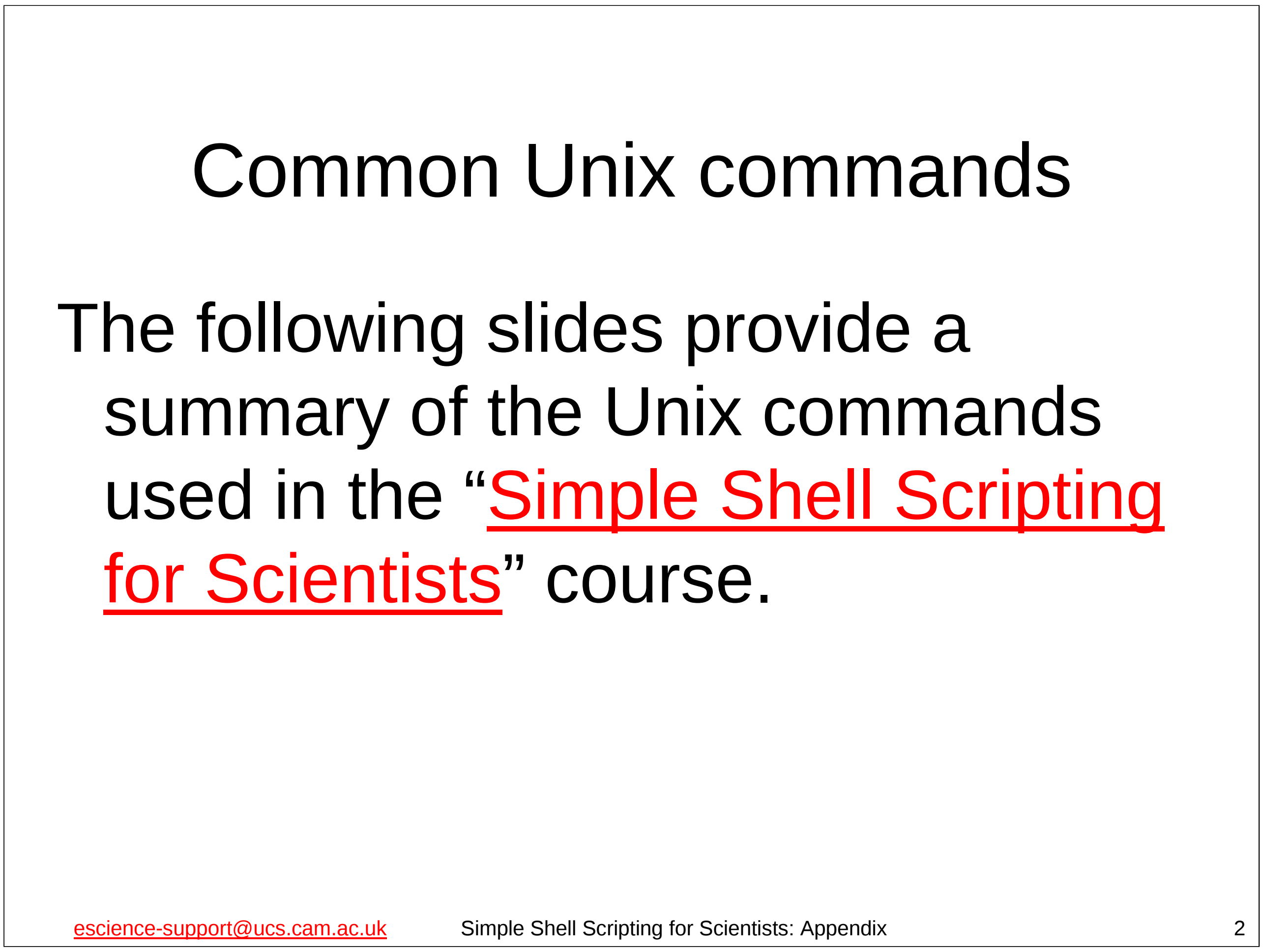
Simple Shell Scripting for Scientists

Appendix

Julian King

Bruce Beckles

University of Cambridge Computing Service



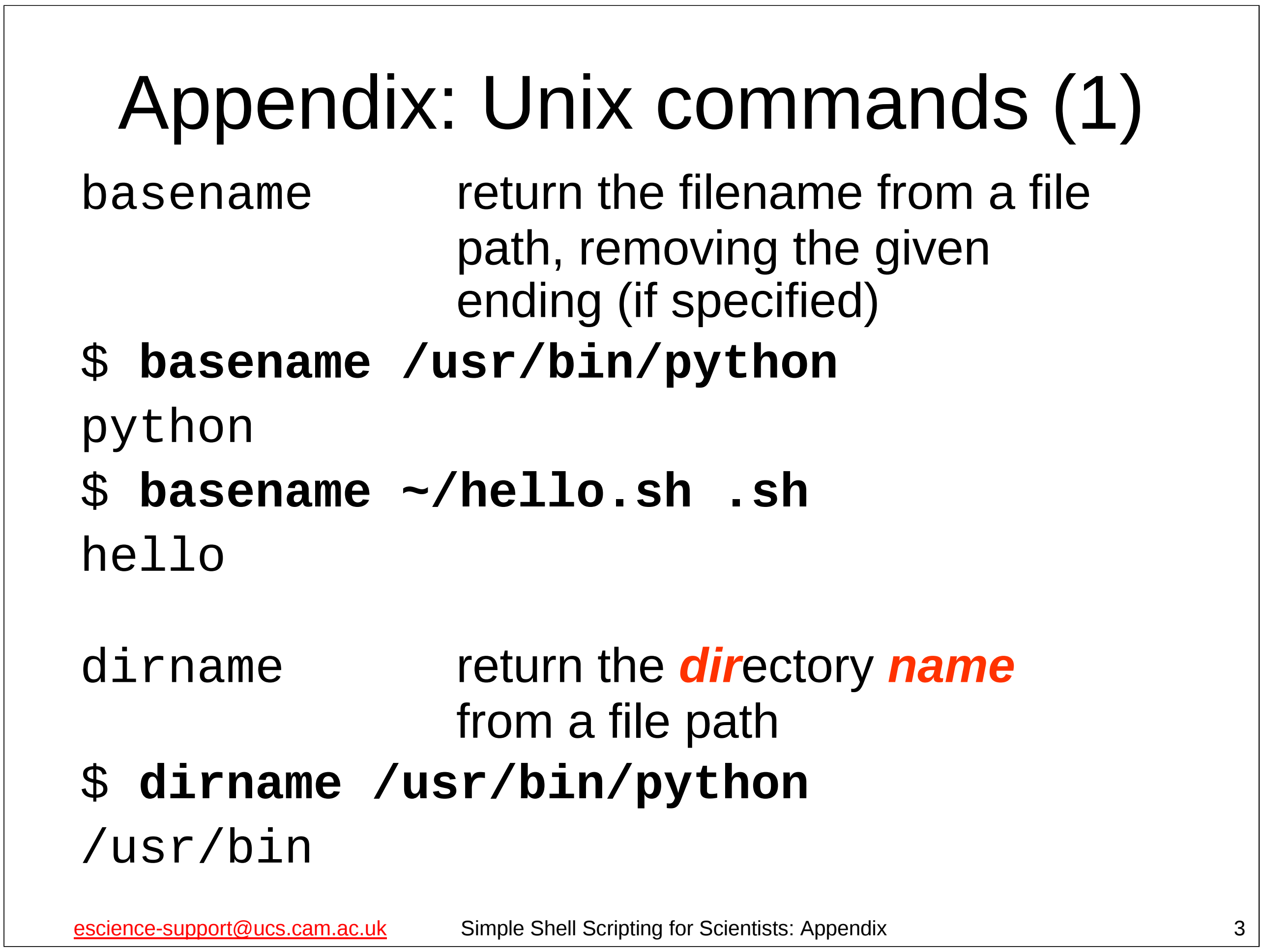
Common Unix commands

The following slides provide a summary of the Unix commands used in the "Simple Shell Scripting for Scientists" course.

escience-support@ucs.cam.ac.uk Simple Shell Scripting for Scientists: Appendix 2

For details of the "Unix: Simple Shell Scripting for Scientists" course, see:

http://training.csx.cam.ac.uk/course/scriptsci



Appendix: Unix commands (1)

`basename`	return the filename from a file path, removing the given ending (if specified)

```
$ basename /usr/bin/python
python
$ basename ~/hello.sh .sh
hello
```

`dirname`	return the ***dir***ectory ***name*** from a file path

```
$ dirname /usr/bin/python
/usr/bin
```

If you have a path to a file, **`dirname`** will give you just the directory, removing the actual filename whilst **`basename`** will give you the filename, removing the directory path. **`basename`** can also remove the endings of filenames.

