UNIVERSITY OF CAMBRIDGE

Simple Shell Scripting for Scientists

Day One

David McBride
Ben Harris
University of Cambridge Information Services

IMPORTANT:

If you are doing the “Simple Shell Scripting for Scientists” course in one of our scheduled classes (as opposed to reading these course notes on-line, for example), then you should have received a user ID to use for the course: it will probably be something like **y*XXX*** (where ***XXX*** is a number).

Make sure you make a note of this user ID and bring it with you to ***all*** sessions of the course. Also, it is a very good idea to bring your copy of earlier sessions’ course notes with you to later sessions as we do not guarantee to keep spare copies of earlier sessions’ course notes on hand should you need to refer to them.

Introduction

- Who:
  - David McBride, Unix Support, UIS
  - Ben Harris, Unix Support, UIS
  - Bruce Beckles, e-Science Specialist, UIS
- What:
  - Simple Shell Scripting for Scientists course, *Day One*
  - Part of the ***Scientific Computing*** series of courses
- Contact (questions, etc):
  - scientific-computing@uis.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 University Information Services, 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 *any* of the next two days of the course then ***please make sure that you fill in the Course Review form online***, accessible under "feedback" on the main MCS Linux menu, or via:

http://feedback.training.cam.ac.uk/uis/

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 3: Introduction for Absolute Beginners"
  - "Python 3: Introduction for Those with Programming Experience"

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:

https://www.gnu.org/software/bash/

We will be using bash 4.4 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 5.0, which was released in January 2019.

For details of the "Python 3: Introduction for Absolute Beginners" course, see:

https://www.training.cam.ac.uk/ucs/course/ucs-python

For details of the "Python 3: Introduction for Those with Programming Experience" course, see:

https://www.training.cam.ac.uk/ucs/course/ucs-python4progs

Outline of Course

1. Prerequistes & recap of Unix commands

SHORT BREAK

2. Very simple shell scripts

SHORT BREAK

3. More useful shell scripts:
   - Variables (and parameters)
   - Simple command-line processing
   - Output redirection
   - Loop constructs: **`for`**

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 *any* of the next two days of the course then ***please make sure that you fill in the Course Review form online***, accessible under "feedback" on the main MCS Linux menu, or via:

http://feedback.training.cam.ac.uk/uis/

Pre-requisites

- Ability to use a text editor under Unix/Linux:
  - Try gedit if you aren't familiar with any other Unix/Linux text editors
- Familiarity with the Unix/Linux command line ("Unix: Introduction to the Command Line Interface" course):
  - Common Unix/Linux commands (`ls`, `rm`, etc)
  - Piping; redirecting input and output
  - Simple use of environment variables
  - File name globbing ("pathname expansion")

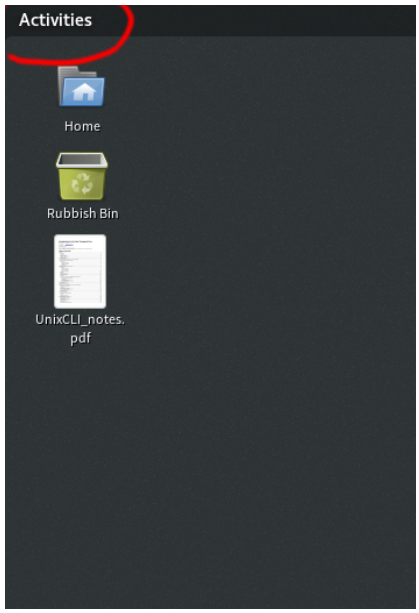
For details of the "Unix: Introduction to the Command Line Interface" course, see:

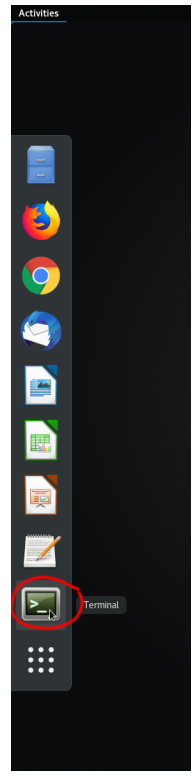
https://www.training.cam.ac.uk/ucs/course/ucs-unixintro1

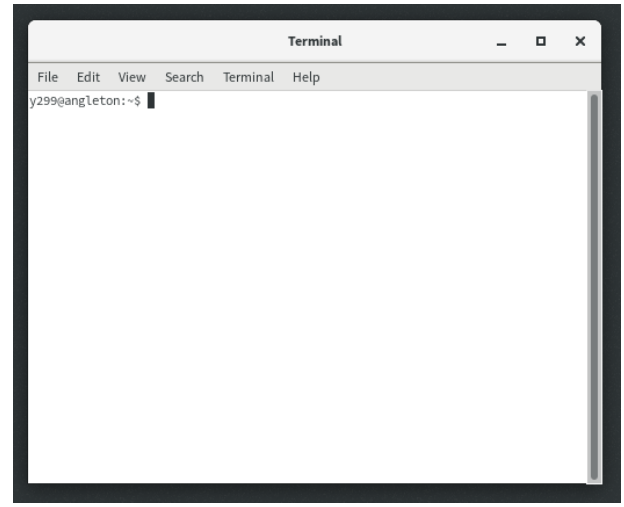
The notes from this course are available on-line at:

https://help.uis.cam.ac.uk/help-support/training/downloads/course-files/programming-student-files/unix-cli

Start a shell







As this is a shell scripting course, we are going to need to interact with the Unix shell.

To start a shell, click on “Activities” in the top-left corner of the screen, then click on the “Terminal” icon in the desktop application bar.

A Terminal window will then appear.

Unix commands (1)

`cat` Display contents of a file

```
$ cat welcome
```

```
Welcome to the Simple Shell Scripting for Scientists course!

This course is intended to be run on MCS Linux 2012/2013.
```

`cd` ***c***hange ***d***irectory

```
$ cd /tmp
$ cd
```

`chmod` ***ch***ange the ***mod***e (permissions) of a file or directory

```
$ chmod a+r treasure.txt
```


If you give the **cd** command without specifying a directory then it will change the directory to your *home directory* (the location of this directory is specified in the **HOME** *environment variable* – more on environment variables later).

The **chmod** command changes the permissions of a file or directory (in this context, the jargon word for "permissions" is "mode"). For instance, the above example gives read access to the file `treasure.txt` for all users on the system. Note that the **chmod** command has little effect on the MCS Linux systems used in this course. Unix permissions were covered in the "Unix: Introduction to the Command Line Interface" course, see:

https://www.training.cam.ac.uk/ucs/course/ucs-unixintro1

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

https://help.uis.cam.ac.uk/help-support/training/downloads/course-files/programming-student-files/unix-cli

Unix commands (2)

`cp` ***c***o***p***y files and/or directories

```
$ cp welcome /tmp/welcome-copy
```

Options:

`-p` ***p***reserve (where possible) files' owner, permissions and date

`-f` if unable to overwrite destination file, delete it and try again, i.e. ***f***orcibly overwrite destination files

`-R` copy any directories ***R***ecursively, i.e. copy their contents

`-i` prompt before overwriting anything (be ***i***nteractive – ask the user)

```
$ cp –p welcome /tmp/welcome-copy
```


Note that the **`cp`** command has many other options than the four listed above, but those are the options that will be most useful to us in this course.

Unix commands (3)

`date` display/set system ***date*** and time

```
$ date
Tue Oct 30 12:21:57 GMT 2012
```

`echo` display text

```
$ echo "Hello"
Hello
```

`env` With no arguments, display ***env***ironment variables (example later)

Please note that if you try out the **`date`** command, you will get a different date and time to that shown on this slide (unless your computer's clock is wrong!). Also, note that usually only the system administrator can use **`date`** to set the system date and time.

Note that the **`echo`** command has a few useful options, but we won't be making use of them today, so they aren't listed.

Note also that the **`env`** command is a very powerful command, but we will not have occasion to use it for anything other than displaying *environment variables* (see later), so we don't discuss its other uses.

Unix commands (4)

`grep` find lines in a file that match a given pattern

```
$ grep 'MCS' welcome
This course is intended to be run on MCS Linux 2012/2013.
```

Options:

`-i` search case ***i***nsensitively

`-w` only match *whole* ***w***ords, not parts of words

`--color=always` ***always*** display the matching text in ***colo***u***r***

```
$ grep 'mcs' welcome
$ grep -i 'mcs' welcome
This course is intended to be run on MCS Linux 2012/2013.
```


The patterns that the **`grep`** command uses to find text in files are called *regular expressions*. We won't be covering these in this course, but if you are interested, or if you need to find particular pieces of text amongst a collection of text, then you may wish to attend the CS "Programming Concepts: Pattern Matching Using Regular Expressions" course, details of which are given here:

https://www.training.cam.ac.uk/ucs/course/ucs-regex

Note that the **`grep`** command has many, many other options than the two listed above, but we won't be using them in this course.

Unix commands (5)

`ln` create a ***l****i****n****k** between files (almost always used with the `-s` option for creating ***s****ymbolic links)

Options:

`-f` ***f****orcibly remove destination files (if they exist)

`-i` prompt before removing anything (be ***i****nteractive – ask the user)

`-s` make ***s****ymbolic links rather than a hard links

```
$ ln –s ~/welcome /tmp/welcome-link
$ cat welcome
Welcome to the Simple Shell Scripting for Scientists course!

This course is intended to be run on MCS Linux 2012/2013.

$ cat /tmp/welcome-link
Welcome to the Simple Shell Scripting for Scientists course!

This course is intended to be run on MCS Linux 2012/2013.
```


The **`ln`** command creates links between files. (Note that it has other options besides those listed above, but we won't be using them in this course.) In the example above, we create a symbolic link to the file `motd` in `/etc` and then use **`cat`** to display both the original file and the symbolic link we've created. We see that they are identical.

