Unix Systems: Shell Scripting (I)

Bruce Beckles
University of Cambridge Computing Service

Introduction

- Who:
  - Bruce Beckles, e-Science Specialist, UCS
- What:
  - Unix Systems: Shell Scripting (I) course
  - Part of the ***Scientific Computing*** series of courses
- Contact (questions, etc):
  - escience-support@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.

What we don't cover

- Different types of shell:
  - We are using the Bourne-Again SHell (bash).
- Differences between versions of bash
- Advanced shell scripting – try these courses instead:
  - "Unix Systems: Shell Scripting (II)"
  - "Unix Systems: Shell Scripting (III)"
  - "Programming: Python for Absolute Beginners"

`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` 3.0 in this course, but everything we do should work in `bash` 2.05 and later. Version 3.0 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` 3.2, which was released on 12 October, 2006.

For details of the "Unix Systems: Shell Scripting (II)" course, see:

http://www.cam.ac.uk/cs/courses/coursedesc/linux.html#script2

For details of the "Unix Systems: Shell Scripting (III)" course, see:

http://www.cam.ac.uk/cs/courses/coursedesc/linux.html#scriptwkshp

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

http://www.cam.ac.uk/cs/courses/coursedesc/prog.html#python

Outline of Course

1. Prerequistes & recap of Unix commands
2. Very simple shell scripts

BREAK

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

Exercise (~16:30)

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 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 and ***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.

Follow-on courses

Unix Systems: Shell Scripting (II):

- More advanced shell scripts

Unix Systems: Shell Scripting (III):

- Better, more robust (handle errors!) shell scripts

We *strongly* encourage you to attend these follow-on courses.

For details of the "Unix Systems: Shell Scripting (II)" course, see:
http://www.cam.ac.uk/cs/courses/coursedesc/linux.html#script2

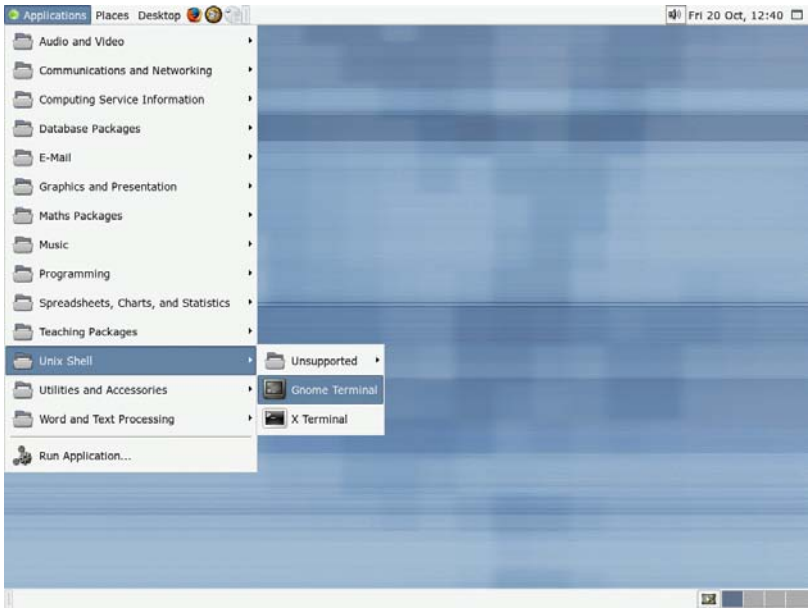
For details of the "Unix Systems: Shell Scripting (III)" course, see:
http://www.cam.ac.uk/cs/courses/coursedesc/linux.html#scriptwkshp

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 System: Introduction" course):
  - Common Unix/Linux commands (`ls`, `rm`, etc)
  - Piping; redirecting input and output
  - Simple use of environment variables
  - Filename expansion ("pathname expansion")

For details of the "Unix System: Introduction" course, see:
http://www.cam.ac.uk/cs/courses/coursedesc/linux.html#unix

Start a shell



Screenshot of newly started shell

Unix commands (1)

`cat` Display contents of a file

```
> cat /etc/motd
```

```
Welcome to PWF Linux 2006/2007.

If you have any problems, please email Help-Desk@ucs.cam.ac.uk.
```

