UNIVERSITY OF CAMBRIDGE

Simple Shell Scripting for Scientists

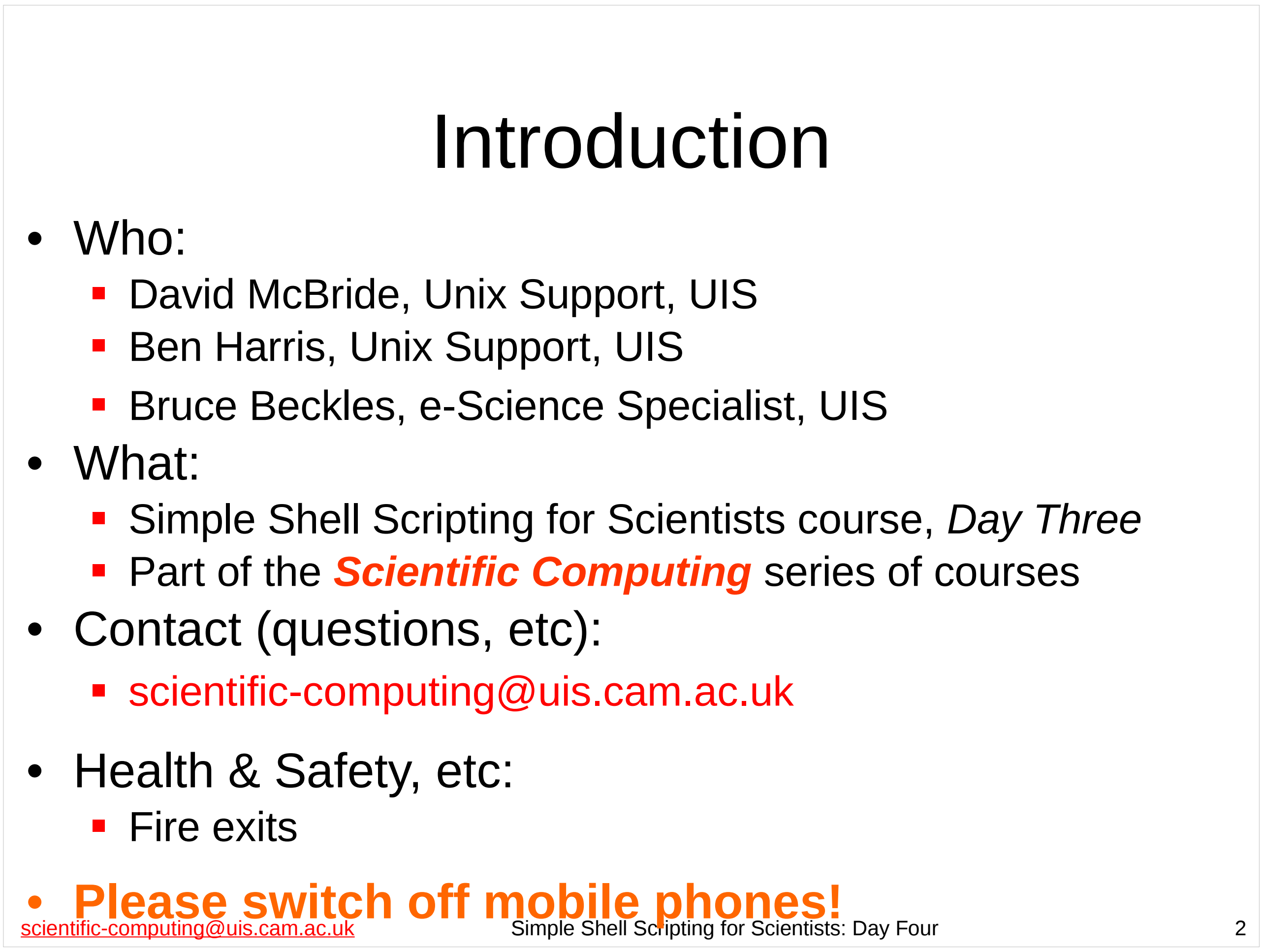
Day Four

David McBride

Ben Harris

University of Cambridge Information Services

Version: 2017-03-01



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 Three*
  - 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!**

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 2

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.

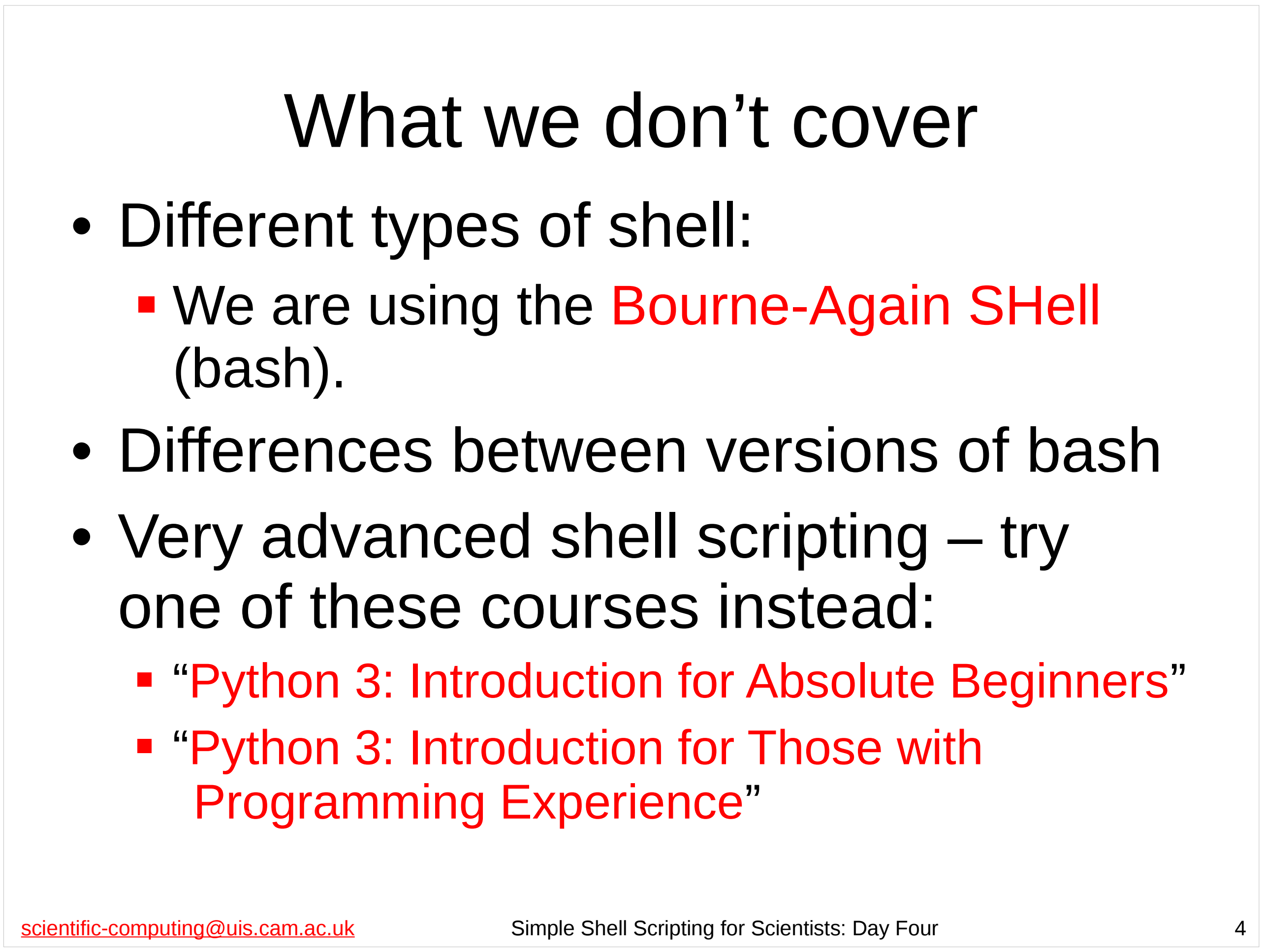
We finish at:

17:00

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 3

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 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.



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”

