as one argument!
for zzARGS in "${@}" ; do
    run_program ${myFIXED_PARAMS} "${zzARGS}"
done
```

The `multi-run.sh` and `run-once.sh` shell scripts are in the `scripts` directory of your home directory. Your task is to get the functionality of the `run-once.sh` script into the `multi-run.sh` script as a *shell function*. Above I've given you the skeleton of what the modified script should look like. You should be able to fill in the rest.

This should be a quick exercise, so when you finish it, take a short break and then we'll start again with the solution. (I really **do** mean take a break – sitting in front of computers for long periods of time is very bad for you. Get up, move around, have a drink, do a little dance, relax...)

You can check that you've done it correctly by trying to run your modified `multi-run.sh` script (*remember to save it after you've made your modifications!*):

```
$ cd
$ rm -f *.dat stdout-* logfile
$ ls
$ scripts/multi-run.sh 50 100 500 1000 3000 5000 10000 50000
$ ls
```

Hint: This exercise is essentially a cut-and-paste (or copy-and-paste) task. If you are trying to do anything much more complicated than that, then you're on the wrong track...

Recap: Shell functions

- “mini-shell scripts”
- Usually used for well-defined tasks (often called repeatedly)
- Specify arguments by listing them after function name when calling function
hello Dave
- Positional parameters (and related special shell parameters) set to function’s arguments within function
In function hello, positional parameter **1** = Dave

One thing worth noticing from the exercise we’ve just done:

The original script had the line:

```
"${HOME}/scripts/run-once.sh" ${myFIXED_PARAMS} "${zzARGS}"
```

The new script has the line:

```
run_program ${myFIXED_PARAMS} "${zzARGS}"
```

Note that arguments that we are passing have not changed in the slightest. In the original script we were calling another shell script with some arguments. In our new script we are calling a *shell function* with the *same* arguments. The syntax for these is almost identical: the main change is the name (and location) of the things being called. See?, I told you shell functions were like “mini-shell scripts”. ☺

Testing

```
#!/bin/bash

function run_program()
{
# This function runs the infect.py program
    ...
}

# Parameters that stay the same each run
myFIXED_PARAMS="1.0 0.1 0.0005"

run_program ${myFIXED_PARAMS} 80

# Run infect.py program once for each argument
# Note: *no* quotes around ${myFIXED_PARAMS}
#       or they'll be interpreted as one argument!
#for zzARGS in "${@}"; do
# run_program ${myFIXED_PARAMS} "${zzARGS}"
#done
```

One of the advantages of writing a shell script using shell functions should be immediately apparent. The main body of this shell script – the **for** loop – is nice and simple. It just calls a function over and over varying one parameter each time. Because we’ve hidden the commands that do the real work in a shell function, we can see this immediately just by looking at the script.

If we’d put all the lines in the **run_program** function in the **for** loop it would have obscured the script’s structure, and we might have spent a lot of time trying to figure out what the individual lines of script did before realising what was going on. It also helps that we’ve chosen a meaningful name for our shell function. So just by looking at the script we can immediately say “Aha! This script probably runs a program (**run_program**) several times, varying one of its parameters each time.” (Of course, at this point we’d be taking it on faith that the author of the shell script wasn’t an evil troll who deliberately chose *misleading* names for his shell functions. Fortunately, most of those spend the majority of their time under bridges harassing goats.)

Another advantage is that we can easily test our shell function by just commenting the other complicating bits of the shell script out (as above) and just running the function once with some test arguments. This is worth doing every time you’ve written a new function (especially if it is complicated) so that you know it behaves the way you expected it to. It also means that you know that, if there is an error, it is *not* in that part of the shell script (that shell function). That makes it much easier to track down errors.

You can save the above modifications and try out the script if you want: it should just run the **run_program** function once, producing two output files (`infect-80.dat` and `stdout-80`) and writing some information about what it is doing to the log file `logfile`.

If you do try it out, make sure that you undo those modifications and return the shell script to its former state (and save it) as we will be using the shell script later.

Command substitution

Sometimes we want to get the output of a command and use it in our shell script, for instance, we might want a shell variable to hold the output of a command. How do we do this?:

```
$( command )
```

```
$ cd /tmp
$ myDIRECTORY="$(pwd)"
$ echo "I will use directory: ${myDIRECTORY}"
I will use directory: /tmp
```

Command substitution is the process whereby the shell runs a *command* and *substitutes* the command's output for wherever the command originally appeared (in a shell script or on the command line).

So, for example, the following line in a shell script:

```
myDIR="$(pwd)"
```

would set the shell variable **myDIR** to the full path of the current working directory. (We don't have to surround the **\$(pwd)** in quotes, but it is a good idea: the path may contain spaces.) This is how it works:

1. The shell runs the **pwd** command. The **pwd** command prints out the full path of the current working directory, i.e. its *output* is the full path of the current working directory. Let's suppose we were in `/tmp`, so the output of the **pwd** command would be `"/tmp"`.
2. The shell takes this output (`"/tmp"`) and substitutes it for where the original expression **\$(pwd)** appeared. So what we now have is:

```
myDIR="/tmp"
```

3. As you probably know by now, this is just the normal way of assigning a value to a shell variable, and, sure enough, that's exactly what the shell does: it assigns the value `"/tmp"` to the shell variable **myDIR**.

Instead of the **\$()** construct you can also use *backquotes*, i.e. you can use ``command`` instead of **\$(command)**, and you are likely to come across these in many shell scripts. However, the use of backquotes is generally a very bad idea for two reasons: (1) it's very easy to misplace or overlook a backquote (with catastrophic results) as the backquote character (```) is so small, and (2) it's very difficult to use backquotes to do *nested* command substitution (one command substitution inside another one).

Improving multi-run.sh (2)