`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 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. Unix permissions were covered in the "Unix System: Introduction" course, see:

http://www.cam.ac.uk/cs/courses/coursedesc/linux.html#unix

Unix commands (2)

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

```
> cp /etc/motd /tmp/motd-copy
```

Options:

`-p` ***p***reserve (if possible) files' owner, permissions & 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

```
> cp -p /etc/motd /tmp/motd-copy
```


Note that the `cp` command has many other options than the three 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
Fri Feb 16 11:52:03 GMT 2007
```

`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 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 'PWF' /etc/motd
Welcome to PWF Linux 2006/2007.
```

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

```
> ln -s /etc/motd /tmp/motd
> cat /etc/motd
Welcome to PWF Linux 2006/2007.
If you have any problems, please email Help-Desk@ucs.cam.ac.uk.
> cat /tmp/motd
Welcome to PWF Linux 2006/2007.
If you have any problems, please email Help-Desk@ucs.cam.ac.uk.
```

escience-support@ucs.cam.ac.uk Unix Systems: Shell Scripting (I) 12

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 "Pattern Matching Using Regular Expressions" course, details of which are given here:

http://www.cam.ac.uk/cs/courses/coursedesc/linux.html#regex

The `ln` command creates links between files. 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:

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

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

Unix commands (5)

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

```
> ls
bin  examples  gnuplot  hello.sh  iterator  scripts  source  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 (6)

`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 (7)

`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 (8)

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

```
> mv /tmp/motd-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, move actually copies the file or directory to the other filesystem and then deletes the original.

Unix commands (9)

`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 (10)

`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 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.

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 +x
```


Recall that the `chmod` command changes the permissions on a file. `chmod +x` sets the executable bit on a file, i.e. it grants permission to execute the file. Unix file permissions were covered in the "Unix System: Introduction" course, see:

http://www.cam.ac.uk/cs/courses/coursedesc/linux.html#unix

Run a simple shell script

```
> ./hello.sh
Hello!  I am a shell script.
Who are you?
>
```


A common naming convention for shell scripts is for them to have the extension `.sh`, and all our shell scripts will follow this convention. This has the advantage that editors like gedit will automatically recognise our files as shell scripts and highlight them appropriately.

The name of the shell script we are running is "`hello.sh`". Since it is in the current directory, we can tell the shell to execute it by typing "`./`" in front of its name, as shown on this slide. This basically means "execute the file `hello.sh` that is to be found in the current directory" – if there is no file of that name in the current directory, the shell returns a "`No such file or directory`" error. It is useful to know how to use "`./`" for two reasons:

1) If you ask the shell to run a program by just typing the name of the program and pressing return, it looks for the program in all the directories specified in the `PATH` *environment variable* (more on environment variables later). If the current directory isn't one of those specified in the `PATH` environment variable, then it wouldn't find the `hello.sh` that we want it to execute. By explicitly telling the shell to look in the current directory, it finds the `hello.sh` that we are looking for.
2) There might be another program called "`hello.sh`" in a directory that is specified in the `PATH` environment variable. The shell looks for programs to execute in the directories specified in the `PATH` environment variable *in the order they are specified in that environment variable*. It then executes the ***first*** program it finds that matches the name given. So if there was a file called "`hello.sh`" in some other directory specified in the `PATH` environment variable, then that might be executed instead.

You can achieve the same effect by asking the shell to run a program and giving it the path to the program, e.g. if `hello.sh` was in the directory `/home/x241`, then typing:

```
/home/x241/hello.sh
```

and pressing return would execute `hello.sh`.

Examining hello.sh

```
> ls -l hello.sh
-rwxr-xr-x  1 x241 x241 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: "`-rwxr-xr-x`"), 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 ("`x241`") and group ("`x241`") 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, 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).

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.)

If the shell is started with the `-x` option, it prints commands and their arguments as they are executed. You can also get this behaviour by using the `set` shell builtin command like this:

```
set -x
```

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.