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 4

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

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

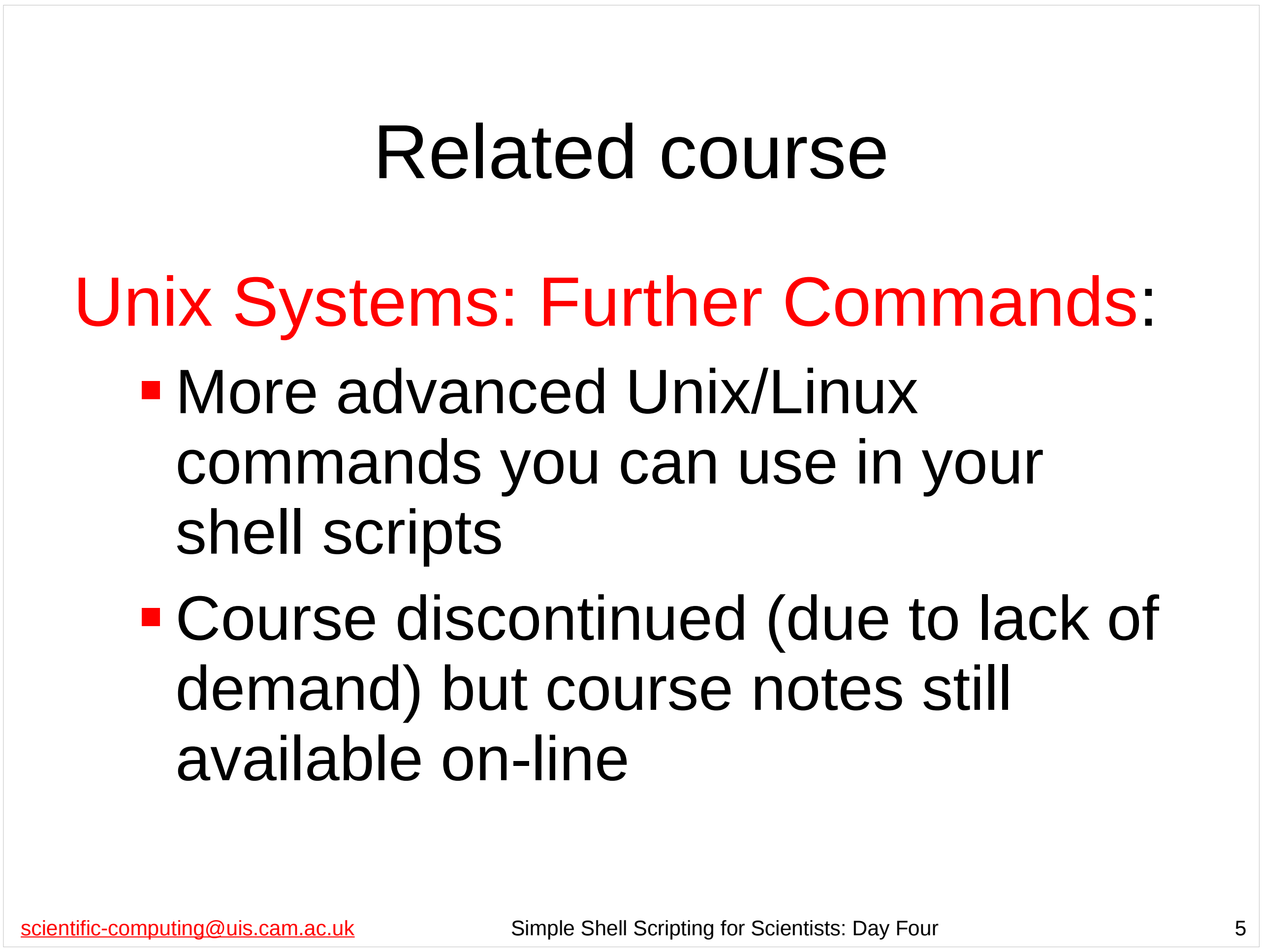
We will be using bash 4.4 in this course, but everything we do should work in bash 2.05 and later. Version 4, version 3 and version 2.05 (or 2.05a or 2.05b) are the versions of bash in most widespread use at present. Most recent Linux distributions will have one of these versions of bash as one of their standard packages. The latest version of bash (at the time of writing) is bash 4.4.5, which was released in November 2016.

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

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

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

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



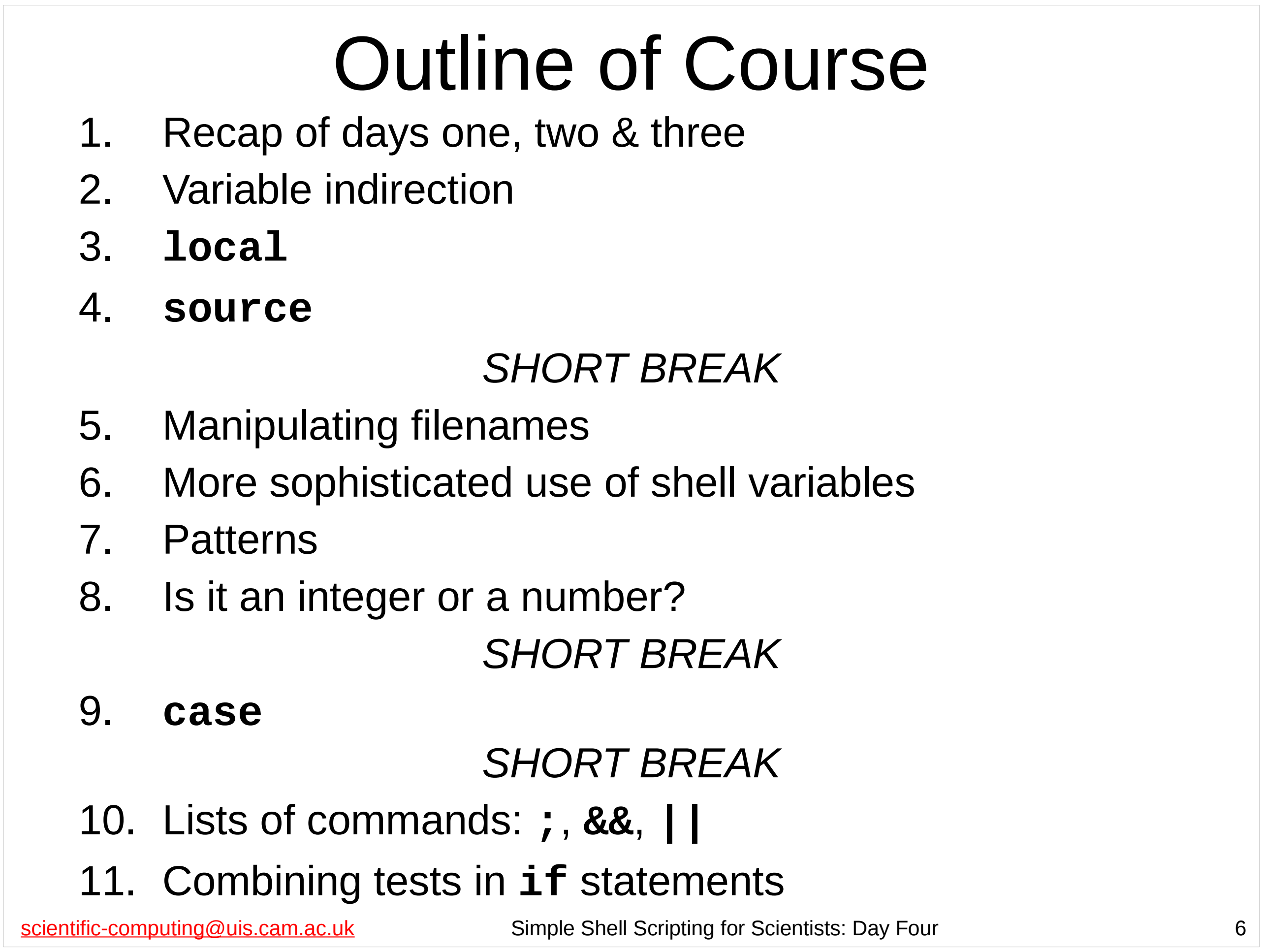
Related course

Unix Systems: Further Commands:

- More advanced Unix/Linux commands you can use in your shell scripts
- Course discontinued (due to lack of demand) but course notes still available on-line

For the course notes from the “Unix Systems: Further Commands” course, see:

http://help.uis.cam.ac.uk/help-support/training/downloads/
course-files/programming-student-files/commands



Outline of Course

1. Recap of days one, two & three
2. Variable indirection
3. **`local`**
4. **`source`**

SHORT BREAK

5. Manipulating filenames
6. More sophisticated use of shell variables
7. Patterns
8. Is it an integer or a number?

SHORT BREAK

9. **`case`**

SHORT BREAK

10. Lists of commands: **`;`**, **`&&`**, **`||`**
11. Combining tests in **`if`** statements

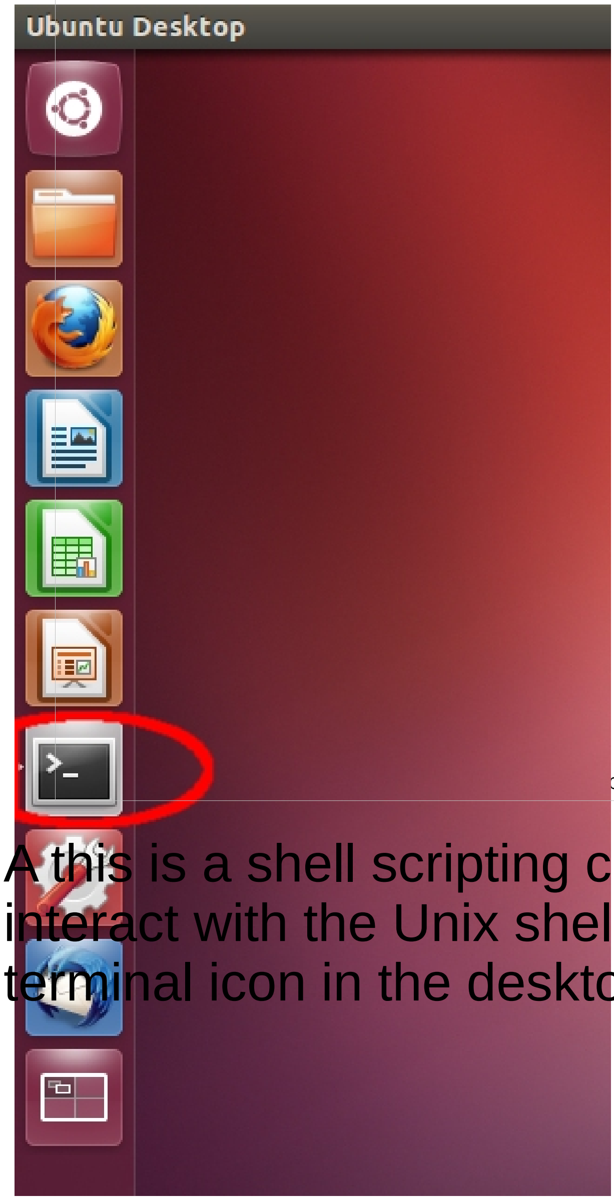
scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 6