There are two sort of links: *symbolic links* (also called *soft links* or *symlinks*) and *hard links*. A symbolic link is similar to a shortcut in the Microsoft Windows operating system (if you are familiar with those) – essentially, a symbolic link points to another file elsewhere on the system. When you try and access the contents of a symbolic link, you actually get the contents of the file to which that symbolic link points. Whereas a symbolic link points to another *file* on the system, a hard link points to *actual data* held on the filesystem. These days almost no one uses **`ln`** to create hard links, and on many systems this can only be done by the system administrator. If you want a more detailed explanation of symbolic links and hard links, see the following Wikipedia articles:

https://en.wikipedia.org/wiki/Symbolic_link

https://en.wikipedia.org/wiki/Hard_link

Unix commands (6)

`ls` ***l****i****s****t the contents of a directory*

```
$ ls
answers  Desktop   gnuplot   zombie.py  source
bin      examples  hello.sh  scripts    treasure.txt
```

Options:

- `-d` List ***d****irectory name instead of its contents*
- `-l` use a ***l****ong listing that gives lots of information about each directory entry*
- `-R` list subdirectories ***R****ecursively, i.e. list their contents and the contents of any subdirectories within them, etc*

If you try out the **`ls`** command, please note that its output 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.

Note also that the **`ls`** command has many, many more options than the three given on this slide, but these three are the options that will be of most use to us in this course.

Unix commands (7)

`less` Display a file one screenful of text at a time

`more` Display a file one screenful of text at a time

```
$ more treasure.txt
The Project Gutenberg EBook of Treasure Island, by Robert Louis Stevenson

This eBook is for the use of anyone anywhere at no cost and with
almost no restrictions whatsoever.  You may copy it, give it away or
re-use it under the terms of the Project Gutenberg License included
with this eBook or online at www.gutenberg.org


Title: Treasure Island

Author: Robert Louis Stevenson

Release Date: February 25, 2006 [EBook #120]

Language: English

Character set encoding: ASCII

*** START OF THIS PROJECT GUTENBERG EBOOK TREASURE ISLAND ***



--More--(0%)
```


(Note that the output of the `more` command may not exactly match that shown on this slide – in particular, the number of lines displayed before the "`--More--(0%)`" message depends on the number of lines it takes to fill up the window in which you are running the `more` command.)

The **`more`** and **`less`** commands basically do the same thing: display a file one screenful of text at a time. Indeed, on some Linux systems the `more` command is actually just another name (an *alias*) for the `less` command.

Why are there two commands that do the same thing? On the original Unix systems, the **`less`** command didn't exist – the command to display a file one screenful of text at a time was **`more`**. However, the original **`more`** command was somewhat limited, so someone wrote a better version and called it **`less`**. These days the **`more`** command is a bit more sophisticated, although the **`less`** command is still much more powerful.

For everyday usage though, many users find the two commands are equivalent. Use whichever one you feel most comfortable with, but remember that every Unix/Linux system should have the **`more`** command, whereas some (especially older Unix systems) may not have the **`less`** command.

Unix commands (8)

`man` Display the on-line reference ***man***ual for a command

`$ man bash`

```
BASH(1)                                                    BASH(1)

NAME
       bash - GNU Bourne-Again SHell

SYNOPSIS
       bash [options] [file]

COPYRIGHT
       Bash is Copyright (C) 1989-2011 by the Free Software Foundation, Inc.

DESCRIPTION
       Bash  is  an  sh-compatible  command language interpreter that executes
       commands read from the standard input or from a file.  Bash also incor␣
       porates useful features from the Korn and C shells (ksh and csh).

       Bash  is  intended  to  be a conformant implementation of the Shell and
       Utilities portion  of  the  IEEE  POSIX  specification  (IEEE  Standard
       1003.1).  Bash can be configured to be POSIX-conformant by default.

OPTIONS
       All  of  the  single-character shell options documented in the descrip␣
       tion of the set builtin command can be used as options

 Manual page bash(1) line 1 (press h for help or q to quit)
```


(Note that the output of the `man` command may not exactly match that shown on this slide – in particular, the number of lines displayed before the "`Manual page bash(1) line 1`" message depends on the number of lines it takes to fill up the window in which you are running the `man` command.)

The **`man`** command displays the on-line reference manual for a command. Such manuals are called "man pages". Whilst not all commands have man pages, many do, and, in particular, most of the Unix commands we use in this course do.

The **`man`** command has the functionality of the **`more`** command built into it so that it can display the man page one screenful of text at a time. To advance a screen, press the space bar. To go *b*ack a screen type "b", and to *q*uit **`man`** press the "Q" key.

Unix commands (9)

`mkdir` ***m****a****k****e*** ***dir****ectories*

```
$ mkdir /tmp/mydir
```

Options:

`-p` make any ***p****arent* directories as required; also if directory already exists, don't consider this an error

```
$ mkdir /tmp/mydir
mkdir: cannot create directory `/tmp/mydir': File exists
$ mkdir –p /tmp/mydir
$
```


Note that the **`mkdir`** command has other options, but we won't be using them in this course.

Unix commands (10)

`mv` ***m*****o*****v*****e or rename files and directories

```
$ mv /tmp/welcome-copy /tmp/junk
```

Options:

`-f` do not prompt before overwriting files or directories, i.e. ***f***orcibly move or rename the file or directory; this is the default behaviour

`-i` prompt before overwriting files or directories (be ***i***nteractive – ask the user)

`-v` show what is being done (be ***v***erbose)

Note that the **`mv`** command has other options, but we won't be using them in this course. Note also that if you move a file or directory between different filesystems, **`mv`** actually copies the file or directory to the other filesystem and then deletes the original.

Unix commands (11)

pwd ***p***rint full path of current ***w***orking ***d***irectory

```
$ cd /tmp
$ pwd
/tmp
```

Options:

`-P` print the full ***P***hysical path of the current working directory (i.e. the path printed will not contain any symbolic links)

Note that the **pwd** command has another option, but we won't be using it in this course.

Unix commands (12)

`rm` ***r****e****m****ove files or directories

```
$ rm /tmp/junk
```

Options:

`-f`	ignore non-existent files and do not ever prompt before removing files or directories, i.e. ***f****orcibly remove the file or directory*
`-i`	prompt before removing files or directories (be ***i****nteractive – ask the user)*
`--preserve-root`	do not act recursively on `/`
`-R`	remove subdirectories (if any) ***R****ecursively, i.e. remove subdirectories and their contents*
`-v`	show what is being done (be ***v****erbose)*

Note that the **`rm`** command has other options, but we won't be using them in this course.

Unix commands (13)

`rmdir` ***r**e**m**ove* *empty* ***dir**ectories*

```
$ rmdir /tmp/mydir
```

`touch` change the timestamp of a file; if the file doesn't exist create it with the specified timestamp (the default timestamp is the current date and time)

```
$ touch /tmp/nowfile
```


The **`rmdir`** and **`touch`** commands have various options but we won't be using them on this course. If you try out the **`touch`** command with the example above, check that it has really worked the way we've described here by using the **`ls`** command as follows:

```
ls -l /tmp/nowfile
```

You should see that the file `nowfile` has a timestamp of the current time and date.

First exercise

Write a simple shell script that:

1) **Removes** the subdirectory `play` in your home directory (whether or not it exists), then
2) **Makes** a *new* subdirectory called `play` in your home directory, then
3) **Copies** the *contents* of the `examples` subdirectory in your home directory to the newly created `play` subdirectory, and finally
4) Prints out the **current date and time**.

Your shell script should also **print out on the screen** what it is about to do before it actually does it.

 **10 minutes**

Previous people who have taken this course have tended to describe it as difficult. A close examination of their feedback has revealed that many of those who have difficulty haven't mastered the basic Unix commands that we assume you are already familiar with and/or aren't familiar enough with the Unix command line. So we now start the course by giving you an exercise that tests your knowledge of basic Unix commands – if you find this exercise difficult, then you aren't ready to do this course, I'm afraid.

If you do find this exercise difficult then I suggest that you ***leave*** now and try the course again when you've practiced your Unix a bit more – it really won't be a good use of your time, that of your fellow course attendees, and that of the course giver, for you to try the course at this point. Sorry, but this course really does require you to be on top of your basic Unix commands.

So, the exercise is to write a trivial shell script called `setup-play.sh` that does what it says on the slide above. It's a trivial shell script so it doesn't need to do any error checking or anything fancy like that. **Note that the first time you run this script the `play` subdirectory won't exist (unless you've created it manually yourself), but your script should still try to remove it.** Your shell script should print out on the screen what it is going to do before it actually does it.

Everything you need to do this exercise was covered on the "Unix: Introduction to the Command Line Interface" course, the course notes of which are available on-line as a PDF file:

https://help.uis.cam.ac.uk/help-support/training/downloads/course-files/programming-student-files/unix-cli/unix-cli-files/notes.pdf

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:

https://www.training.cam.ac.uk/ucs/course/ucs-unixintro1

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

https://www.ucs.cam.ac.uk/docs/course-notes/unix-courses/UnixCLI

Examining hello.sh

```
$ ls –l hello.sh
-rwx------  1 y250 y250 69 2006-11-06 11:35 hello.sh
$ cat hello.sh
#!/bin/bash

echo "Hello.  I am a shell script."
echo "Who are you?"
$ gedit hello.sh &
```


Remember that the **`ls`** command lists the files in a directory and that it can take options that modify its behaviour. **`ls -l <file>`** gives us a lot of information about the particular file **`<file>`**. In particular, it shows us the file's permissions (in this case: "`-rwx------`"), and we see that this file indeed has its execute bits set. Note that the exact text you see when you execute "**`ls -l hello.sh`**" on the computer in front of you may be slightly different – in particular, the owner ("`y250`") and group ("`y250`") of the file will be different.

Recall that **`cat <file>`** displays the contents of the file **`<file>`**.

`gedit <file>` starts the editor gedit and loads the file **`<file>`**. The "**`&`**" tells the shell to run gedit in the background, so that we go straight back to the shell prompt and can carry on doing other things rather than waiting until we quit gedit. Note that because we're running gedit in the background, after we quit gedit the shell will print a message saying "`Done`" (along with some other text) to indicate that the gedit program that was running in the background has finished.

You don't have to use gedit to edit the file, you can use whatever editor you are most comfortable with.

Remember that the **`echo`** command prints out the text that it has been given on standard output (normally the screen). It is a *shell builtin command*, i.e. a command that is implemented by the shell itself as opposed to an external program that the shell runs on your behalf. For example, the **`ls`** command is ***not*** a shell builtin command – it is an external program that the shell executes when you type "**`ls`**".