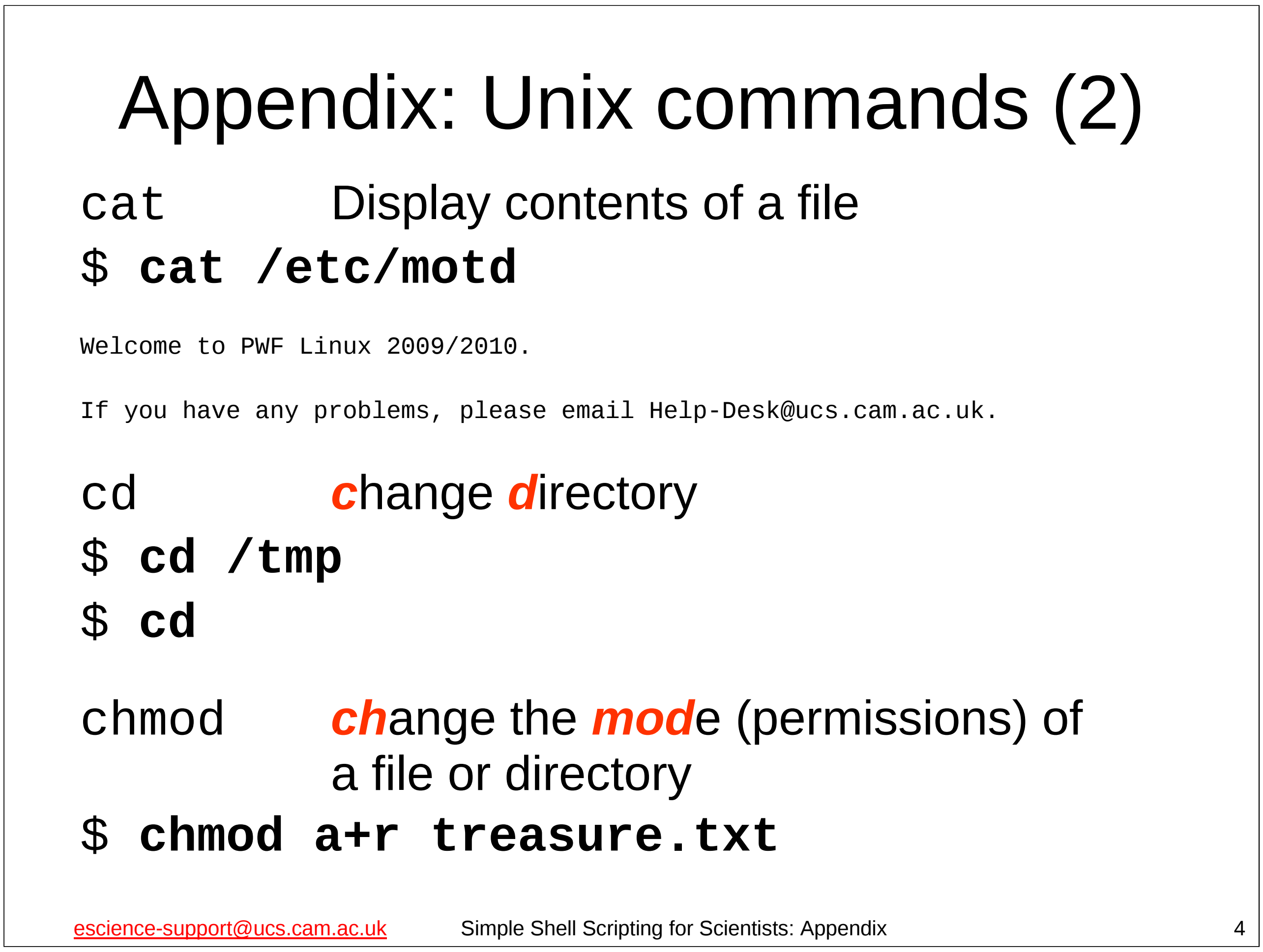
If you need to do more advanced filename (or file) manipulation, then you should look at the **`find`** and **`xargs`** commands. The **`find`** command is covered in the "Unix Systems: Further Commands" course, the notes for which are available here:

http://www-uxsup.csx.cam.ac.uk/courses/Commands/

The **`find`** command searches for files in a directory tree, and having found the specified files, can run a command on each file.

The **`xargs`** command builds a command line from a combination of values read from standard input and arguments specified on the command line, and then executes that command line a certain number of times. You can find out more about **`xargs`** from its man page:

```
man xargs
```



Appendix: Unix commands (2)

`cat` Display contents of a file

```
$ cat /etc/motd

Welcome to PWF Linux 2009/2010.

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
```

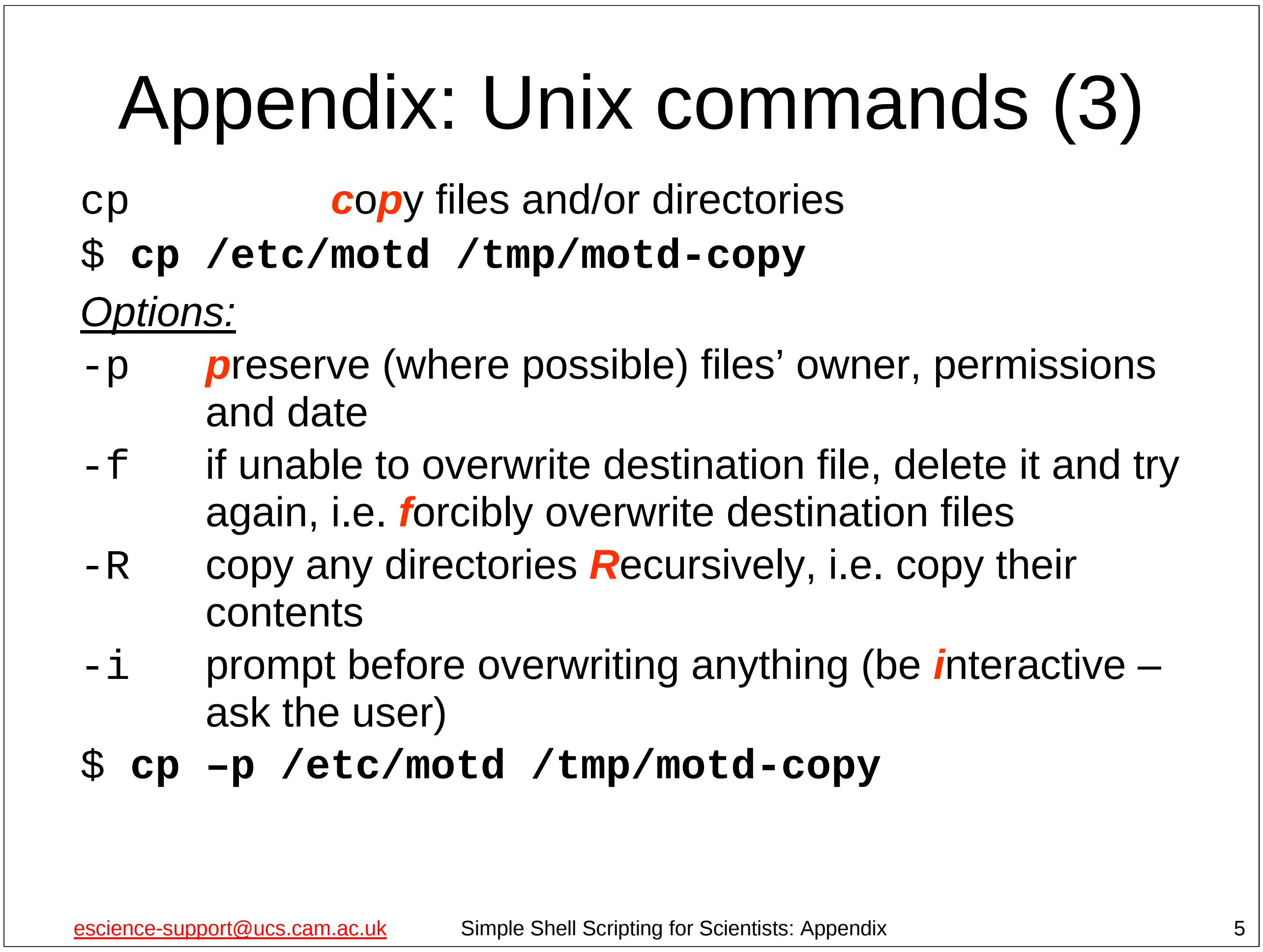
If you give the **`cd`** command without specifying a directory then it will change the directory to your *home directory* (the location of this directory is specified in the **`HOME`** environment variable).