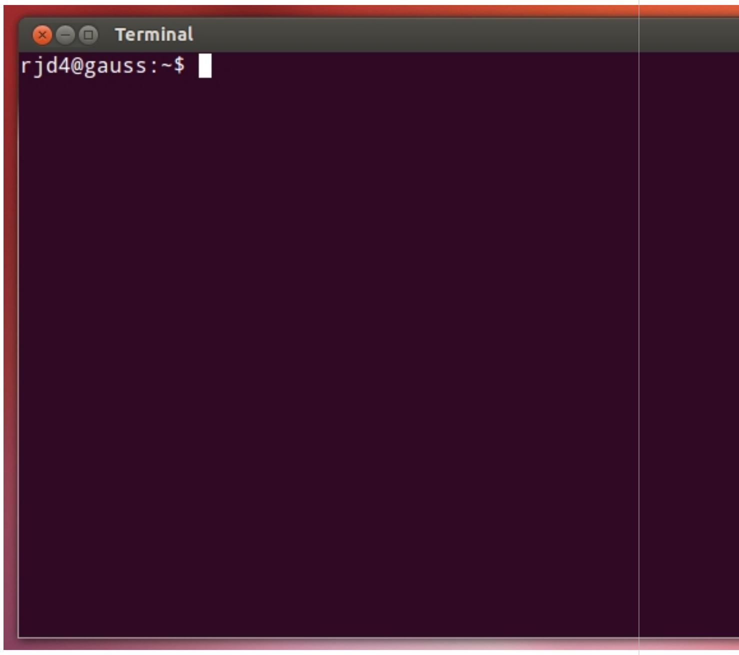
The course officially finishes at 17.00, but the intention is that the lectured part of the course will be finished by about 16.30 or soon after, and the remaining time is for you to ask questions about anything that is still puzzling you. If you need to leave before 17.00 (or even before 16.30), please do so, but don’t expect the course to have finished before then.

Before the end of today's session, 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/

Start a shell



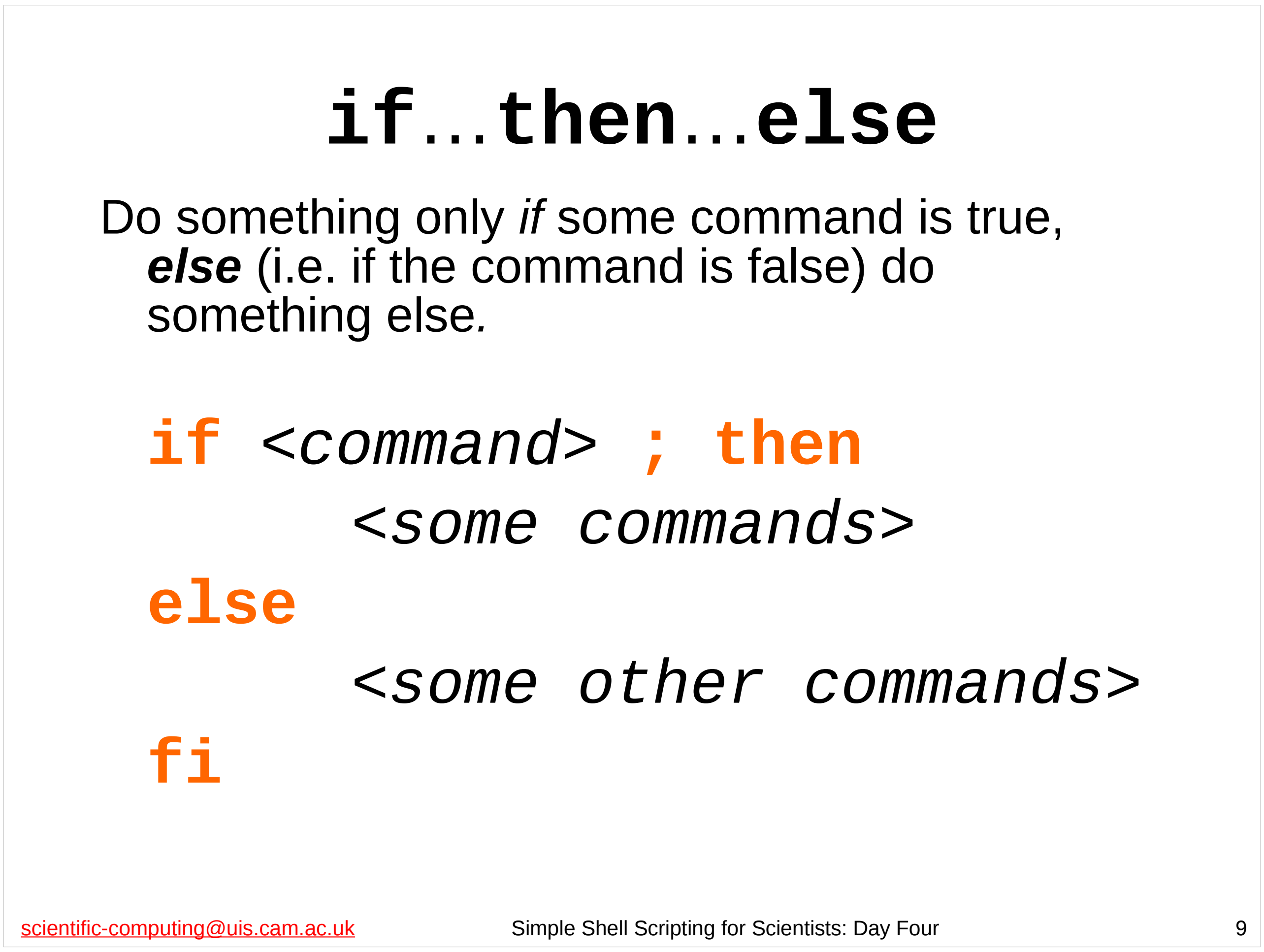


```
rjd4@gauss:~$
```

A this is a shell scripting course, we are going to need to interact with the Unix shell. To start a shell, click on the terminal icon in the desktop application bar.

Recap: Days One, Two & Three

- Shell scripts as linear lists of commands
- Simple use of shell variables and parameters
- Simple command line processing
- Shell functions
- Pipes and output redirection
- Accessing standard input using **`read`**
- **`for`** and **`while`** loops
- Tests
- Command substitution and (integer) arithmetic expansion
- The **`mktemp`** command
- **`if`** statement
- Error handling (including standard error and **`set -e`**, **`set +e`**)
- **`exit`** and **`return`** to quit scripts and functions



`if`…`then`…`else`

Do something only *if* some command is true, ***else*** (i.e. if the command is false) do something else.

```
if <command> ; then
        <some commands>
else
        <some other commands>
fi
```

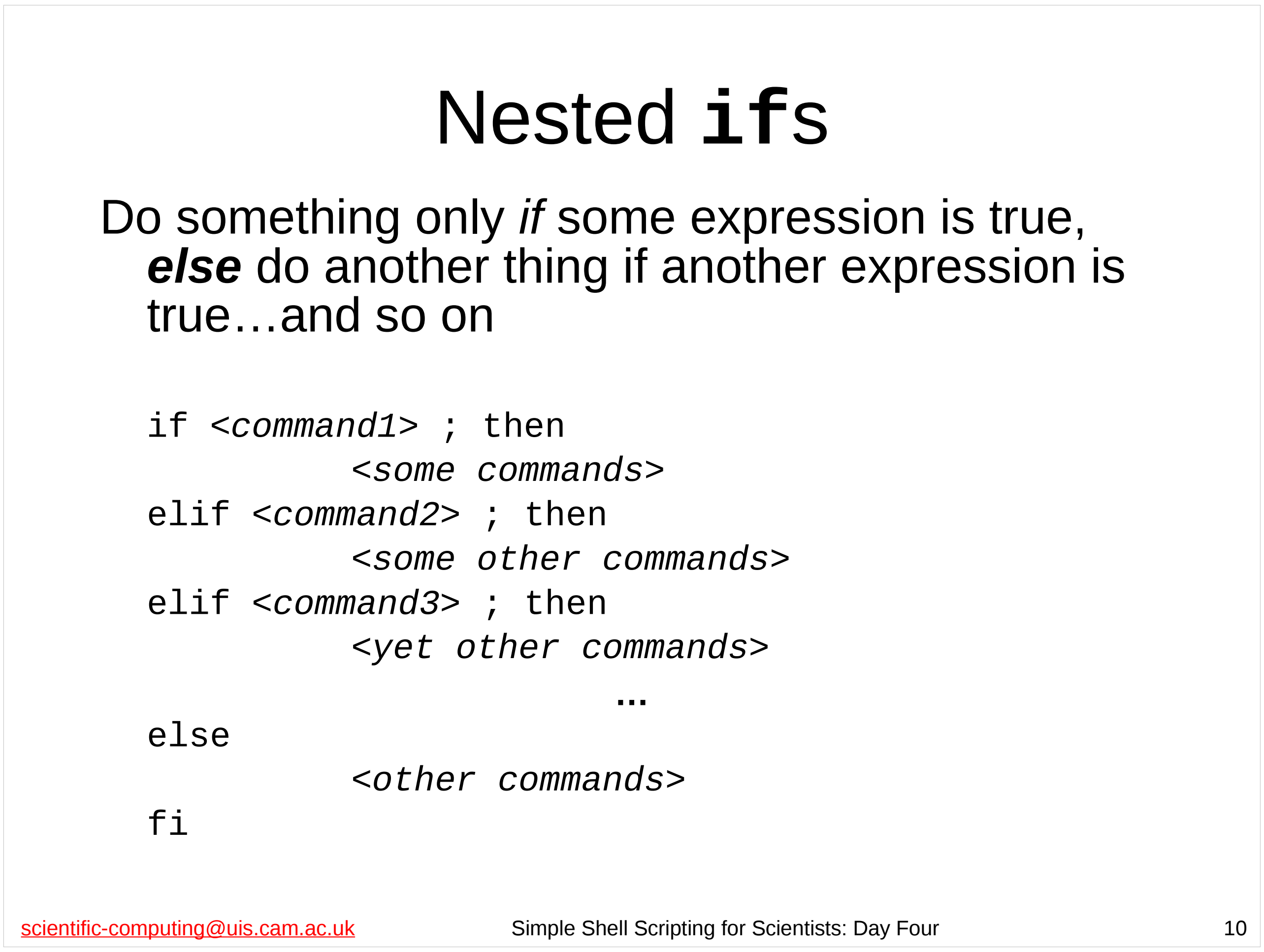
We can decide whether a collection of commands should be executed using an **`if`** statement. An **`if`** statement executes a collection of commands ***if and only if*** the result of some command or test is true. (Recall that the result of a command is considered to be true if it returns an exit status of 0 (i.e. if the command succeeded)).