Errors in shell scripts (1)

Change:

```
echo "Hello.  I am a shell script."
```

to:

```
echoq "Hello.  I am a shell script."
```

```
$ ./hello.sh
./hello.sh: line 3: echoq: command not found
Who are you?
$
```

(Now change "echoq" back to "echo".)

Make sure you save the file before running it again, or the changes won't take effect.

As you can see, even if there is an error in the shell script, the shell script simply reports the error and merrily continues running. There are many different sorts of errors one can make in writing a shell script, and for most of them the shell will report the error but continue running. There **is** one type of error that will stop the execution of the shell script: a *syntax error* (see next slide).

Also note that the shell tells us what the error is – "`command not found`" (as there is no "`echoq`" command) – and the line on which it occurred (line 3). This makes it easier to track down the error and fix it.

You can force the shell script to quit when it encounters an error by using the `set` shell builtin command like this:

```
set -e
```

as we will see later.

Errors in shell scripts (2)

Change:

```
echo "Who are you?"
```

to:

```
(echo "Who are you?"
```

```
$ ./hello.sh
Hello.  I am a shell script.
./hello.sh: line 5: syntax error: unexpected end of file
$
```

(Now remove the extraneous open bracket "(".)

Make sure you save the file before running it again, or the changes won't take effect.

If there is a **syntax** error in the shell script, the shell script will abort once it encounters the error and *won't* run the rest of the script, because it doesn't understand what it should do.

Note that although the error is actually at line 4, it is not until line 5 that the shell decides something is wrong and tells us anything. **Get used to this behaviour!** – it is very annoying, as it makes debugging shell scripts painful, but that's just the way it is. When the shell tells you there is a syntax error at line *n*, you should take that to mean that there is a syntax error somewhere between the last command the script managed to execute and line *n* (inclusive).

Explicitly using bash to run the script

```
$ bash ./hello.sh
Hello.  I am a shell script.
Who are you?

$ bash -x ./hello.sh
+ echo 'Hello.  I am a shell script.'
Hello.  I am a shell script.
+ echo 'Who are you?'
Who are you?
$
```


There's another way we can run a shell script, which is by explicitly starting a new instance of the `bash` shell and telling it to execute the commands in the file containing the shell script. (Once the shell script has finished the instance of `bash` that was running it will silently exit, leaving us in our original shell.)

A very important point to note is that if we run the shell script this way, the magic `#!`… first line of the shell script is ***ignored***. (This behaviour is not as surprising as it might first seem, since that first line doesn't mean anything to `bash`, our shell – it is used by the *operating system* to work out what program it should use to run our shell script.)

(You may wonder why we would ever bother running a shell script like this: after all, the whole point of the magic `#!`… first line of the shell script is so that the operating system knows how to run the script without us having to explicitly tell it. Well, what if someone hasn't put that magic first line in? Or what if we have permission to read the file containing the shell script, but not to execute it? In either of these situations, explicitly telling `bash` to execute the commands in the file will solve the problem for us.)

One very nice consequence of running a shell script like this is that we can change how the shell script is run *without* modifying the file containing the script: we can use **`bash -x`** to run the shell script. Starting `bash` with the **`-x`** option causes it to print commands and their arguments as they are executed.

Changing how the shell script is run

Change:

```
#!/bin/bash
```

to:

```
#!/bin/bash -x
```

```
$ ./hello.sh
+ echo 'Hello.  I am a shell script.'
Hello.  I am a shell script.
+ echo 'Who are you?'
Who are you?
$
```

(Now remove the “ -x” you added.)

…and we can also modify the shell script itself so that the `bash` shell used to run it is started with the **`-x`** option by modifying the magic `#!`… first line of the script.

As we have just seen, starting the `bash` shell with the `-x` option makes it print commands and their arguments as they are executed.

There’s also another way we can get this behaviour, which we are about to meet.

`set -x`, `set +x`

Print commands and their arguments just before they are executed:

```
set -x
```

Don't print commands and their arguments just before they are executed (default):

```
set +x
```


We already know that if the first "magic" line of our shell script is:

```
#!/bin/bash -x
```

then the commands in our shell script and their arguments are printed out just before they are executed.

You can also get this behaviour by using the **`set`** shell builtin command like this:

```
set -x
```

You can return to the normal behaviour of just executing the command rather than first printing out the command and its arguments by using the **`set`** shell builtin command like this:

```
set +x
```

`set -x`, `set +x` example

Add the lines in red to `hello.sh` as shown:

```
set -x
echo "Hello.  I am a shell script."
set +x
echo "Who are you?"
```

```
$ ./hello.sh
+ echo 'Hello.  I am a shell script.'
Hello!  I am a shell script.
+ set +x
Who are you?
$
```

(Now remove the "`set -x`" and "`set +x`" lines.)

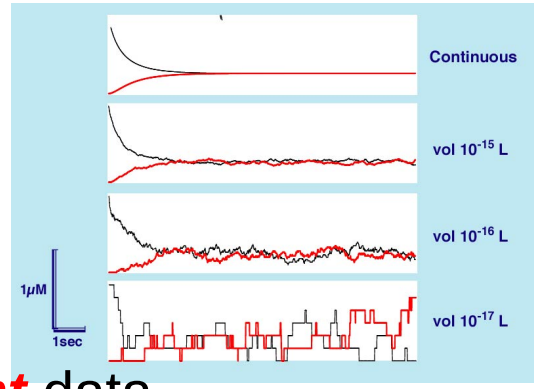
(Make sure you save the file before running it again, or the changes won't take effect.)

By using `set -x` and `set +x` we can turn off and on the "print the command and its arguments before executing it" behaviour. This can be extremely helpful in debugging, as it allows us to see what is happening in the particular part of the script we are interested in, rather than having to see a list of *all* the commands (and their arguments) that came before that part of the script as well.

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*.

Now, 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.

Automating repetitive tasks (1)

Imagine I'm working on a program. Every time I change it, I save it, then compile and run it. My editor makes a backup copy of the file, and the compiler produces one or more files that are of no interest to me, as well as the executable that I actually run. At some point I need to clean up these files.

So, in keeping with this general aim, we'll start out by looking at automating a simple sequence of Unix commands that we might use on a regular basis.

We'll then build up to doing something more complicated like running a program that does some numerical calculations repeatedly with different input data (parameters).

Automating repetitive tasks (2)

How do I do this?:

1. Change into my home directory:

```
$ cd
```

2. Create a backup directory:

```
$ mkdir backup
```

3. Move editor backups from source directory to backup directory:

```
$ mv source/*~ backup
$ mv source/*.bak backup
```


Different editors tend to backup files in different ways. gedit's backups have the same name as the original file with a ~ added to the end of the name (e.g. the backup of `myprog.c` would be `myprog.c~`). Some editors' backups will have the same name as the original file with a `.bak` added to the end of the name. For the sake of this example, let's suppose I sometimes use different editors as the mood takes me so I want to handle whatever backup files there might be, regardless of which editor(s) I've been using.

Automating repetitive tasks (3)

4. Delete extraneous compiler files

```
$ rm source/*.o
```

If I put those commands together…:

```
cd
mkdir backup
mv source/*~ backup
mv source/*.bak backup
rm source/*.o
```


Instead of typing out those commands each time I want to do this, I could just put them all together…

Automating repetitive tasks (4)

…I can make a simple shell script:

```
$ gedit cleanup-prog-dir.sh &
```

```
#!/bin/bash
cd
mkdir backup
mv source/*~ backup
mv source/*.bak backup
rm source/*.o
```

```
$ chmod a+x cleanup-prog-dir.sh
```


…into a very simple shell script. Note that this shell script is just a linear list of the commands I would type at the command line in the order I would type them. Now I can just type:

`./cleanup-prog-dir.sh`

if I'm in my `source` directory, or:

`~/source/cleanup-prog-dir.sh`

if I'm in another directory, instead of all those separate commands. Simple, really.

(After creating the shell script in gedit (or another editor of your choice) remember to save it *and* set the executable bit on the script using `chmod` before trying to run it.)

Improving my shell script (1)

```
#!/bin/bash
cd
mkdir -p backup
mv source/*~ backup
mv source/*.bak backup
rm -f source/*.o
```


Of course, my shell script is very simple, so it gives me errors if I run it more than once, or if some of the files I want to handle don't exist. I can fix some of these errors quite simply:

- If I use the **`-p`** option with **`mkdir`**, then it won't complain if the `backup` directory already exists.
- If I use the **`-f`** option with **`rm`**, then it won't complain if there aren't any `.o` files.

Unfortunately, there's no correspondingly easy way to deal with `mv` complaining if there aren't any files ending in ~ or `.bak`. We need to know more shell scripting to deal with that problem.

Note, though, that it doesn't prevent our shell script from running, it just gives us some annoying error messages when we do run it. So our shell script is still perfectly usable, if not very pretty.

(Remember to save your shell script after making these changes.)

Improving my shell script (2)

```
#!/bin/bash
# Change to my home directory
#  as directories are relative to home
cd
# Make backup directory
mkdir –p backup
# Move editor backups to backup dir
mv source/*~ backup
mv source/*.bak backup
# Delete compiler object files
rm –f source/*.o
```


One of the most important improvements I can make to even this simple shell script is to add some documentation, in the form of *comments*, to it.

Any line that starts with the hash character (#) is ignored by the shell. Such lines are called comments, and are used to add notes, explanations, instructions, etc to shell scripts and programs.

This is very important, because I may well have forgotten what this shell script is supposed to do in several months when I come to use it again. If I've put sensible comments in it though, then it is immediately obvious.

This also makes it easier to debug if I've made a mistake: the comment tells me what the shell script is *supposed* to be doing at that point, so if there is a discrepancy between that and what it *actually* does when I run it, then it is clear there's a bug in the script, probably somewhere around that point.

(Remember to save your shell script after adding the comments.)

Second exercise (1)

We have a program, **`zombie.py`**, that takes five parameters and produces some output (on the screen and also in a file). We want to run it several times with different parameters, storing the output in the file from each run.

The **`zombie.py`** program is in your home directory. It is a program written specially for this course, but this is a pretty general task you might want to do with many different programs. Think of **`zombie.py`** as just some program that takes some input on the command line and then produces some output in a file, e.g. a scientific simulation or data analysis program.