```
#!/bin/bash

function run_program()
{
    ...
    # Run program with passed arguments
    "${myPROG}" "${@"}" > "stdout-${4}"
    ...
}

# Program to run: infect.py
myPROG="$(pwd -P)/infect.py"

# Set up environment variables for program
export INFECT_FORMAT="NORMAL"

# Parameters that stay the same each run
myFIXED_PARAMS="1.0 0.1 0.0005"
    ...
```

scientific-computing@ucs.cam.ac.uk

Simple Shell Scripting for Scientists: Day Two

37

Let's make a small, but major, improvement to the multi-run.sh script (this script is in the scripts directory of your home directory). Change the lines:

```
# Run infect.py with passed arguments
INFECT_FORMAT="NORMAL" ./infect.py "${@"}" > "stdout-${4}"
to:
# Run program with passed arguments
"${myPROG}" "${@"}" > "stdout-${4}"
```

And add the lines:

```
# Program to run: infect.py
myPROG="$(pwd -P)/infect.py"

# Set up environment variables for program
export INFECT_FORMAT="NORMAL"
```

immediately **before** the line:

```
# Parameters that stay the same each run
```

Why is this such a major improvement?

Firstly, by replacing the hard-coded `./infect.py` with a shell variable, we have made it much easier to modify the script to use other programs instead of `infect.py`. (Not to mention making it much more obvious where we make such a modification. And by explicitly setting the `INFECT_FORMAT` environment variable in an adjacent part of the script we have also made it more obvious where any environment variables the program uses should be changed, should we need to do so.)

Secondly, by obtaining the full path of the `infect.py` program our shell script can now work in another directory than the one we start off in, as we now have a full path to the `infect.py` program and so can run it from whatever directory we may be in. We'll see why this is a good idea in a minute.

For those wondering what "`pwd -P`" does, recall that, as already mentioned, `pwd` prints out the full path of the current working directory. With the `-P` option it prints out the full *physical* path, i.e. the path will contain no symbolic links which might change over time (or otherwise confuse things).

You should check that this modified multi-run.sh script still works – *remember to save it after you've made your modifications* – with the same sequence of commands given for this purpose on the page 33 of your notes.

The **mktemp** command

Safely **m**akes **t**emporary files or directories for you

Options:

- d make a **d**irectory instead of a file
- t make file or directory in a **t**emporary directory (usually /tmp)

```
$ mktemp -t -d infect.XXXXXXXXXX  
/tmp/infect.khhcE30735
```

The **mktemp** command is an extremely useful command that allows users to *safely* create temporary files or directories on multi-user systems. It is very easy to *unsafely* create a temporary file or directory to work with from a shell script, and, indeed, if your shell script tries to create its own temporary files or directories using the normal Unix commands then it is almost certainly doing so unsafely. Use the **mktemp** command instead.

Note that if you try the example above you will almost certainly get a directory with a different name created for you.

Note also that **mktemp** has more options than the two listed above, but we won't be using them in this course. Note also that if you use a version of **mktemp** earlier than version 1.3 (or a version derived from BSD, such as that shipped with MacOS X) then you can't use the **-t** option, and will have to specify /tmp (or another temporary directory) explicitly, e.g.

```
mktemp -d /tmp/infect.XXXXXXXXXX
```

How do you use **mktemp**? You give it a "template" which consists of a name with some number of **X**'s appended to it (note that is an **UPPER CASE** letter **X**), e.g. **infect.XXXXX**. **mktemp** then replaces the **X**'s with random letters and numbers to make the name unique and creates the requested file or directory. It outputs the name of the file or directory it has created.

Improving multi-run.sh (3a)

```
#!/bin/bash
set -e

...

# My current directory
myDIR="$(pwd -P)"
# Temporary directory for me to work in
myTEMP_DIR="$(mktemp -t -d infect.XXXXXXXXXX)"

# Change to temporary directory
cd "${myTEMP_DIR}"
# Run program once for each argument
# Note: *no* quotes around ${myFIXED_PARAMS}
#       or they'll be interpreted as one argument!
for zzARGS in "${@}"; do
    run_program ${myFIXED_PARAMS} "${zzARGS}"
done

# Copy files back to my directory
cp -fpR . "${myDIR}"
# Go back to my directory
cd "${myDIR}"
# Clean up
rm -Rf "${myTEMP_DIR}"
```

Modify the multi-run.sh script in the scripts directory as shown above.

The improvement we've made here is to now do all our calculations in a temporary directory, and only copy the output files (and log file) back to our working directory when we've finished.

(You should understand what all the lines of shell script we've just added are doing – if you don't please ask the course giver or demonstrator to explain.)

Why is this an improvement? Well, if, as in this course, the directory we are working from (our home directory) is actually on a network filesystem, then this can have a major impact on performance, particularly when the network is busy (like when a whole classroom is doing this course). By working in /tmp, which is usually a local filesystem (as it is for PWF Linux machines) we no longer have to deal with the network overheads and bottlenecks except right at the very end of the process. This should make things much quicker. It also potentially makes things more reliable as well, as it minimises the opportunity for network problems to mess up our work. (Hurrah!)

One other important thing to note is that we've told our script to abort as soon as it hits an error. That's what adding the "set -e" line immediately after "#!/bin/bash" at the start of the file does (you did remember to make that modification, right?). (We can also get the same effect by starting the bash shell with the -e option, for instance by changing the "#!/bin/bash" line at the start of the file to "#!/bin/bash -e" although it is better to use "set -e".)

Why do this now? The reason is that our shell script is now doing something dangerous: **it is changing the working directory**. Why is that dangerous? Well, imagine I tried to change to a directory and failed for some reason. Thinking I'm in a different directory than I actually am, I promptly delete everything in it. Oops!

We have one more change to make (see the next slide) and then you can check that you've modified your script correctly by trying to run your modified multi-run.sh script (*remember to save it after you've made your modifications!*).

Improving multi-run.sh (3b)

```
#!/bin/bash
set -e

function run_program()
{
    ...