As well as deciding whether a collection of commands should be executed at all, we can also decide whether one or other of two collections of commands should be executed using a more advanced form of the **`if`** statement. If there is an **`else`** section to an **`if`** statement the collection of commands in the **`else`** section will be executed ***if and only if*** the given <*`command`*> is ***false***. Note the syntax above. (Note that we don't have to have an **`else`** section; it is completely optional.)

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

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

the shell script will not exit if the result of the command or test the **`if`** statement depends on is false (i.e. it returns a non-zero exit status), since if it did, this would make **`if`** statements fairly useless(!).



Nested **`if`**s

Do something only *if* some expression is true, ***else*** do another thing if another expression is true…and so on

```
if <command1> ; then
          <some commands>
elif <command2> ; then
          <some other commands>
elif <command3> ; then
          <yet other commands>
                         …
else
          <other commands>
fi
```

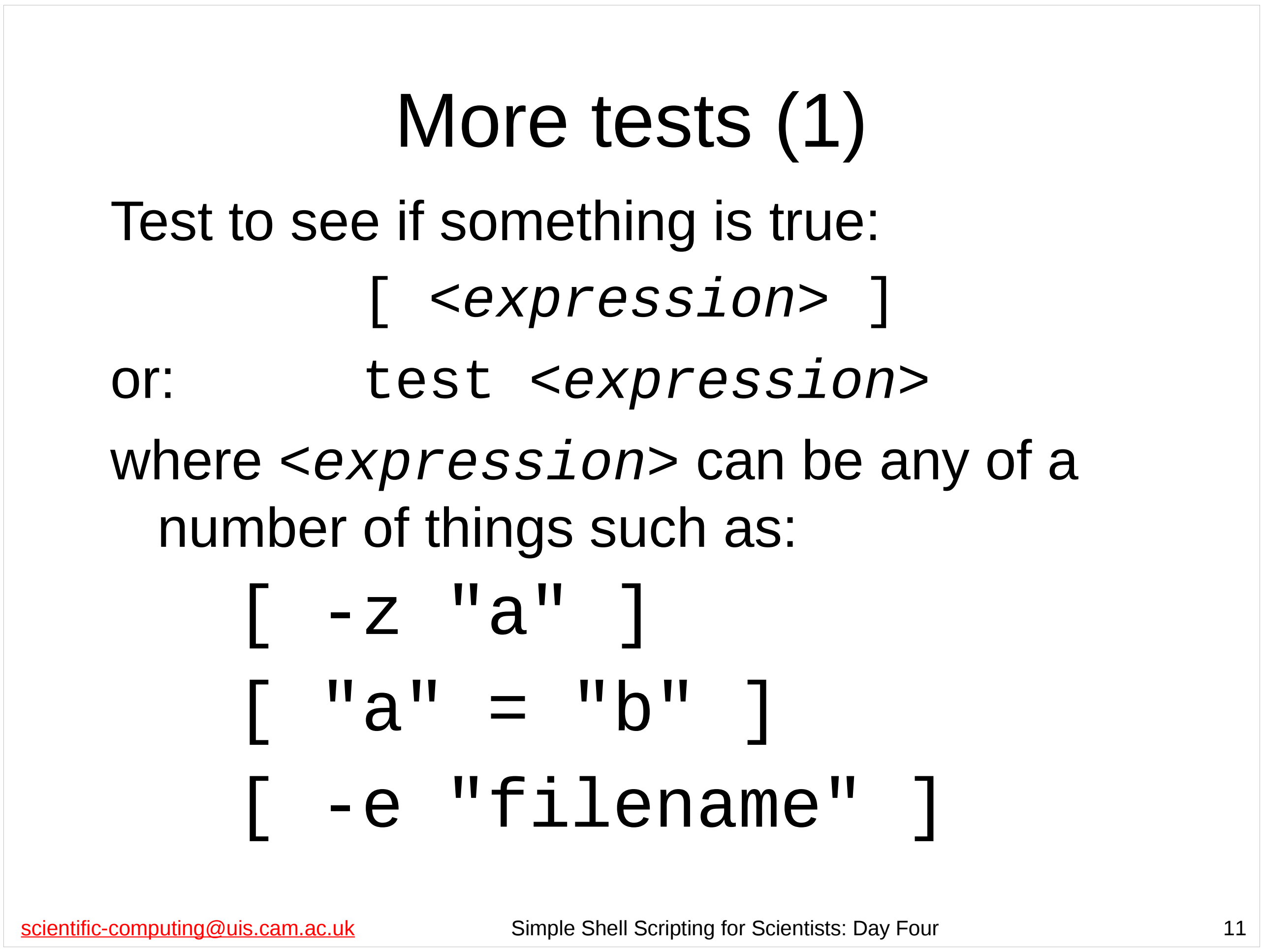
scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 10

We can have even more complicated **`if`** statements than the simpler **`if`**…**`then`**…**`else`** form shown on the previous slide. We can *nest* **`if`** statements: if one command (or test) is true, do one thing, if a different command (or test) is true do something else and so on, culminating in an optional **`else`** section ("if none of the previous expressions were true, do this").

One of the easiest ways of doing this is by using **`elif`** (short for ***`el`***`se` ***`if`***) for all the alternative expressions we want to test.

Why would we do this? Imagine that we had a shell script that could do several different things and the decision as to which it should do was made by the user specifying different arguments on the command line. We might want our script to have the following logic: if the user said "a" do this, else if they said "b" do that, else if they said "c" do something else, and so on, ending with else if they said something that was none of the previous things say "I don't know what you are talking about".

There are better ways to do that than using this sort of **`if`** statement which involve a construct (**`case`**) and a shell builtin command (**`shift`**) that we will cover later today.



More tests (1)

Test to see if something is true:

```
        [ <expression> ]
```

or:

```
        test <expression>
```

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

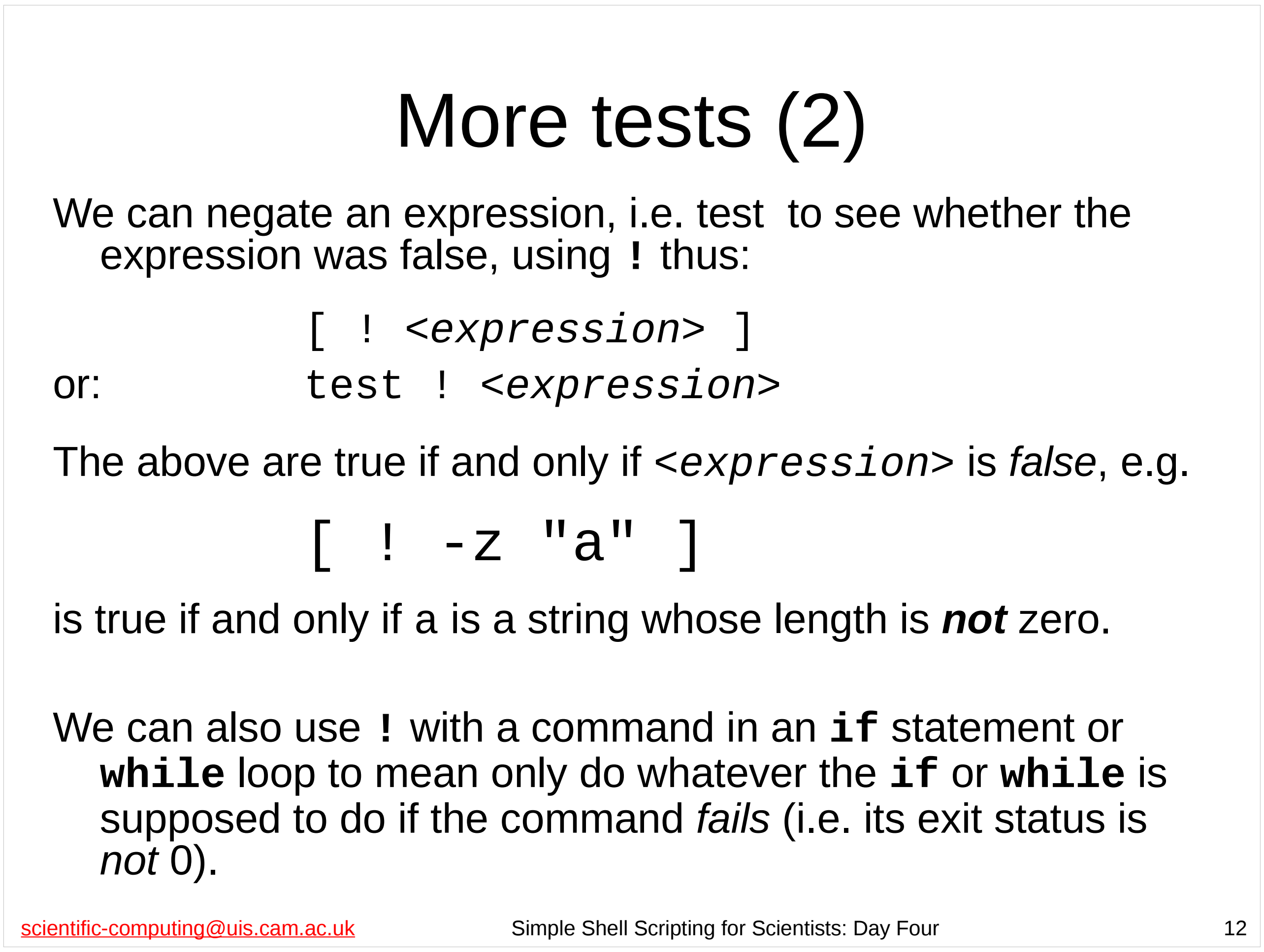
```
[ -z "a" ]
[ "a" = "b" ]
[ -e "filename" ]
```