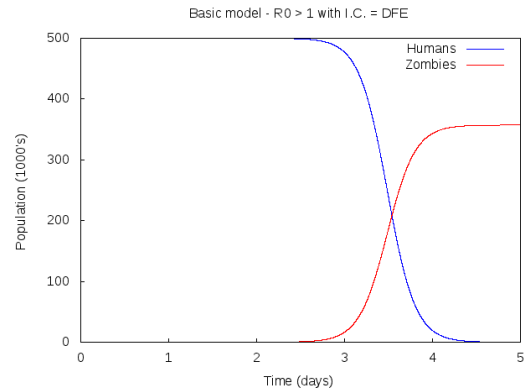
What is **`zombie.py`**?

When Zombies Attack!

Simulation of an outbreak of a zombie infection in a closed population

Photo: Melbourne Zombie Shuffle by Andrew Braithwaite
Licensed under CC BY 2.0
http://www.flickr.com/photos/bratha/2578784637/

blue = Humans
red = Zombies



The **`zombie.py`** program uses a variant of the SIR model from epidemiology to simulate an outbreak of a zombie infection in a closed (i.e. no one enters or leaves) population. Obviously, since zombies don't actually exist, it would be a mistake to try and take this program too seriously. You should think of **`zombie.py`** as just a program that takes some input on the command line and then produces some output on the screen and in a file, and whose output can then be fed to yet other programs for further processing (as we'll see later this afternoon).

However, as it happens, the program is based on actual academic modelling of the spread of disease, as found in Chapter 4 (pp. 133-150) of Infectious Disease Modelling Research Progress (2009), which is entitled "When Zombies Attack!: Mathematical Modelling of an Outbreak of Zombie Infection", and which you can find here:

http://mysite.science.uottawa.ca/rsmith43/zombies.pdf

And in case you are interested in the book from which that chapter is taken, the ISBN of Disease Modelling Research Progress is 978-1-60741-347-9, it's edited by J. M. Tchuenche & C. Chiyaka and published by Nova Science Publishers, Inc.

Note that the **`zombie.py`** program writes its output to a file of numbers rather than producing graphical output. At the end of this afternoon we will see how to produce graphs of its output.

Know Your Enemy (1)

```
$ ./zombie.py
Wrong number of arguments!
5 required, 0 found.
Usage: ./zombie.py zom_death infect resurrect death size

$ ls *zom*
zombie.py

$ ./zombie.py 0.005 0.0175 0.01 0.01 500

When Zombies Attack!: Basic Model of outbreak of zombie infection

Population size:        5.0000e+05
Model run time:         1.0e+01 days

Zombie destruction rate (alpha):        5.000000e-03
Zombie infection rate (beta):           1.750000e-02
Zombie resurrection rate (zeta):        1.000000e-02
Natural death [and birth] rate (delta): 1.000000e-02

Output file:            zombie.dat

Model took 6.947398e-02 seconds
```


The **`zombie.py`** program, which is located in your home directory, takes 5 numeric arguments: 4 positive floating-point numbers and 1 positive integer. It always writes its output to a file called `zombie.dat` in the current working directory, and also writes some informational messages to the screen, which we'll ignore for now.

`*zom*` is a file name glob that means "all the files containing '`zom`' (in the current directory)". File name globbing was covered in the "Unix: Introduction to the Command Line Interface" course, see:

https://www.training.cam.ac.uk/ucs/course/ucs-unixintro1

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

https://help.uis.cam.ac.uk/help-support/training/downloads/course-files/programming-student-files/unix-cli

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.

Know Your Enemy (2)

```
$ ls *zom*
zombie.py  zombie.dat  running-zombie

$ ./zombie.py 0.005 0.0175 0.01 0.01 500
Zombie Attack already running in this directory!

$ rm running-zombie
$ ./zombie.py 0.005 0.0175 0.01 0.01 500

When Zombies Attack!: Basic Model of outbreak of zombie infection

Population size:        5.0000e+05
Model run time:         1.0e+01 days

Zombie destruction rate (alpha):        5.000000e-03
Zombie infection rate (beta):           1.750000e-02
Zombie resurrection rate (zeta):        1.000000e-02
Natural death [and birth] rate (delta): 1.000000e-02

Output file:            zombie.dat

Model took 6.758285e-02 seconds
```


Recall that **`*zom*`** is a file name glob that means “all the files containing ‘`zom`’ (in the current directory)”. Also, once again 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.

The **`zombie.py`** program is not as well behaved as we might like: every time it runs it creates a file called `running-zombie` 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.

Of course, if we run it multiple times in the same directory, we will overwrite any file called `zombie.dat` each time. So if we want to keep the output of each run we’ll need to rename the `zombie.dat` file or copy it to somewhere else before we run the program again.

Second exercise (2)

What we want to do is:

1. Delete any file called `running-zombie`

```
$ rm running-zombie
```

2. Run `zombie.py` with some parameters

```
$ ./zombie.py 0.005 0.0175 0.01 0.01 50
```

3. Rename `zombie.dat`

```
$ mv zombie.dat zombie-50.dat
```

4. Repeat the above steps for the following parameter sets:

```
0.005 0.0175 0.01 0.01 500
0.005 0.0175 0.01 0.01 5000
```

5 minutes

Your task is to create a simple shell script that does the above task. Basically, you want to run the **`zombie.py`** program three times with a different parameter set each time. Note that only the last parameter changes between each run, and that is the parameter we insert into the output file name when we rename it to stop it being overwritten by the next run.

We have gone through everything you need to do this exercise (remember the shell script should be very simple, nothing fancy). You should comment your shell script, preferably as you are writing it, and you should try to avoid it producing errors if you can. (However, the important thing is to produce a shell script that completes the above task, even if it produces some error messages along the way.)

When you've finished this exercise, take a break from the computer – and I do mean "from the computer" – sitting at a computer for too long is bad for you! Don't check your e-mail, get up, stretch, move around, get something to drink.

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

```
$ VAR="My variable"
$ echo "${VAR}"
My variable
$ VAR1="${VAR}"
$ VAR="567"
$ echo "${VAR}"
567
$ echo "${VAR1}"
My variable
```


We create shell variables by simply assigning them a value (as above for the shell variables **VAR** and **VAR1**). We can access a 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. **VAR1** is *not* a "pointer" to or an "alias" for **VAR**.

Improving my shell script (3)

```
#!/bin/bash
# Configuration section
myPROGS="${HOME}/source"
myBACKUPS="${HOME}/backup"
# Change to my home directory
#  as directories are relative to home
cd
# Make backup directory
mkdir –p "${myBACKUPS}"
# Move editor backups to backup dir
mv "${myPROGS}"/*~ "${myBACKUPS}"
mv "${myPROGS}"/*.bak "${myBACKUPS}"
# Delete compiler object files
rm –f "${myPROGS}"/*.o
```


I can use shell variables to store (almost) any values I like, much as I can use variables in a program. I can define my program directory and backup directory in shell variables, and then use those variables in the rest of my shell script wherever I would have previously used the corresponding values. This has the huge advantage that if I want to change the location of my program directory or backup directory, I only need to do it in one place.

Another advantage is, if I am disciplined and define all my important shell variables at the start of my shell script, I know immediately, just by looking at the start of the shell script, what values are important to my shell script. Note that I've used variable names that have some relation to what their values represent rather than generic variable names like **VAR1**, **VAR2**, etc. Using sensible variable names can be a huge help in figuring out what the shell script is supposed to do.

Note that I specify my program and backup directories as being subdirectories of my home directory, which means my script now knows where to find them regardless of the directory it is operating in. The way I get my my home directory is by getting the value of the environment variable **HOME** – more on environment variables in a moment – using **${HOME}** (exactly the same as I would for a shell variable called **HOME**). At the command line we will often use **~** to mean "my home directory", but, unfortunately, this does not work if the **~** character appears in double quotes.

(Remember to save your shell script after making the changes above.)

Improving my shell script (4)

```
#!/bin/bash
# Configuration section
myPROGS="${HOME}/source"
myBACKUPS="${HOME}/backup"
# Make backup directory
mkdir –p "${myBACKUPS}"
# Move editor backups to backup dir
mv "${myPROGS}"/*~ "${myBACKUPS}"
mv "${myPROGS}"/*.bak "${myBACKUPS}"
# Delete compiler object files
rm –f "${myPROGS}"/*.o
```


Now delete the following three lines from the shell script:

```
# Change to my home directory
#  as directories are relative to home
cd
```

(Remember to save your shell script after deleting the two lines above.)

We now see another benefit I've gotten from improving my script: because I am now specifying the *full* path (i.e. a path that starts with `/`) rather than a *relative* path (i.e. a path that depends on which directory I'm currently in) for my program and backup directories, I no longer need to change directory to start with. As an added benefit, using a full path rather than a relative path means I don't have to worry about what happens if my script fails to change directory for some reason. With relative paths, if the script fails to change to the correct directory to start with, then all the subsequent commands that use relative paths will try and operate on the wrong files, since the script will be in the wrong part of the filesystem!

Environment Variables (1)

```
$ env
TERM=xterm
SHELL=/bin/bash
.
.
.

$ set
BASH=/bin/bash
BASH_VERSION='4.2.24(1)-release'
.
.
.

$ set | more
BASH=/bin/bash
BASH_VERSION='4.2.24(1)-release'
.
.
.
```


When used with no arguments, the **env** command displays the *environment variables* (and their values). The environment variables are simply a collection of variables and values that are passed to a program (including the shell and any shell scripts) when the program starts up. Typically they contain information that may be used by the program or that may modify its behaviour. Two environment variables you may have already met are `PATH` and `HOME`. `PATH` specifies which directories the system should search for executable files when you ask it to execute a program and don't give it a path to the executable. `HOME` is usually set by the system to the location of the user's home directory.

The **`set`** shell builtin command (when issued with no arguments) displays all the environment variables, shell variables, various *shell parameters* and any *shell functions* that have been defined. (We'll be meeting shell parameters in context a little later, which should make clear what they are, and shell functions are covered on the next day of this course.) Thus **`set`** displays many more variables than the **env** command.

So many more, in fact, that we probably want to *pipe* its output through the **`more`** command. (The **`more`** command displays its input on the screen one screenful (page) at a time.) Piping is the process whereby the *output* of one command is given to another command as *input*. We tell the shell to do this using the **|** symbol. So:

```
set | more
```

takes the the output of the **`set`** shell builtin command and passes it to the **`more`** command, which displays it for us one screen at a time.