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: Introduction to the Command Line Interface" course, see:

http://training.csx.cam.ac.uk/course/unixintro1

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

http://www-uxsup.csx.cam.ac.uk/courses/UnixCLI/



Appendix: Unix commands (3)

`cp` — **co**py files and/or directories

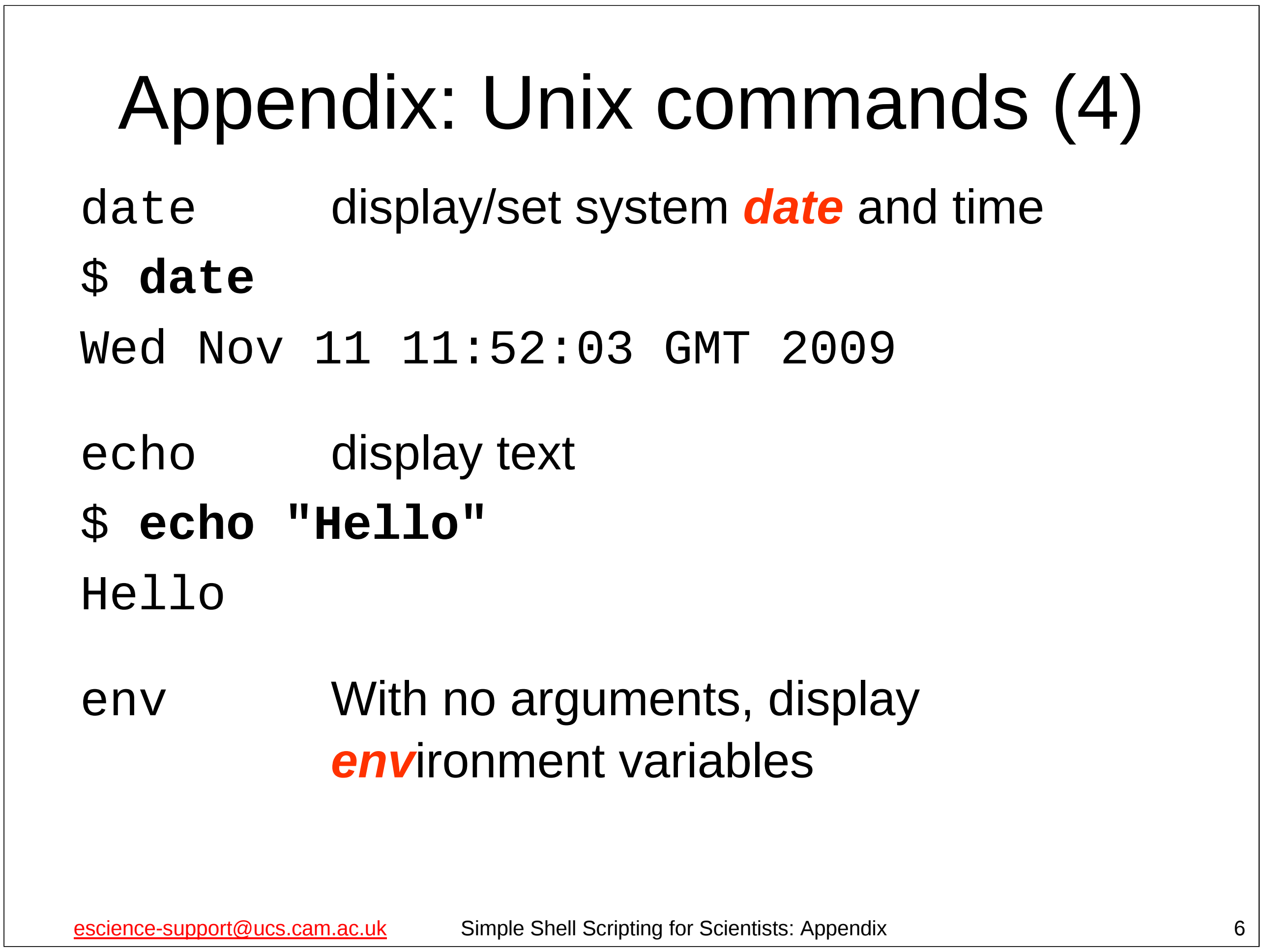
```
$ cp /etc/motd /tmp/motd-copy
```

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 /etc/motd /tmp/motd-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.



Appendix: Unix commands (4)

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

```
$ date
Wed Nov 11 11:52:03 GMT 2009
```

`echo` display text

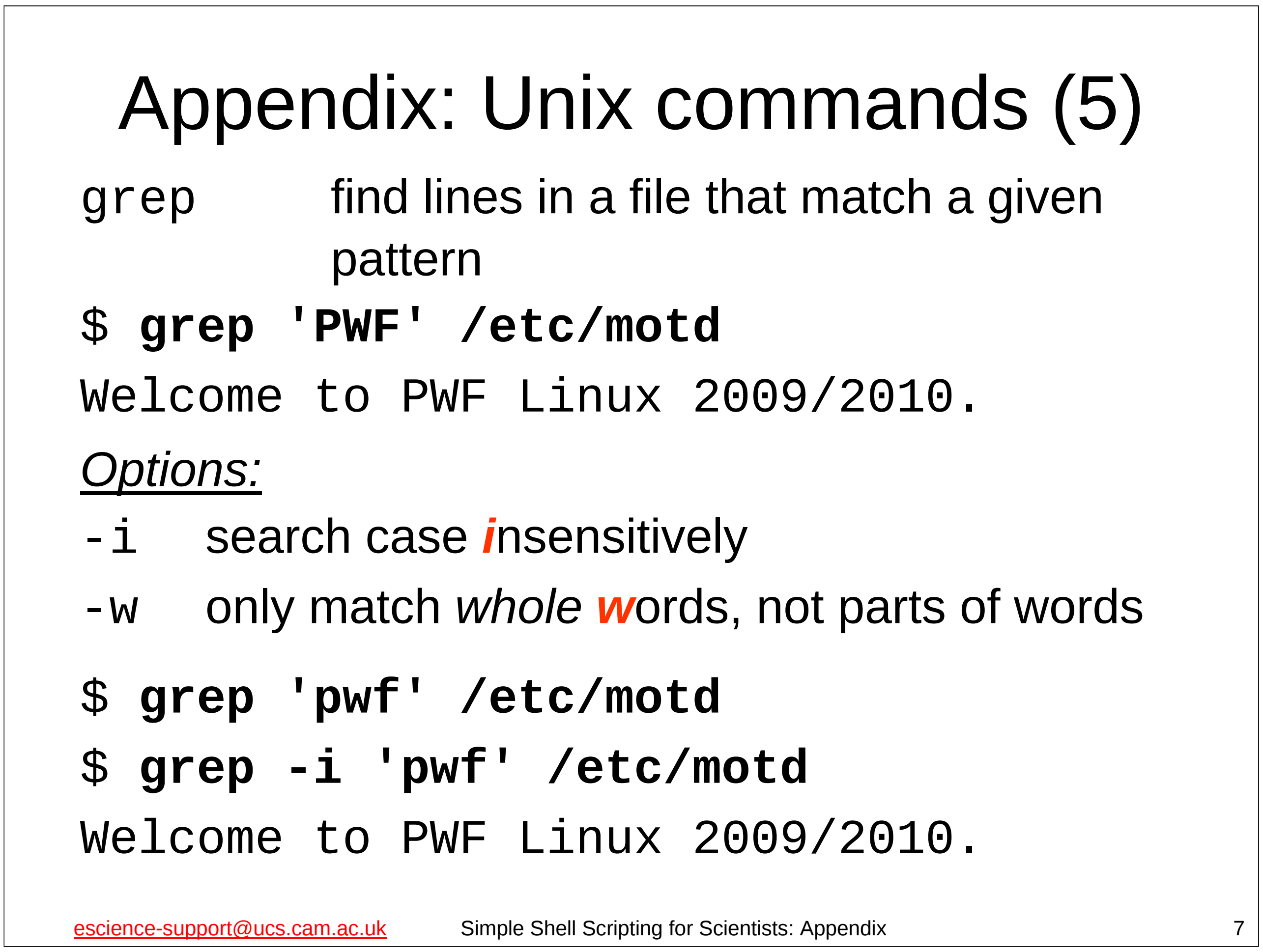
```
$ echo "Hello"
Hello
```

`env` With no arguments, display ***env***ironment variables

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 or you have fallen into a worm-hole in the space-time continuum). 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, so we don't discuss its other uses.