As well as the (integer) arithmetic tests we met on the second day of the course, there are a number of other tests we can do. They fall into two main categories: tests on files and tests on strings. There are many different such tests and we only list a few of the most useful below:

`–z "a"`	true if and only if **`a`** is a string whose length is zero
`"a" = "b"`	true if and only if the string **`a`** is equal to the string **`b`**
`"a" == "b"`	true if and only if the string **`a`** is equal to the string **`b`**
`"a" != "b"`	true if and only if the string **`a`** is not equal to the string **`b`**
`–d "filename"`	true if and only if the file **`filename`** is a directory
`–e "filename"`	true if and only if the file **`filename`** exists
`–h "filename"`	true if and only if the file **`filename`** is a symbolic link
`–r "filename"`	true if and only if the file **`filename`** is readable
`–x "filename"`	true if and only if the file **`filename`** is executable

You can often omit the quotation marks but it is good practice to get into the habit of using them, since if the strings or file names have spaces in them then *not* using the quotation marks can be disastrous. (Note that string comparison is *always* done **case sensitively**, so “`HELLO`” is not the same as “`hello`”.)

You can get a complete list of all the tests by looking in the CONDITIONAL EXPRESSIONS section of `bash`’s man page (type “**`man bash`**” at the shell prompt to show `bash`’s man page.)



More tests (2)

We can negate an expression, i.e. test to see whether the expression was false, using **`!`** thus:

```
            [ ! <expression> ]
or:         test ! <expression>
```

The above are true if and only if <*`expression`*> is *false*, e.g.

```
[ ! -z "a" ]
```

is true if and only if a is a string whose length is ***not*** zero.

We can also use **`!`** with a command in an **`if`** statement or **`while`** loop to mean only do whatever the **`if`** or **`while`** is supposed to do if the command *fails* (i.e. its exit status is *not* 0).

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 12

Recall that in a **`while`** loop or an **`if`** statement we can use commands as well as tests. The command is considered true if it succeeds, i.e. its exit status is 0. In a **`while`** loop or an **`if`** statement we can negate a command in exactly the same way we negate <*`expression`*>, using **`!`** – negating a command means that the **`while`** loop or **`if`** statement will only consider it true if the command *fails*, i.e. its exit status is *non-zero*.

So:

```
        while ! ls datafile ; do
                echo "Can't list file datafile!"
        done
```

…would print the string "`Can't list file datafile!`" on the screen as long as **`ls`** was unable to list the file `datafile`, i.e. as long as the **`ls`** command returns an error when it tries to list the file `datafile` (for instance, if the file didn't exist).

Similary:

```
        if ! ./infect.py ; then
                echo "Unable to run ./infect.py successfully"
        fi
```

…will only print the message "`Unable to run ./infect.py successfully`" if the **`infect.py`** program in the current directory returns a non-zero exit status (i.e. it fails for some reason).

Standard Error (2)

To redirect standard error to a file we use the following construct:

```
command 2> file
```

To send the output of a command to standard error, we use the following construct:

```
command >&2
```

Standard output is one of the *standard streams* that all programs (whether they are shell scripts or not) have. (The idea of a *stream* here is that there is a “stream” of data flowing to/from our program and to/from somewhere else, like the screen.) Another standard stream that we have already met is standard input (which by default comes from the keyboard unless we redirect it).

There is actually a *third* standard stream called *standard error*. Like standard output, this is an “output stream” – data flows *from* our program along this stream *to* somewhere else. This stream is not for ordinary output though, but for any ***error messages*** our program may generate (and by default it also goes to the screen).

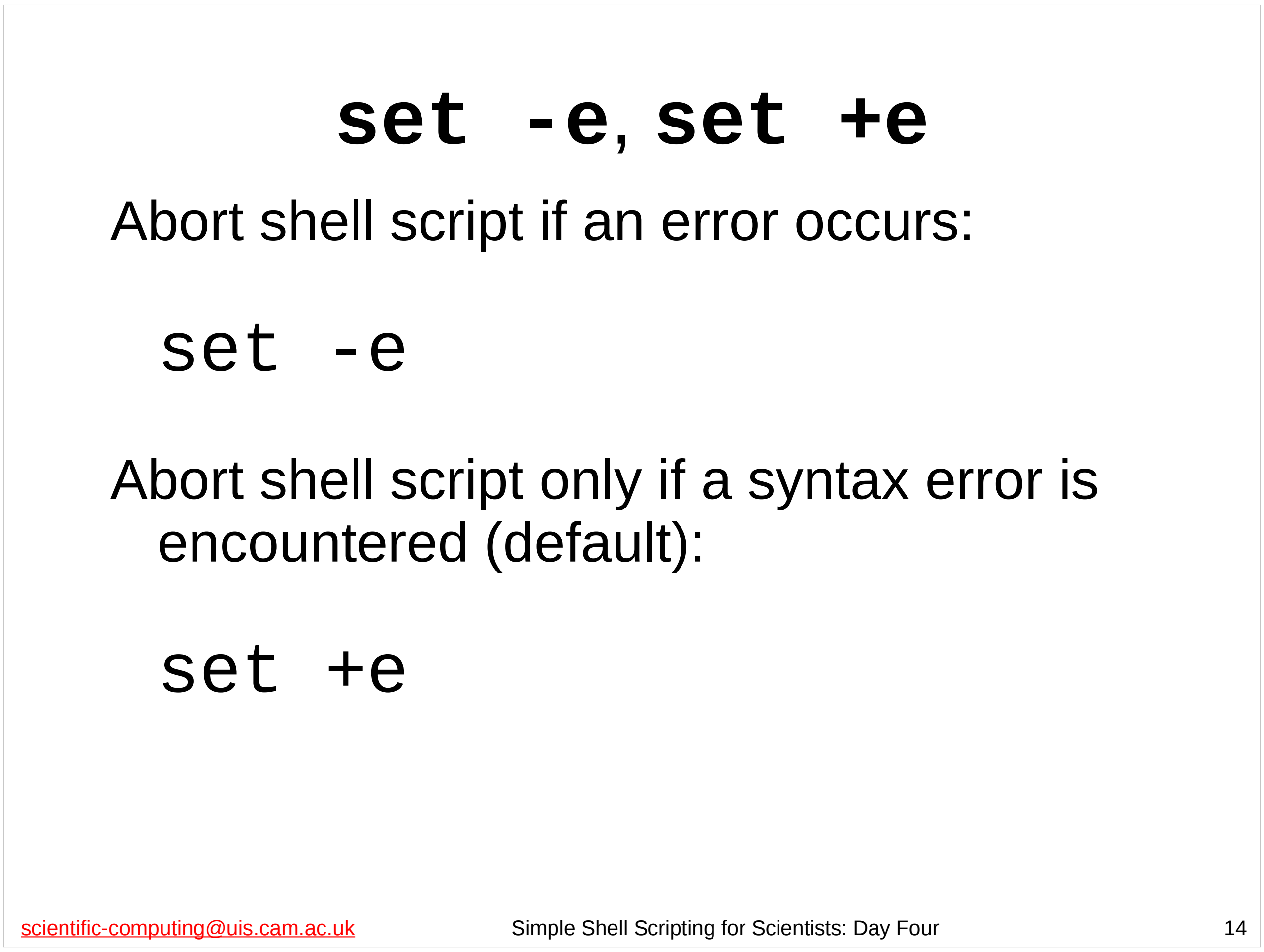
Why have two output streams? The reason is that this allows error messages to be easily separated from a program’s output, e.g. for ease of debugging, etc.

Note that when using standard error there is ***no*** space between the “2” and the “>” or the “>” and the “&2”, i.e.

it is	“2>”	***not***	“2 >”
and	“>&2”	***not***	“> &2” or “> & 2”

This is very important – if you put erroneous space characters in these constructs, the shell will not understand what you mean and will either produce an error message, or worse, do the wrong thing.

For more information on standard error and the other standard streams (standard input and standard output) see the following Wikipedia article:
http://en.wikipedia.org/wiki/Standard_streams



`set -e`, `set +e`

Abort shell script if an error occurs:

```
set -e
```

Abort shell script only if a syntax error is encountered (default):

```
set +e
```

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

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

then the shell script will abort if it encounters an error. We also know we can make this happen by using **`set -e`** instead, if we prefer.