Note that the output of **env** and **`set`** may be different from that shown here, and also, since both commands produce so much output, not all of their output is shown on this slide, as is indicated by the three dots on separate lines:

.
.
.

Environment Variables (2)

```
$ env | grep 'zzTEMP'
$ zzTEMP="Temp variable"
$ env | grep 'zzTEMP'
$ set | grep 'zzTEMP'
zzTEMP='Temp variable'
$ export zzTEMP
$ env | grep 'zzTEMP'
zzTEMP=Temp variable
```


Recall that we create shell variables by simply assigning them a value (as above for the shell variable **zzTEMP**). A shell variable is **not** the same as an environment variable however, as we can see by searching for the shell variable **zzTEMP** in the output of the **env** command. However, we have indeed created a ***shell*** variable with that name, as we see by examining the output of the **set** shell builtin command.

(The **grep** command searches for strings of text in other text. In the example above we are using it to search for the text "zzTEMP" in the output of various commands.)

The **export** shell builtin command adds a shell variable to the shell's environment. Once we've done this, we see that if we run the **env** command we will find the **zzTEMP** variable. It is has become an *environment variable*.

Environment Variables (3)

```
$ export -n zzTEMP
$ env | grep 'zzTEMP'
$ set | grep 'zzTEMP'
zzTEMP='Temp variable'
$ export zzVAR="Another variable"
$ env | grep 'zzVAR'
zzVAR=Another variable
```


We can remove a variable from the shell's environment by using the **`export`** shell builtin command with the **`-n`** option. Note that this does *not* destroy the variable, and it remains a shell variable, but is no longer an environment variable.

Once a shell variable has been added to the shell's environment, it remains an environment variable even if we change its value. Thus we do not have to keep using the **`export`** shell builtin command on a variable every time we change its value.

We can also set a shell variable and add it to the shell's environment all in one go using the **`export`** shell builtin command, as in the above example with the **`zzVAR`** variable.

Environment Variables (4a)

```
$ grep --color=always 'MCS' welcome
This course is intended to be run on MCS Linux 2012/2013.

$ GREP_COLORS='ms=01;36'
$ grep --color=always 'MCS' welcome
This course is intended to be run on MCS Linux 2012/2013.


$ GREP_COLORS='ms=01;36' grep
   --color=always 'MCS' welcome
This course is intended to be run on MCS Linux 2012/2013.
$ grep --color=always 'MCS' welcome
This course is intended to be run on MCS Linux 2012/2013.
```


The **grep** command will highlight any matching text if you give it the **--color=always** option. You can control the colours it uses by using the **GREP_COLORS** environment variable (see the man page for **grep** for details of how you specify the colours). However, setting an environment variable means every time you run **grep** with the appropriate option it will use those colours. But suppose you just want to change the colours for just a single use of the **grep** command? Is this possible?

We can set (or change) an environment variable for just a single run of a command by setting the variable immediately before the command ***on the same line***.

Note that the environment variable is added to the environment (or its value is changed) just for that command. The environment variables in the shell's environment remain unaffected by this change, as we can see in the example above: the **GREP_COLORS** variable does not exist in the shell's environment, although we set it once for a particular use of the **grep** command (we can see this by running the **grep** command again with the same arguments and inspecting its output).

Environment Variables (4b)

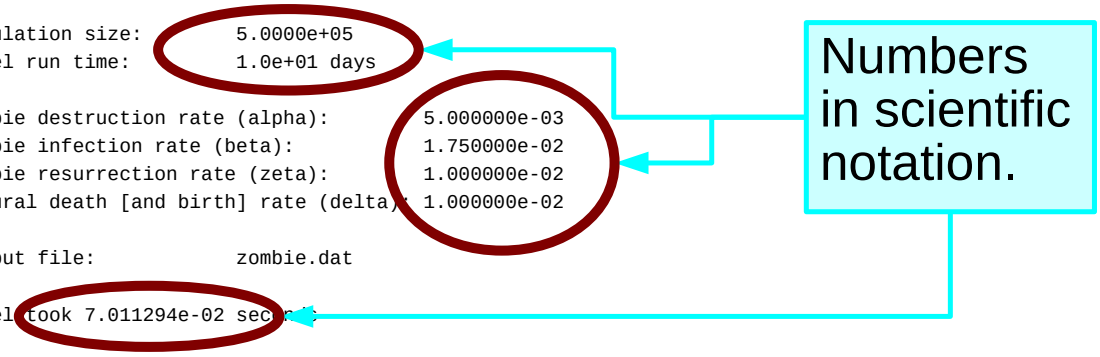
```
$ echo "${zzJUNK}"

$ zzJUNK="123" env | grep 'zzJUNK'
zzJUNK=123
$ echo "${zzJUNK}"

$
```


To recap: we can set (or change) an environment variable for just a single run of a command by setting the variable immediately before the command ***on the same line***.

As previously mentioned, the environment variable is added to the environment (or its value is changed) just for that command. The environment variables in the shell's environment remain unaffected by this change, as we can see in the example above: the **`zzJUNK`** variable does not exist in the shell's environment, although we set it for the **`env`** command (which we can verify by searching the output of the **`env`** command with the **`grep`** command).

Know Your Enemy (3a)

```
$ rm running-zombie
$ ./zombie.py 0.005 0.0175 0.01 0.01 500

When Zombies Attack!: Basic Model of outbreak of zombie infection

Population size:                    5.0000e+05
Model run time:                     1.0e+01 days

Zombie destruction rate (alpha):    5.000000e-03
Zombie infection rate (beta):       1.750000e-02
Zombie resurrection rate (zeta):    1.000000e-02
Natural death [and birth] rate (delta): 1.000000e-02

Output file:                        zombie.dat

Model took 7.011294e-02 seconds
```



By default, all the numbers that the **`zombie.py`** program displays are in *scientific notation* (also called *standard form*). For very large or very small numbers, scientific notation makes sense, but for numbers that aren't very big or very small (like the ones we are using), it makes less sense, and can be difficult to read.

Is there anything we can do about this?

Note that the output of the **`zombie.py`** program may not exactly match what is shown on this slide – in particular, the model may take more or less time to run.

Know Your Enemy (3b)

```
$ rm running-zombie
$ ZOMBIE_FORMAT="NORMAL" ./zombie.py 0.005 0.0175 0.01 0.01 500

When Zombies Attack!: Basic Model of outbreak of zombie infection

Population size:        500000
Model run time:         10.0 days

Zombie destruction rate (alpha):        0.0050
Zombie infection rate (beta):           0.0175
Zombie resurrection rate (zeta):        0.0100
Natural death [and birth] rate (delta): 0.0100

Output file:            zombie.dat

Model took 0.072 seconds
```

Numbers displayed normally

Like many programs, the way the **`zombie.py`** program behaves can be affected by the environment variables defined at the time it is run. In the case of the **`zombie.py`** program, the way in which it displays numbers on the screen is controlled by the `ZOMBIE_FORMAT` environment variable.

If the `ZOMBIE_FORMAT` environment variable is set to the value:

```
NORMAL
```

when the **`zombie.py`** program is run, the numbers it displays on the screen will ***not*** be in scientific notation.

In the example above, we've only set this environment variable for a single run of the **`zombie.py`** program. If we wanted to set it for all future runs of the **`zombie.py`** program started from this session of the shell then we need to use the **`export`** shell builtin command as follows:

```
export ZOMBIE_FORMAT="NORMAL"
```

Note that the output of the **`zombie.py`** program may not exactly match what is shown on this slide – in particular, the model may take more or less time to run.

Positional parameters

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

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.

@,

- @ expands to values of all positional parameters (starting from **1**)

  In double quotes each parameter is treated as a separate *word* (value)

  "${@}"

- # expands to the number of positional parameters (not including **0**)

  ${#}

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: **${@}**
- Number of arguments: **${#}**

```
$ cd
$ 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 shell parameters introduced earlier 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.

Using positional parameters

```
#!/bin/bash
# Remove left over running-zombie file
rm –f running-zombie
# Run zombie.py with passed arguments
ZOMBIE_FORMAT="NORMAL" ./zombie.py "${@}"
# Rename output file
mv zombie.dat "zombie-${5}.dat"
# Remove left over running-zombie file
rm –f running-zombie
```


The file **`run-once.sh`** in the `scripts` directory (shown above) accepts some command line arguments and then tries to run the **`zombie.py`** program with them. (Note that it does no checking whatsoever of the arguments it is given.) On the assumption that only the fifth argument will change between runs, it renames the output file to a new name based on that argument. Change to your home directory and try this:

```
scripts/run-once.sh 0.005 0.0175 0.01 0.01 50
```

Do an **`ls`** of your home directory and see what it has produced.

Redirection (>)

Redirect output to a file, *overwriting* file if it exists:

```
command > file
```

Equivalently:

```
command 1> file
```

Redirect standard error to a file, *overwriting* file if it exists:

```
command 2> file
```


The **>** operator redirects the output from a command to a file, ***overwriting*** that file if it exists. You place this operator at the end of the command, after all of its arguments. The place where the output of a command normally goes is known as "*standard output*" or "*file descriptor 1*". So, we can also use **`1> filename`** which means "redirect file descriptor 1 (i.e. standard output) to the file **`filename`**, ***overwriting*** it if it exists".

(Unsurprisingly, **`2> filename`** means "redirect file descriptor 2 (*standard error*) to the file **`filename`**, ***overwriting*** it if it exists". And it will probably come as no shock to learn that ***`descriptor`*****`> filename`** means "redirect file descriptor ***`descriptor`*** to the file **`filename`**, ***overwriting*** it if it exists", where ***`descriptor`*** is the number of a valid file descriptor. We'll meet *standard error*, also known as *file descriptor 2*, on the third day of the course.)

You may think that this "overwriting" behaviour is somewhat undesirable – you can make the shell refuse to overwrite a file that exists, and instead return an error, using the `set` shell builtin command as follows:

```
set -o noclobber
```

or, equivalently:

```
set -C
```

Redirection to nowhere (`> /dev/null`)

Redirect output to “nowhere” (i.e. the command’s output is discarded):

```
command > /dev/null
```

Equivalently:

```
command 1> /dev/null
```


If you don’t want a command’s output for some reason, then you can discard it by redirecting the output to a special “*device*” that is part of the operating system: **`/dev/null`**.

Anything sent to `/dev/null` is simply thrown away. This is particularly useful if you have a shell script where you want to run some commands that produce output on the screen, but that output would be confusing or of no interest to the users of you shell script.

Using redirection

```
#!/bin/bash
# Remove left over running-zombie file
rm –f running-zombie
# Run zombie.py with passed arguments
ZOMBIE_FORMAT="NORMAL" ./zombie.py "${@}" > "stdout-${5}"
# Rename output file
mv zombie.dat "zombie-${5}.dat"
# Remove left over running-zombie file
rm –f running-zombie
```


Modify the file **`run-once.sh`** in the `scripts` directory as shown above (remember to save it when you've finished). Now it captures what the **`zombie.py`** program outputs to the screen (standard output) as well (hurrah!). Change to your home directory and try this:

```
scripts/run-once.sh 0.005 0.0175 0.01 0.01 50
```

Do another **`ls`** of your home directory and see what it has produced.

Appending output (**>>**)

- Redirect output of a command…
- …to a file…
- …*appending* it to that file

```
command >> file
```


The **>>** operator redirects the output from a command to a file, ***appending*** it to that file. You place this operator at the end of the command, after all of its arguments. If the file does not exist, it will be created.

(You can also use **1>>** instead of just **>>**. If you want to redirect *standard error* of a command (also known as *file descriptor 2*) to a file, appending it to that file, you would use **2>>**. (And ***descriptor*>> filename** means "redirect file descriptor ***descriptor*** to the file **filename**, ***appending*** to the file **filename**".) Don't worry if you don't know what standard error is – we'll meet it properly in the third session of this course; this note here is just for completeness.)

Keeping a record

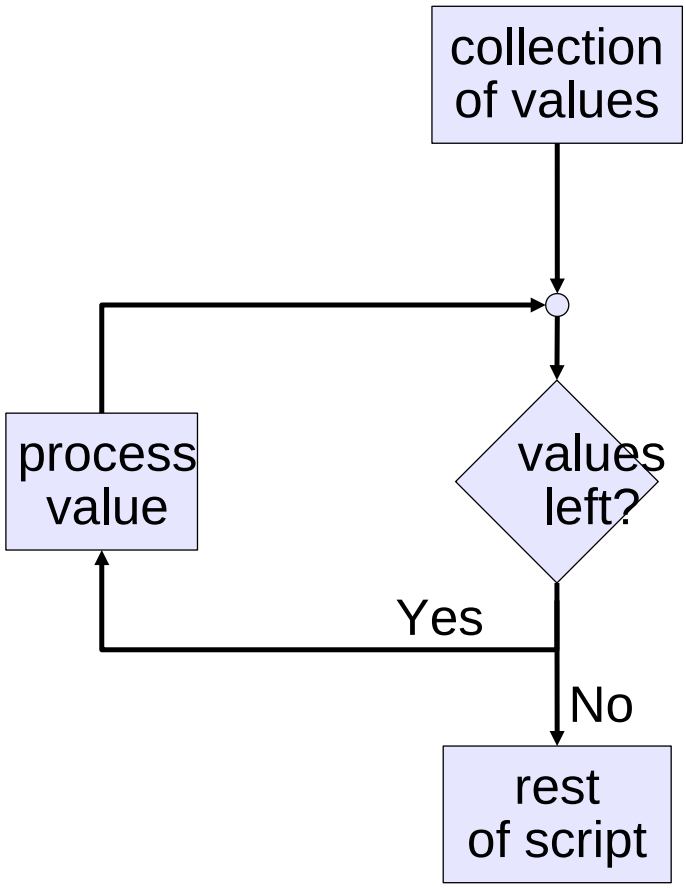
```
#!/bin/bash
# Remove left over running-zombie file
rm –f running-zombie
# Write to logfile
echo "" >> logfile
date >> logfile
echo "Running zombie.py with ${@}" >> logfile
# Run zombie.py with passed arguments
ZOMBIE_FORMAT="NORMAL" ./zombie.py "${@}" > "stdout-${5}"
# Rename output file
mv zombie.dat "zombie-${5}.dat"
# Remove left over running-zombie file
rm –f running-zombie
# Write to logfile
echo "Output file: zombie-${5}.dat" >> logfile
echo "Standard output: stdout-${5}" >> logfile
```


Modify the file **`run-once.sh`** in the `scripts` directory as shown above (remember to save it when you've finished). Now every time it runs it stores a record of what it is doing in the file `logfile` in the current directory. Making your shell scripts keep a record of what they are doing is an extremely good idea, especially if they are going to run for a long time or on a remote machine or when you are not around.

Notice that we have something written to the `logfile` ***before*** we start running the **`zombie.py`** program ***and*** something ***after*** it is finished. This means that if the shell script crashes or is stopped before it is finished there is a very good chance we'll be able to tell from the log file as it will not have the "`Output file:`" or "`Standard output:`" lines in it. There are better, more sophisticated ways of checking whether things have gone wrong, but this is a nice simple one that is well worth remembering.

Now change to your home directory and try this:

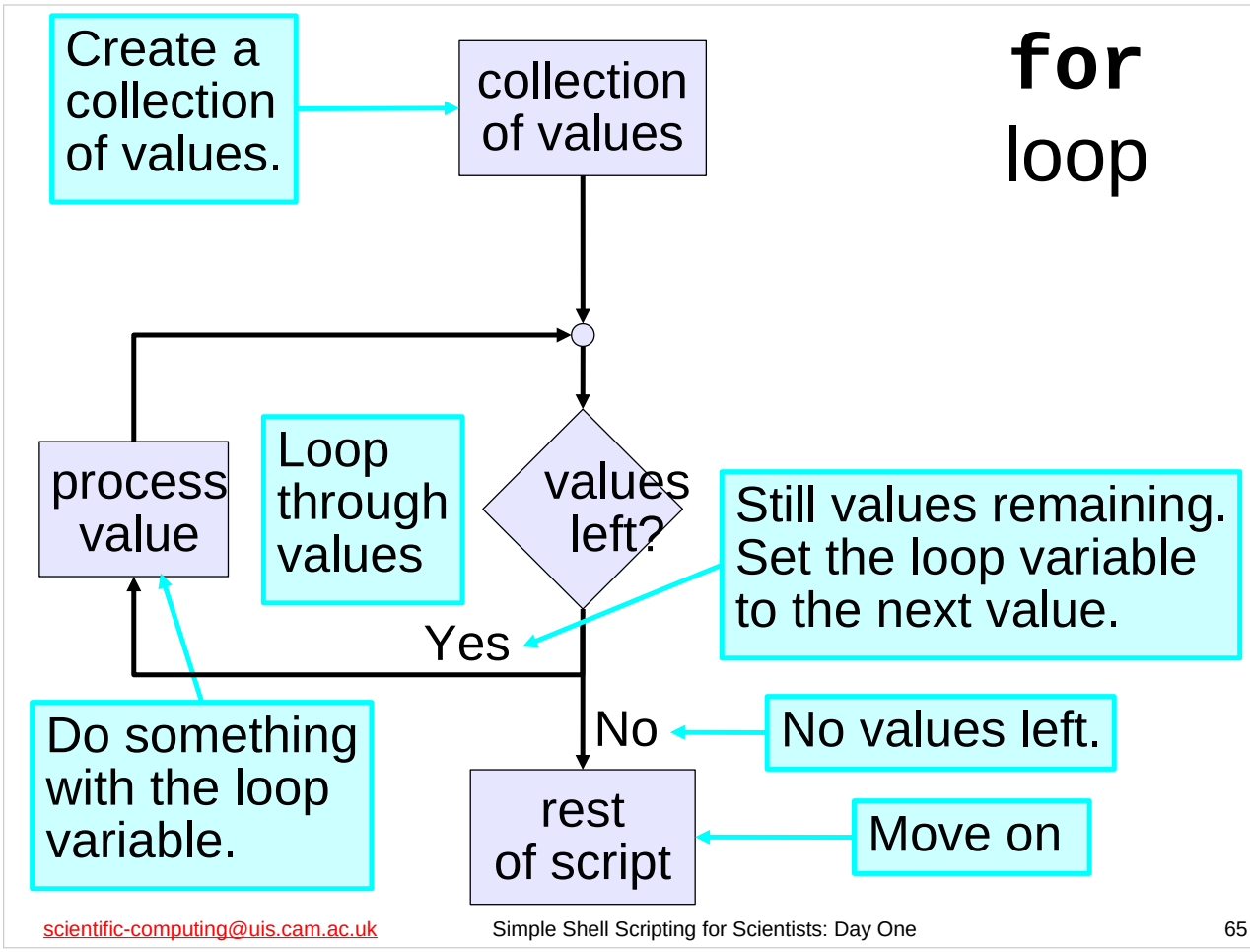
```
scripts/run-once.sh 0.005 0.0175 0.01 0.01 50
cat logfile
```



`for` loop

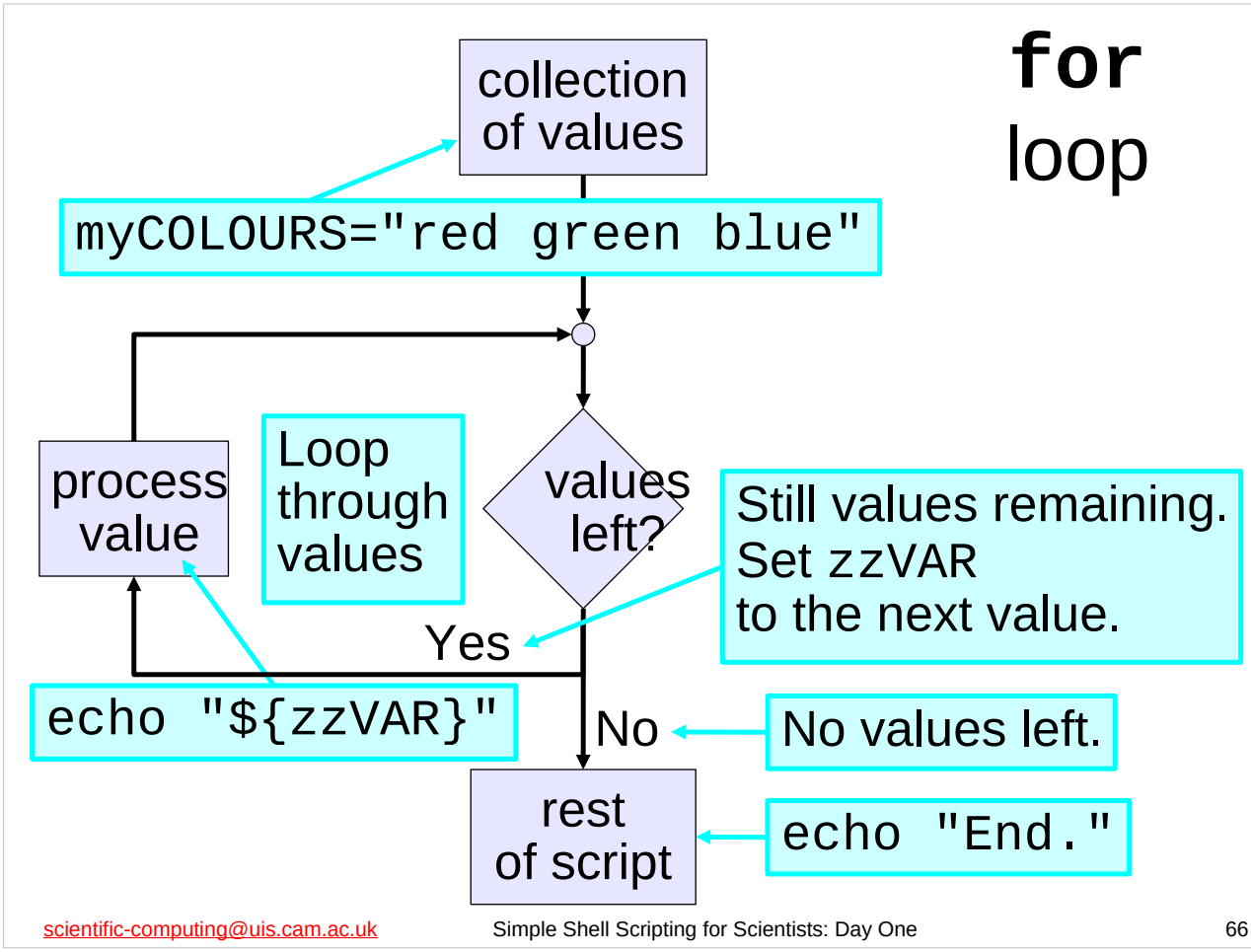
Execute some commands once for each value in a collection of values

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 (strings of characters) it has been given.



The idea behind the **`for`** loop is that the shell script works its way through the collection of values, one value at a time, asking if there are any values left to process. If so, a variable (called the "loop variable") gets set to refer to the current value in the collection and a set of commands is run. Then the question is asked again and so on. When there are no more values, the loop finishes and the shell script carries on. (Of course, if the **`for`** loop is at the end of the shell script, then when the for loop finishes, so will the shell script.)

The set of commands that the **`for`** loop runs for each value in the collection can be as large or small as we like, provided that it contains at least one command.

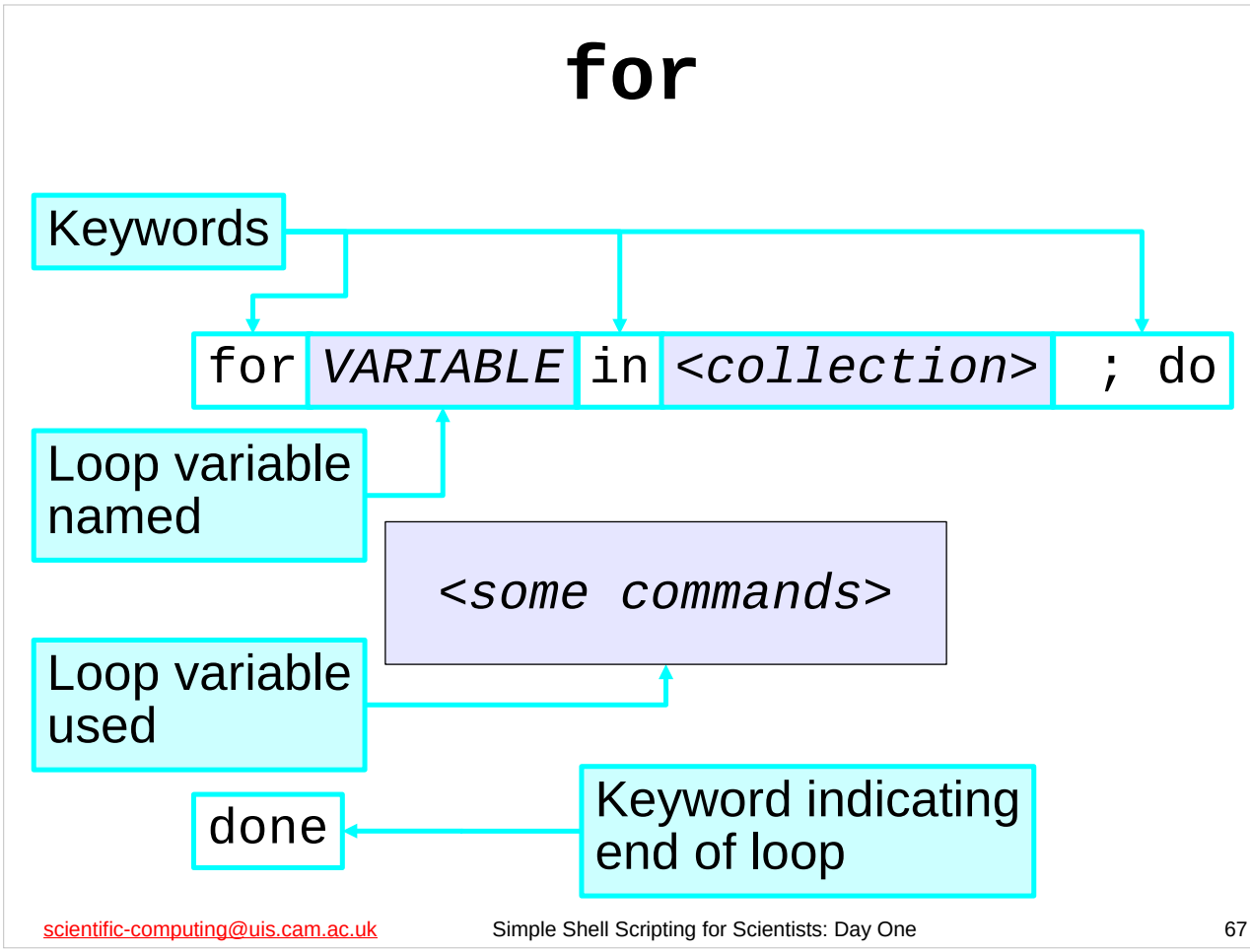


So in the example we're about to look at we start by creating a collection of values – the names of the colours red, green and blue – and putting them in a shell variable, **`myCOLOURS`**.

We will assign each value in our collection to the loop variable **`zzVAR`**, one at a time.

The action we are going to perform is simply to print the value of **`zzVAR`**. Obviously we could do a lot more.

Once we're finished and have left the loop we will print "`End.`" just to establish that control has passed on beyond the loop and that the script hasn't just silently aborted.



So how do we build this into a shell script? 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) and **`<some commands>`** is a list of one or more commands to be executed.

The words "**`for`**" and "**`in`**", "**`do`**", "**`done`**" and the semi-colon ("**`;`**") are just syntax. The word immediately after "**`for`**" is the name of the loop variable that is going to be used to track the values in the collection that appears immediately after "**`in`**".

The "**`; do`**" indicates that we've finished giving the collection of values to loop over and we're about to start giving the commands we want executed for each iteration of the loop.

The word "**done**" indicates that we've finished specifying the commands we want executed for each iteration of the loop.

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
```