Appendix: Unix commands (5)

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

```
$ grep 'PWF' /etc/motd
Welcome to PWF Linux 2009/2010.
```

Options:

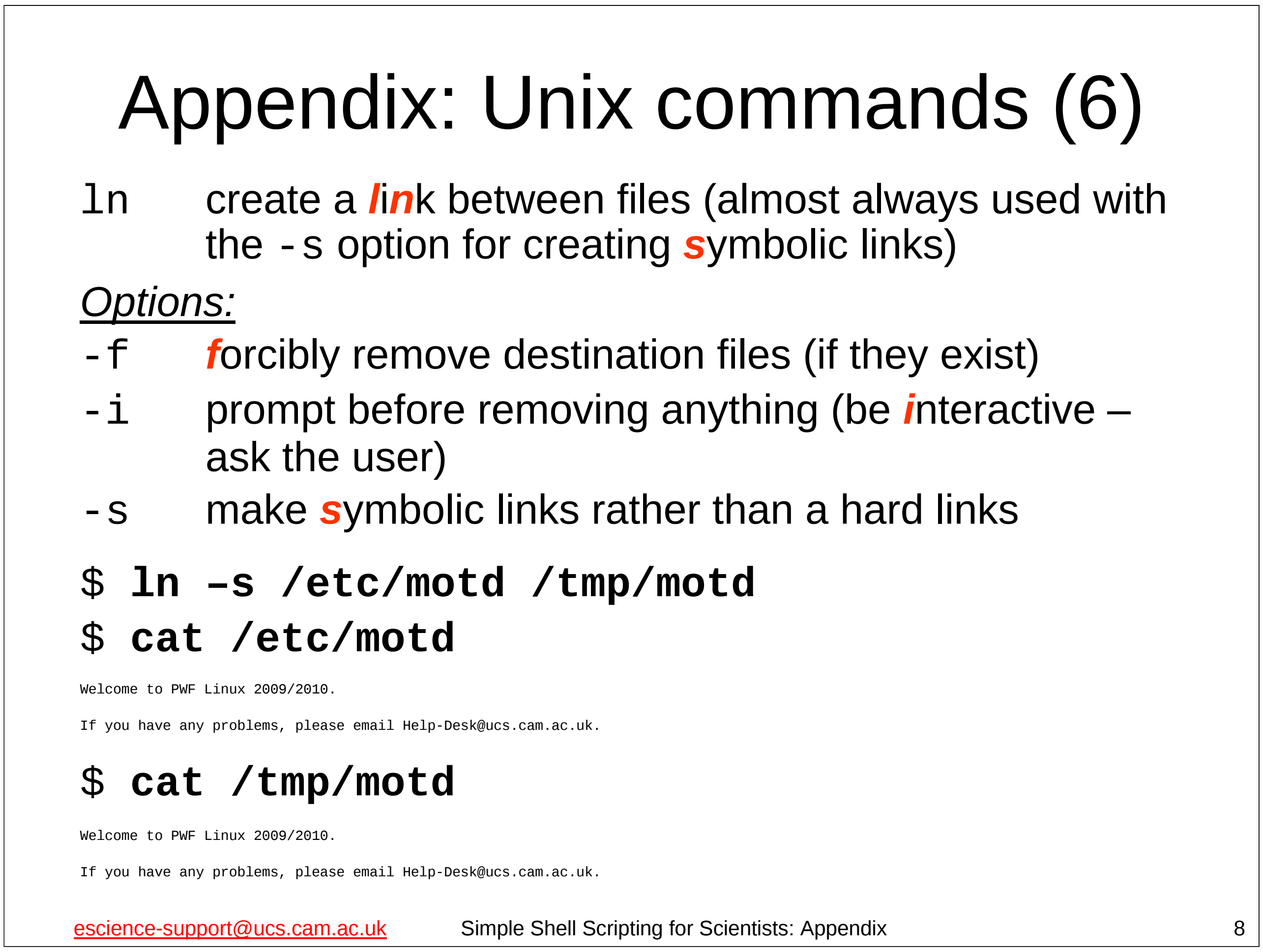
`-i`	search case ***i***nsensitively
`-w`	only match *whole* ***w***ords, not parts of words

```
$ grep 'pwf' /etc/motd
$ grep -i 'pwf' /etc/motd
Welcome to PWF Linux 2009/2010.
```

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://training.csx.cam.ac.uk/course/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.



Appendix: Unix commands (6)