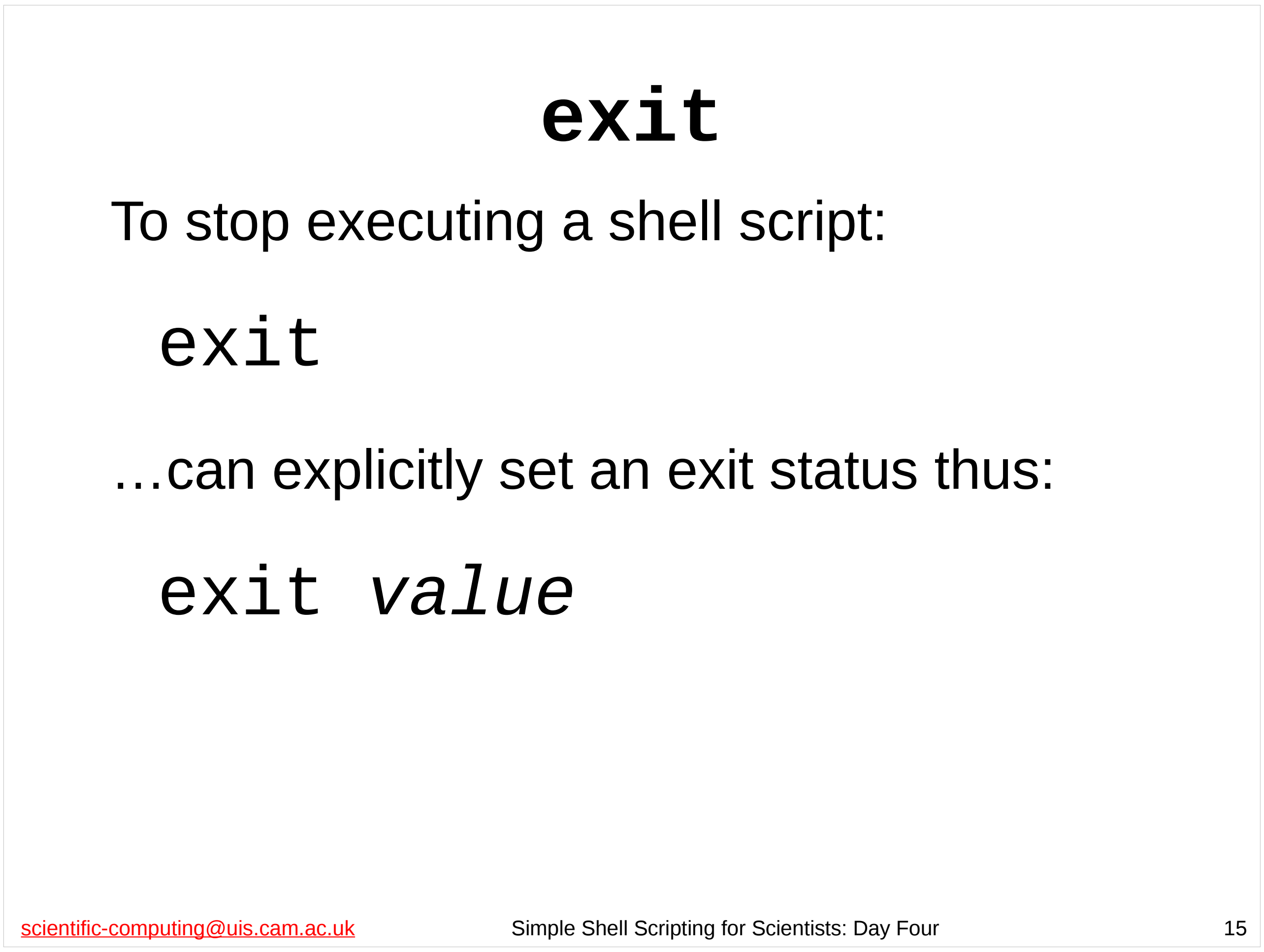
Sometimes though, we may want to handle errors ourselves, rather than just having our shell script fall over in a heap. So it would be nice if we could turn this behaviour off and on at the appropriate points in the shell script, and `bash` provides a mechanism for us to do just that:

- As we know, **`set -e`** tells the shell to quit when it encounters an error in the shell script. Whenever you are not doing your own *error handling* (i.e. checking to make sure the commands you run in your shell script have executed successfully), you ***should*** use **`set -e`**.
- **`set +e`** returns to the default behaviour of continuing to execute the shell script even after an error (other than a syntax error) has occurred.

A good practice to get into is to always have the following as the first line of your shell script that isn’t a comment (i.e. doesn’t start with a `#`):

```
set -e
```

and then to turn this behaviour off ***only*** when you are actually dealing with the errors yourself.



exit

To stop executing a shell script:

```
exit
```

…can explicitly set an exit status thus:

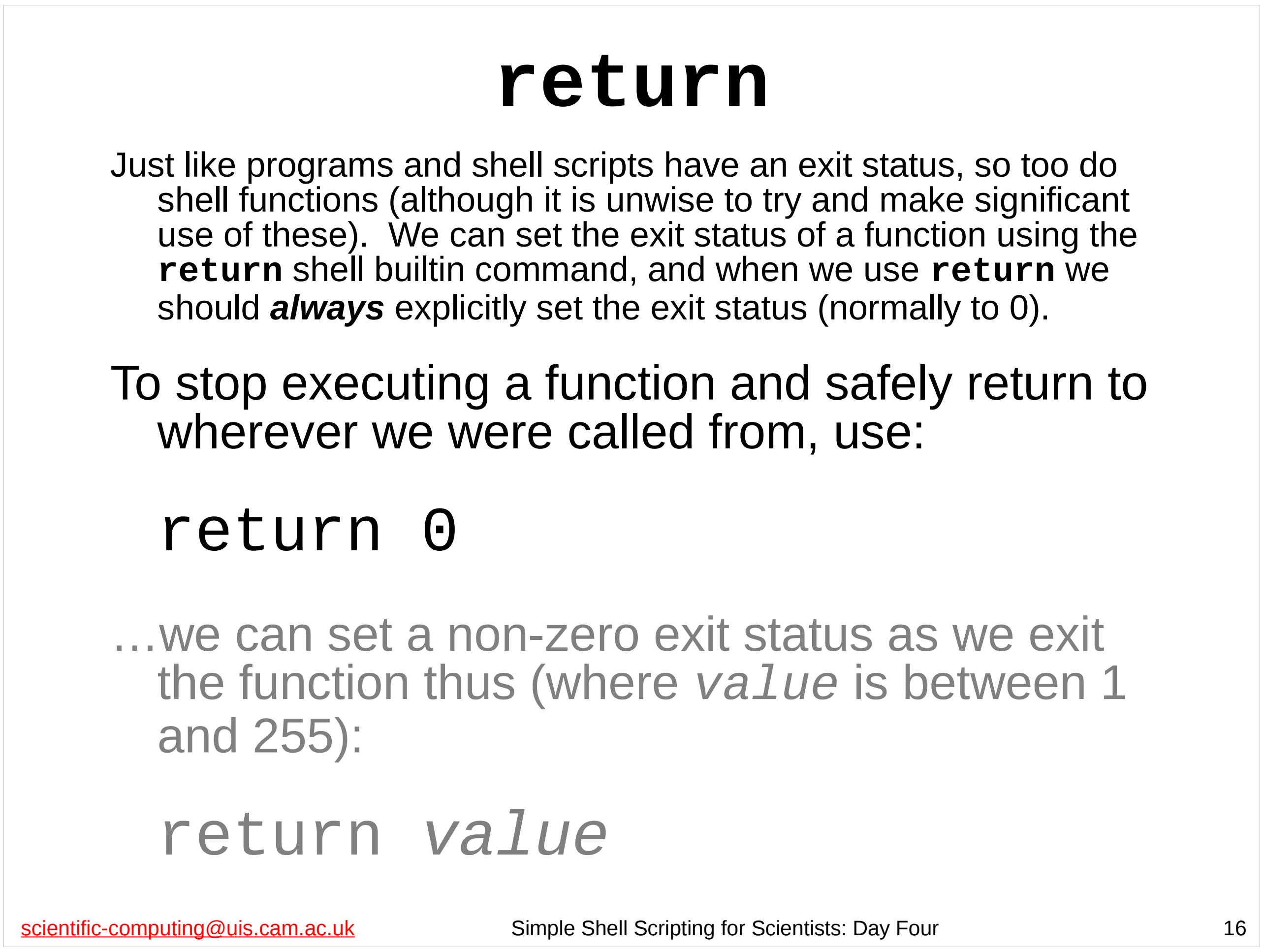
```
exit value
```

The **`exit`** shell builtin command causes a shell script to *exit* (stop executing) and can also explicitly set the exit status of the shell script (if you specify a value for the exit status).

Recall that the exit status is an integer between 0 and 255, and should be 0 ***only*** if the script was successful in what it was trying to do. If the script encounters an error it should set the exit status to a non-zero value.

If you don't give **`exit`** an exit status then the exit status of the shell script will be the exit status of the last command executed by the script before it reached the **`exit`** shell builtin command.

(If you don't have a **`exit`** shell builtin command in your shell script, then your script will exit when it executes its last command. In this case its exit status will be the exit status of the last command executed by your script.)



`return`

Just like programs and shell scripts have an exit status, so too do shell functions (although it is unwise to try and make significant use of these). We can set the exit status of a function using the **`return`** shell builtin command, and when we use **`return`** we should ***always*** explicitly set the exit status (normally to 0).