    # Write to logfile
    echo "" >> "${myLOGFILE}"
    date >> "${myLOGFILE}"
    echo "Running ${myPROG} with ${@}" >> "${myLOGFILE}"

    ...

    # Write to logfile
    echo "Output file: infect-${4}.dat" >> "${myLOGFILE}"
    echo "Standard output: stdout-${4}" >> "${myLOGFILE}"

    ...

    # My current directory
    myDIR="$(pwd -P)"
    # Location of log file
    myLOGFILE="${myDIR}/logfile"

    ...
}
```

Now modify the `multi-run.sh` script in the `scripts` directory as shown above.

We've made two improvements here. The first is to use a shell variable to hold the location of our log file (so we only have to change its location in one place in the future). The second (and more important) is to make our script write to the end of the existing logfile in the current directory when we run the script rather than *overwriting* logfile each time we run the script. Since the log file is supposed to contain a record of *all* the runs of the script that we do for posterity (and debugging), we normally wouldn't want it to be replaced with a new log file each time we run the script.

(You should understand what all the lines of shell script we've just added are doing – if you don't please ask the course giver or demonstrator to explain.)

You can check that you've done it correctly by trying to run your modified `multi-run.sh` script (*remember to save it after you've made your modifications!*):

```
$ cd
$ rm -f *.dat stdout-*
$ ls
$ scripts/multi-run.sh 50 100 500 1000 3000 5000 10000 50000
$ ls
```

Now do it!

Make the changes to `multi-run.sh` indicated on the previous slides (37, 39 & 40) and then ***try*** the improved script.

Then take a short break. We'll start again in 5 minutes or thereabouts.

The `multi-run.sh` shell script is in the `scripts` directory of your home directory. Make the modifications indicated on the previous slides (37, 39 & 40), if you haven't already.

Now check that you've done it correctly by trying to run your modified `multi-run.sh` script (*remember to save it after you've made your modifications!*):

```
$ cd
$ rm -f *.dat stdout-*
$ ls
$ scripts/multi-run.sh 50 100 500 1000 3000 5000 10000 50000
$ ls
```

And when you finish doing this, please do take a quick break before we continue. (And that's "break" as in "break from the computer" not "break to check my e-mail".)

read

Get input from standard input...

...try to put each word (value) in as many separate variables as are provided...

read VAR1 VAR2 VAR3

Options:

-p Use the following string as a **p**rompt for the user

```
$ read -p "What is the answer?: " myANSWER
What is the answer?: 42
$ echo "${myANSWER}"
42
```

The **read** shell builtin command takes input from standard input (usually the keyboard) and returns it in the specified shell variable. If you don't specify a shell variable, it will return it in a shell variable called **REPLY**.

The **-p** option gives **read** a string that it displays as a prompt for the user.

You can give **read** more than one shell variable in which to return its input. What happens then is that the first *word* it reads goes into the first shell variable, the second word into the second shell variable and so on.

If there are more words than shell variables, the extra words all are put into the last shell variable.

If there are more shell variables than words, each of the extra variables are set to the empty string.

As far as **read** is concerned a “word” is a sequence of characters that does *not* contain a space, i.e. it considers spaces as the thing that separates one word from another. (The technical term for “thing that separates one thing from another” is “*delimiter*”.)

Using read (1)

```
$ cd
$ gedit scripts/run-once-using-read.sh &

#!/bin/bash

# Read in parameters from standard input
read -p "Input parameters for infect.py: " myI myR myB mySIZE
...

$ cd
$ scripts/run-once-using-read.sh
Input parameters for infect.py: 1.0 0.1 0.05 70
$
```

In the `scripts` directory there is a shell script called **`run-once-using-read.sh`**. Open this up with your favourite editor (or `gedit`) and have a look at it.

The first line (that doesn't start with a `#` character) is a **`read`** shell builtin command that reads some values from standard input and puts them in some shell variables. (You should be able to work out how the rest of the script has been modified to use these shell variables – if there is anything you don't understand, ask the course giver or demonstrator.)

Let's try this script out and see how it behaves.

Using read (2)

```
$ cd
$ rm -f *.dat stdout-* logfile *.png
$ scripts/run-once-using-read.sh
Input parameters for infect.py: 1.0 0.1 0.05 70 garbage
Traceback (most recent call last):
  File "infect.py", line 149, in <module>
    N0 = long(sys.argv[4])
ValueError: invalid literal for long() with base 10: '70 garbage'
mv: cannot stat `infect.dat': No such file or directory

$ gedit scripts/run-once-using-read.sh &
```

```
#!/bin/bash

# Read in parameters from standard input
read -p "Input parameters for infect.py: " myI myR myB mySIZE myJUNK
```

So, on first try it seemed to do what we'd expect. However, if we give it some input that should be invalid something slightly strange happens. If we give it 5 input parameters instead of 4, instead of complaining, or only using the first 4 parameters, it puts the last two parameters together to form one argument ("0.7 garbage") in the above example and runs the **infect.py** program with that (we can see this is what is happening by inspecting the contents of the log file `logfile`). This causes the **infect.py** program to crash with an error message that is less clear than one might hope (an indication that the **infect.py** program is (yet again) not as well written as we might like (an all too common complaint with software)). (Also, as a result of **infect.py** crashing, the **mv** command our shell script uses to rename `infect.dat` file then complains that there is no file for it to rename.)

Regardless of how well or badly the **infect.py** program handles invalid parameters, that fact that our script gives it mangled input to work with is an indication that our script is broken. What is the problem and how can we fix it?

Recall how **read** works: if it reads more words (values) than it was given shell variables, it puts all the extra ones together in the last shell variable. This is what is happening here, and it is undesirable. We can fix this by giving **read** an extra "dummy" shell variable that we never use, but that is simply there to hold any extra junk it may read in.

Modify the `run-once-using-read.sh` shell script in the `scripts` directory as shown above (remember to save it when you've finished).

Using read (3)