`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

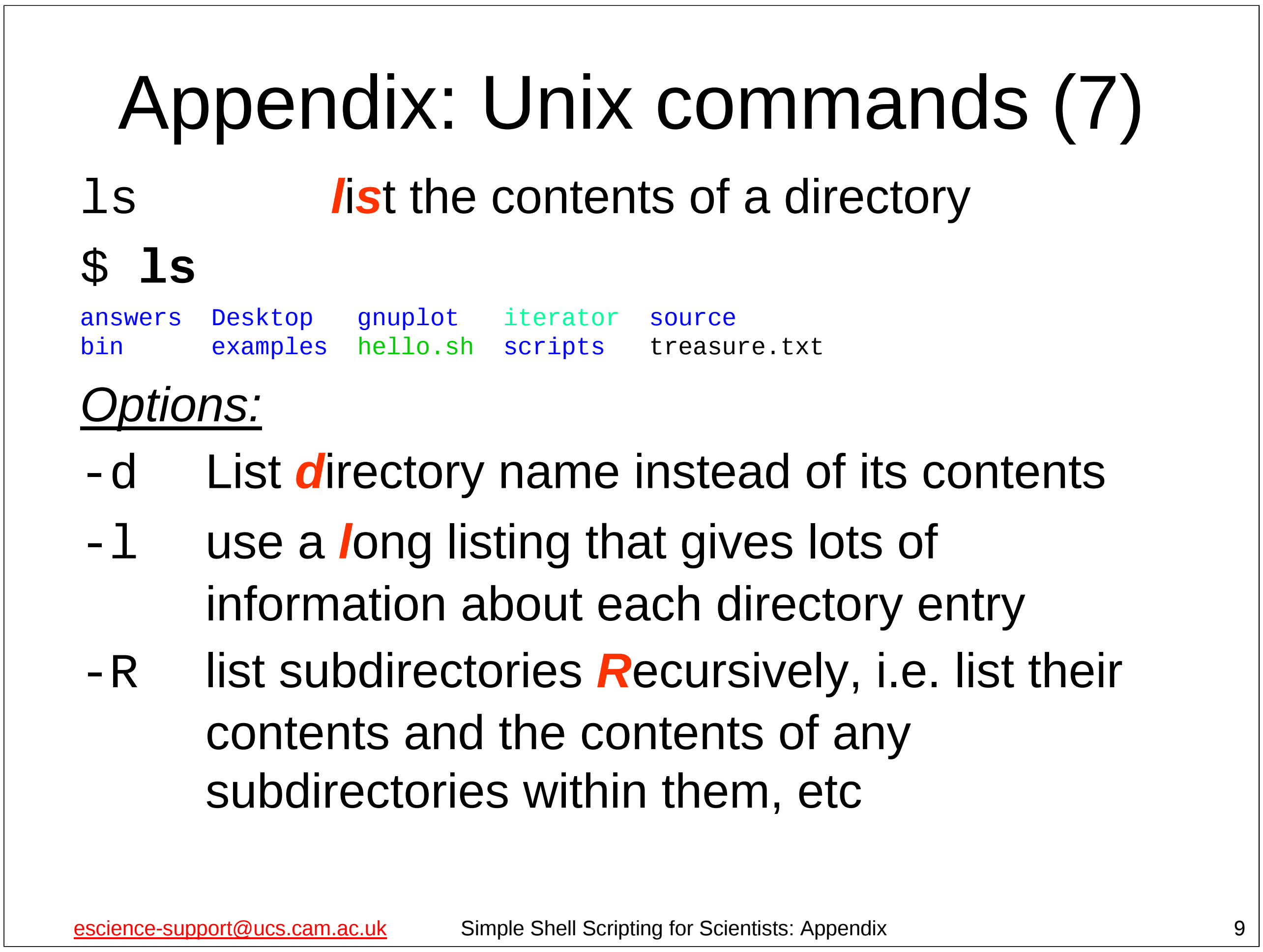
```
$ ln –s /etc/motd /tmp/motd
$ cat /etc/motd
Welcome to PWF Linux 2009/2010.

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

$ cat /tmp/motd
Welcome to PWF Linux 2009/2010.

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

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:

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

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



Appendix: Unix commands (7)

`ls` ***l****is*t the contents of a directory

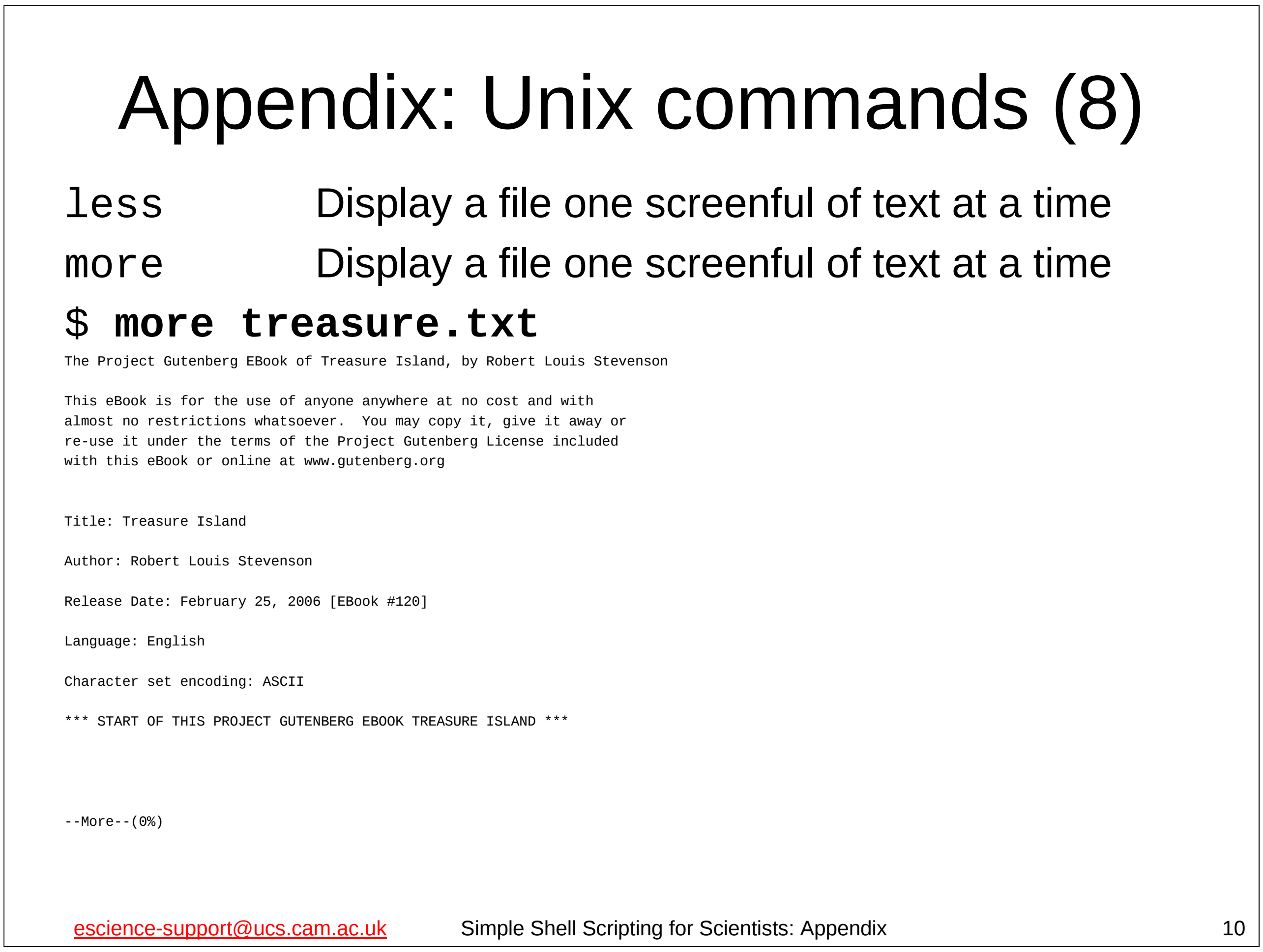
```
$ ls
answers  Desktop   gnuplot  iterator  source
bin      examples  hello.sh scripts   treasure.txt
```

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.



Appendix: Unix commands (8)

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

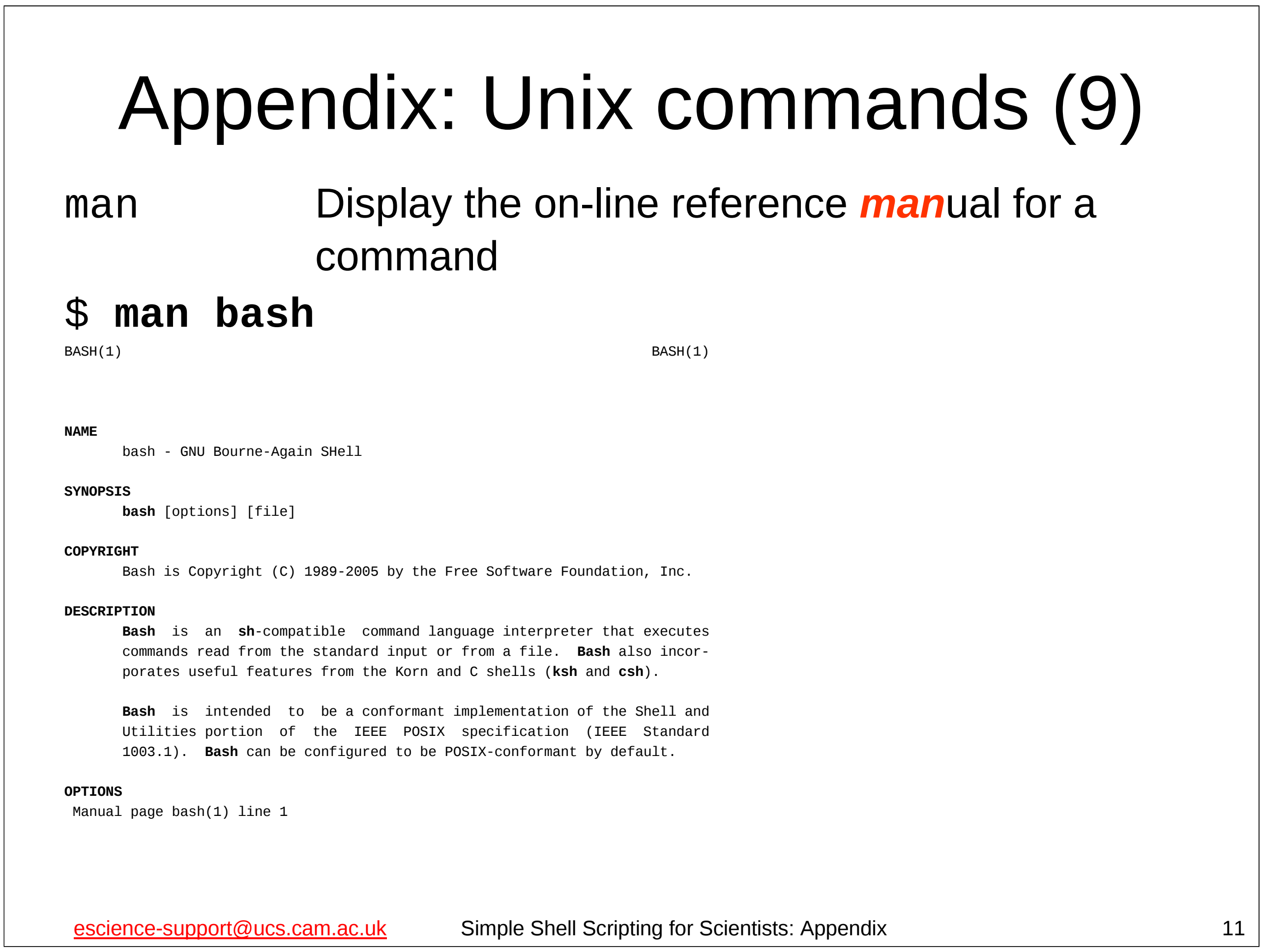
```
$ more treasure.txt
The Project Gutenberg EBook of Treasure Island, by Robert Louis Stevenson

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.