Automating repetitive tasks (2)

How do I do this?:

1. Change into my program directory:

```
> cd ~/source
```

2. Create a backup directory:

```
> mkdir ~/backup
```

3. Move editor backups to backup directory:

```
> mv *~ ~/backup
> mv *.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 *.o**

If I put those commands together…:

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

```
> chmod +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 ~/source
mkdir -p ~/backup
mv *~ ~/backup
mv *.bak ~/backup
rm -f *.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 program directory
cd ~/source
# Make backup directory
mkdir -p ~/backup
# Move editor backups to backup dir
mv *~ ~/backup
mv *.bak ~/backup
# Delete compiler object files
rm -f *.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.)

First exercise (1)

We have a program, `iterator`, that takes four 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 `iterator` 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 `iterator` 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.

Know Your Enemy (1)

```
> ./iterator
Wrong number of arguments!
4 expected, 0 found.
Usage: ./iterator Nx Ny n_iterations epsilon

> ls
hello.sh  iterator

> ./iterator 10 10 100 0.1
x dimension of grid:    10
y dimension of grid:    10
Number of iterations:   100
Epsilon:                0.100000

Output file:            output.dat

Iterations took 0.000 seconds
```


The `iterator` program, which is located in your home directory, takes 4 numeric arguments: 3 positive integers and 1 floating-point number. It always writes its output to a file called `output.dat` in the current working directory, and also writes some informational messages to the screen, which we'll ignore for now.

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.

Know Your Enemy (2)

```
> ls
hello.sh  iterator  output.dat  running
> ./iterator 10 10 100 0.2
Already running in this directory!
> rm running
> ./iterator 10 10 100 0.2
x dimension of grid:    10
y dimension of grid:    10
Number of iterations:   100
Epsilon:                0.200000

Output file:            output.dat

Iterations took 0.000 seconds
```


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 and there may be additional files and/or directories shown.

The `iterator` program is not as well behaved as we might like: 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.

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

First exercise (2)

What we want to do is:

1. Delete any file called `running`

```
> rm running
```

2. Run `iterator` with some parameters

```
> ./iterator 100 100 1000 0.05
```

3. Rename `output.dat`

```
> mv output.dat output-0.05.dat
```

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

```
100 100 1000 0.1
100 100 1000 0.15
```


Your task is to create a simple shell script that does the above task. Basically, you want to run the `iterator` 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
myPROGS=~/source
myBACKUPS=~/backup
# Change to my program directory
cd "${myPROGS}"
# Make backup directory
mkdir -p "${myBACKUPS}"
# Move editor backups to backup dir
mv *~ "${myBACKUPS}"
mv *.bak "${myBACKUPS}"
# Delete compiler object files
rm -f *.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. (Whilst in this shell script I only use the location of my program directory in one place, I use the location of the backup directory in three places, and previously I would have needed to change it in all three places if I decided to store my backups somewhere else.)

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.

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

Environment Variables (1)

```
> env
LESSKEY=/etc/lesskey.bin
INFODIR=/usr/local/info:/usr/share/info:/usr/info
.
.
.

> set
ACLOCAL_FLAGS='-I /opt/gnome/share/aclocal'
BASH=/bin/bash
.
.
.

> set | more
ACLOCAL_FLAGS='-I /opt/gnome/share/aclocal'
BASH=/bin/bash
.
.
.
```


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 in the "Unix Systems: Shell Scripting (II)" 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.

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 first)

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

```
> ~/examples/params.sh 0.5 62 38 hydrogen
This script is /home/x241/examples/params.sh

There are 4 command line parameters.

The first command line parameter is: 0.5
The second command line parameter is: 62
The third command line parameter 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 file
rm -f running
# Run iterator with passed arguments
./iterator "${@}"
# Remove left over running file
rm -f running
# Rename output file
mv output.dat "output-${4}.dat"
```


The file `run-once.sh` in the `scripts` directory (shown above) accepts some command line arguments and then tries to run the `iterator` program with them. (Note that it does no checking of the arguments it is given whatsoever.) On the assumption that only the fourth 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 100 100 1000 0.05
```

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. This is equivalent to using `1>filename` which means "redirect file descriptor 1 (*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.

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
```

Using redirection

```
#!/bin/bash
# Remove left over running file
rm -f running
# Run iterator with passed arguments
./iterator "${@}" > "stdout-${4}"
# Remove left over running file
rm -f running
# Rename output file
mv output.dat "output-${4}.dat"
```


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 `iterator` program outputs to the screen (standard output) as well (hurrah!). Change to your home directory and try this:

```
scripts/run-once.sh 100 100 1000 0.05
```

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.