```
$ cd
$ rm -f *.dat stdout-* logfile
$ scripts/run-once-using-read.sh
Input parameters for infect.py: 1.0 0.1 0.05 70 garbage
$ ls
answers  gnuplot          infect.gplt      scripts
bin      hello-function.sh logfile          source
Desktop  hello.sh         infect-70.dat   stdout-70
examples infect.py         run-infect.sh   treasure.txt
```

Now it works better. If we give it more than 4 input parameters it doesn't mangle the 4th argument that it passes to the **infect.py** program.

(Note that the output of the **ls** command may not exactly match what is shown above – in particular there may be other files or directories shown, and the colours may be slightly different.)

Now this may seem like a lot of trouble to go to for not much in the way of improvement to our script. After all, the original `run-once.sh` script could perfectly well accept a single set of 4 parameters without all these problems – it just wanted them on the command line rather than from standard input.

So, what's the big deal about standard input? After all, if I have lots of parameter sets to run I'm hardly going to sit there and type them all in one at a time!

Well, how many command line arguments can a shell script have? The answer is quite a few but **not** an unlimited number. In fact, if I have thousands of parameter sets, that's definitely going to be too many for me to pass to my shell script all in one go (or even a small number of goes) on the command line. So, how do we deal with situation?

Hmmm, maybe if I could put all my thousands of parameter sets into a file, and then could somehow get my shell script to read in that file, one parameter set at a time, that might do it... we need to be able to do a few more things to make that particular idea fly, so let's have a look at some of them now...

Pipes

A *pipe* takes the
...*output* of one command...
...and passes it to another command as
input...

```
command1 | command2
```

Pipes can be combined:

```
command1 | command2 | command3
```

A set of one or more pipes is known as a
pipeline

A *pipe* takes the *output* of one command and feeds it to another command as *input*. We tell the shell to do this using the `|` symbol. So:

```
ls | more
```

takes the the output of the **ls** command and passes it to the **more** command, which displays the output of the **ls** command one screenful at a time. We can combine several pipes by taking the output of the last command of each pipe and passing it to the first command in the next pipe, e.g.

```
ls | grep 'fred' | more
```

takes the output of **ls** and passes it to **grep**, which searches for lines with the string “fred” in them, and then the output of **grep** is passed to the **more** command to display one screenful at a time. A set of one or more pipes is known as a *pipeline*. This pipeline would show us all the files with the string “fred” in their name, one screenful at a time.

Using pipes

```
$ cd
$ rm -f *.dat stdout-* logfile
$ cat scripts/basic_param_set
1.0 0.1 0.0005 50
1.0 0.1 0.0005 100
1.0 0.1 0.0005 500
1.0 0.1 0.0005 1000
1.0 0.1 0.0005 3000
1.0 0.1 0.0005 5000
1.0 0.1 0.0005 10000
1.0 0.1 0.0005 30000
1.0 0.1 0.0005 50000
1.0 0.1 0.0005 500000
$ cat scripts/basic_param_set | scripts/run-once-using-read.sh
$ ls
answers      gnuplot          infect.gplt      scripts
bin          hello-function.sh logfile          source
Desktop     hello.sh         infect-50.dat   stdout-50
examples    infect.py        run-infect.sh   treasure.txt
```

In the `scripts` directory there is a file called `basic_param_set` that contains a number of parameter sets. We can use the `cat` command to display the contents of this file. In fact, if we use the `cat` command on this file, the *output* of the `cat` command will be a list of parameter sets...

...and our `run-once-using-read.sh` shell script will accept a complete parameter set as its *input*, so...

...if we connect the *output* of the `cat` command to the *input* of our shell script – by, say, using a pipe – maybe that will give us what we want? Let's try it!

Well, it almost does!, i.e. it does it for the first parameter set, but none of the others. If we try running it again and again it will still only do it for the first parameter set in the file, so we're not quite there, but close. What we want is some way of telling the script to keep reading until there is no more stuff to read.

In fact, what we want is for the script to do some sort of *loop*: reading in a set of values, then running the `infect.py` program, then reading in the next set of values, and so on. How can we get it to do that? Before we look at that, we need to understand something else first...

Exit Status (1)

- Every program (or shell builtin command) returns an *exit status* when it completes
- Number between 0 and 255
- **Not** the same as the program's (or shell builtin command's) output
- By convention:
 - 0 means the command succeeded
 - Non-zero value means the command failed
- Exit status of the last command ran stored in special shell parameter named `?`

The exit status of a program is also called its *exit code*, *return code*, *return status*, *error code*, *error status*, *errorlevel* or *error level*.

Exit Status (2)

```
$ ls
```

```
answers  gnuplot          infect.gplt      scripts
bin      hello-function.sh logfile          source
Desktop  hello.sh         infect-50.dat   stdout-50
examples infect.py         run-infect.sh   treasure.txt
```