Appendix: Unix commands (9)

`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) 1989-2005 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
 Manual page bash(1) line 1
```

(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 back a screen type "b", and to quit **`man`** press the "`Q`" key.

Appendix: Unix commands (10)

`mkdir`   ***ma****k*e ***dir****ectories*

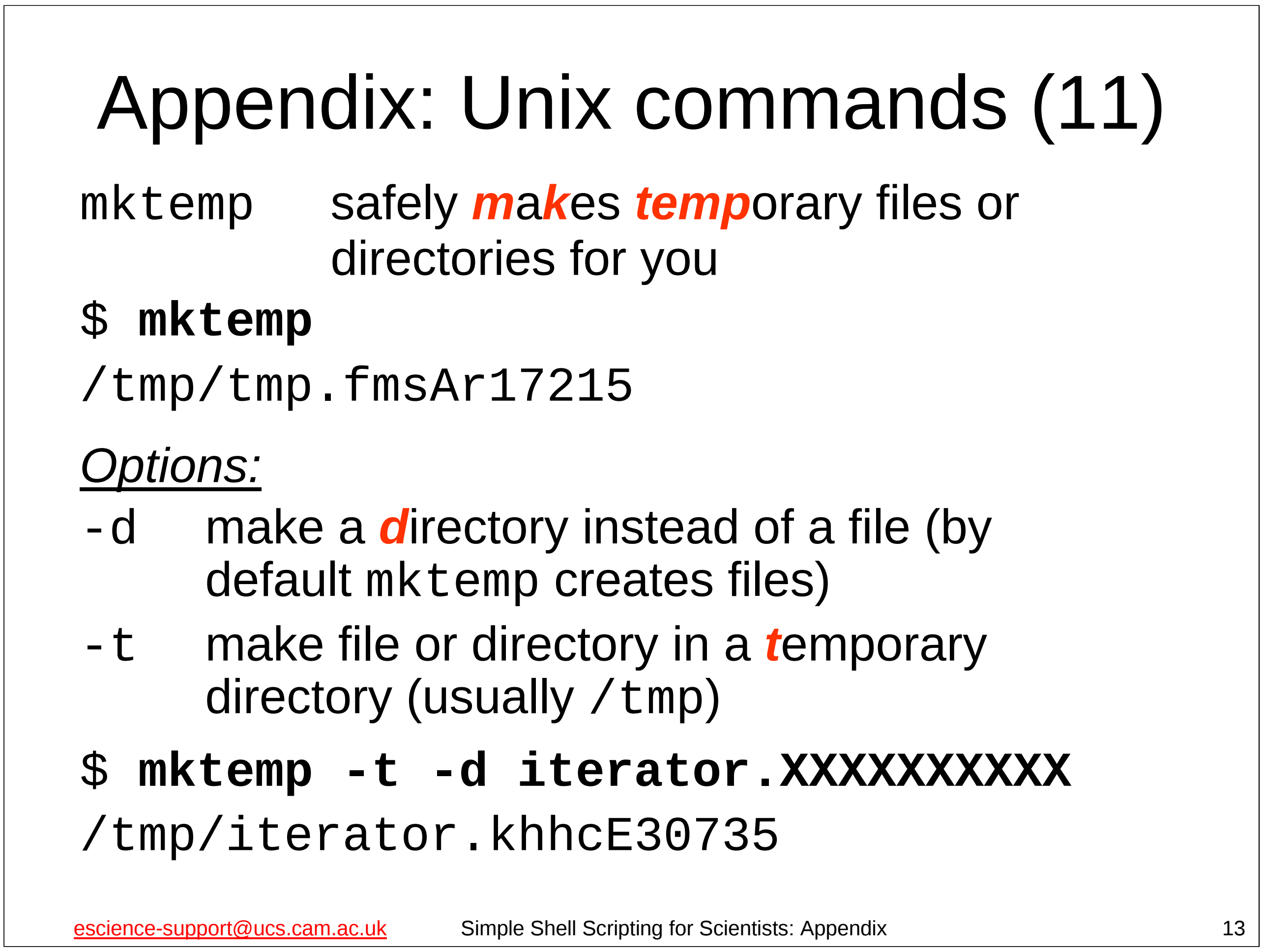
```
$ mkdir /tmp/mydir
```

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.



Appendix: Unix commands (11)

`mktemp` safely ***mak***es ***temp***orary files or directories for you