To stop executing a function and safely return to wherever we were called from, use:

```
return 0
```

…we can set a non-zero exit status as we exit the function thus (where *`value`* is between 1 and 255):

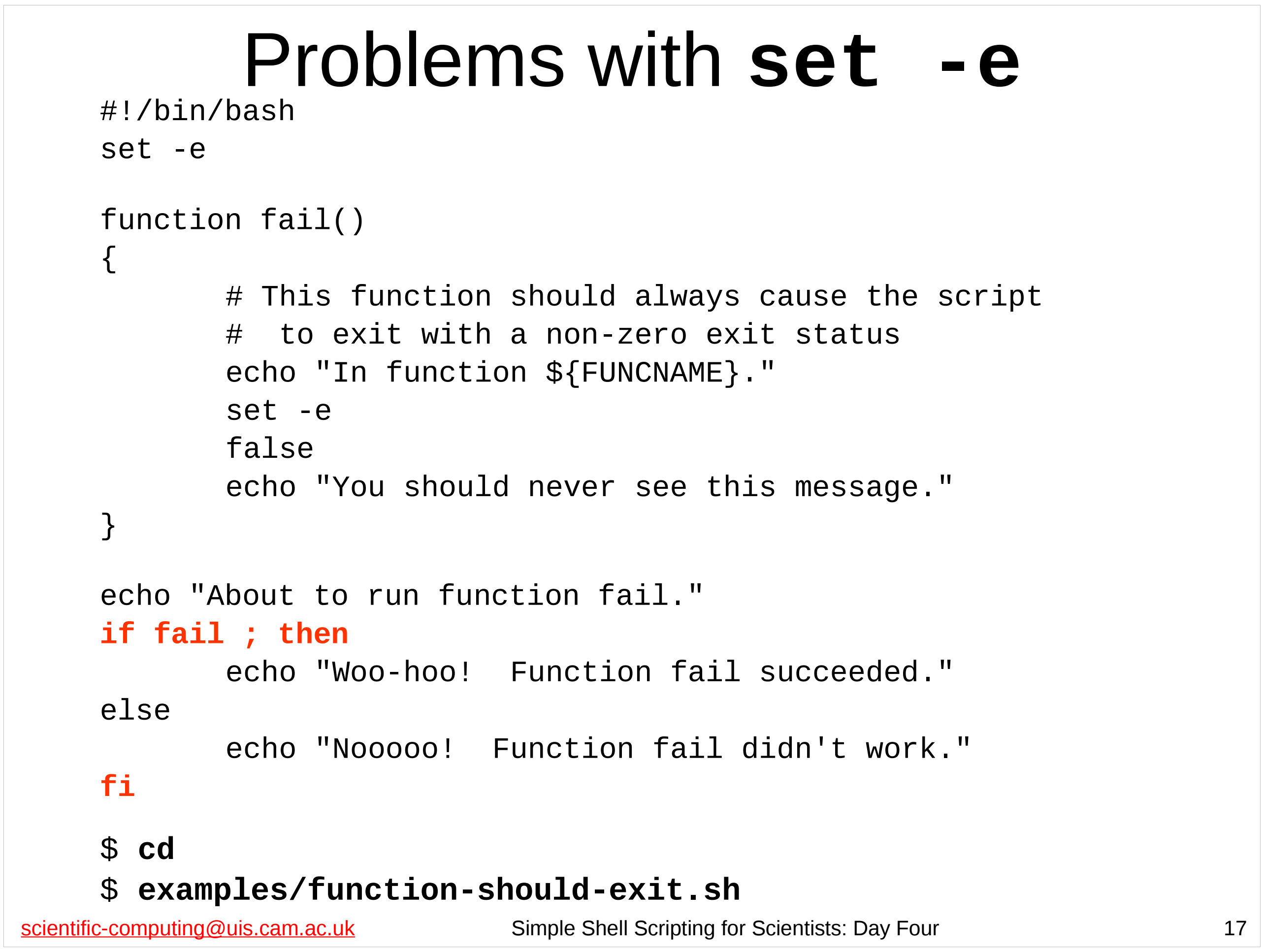
```
return value
```

The **`return`** shell builtin command causes a shell function to stop executing and return control to whatever part of the shell script called it. It can also explicitly set the exit status of the function, and when we use **`return`** we should explicitly set the status (normally to 0).

As with ordinary programs and shell scripts themselves, the exit status of a shell function is an integer between 0 and 255, and, as one might expect, the convention is that the exit status should be 0 only if the function was successful in what it was trying to do. Unfortunately, if the function returns a non-zero exit status, this can cause very subtle (i.e. difficult to track down) types of misbehaviour, so it is actually safest to always use **`return`** with an exit status of 0 (i.e. "**`return 0`**").

(If you don't give **`return`** an exit status then the exit status of the shell function will be the exit status of the last command executed by the function before it reached the **`return`** shell builtin command, but this can lead to extremely subtle types of misbehaviour – use "**`return 0`**" instead.

And if you don't have a **`return`** shell builtin command in your shell function, then your function will exit when it executes its last command. In this case its exit status will be the exit status of the last command executed in your function – this can also cause subtle problems, so your functions should really always end with "**`return 0`**".)



Problems with **set -e**

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

function fail()
{
       # This function should always cause the script
       #  to exit with a non-zero exit status
       echo "In function ${FUNCNAME}."
       set -e
       false
       echo "You should never see this message."
}

echo "About to run function fail."
if fail ; then
       echo "Woo-hoo!  Function fail succeeded."
else
       echo "Nooooo!  Function fail didn't work."
fi
```

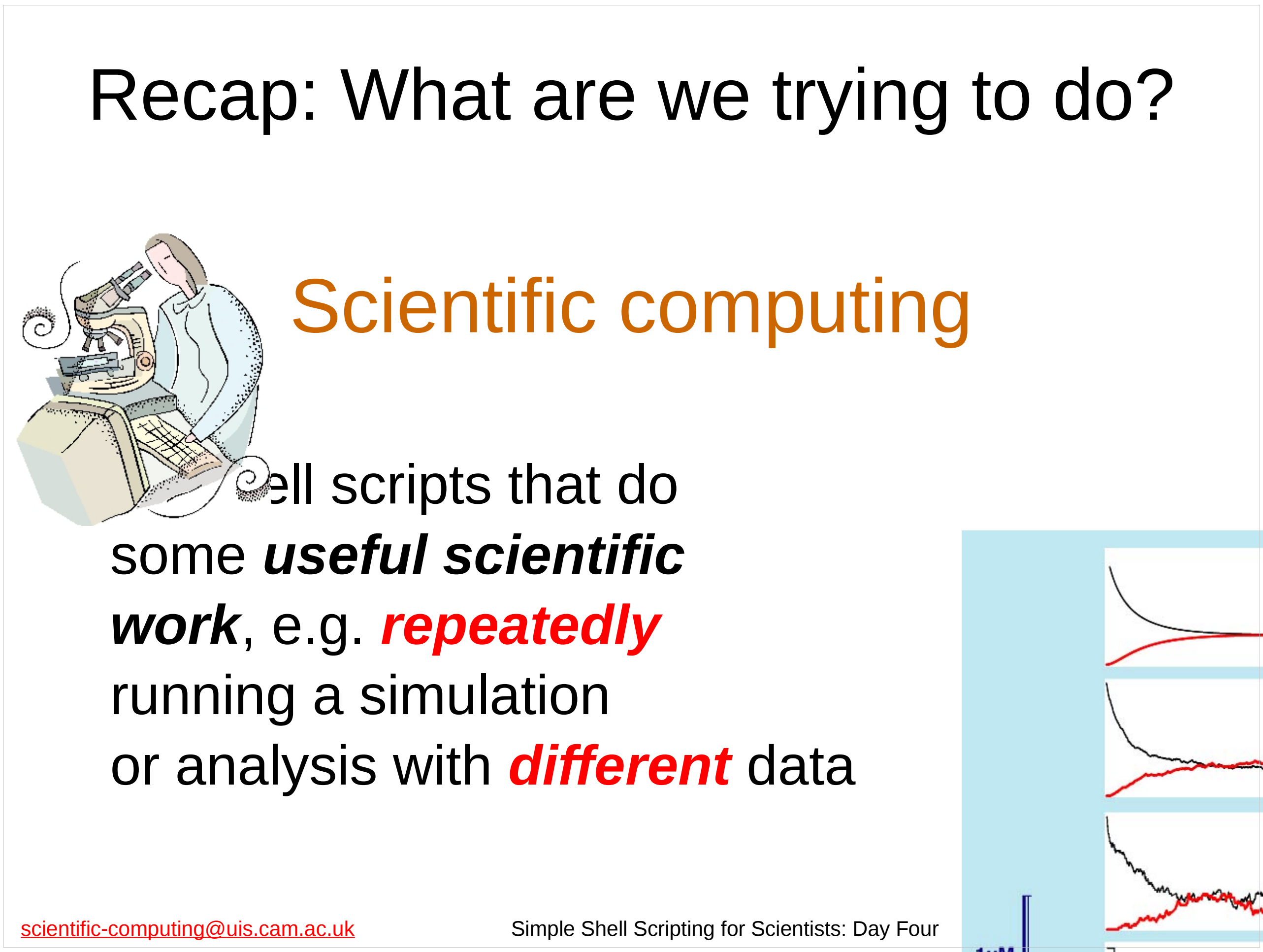
```
$ cd
$ examples/function-should-exit.sh
```

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 17

There are subtle problems with **`set -e`** we need to be aware of, particularly where functions are concerned. For an example of this, see the `function-should-exit.sh` script in the `examples` subdirectory (shown on the slide above).

We might expect that, because we use **`set -e`** within the function **`fail`**, when we run that function it will cause the script to exit. However, because we run the function as the command checked by an **`if`** statement, this doesn't happen! (We would have the same problem if we ran the function as the command checked by a **`while`** loop.)

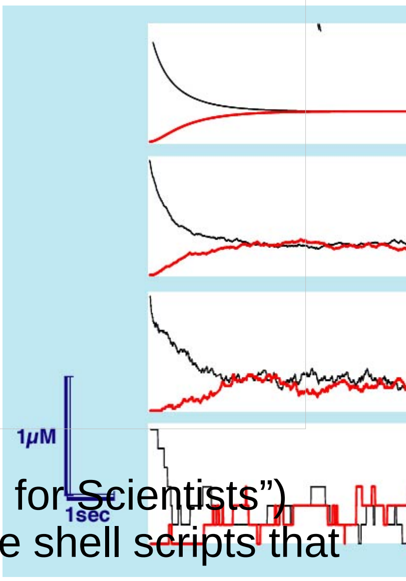
Basically, if you run a shell function as the command checked by an **`if`** statement or a **`while`** loop, **`set -e`** is ***disabled*** whilst the function is running, even if you explicitly use it within the function. This makes using shell functions as the command checked by an **`if`** statement or a **`while`** loop extremely dangerous, so we advise you not to do it.



Recap: What are we trying to do?

Scientific computing

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

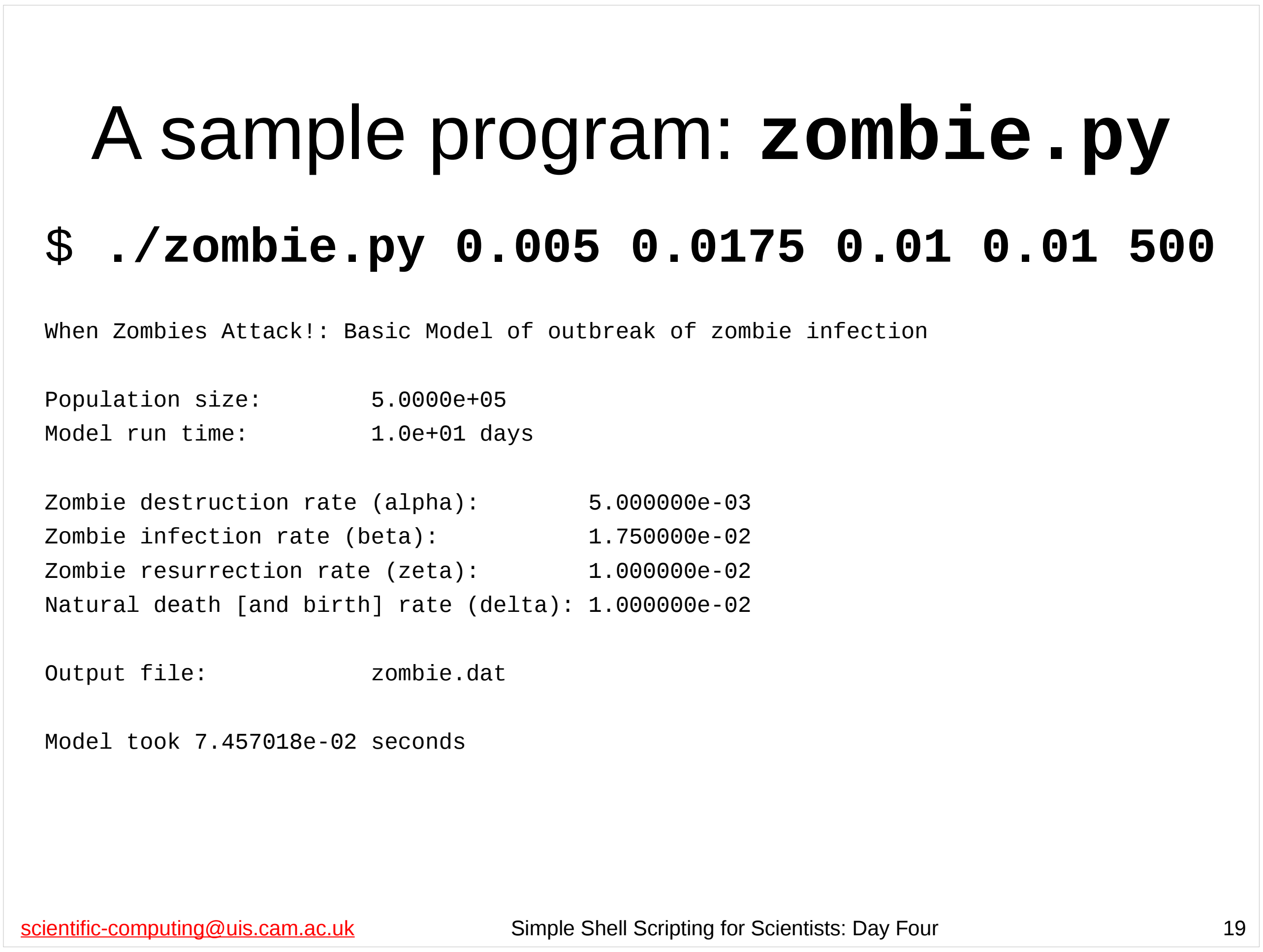


scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four

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

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

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



A sample program: **`zombie.py`**