```
$ echo "${?}"
```

```
0
```

```
$ ls zzzzfired
```

```
/bin/ls: zzzzfired: No such file or directory
```

```
$ echo "${?}"
```

```
2
```

You get the value of the special parameter `?` by using the construct `${?}`, as in the above example.

Note that when the `ls` command is successful, its exit status is 0. When, however, it fails (for example because the file does not exist, as here), its exit status is non-zero ("2", in this case). In our shell scripts, we will make significant use of the fact that a non-zero exit status of a program (or a shell builtin command) means that there was an error.

Please note that the output of the `ls` command may not exactly match what is shown on this slide – in particular, the colours may be slightly different shades and there may be additional files and/or directories shown (and/or – if you've recently cleaned up your home directory – you may not have all of the files shown here).

true, false

`true` do nothing, *successfully*

```
$ true
```

```
$ echo "${?}"
```

```
0
```

`false` do nothing, *unsuccessfully*

```
$ false
```

```
$ echo "${?}"
```

```
1
```

It's worth introducing a couple of commands at this point which do nothing. (No, really.)

true does nothing and always *succeeds*, i.e. its exit status of 0.

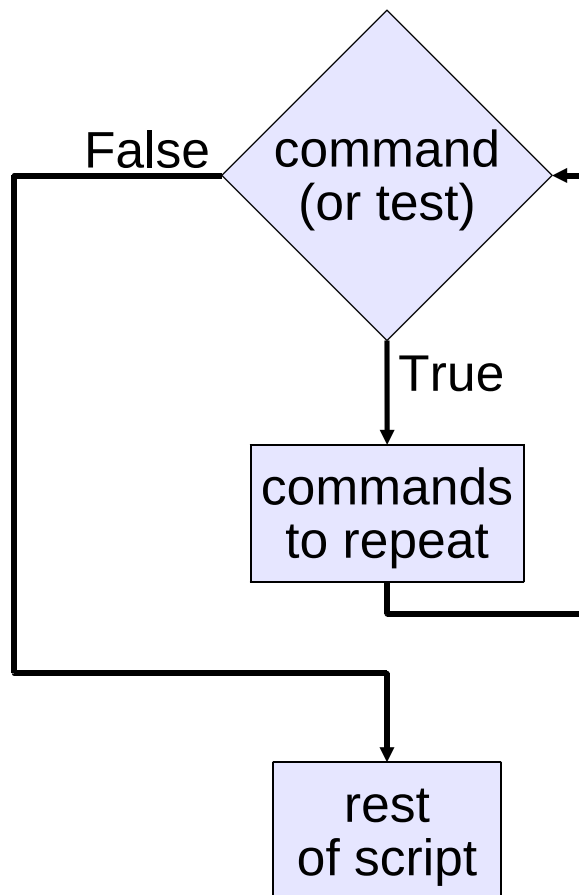
false does nothing and always *fails*, i.e. its exit status is non-zero.

You may be wondering what possible use there could be for such commands. The most obvious use is for debugging: suppose you have a script that runs a program that take a long time, and you want to test the script to make sure it works. You could replace the program that takes a long time with **true** to see what your script does if it thinks the program has succeeded. Similarly, you could replace the program your script is calling with **false** if you want to see what your script will do if it thinks the program has failed.

Another use for **true** is when you want the shell to do nothing (this is known as a *NOP* or *no-op* command): for instance, shell functions and **for** loops must contain at least one command. If, for some reason, you want a shell function or a **for** loop that does nothing (maybe because you haven't gotten around to writing it yet but you want to be able to test the rest of your script) you can use **true**. Then the shell won't complain about the definition of your function or the syntax of your **for** loop being incorrect, but they won't actually do anything.

while loop

Repeat some commands *while* some command (or test) is *true*



Now that we know about the exit status of a command we are ready to meet the loop structure alluded to earlier:

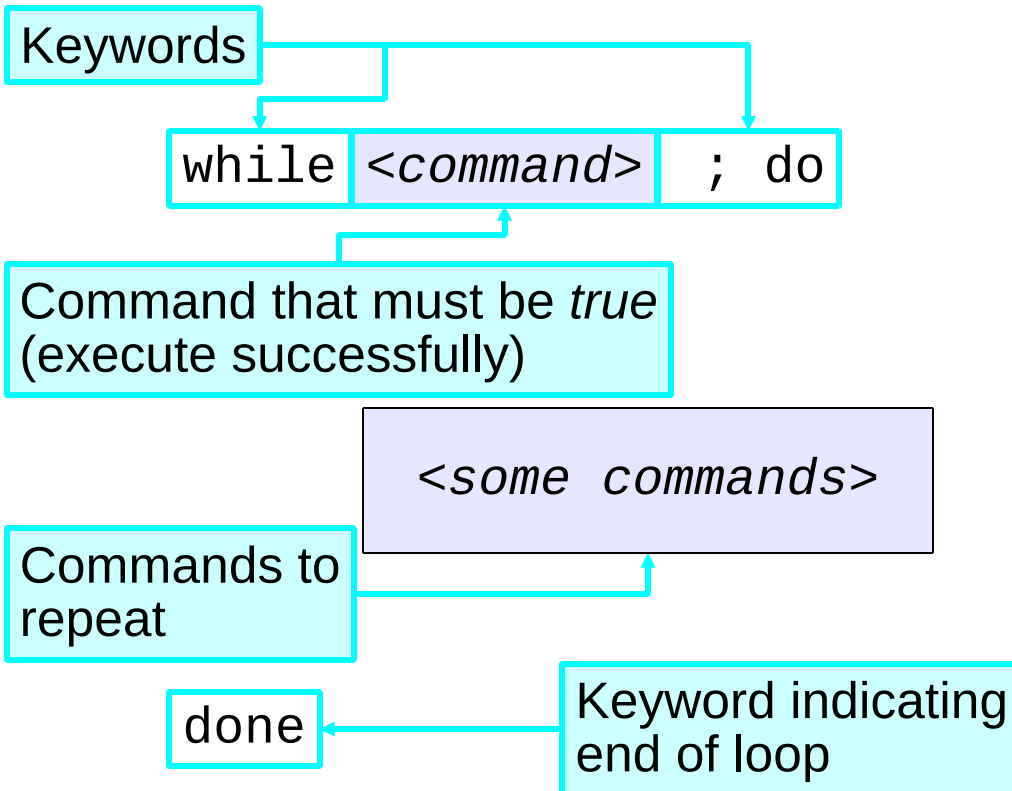
We can repeat a collection of one or more commands using a **while** loop. A **while** loop repeats a collection of commands as long as the result of some command is true. The result of a command is considered to be true if it returns an exit status of 0 (i.e. if the command succeeded). (The command we use in a **while** loop could also be a *test* of whether some expression is true. We'll see how to do that shortly.)

Note that even if `set -e` is in effect, or the first line of our shell script is