for1.sh

```
myCOLOURS="red green blue"

for zzVAR in ${myCOLOURS} ; do

echo "${zzVAR}"

done

echo "End."
```

```
$ cd
$ examples/for1.sh
```


So here is an example of a **`for`** loop. It is in the file **`for1.sh`** in the `examples` directory of your home directory. We create a collection of values (`"red green blue"`) and assign it to a shell variable, **`myCOLOURS`**. This will be the collection of values our loop will use. Then we tell the loop to create a loop variable (**`zzVAR`**) to hold each value in turn and to loop over the collection of values in **`myCOLOURS`**. Recall that the construct `${VARNAME}` means "the values in the variable *VARNAME*". We then use the loop variable in the loop – in this example we just print out its value (**`echo "${zzVAR}"`**).

Note that our shell script does not explicitly define the shell variable **`zzVAR`** anywhere. This variable is created by the **`for`** loop (and persists even after the loop is finished, so we can see what the last value the loop processed was if we should so wish).

Note also how we *don't* put any quotation marks around **`${myCOLOURS}`** when we tell our **`for`** loop that this is the collection of values we want it to loop over. If we were to put quotation marks around **`${myCOLOURS}`**, then it would be regarded as a single value ("`red green blue`") rather than as 3 separate values ("`red`", "`green`" and "`blue`").

You should now be able to work out what this **`for`** loop will do. When you think you know, run the **`for1.sh`** shell script in the `examples` directory of your home directory and see if you were correct. If there is anything you do not understand in the shell script shown above, please ask the course giver or a demonstrator now.

for1a.sh

```
myCOLOURS="red green blue"

for zzVAR in "${myCOLOURS}"; do

echo "${zzVAR}"

done

echo "End."
```

```
$ cd
$ examples/for1a.sh
```


What happens if we *do* put double quotation marks around **`${myCOLOURS}`**?

A mentioned previously, if we put double quotation marks around **`${myCOLOURS}`**, then it is regarded as a single value ("`red green blue`") rather than as 3 separate values ("`red`", "`green`" and "`blue`").