```
$ mktemp
/tmp/tmp.fmsAr17215
```

Options:

`-d`	make a ***d***irectory instead of a file (by default `mktemp` creates files)
`-t`	make file or directory in a ***t***emporary directory (usually `/tmp`)

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

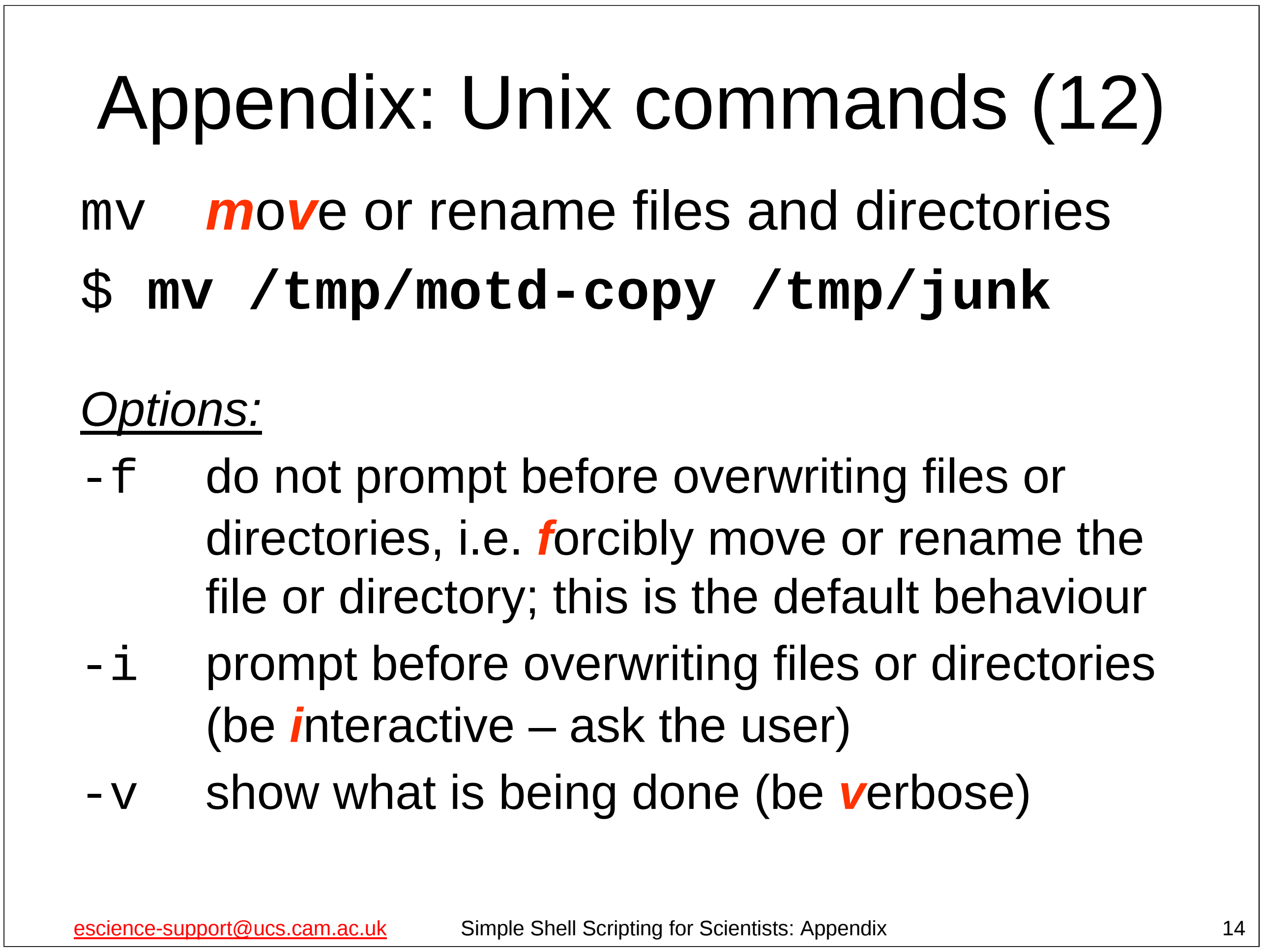
The **`mktemp`** command is an extremely useful command that allows users to *safely* create temporary files or directories on multi-user systems. It is very easy to ***un****safely* create a temporary file or directory to work with from a shell script, and, indeed, if your shell script tries to create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

Note that if you try the examples above you will almost certainly get files and directories with different names created for you.

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

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

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



Appendix: Unix commands (12)

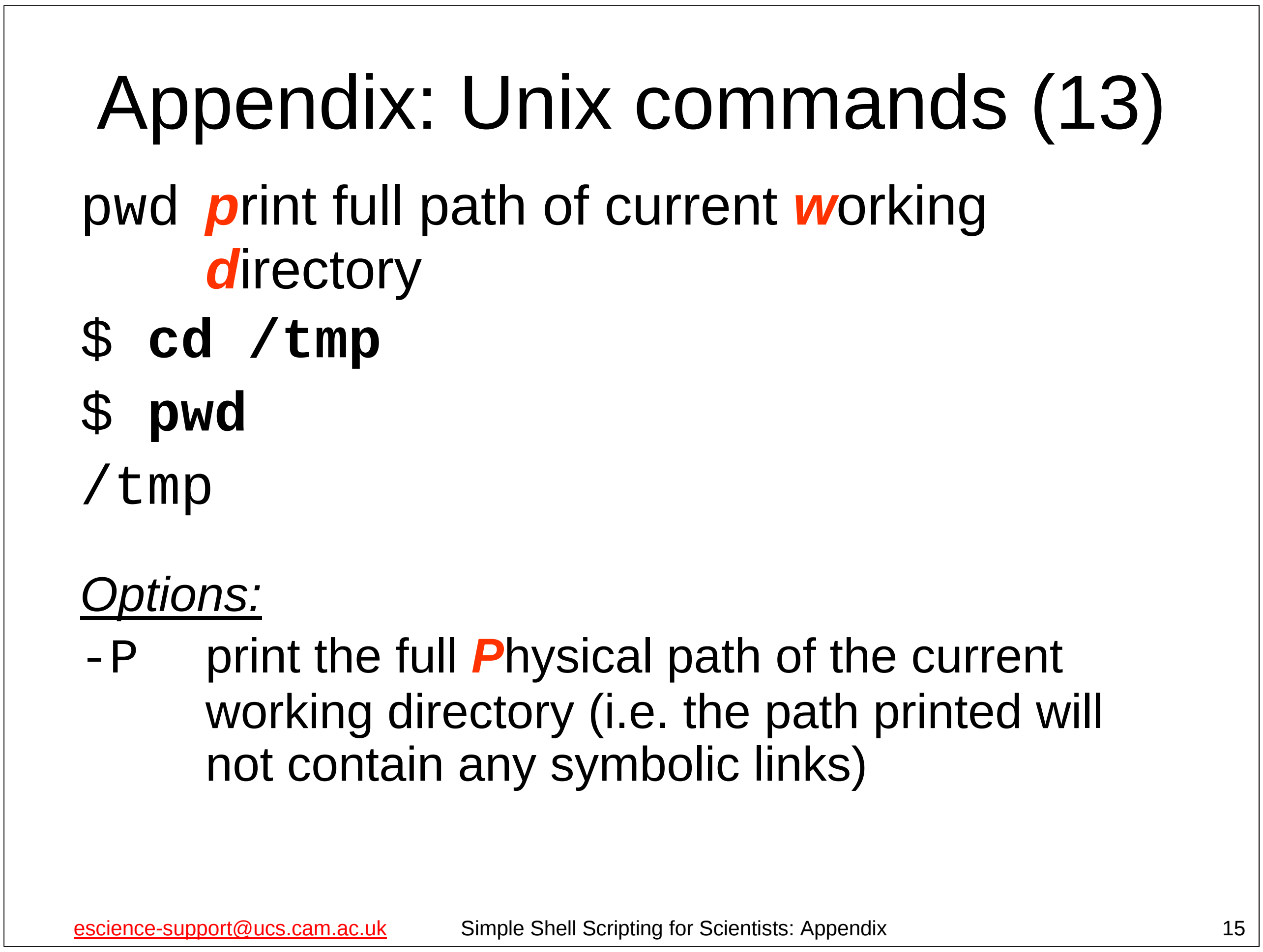
`mv`	***mo**v**e* or rename files and directories

```
$ 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, **`mv`** actually copies the file or directory to the other filesystem and then deletes the original.



Appendix: Unix commands (13)