```
#!/bin/bash -e
```

the shell script will not exit if the result of the command the **while** loop depends on gives a non-zero exit status, since if it did, this would make **while** loops unusable(!).

while



scientific-computing@ucs.cam.ac.uk

Simple Shell Scripting for Scientists: Day Two

52

We use a **while** loop like this:

```
while <command> ; do  
    <some commands>  
done
```

where **<command>** is a command (which could be a test; more on tests later), and **<some commands>** is a collection of one or more commands. Note that if **<command>** is false the shell script will *not* exit, even if `set -e` is in effect or the first line of the shell script is

```
#!/bin/bash -e
```

As with a **for** loop, you can put the **do** on a separate line, in which case you can omit the semi-colon (;).

There are some examples of how to use **while** loops in the following files in the `examples` directory:

```
while1.sh
```

```
while2.sh
```

...but don't look at those files just yet as we need to meet a few more things first...

while

Repeat *while* some command is *true*

```
while <command> ; do  
    <some commands>  
done
```

To recap: we can repeat a collection of commands using a **while** loop. A **while** loop repeats a collection of commands as long as the result of some command is true. The result of a command is considered to be true if it returns an exit status of 0 (i.e. if the command succeeded). (The command we use in a **while** loop could also be a *test* of whether some expression is true. We'll see how to do that shortly.) We use a **while** loop like this:

```
while <command> ; do  
    <some commands>  
done
```

where **<command>** is a command (which could be a test), and **<some commands>** is a collection of one or more commands. Note that even if `set -e` is in effect, or the first line of the shell script is `#!/bin/bash -e`, the shell script will *not* exit if the result of **<command>** is not true.

As with a **for** loop, you can put the **do** on a separate line, in which case you can omit the semi-colon (;).

There are some examples of how to use **while** loops in the following files in the `examples` directory:

`while1.sh`

`while2.sh`

...but don't look at those files just yet as we need to meet a few more things first...

Using `while` (1)

```
$ cd
$ cp -p scripts/run-once-using-read.sh scripts/run-while-read.sh
$ gedit scripts/run-while-read.sh &
#!/bin/bash

# Read in parameters from standard input
#   and then run infect.py with them
#   and run it again and again until there are no more
while read myI myR myB mySIZE myJUNK ; do
    ...
echo "Standard output: stdout-${mySIZE}" >> logfile
done
```

Create a copy of the `run-once-using-read.sh` shell script in the `scripts` directory called `run-while-read.sh`. Open this up with your favourite editor (or `gedit`) and modify it as shown above.

Basically, replace the line:

```
read -p "Input parameters for infect.py: " myI myR myB mySIZE myJUNK
```

with:

```
#   and then run infect.py with them
#   and run it again and again until there are no more
while read myI myR myB mySIZE myJUNK ; do
```

And at the very end of the file add the following line:

```
done
```

Remember to save the script when you've finished.

Now let's try this script out and see if it does what we want:

```
$ cd
$ rm -f *.dat stdout-* logfile
$ cat scripts/basic_param_set | scripts/run-while-read.sh
$ ls
```

Second exercise

Make a copy of `multi-run.sh` and make it read all the arguments for `infect.py` in from standard input using a **while** loop:

```
$ cd
$ cp -p scripts/multi-run.sh scripts/multi-run-while.sh
```

```
#!/bin/bash
set -e

...

# Run program once for each argument
# Note: *no* quotes around ${myFIXED_PARAMS}
#       or they'll be interpreted as one argument!
for zzARGS in "${@}"; do
    run_program ${myFIXED_PARAMS} "${zzARGS}"
done

...
```

The `multi-run.sh` shell script is in the `scripts` directory of your home directory. Make a copy of it called `multi-run-while.sh`, also in the `scripts` directory, and work on that. Your task is to get `multi-run-while.sh` to **read** in all the arguments for `infect.py` from standard input (all its arguments, not just the fourth one) using a **while** loop.

Start by deleting the following two lines:

```
# Parameters that stay the same each run
myFIXED_PARAMS="1.0 0.1 0.0005"
```

...and you should also get rid of any other references to the shell variable `myFIXED_PARAMS` – you won't be using it in this script.

We have gone through everything you need to do this exercise. You should comment the modifications you make to your shell script, preferably as you are writing it.

And when you finish this exercise, please do take a short break before we start again with the solution. (And that's "break" as in "break from the computer" not "break to check my e-mail".)

You can check that you've done it correctly by trying to run your `multi-run-while.sh` script (*remember to save it after you've made your modifications!*):