Keeping a record

```
#!/bin/bash
# Remove left over running file
rm -f running
# Write to logfile
echo "" >> logfile
date >> logfile
echo "Running iterator with ${@}" >> logfile
# Run iterator with passed arguments
./iterator "${@}" > "stdout-${4}"
# Remove left over running file
rm -f running
# Rename output file
mv output.dat "output-${4}.dat"
# Write to logfile
echo "Output file: output-${4}.dat" >> logfile
echo "Standard output: stdout-${4}" >> 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 `iterator` 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 100 100 1000 0.05
cat logfile
```

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 element 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 (e.g. `*`) to specify a collection of file names (*pathname expansion*). *<some commands>* is a list of one or more commands to be executed.

There are some examples of how to use it in the `for.sh` in the `examples` directory of your home directory.

Multiple runs

```
#!/bin/bash

# Parameters that stay the same each run
myFIXED_PARAMS="100 100 1000"

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

> cd
> rm -f *.dat stdout-* logfile
> scripts/multi-run.sh 0.05 0.1 0.15 0.2 0.25 0.3 0.35 0.4 0.45 0.5
```


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 `iterator` program) with 4 arguments – 3 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 3 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 0.05 0.1 0.15 0.2 0.25 0.3 0.35 0.4 0.45 0.5
more logfile
```

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

Final exercise (1)

We have a directory that contains the output of several runs of the `iterator` 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.

We are specifically using the `gnuplot` program and the output of the `iterator` 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/

Let's take a closer look… (1)

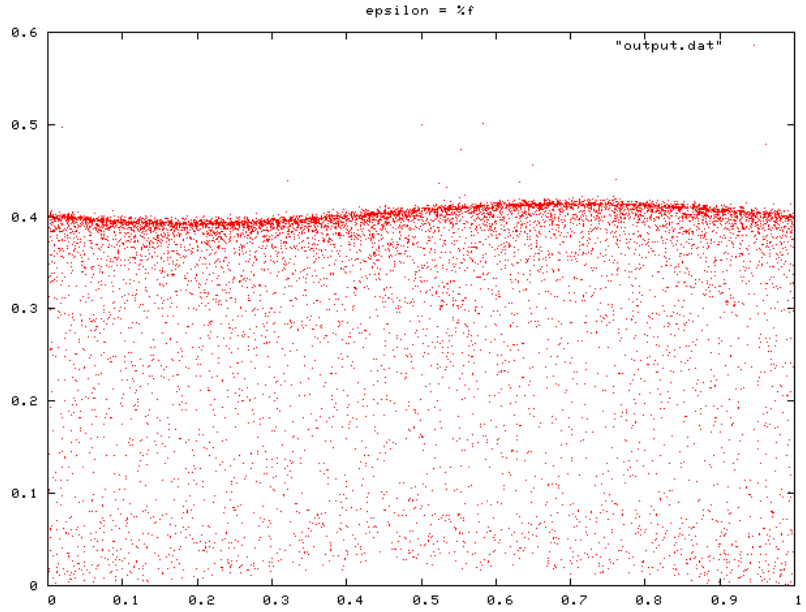
```
> cd
> cp gnuplot/iterator.gplt .
> cp output-0.05.dat output.dat

> ls output.png
/bin/ls: output.png: No such file or directory

> gnuplot iterator.gplt
> rm output.dat
> ls output.png
output.png
> eog output.png &
```


Note that the output of "`ls output.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 `output.dat`

```
> mv output-0.05.dat output.dat
```

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

```
> gnuplot iterator.gplt
```

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

```
> mv output.dat output-0.05.dat
```

4. Rename `output.png`

```
> mv output.png output-0.05.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 `iterator.gplt` file you've been given will only work if the `.dat` file is called `output.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.

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 `for.sh` file in the `examples` directory.