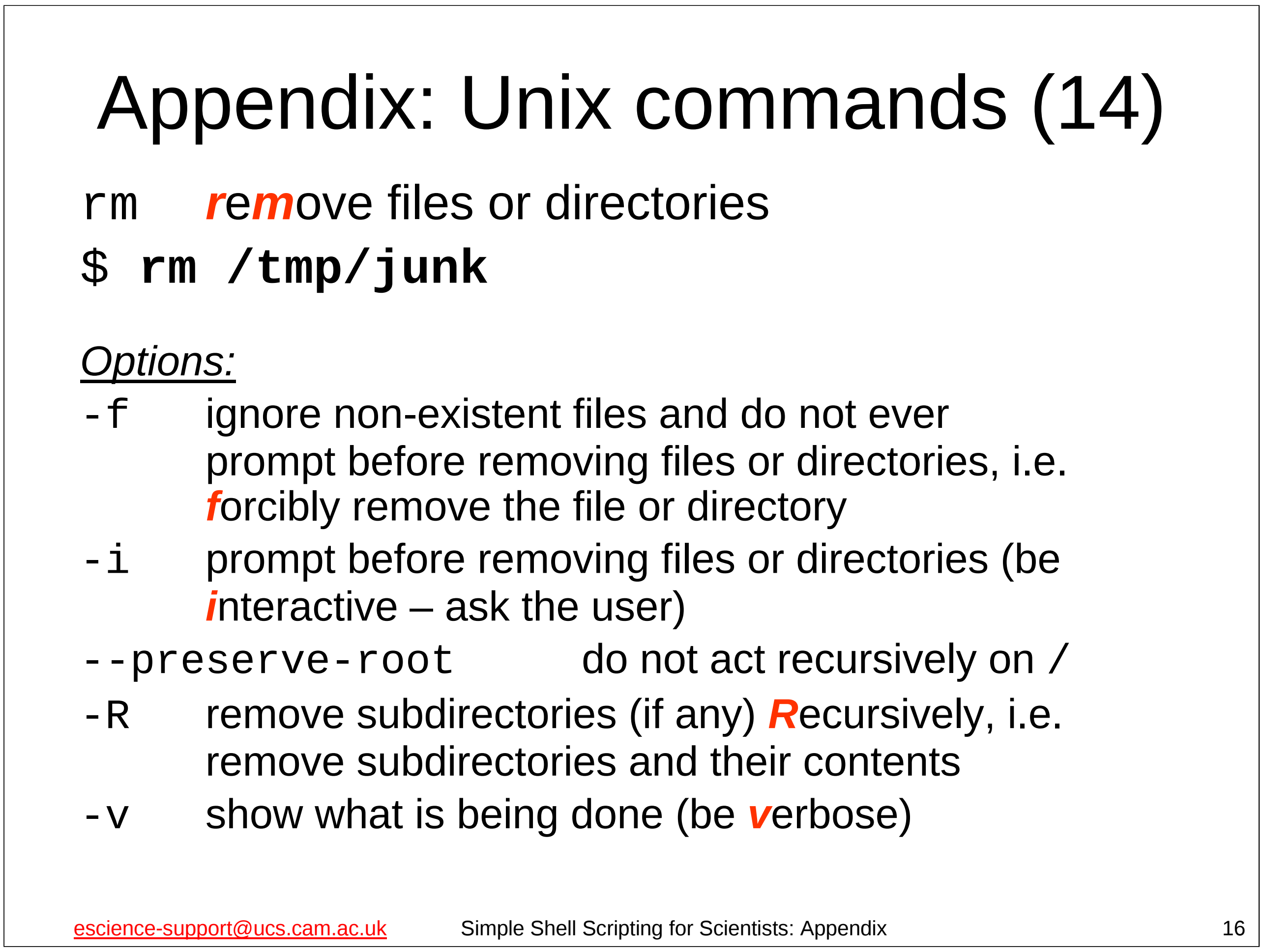
`pwd` ***p***rint full path of current ***w***orking ***d***irectory

```
$ 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.



Appendix: Unix commands (14)

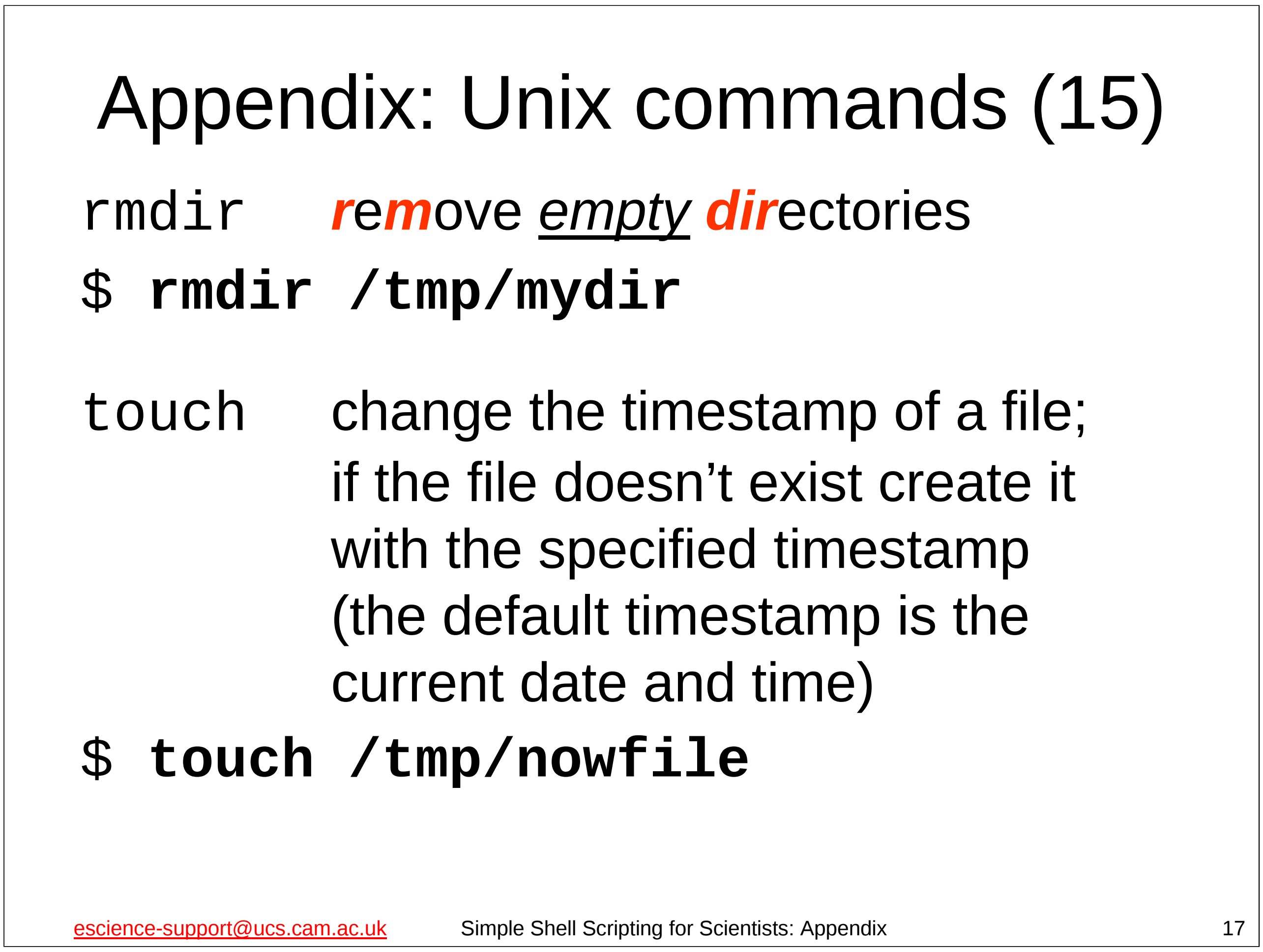
`rm` ***r***e***m***ove files or directories

```
$ 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.



Appendix: Unix commands (15)

`rmdir` ***re**move* *m*ove <u>*empty*</u> ***dir**ectories*

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

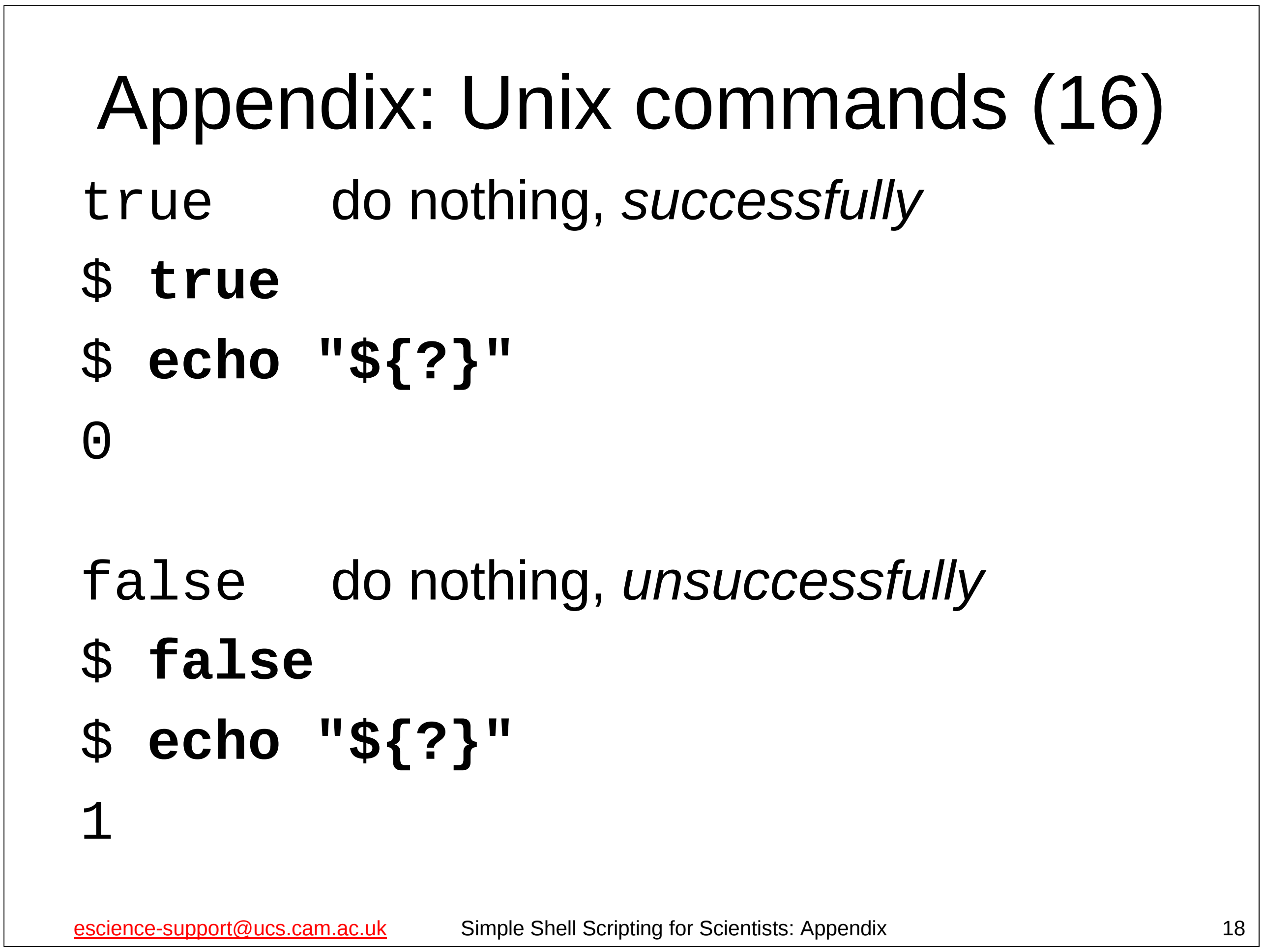
`touch` change the timestamp of a file; if the file doesn't exist create it with the specified timestamp (the default timestamp is the current date and time)

```
$ 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.



Appendix: Unix commands (16)

`true` do nothing, *successfully*

```
$ true
$ echo "${?}"
0
```

`false` do nothing, *unsuccessfully*

```
$ false
$ echo "${?}"
1
```

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

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

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

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