```
$ cd
$ rm -f *.dat stdout-* logfile
$ ls
$ cat scripts/basic_param_set | scripts/multi-run-while.sh
$ ls
```

Hint: Try copying the `run-while-read.sh` script...

Recap: standard input/while loops

- Command substitution $\$(command)$ can be used to get the output of a command into a shell variable
- Use **mktemp** to make temporary files and directories
- **read** gets values from standard input
- Pipes connect one command's output to another's input
- The command **true** does nothing but is considered to be true (its exit status is 0); the command **false** does nothing but is not considered to be true (non-zero exit status).
- **while** loops repeat some commands while something is true – can be used to read in multiple lines of input with **read**

Note that **while** loops can contain other **while** loops, and they can also contain **for** loops (or both). Similarly, **for** loops can contain **while** loops or other **for** loops (or both).

Tests

Test to see if something is true:

```
[ <expression> ]
```

or: `test <expression>`

where `<expression>` can be any of a number of things such as:

```
[ "a" -eq "b" ]
```

```
[ "a" -le "b" ]
```

```
[ "a" -gt "b" ]
```

A test is basically the way in which the shell evaluates an expression to see if it is true. (Recall that they I said they can be used with **while**; we'll see how in a minute.) There are many different tests that you can do, and we only list a few here:

`"a" -lt "b"` true if and only if the integer **a** is less than the integer **b**

`"a" -le "b"` true if and only if the integer **a** is less than or equal to the integer **b**

`"a" -eq "b"` true if and only if the integer **a** is equal to the integer **b**

`"a" -ne "b"` true if and only if the integer **a** is not equal to the integer **b**

`"a" -ge "b"` true if and only if the integer **a** is greater than or equal to the integer **b**

`"a" -gt "b"` true if and only if the integer **a** is greater than the integer **b**

You can often omit the quotation marks, particularly for arithmetic tests (we'll meet other sorts of tests on the next day of this course), but it is good practice to get into the habit of using them, since there are times when *not* using them can be disastrous.

In the above tests, **a** and **b** can be any integers. Recall that shell variables can hold pretty much any value we like – they can certainly hold integer values, so **a** and/or **b** in the above expressions could come from shell variables, e.g.

```
[ "${VAR}" -eq "5" ]
```

Or, equivalently:

```
test "${VAR}" -eq "5"
```

is true if and only if the shell variable **VAR** contains the value "5".

Note that you *must* have a space between the square brackets [] (or the word **test** if you are using that form) and the expression you are testing – if you do not then the shell will not realise that you are trying to do a test.

Arithmetic Expansion: `$(())`

- Returns the value of an *integer* arithmetic operation
- Operands *must* be integers (so *no* decimals, e.g. 2.5, etc)
- Do ***not*** use quotes within the arithmetic expression

```
$( ( <arithmetic-expression> ))
```

Example:

```
$( ( ${VAR} + 56 ))
```

The shell can also do (primitive) integer arithmetic, which can be very useful, as we will see in a minute.

The construct `$((<arithmetic-expression>))` means replace `$((<arithmetic-expression>))` with the result of the *integer* arithmetic expression `<arithmetic-expression>`. This is known as *arithmetic expansion*. (The arithmetic expression is evaluated as integer arithmetic.)

Note that we ***don't*** use quotes around variables within our arithmetic expression as that would cause the shell to treat the values as strings rather than numbers. (This is, alas, somewhat inconsistent with the shell's behaviour elsewhere, because the syntax used for arithmetic expansion is actually a completely different language to everything else we've met in bash.) We ***can*** put quotes around the entire arithmetic expansion construct, though.

while loops that count

Consider the following **while** loop:

```
zzCOUNT="1"
while [ "${zzCOUNT}" -le "6" ] ; do
    echo "${zzCOUNT}"
    zzCOUNT=$(( ${zzCOUNT} + 1 ))
done
```

When we put together arithmetic tests, **while** loops and arithmetic expansion, we can construct a **while** loop that counts for us, as in the above example. Can you figure out what the above loop will do?

When you think you know, try running the script **while2.sh** in the `examples` directory of your home directory. That will show you the output of the above **while** loop, immediately followed by the output of a very similar **while** loop where **zzCOUNT** starts off with the value 0 rather than 1.

Note that **while** loops can (and often do) contain other **while** loops (or **for** loops). We say that one loop is *nested* inside the other one.

Using `while` (2)

```
$ cd
$ cat scripts/generate-params.sh

#!/bin/bash

myINFECTS="1.0 1.5 2.0 2.5 3.0"
myR="0.1"
myB="0.0005"

for zzINFS in ${myINFECTS} ; do
    zzSIZE="50"
    while [ "${zzSIZE}" -le "50000" ] ; do
        echo "${zzINFS} ${myR} ${myB} ${zzSIZE}" >> new_param_set
        zzSIZE=$(( ${zzSIZE} * 10 ))
    done
done

$ scripts/generate-params.sh
$ more new_param_set
```

Examine the file called `generate-params.sh` in the `scripts` directory of your home directory (shown above).

Then try it out and see what it does.

Generalising multi-run-while.sh

```
#!/bin/bash
set -e

...