You should now be able to work out what the **`for`** loop above will do. When you think you know, run the **`for1a.sh`** shell script in the `examples` directory of your home directory and see if you were correct. If there is anything you do not understand in the shell script shown above, please ask the course giver or a demonstrator now.

for1b.sh

```
for zzVAR in "${@}" ; do
echo "${zzVAR}"
done
echo "End."
```

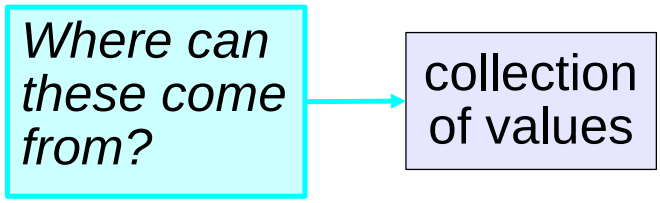
```
$ cd
$ examples/for1b.sh red green blue
```


There's another common place where we might get collections of values for our **`for`** loops: from shell parameters.

We can also use a shell parameter as the source of the collection of values for our **`for`** loop. A very common example of this is to use the special parameter @ to loop through the arguments that were passed to the shell script. Note that if we use @ for a **`for`** loop, we'll almost always want to surround it in quotation marks, i.e. **`"${@}"`**, so that we loop over each of the individual arguments separately.

You should now be able to work out what the **`for`** loop above will do. When you think you know, run the **`for1b.sh`** shell script in the `examples` directory of your home directory as shown on the slide and see if you were correct. If there is anything you do not understand in the shell script shown above, please ask the course giver or a demonstrator now.

Values for **`for`** loops



From a shell variable (or environment variable) or shell parameter:

```
for zzVAR in ${myVARIABLE} ; do
for zzVAR in ${LS_OPTIONS} ; do
for zzVAR in "${@}" ; do
```

…or specify them directly:

```
for zzVAR in "red" "green" "blue" ; do
```


So where can we get collections of values for our **`for`** loops?

Probably the most common way to specify a collection of values for a **`for`** loop is to use a shell variable (or environment variable), either one we've created ourselves, or one which something else has created for us. Note that in such cases we normally would ***not*** surround the **`${}`** with quotation marks.

(For the curious: `LS_OPTIONS` (used as an example on the slide above) is an environment variable that may be used to control the behaviour of the `ls` command on some systems.)

As we have seen, we can also use a shell parameter, most notably the special parameter @ to loop through the arguments that were passed to the shell script. Note that if we use @ for a **`for`** loop, we'll almost always want to surround it in quotation marks, i.e. **`"${@}"`**, so that we loop over each of the individual arguments separately.

We can also just give the collection of values to the for loop directly, by specifying them after the in keyword, one after the other, separated by one or more spaces, e.g.

```
for zzVAR in "red" "green" "blue" ; do
    echo "${zzVAR}"
done
```

More values for `for` loops

Where can these come from? → collection of values

…or from a file name glob or wildcard:

```
for zzVAR in *.dat ; do
for zzVAR in zombie?.dat ; do
```


Where else can we get collections of values for our **`for`** loops?

Another extremely common way of getting collections of values for a **`for`** loop is to use a wildcard or file name glob to specify one or more files in a directory. For example, **`*.dat`** means all the files ending in **`.dat`** in the current directory. **`zombie?.dat`** means all the files with names like **`zombie1.dat`**, **`zombie2.dat`**, etc. (basically all files whose name starts with **`zombie`**, followed by a single character and finishing in **`.dat`**) in the current directory. (You can also tell the shell to look in a particular directory for matching files by preceding the file name glob with the path of the directory, e.g. **`/tmp/*`** means all the files in the **`/tmp`** directory.)

File name globbing was covered in the “Unix: Introduction to the Command Line Interface” course, see:

https://www.training.cam.ac.uk/ucs/course/ucs-unixintro1

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

https://help.uis.cam.ac.uk/help-support/training/downloads/course-files/programming-student-files/unix-cli

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
```


To summarise: 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 (this is known as *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 one **`for`** loop can contain another **`for`** loop (the technical term for this is *nesting*).

Multiple runs

```
#!/bin/bash

# Parameters that stay the same each run
myFIXED_PARAMS="0.005 0.0175 0.01 0.01"

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

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


The file **`multi-run.sh`** in the `scripts` directory (shown above) takes one or more command line arguments and then runs the **`run-once.sh`** script (which in turn runs the **`zombie.py`** program) with 5 arguments – 4 that are always the same and that are hard-coded into the script, and one of its command line arguments. It does this repeatedly until there are no more of its command line arguments. This script is much more versatile than the script we wrote for the earlier exercise. Modifying that script for each different set of values we might want to run would have rapidly become extremely tedious, whereas we don't need to modify this script at all – we just run it with different arguments.

Note that when we use the value of the shell variable **`myFIXED_PARAMS`** we ***don't*** surround it with quotes – if we did then it would be treated as a single value instead of as 4 separate values (when the shell treats spaces in this way – as a separator between values – it is called *word splitting*).

Give it a try – change to your home directory and type the following commands (the **`rm`** command is to remove the files produced by our previous runs of earlier scripts):

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

And finally do a **`ls`** of your home directory and see what files have been produced.

If you're not coming back: Give us Feedback!

If, and only if, you will not be attending *any* of the next three days of the course then ***please make sure that you fill in the Course Review form online***, accessible under "`feedback`" on the main MCS Linux menu, or via:

http://feedback.training.cam.ac.uk/uis/

If you *are* coming to further sessions of this course, then you should fill in the feedback form at the **last** session you attend.

Final exercise (1)

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

In this particular case, we happen to be specifically using the **`gnuplot`** program and the output of the **`zombie.py`** program we've met before. (**`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:

https://www-uxsup.csx.cam.ac.uk/courses/moved.Gnuplot/

Let's take a closer look… (1)

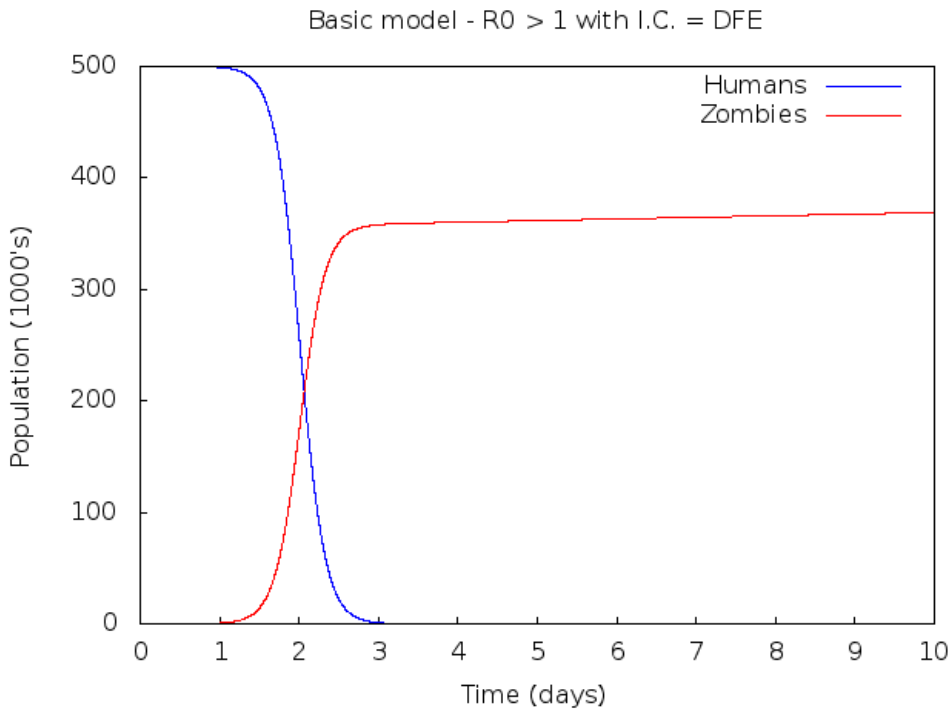
```
$ cd
$ cp gnuplot/zombie.gplt .
$ cp zombie-500.dat zombie.dat

$ ls zombie.png
/bin/ls: zombie.png: No such file or directory

$ gnuplot zombie.gplt
$ rm zombie.dat
$ ls zombie.png
zombie.png
$ eog zombie.png &
```


If you don't already have an `zombie-500.dat` file, do the following before trying the commands above:

```
cd
scripts/multi-run.sh 500
```

Note that the output of "**`ls zombie.png`**" may look slightly different – in particular, the colours may be slightly different shades.

Let’s take a closer look… (2)



Final exercise (2)

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

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

```
$ mv zombie-500.dat zombie.dat
```

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

```
$ gnuplot zombie.gplt
```

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

```
$ mv zombie.dat zombie-500.dat
```

4. Rename `zombie.png`

```
$ mv zombie.png zombie-500.dat.png
```


Your task is to create a shell script that does the above task. Basically, for each of the `.dat` files we've just produced, you want to run **`gnuplot`** on it to create a graph (which will be stored as a `.png` file). The `zombie.gplt` file you've been given will only work if the `.dat` file is called `zombie.dat` and is in the current directory. Also, you don't want **`gnuplot`** to overwrite each `.png` file, so you'll need to rename it after **`gnuplot`**'s created it.

If you don't already have a directory of several `.dat` files, do the following before starting the exercise:

```
cd
scripts/multi-run.sh 50 100 500 1000 3000 5000 10000 50000
```

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

Hint: the best way to do this is with a **`for`** loop over all the `.dat` files in the directory – we haven't used that kind of **`for`** loop much yet, but you've seen the syntax for it, and there is an example of that sort of **`for`** loop in the **`for2.sh`** file in the `examples` directory.

Final exercise – Files

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

https://help.uis.cam.ac.uk/help-support/training/downloads/course-files/programming-student-files/shellscriptingsci/shellscripting-files/exercises/day-one

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*.

Also, if you are doing this course in one of our scheduled classes (as opposed to reading these course notes on-line, for example), then you should have received a user ID to use for the course: it will probably be something like **y*XXX*** (where ***XXX*** is a number). **Make sure** you make a note of this user ID and **bring** it with you to the next session of the course.

If you're not coming back: Give us Feedback!

If, and only if, you will not be attending *any* of the next two days of the course then ***please make sure that you fill in the Course Review form online***, accessible under "`feedback`" on the main MCS Linux menu, or via:

http://feedback.training.cam.ac.uk/uis/

If you *are* coming to further sessions of this course, then you should fill in the feedback form at the **last** session you attend.