"${myPROG}" "${@"} > "stdout-${1}-${2}-${3}-${4}"

...

mv infect.dat "infect-${1}-${2}-${3}-${4}.dat"

...

echo "Output file: infect-${1}-${2}-${3}-${4}.dat" >> "${myLOGFILE}"
echo "Standard output: stdout-${1}-${2}-${3}-${4}" >> "${myLOGFILE}"
```

Modify the multi-run-while.sh script in the scripts directory as shown above.

(Remember to save it when you've finished.)

Basically we are replacing all the instances of the string “\${4}” with the string “\${1} - \${2} - \${3} - \${4}”. This means that now, instead of our output files being based on the fourth argument that is passed to **infect.py**, they are based on all the parameters in the parameter set. This is clearly necessary as we start to experiment with varying parameters other than just the fourth one.

And we finish with an exercise.

If you want to do the exercise outside of class, the files you'll need can be found at:

<http://www-uxsup.csx.cam.ac.uk/courses/ShellScriptingSci/exercises/day-two.html>

Final exercise – Part One

Improve the `run_program` function in `multi-run-while.sh` so that as well as running `infect.py` it also runs **gnuplot** (using the `infect.gplt` file) to plot a graph of the output.

This exercise should be fairly straightforward. One sensible way of approaching it would be as follows:

1. Figure out the full path of the `infect.gplt` file. Store it in a shell variable (maybe called something like `myGPLT_FILE`).
2. Immediately after running `infect.py`, run **gnuplot**:

```
gnuplot "${myGPLT_FILE}"
```
3. Rename the `infect.png` file produced by **gnuplot** along the same lines as the `infect.dat` file produced by `infect.py` is renamed.

Make sure you test the script after you've modified it and check that it does what you would expect.

This exercise highlights one of the advantages of using functions: we can improve or change our functions whilst leaving the rest of the script unchanged. In particular, the *structure* of the script remains unchanged. This means two things: (1) if there are any errors after changing the script they are almost certainly in the function we changed, and (2) the script is still doing the same *kind* of thing (as we can see at a glance) – we've just changed the particulars of one of its functions.

Final exercise – Part Two

Now create a new shell script based on `multi-run-while.sh` that will run `infect.py` three times for *each parameter set* the script reads in on standard input, *changing* the fourth parameter each time as follows:

For a given parameter set `a b c d`, first your script should run `infect.py` with the parameter set:

`a b c 50`

...then with the parameter set:

`a b c 500`

...and then with the parameter set:

`a b c 5000`

An example may help to make this task clearer. Suppose your script **reads** in the parameter set:

`1.0 0.1 0.0005 50`

...it should then run the `infect.py` program 3 times, once for each of the following parameter sets:

`1.0 0.1 0.0005 50`

`1.0 0.1 0.0005 500`

`1.0 0.1 0.0005 5000`

The first thing to do is to make a copy of the `multi-run-while.sh` script and work on the copy – I suggest you call your copy something like `multi-50-500-5000.sh`:

```
$ cd
```

```
$ cp -p scripts/multi-run-while.sh scripts/multi-50-500-5000.sh
```

Now, currently the script will **read** in a parameter set and then call the `run_program` function to process that parameter set. Clearly, instead of passing all four parameters that the script reads in, your new script will now only be passing the first (**myI**), second (**myR**), and third (**myB**) parameters that it has **read** in. However, the `infect.py` program *requires* 4 parameters (and it cares about the order in which you give them to it), so your script still needs to give it 4 parameters, it is just going to ignore the fourth parameter it has read (**mySIZE**) and substitute values of its own instead.

There are two approaches you could take. One would be to call the `run_program` function 3 times, once with 50 as the fourth parameter, once with 500 as the fourth parameter and once with 5000 as the fourth parameter. The other would be to use some sort of loop that calls the `run_program` function, using the appropriate value (50, 500 or 5000) for the fourth parameter on each pass of the loop. **I want you to use the loop approach.**

Hint: Use a **for** loop.

Final exercise – Part Three

Now create a new shell script, based on the script you created in the previous part of the exercise, that does the following:

Instead of running **infect.py** three times for each parameter set it **reads** in, this script should accept a set of values on the command line, and use those instead of the hard-coded 50, 500, 5000 previously used.

Thus, for each parameter set it reads in on standard input, it should run **infect.py** substituting, in turn, the values from the command line **for** the fourth parameter in the parameter set it has **read** in.

So, if the script from the previous part of the exercise was called `multi-50-500-5000.sh`, and we called this new script `multi-sizes.sh` (and stored both in the `scripts` directory of our home directory), then running the new script like this:

```
$ cat ~/scripts/param_set | ~/scripts/multi-sizes.sh 50 500 5000
```

should produce **exactly** the same output as running the old script with the same input file:

```
$ cat ~/scripts/param_set | ~/scripts/multi-50-500-5000.sh
```

The first thing to do is to make a copy of the previous script (which I suggested you call `multi-50-500-5000.sh`) and work on the copy – I suggest you call your copy something like `multi-sizes.sh`:

```
$ cd
```

```
$ cp -p scripts/multi-50-500-5000.sh scripts/multi-sizes.sh
```

You may be wondering what the point of the previous script and this script are. Consider what these scripts actually do: they take a parameter set, vary one of its parameters and then run some program with the modified parameter sets. Why would we want to do this?

Well, in this example the parameter we are varying specifies the size of the population which our program will model. You can easily imagine that we might have a simulation or calculation for which, for any given parameter set, interesting things happened in various population sizes. These scripts allow us to take each parameter set and run it several times for different sizes of populations. We can then look at each parameter set and see how varying the size of the population affects the program's output for that parameter set.

If we were using the parameter sets in the `scripts/param_set` file, we might notice that these parameters are the same except for the first parameter which varies. So if we pipe those parameter sets into one of these scripts, we are now investigating how the output of the **infect.py** program varies as we vary *two* of its input parameters, which is kinda neat, doncha think? 😊

Hint: Modify the loop you used in the previous script to loop over all the command line arguments rather than some hard coded values. If you don't remember the construct that gives you all the command line arguments have a look at the recap of the previous day of this course.

Final exercise – Files

All the files (scripts, **infect.py** program, etc) used in this course are available on-line at:

<http://www-uxsup.csx.cam.ac.uk/courses/ShellScriptingSci/exercises/day-two.html>

We'll be looking at the answers to this exercise on the next day of this course, so *please make sure you have attempted this exercise **before** you come to the next day of this course.*