```
$ ./zombie.py 0.005 0.0175 0.01 0.01 500
```

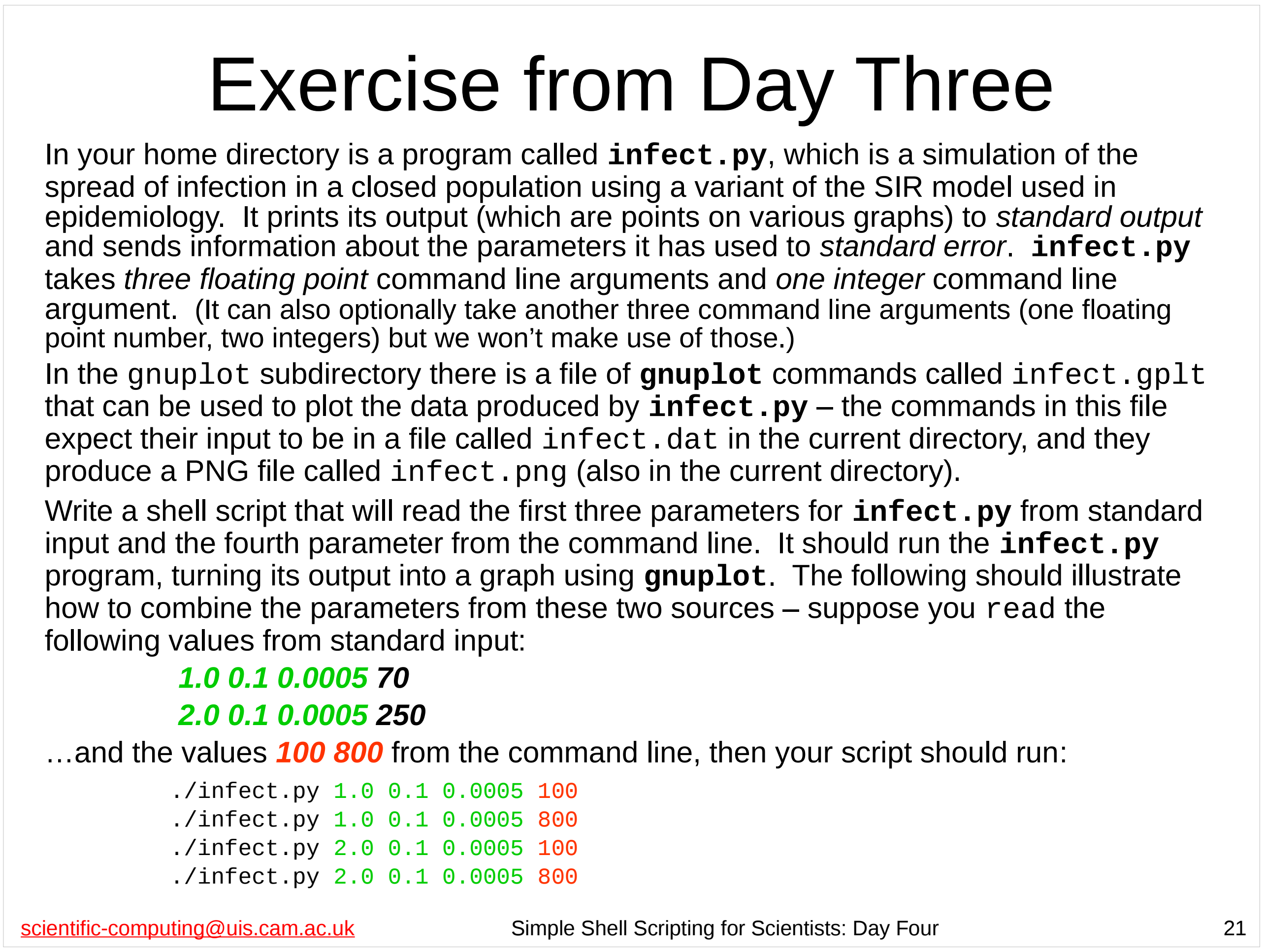
```
When Zombies Attack!: Basic Model of outbreak of zombie infection

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 7.457018e-02 seconds
```

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 19

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

The **`zombie.py`** program takes 5 numeric arguments on the command line: 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.

The **`zombie.py`** program is not as well behaved as we might like (which, sadly, is also typical of many programs you will run). The particular way that **`zombie.py`** is not well behaved is this: 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.



Exercise from Day Three

In your home directory is a program called **`infect.py`**, which is a simulation of the spread of infection in a closed population using a variant of the SIR model used in epidemiology. It prints its output (which are points on various graphs) to *standard output* and sends information about the parameters it has used to *standard error*. **`infect.py`** takes *three floating point* command line arguments and *one integer* command line argument. (It can also optionally take another three command line arguments (one floating point number, two integers) but we won't make use of those.)

In the `gnuplot` subdirectory there is a file of **`gnuplot`** commands called `infect.gplt` that can be used to plot the data produced by **`infect.py`** – the commands in this file expect their input to be in a file called `infect.dat` in the current directory, and they produce a PNG file called `infect.png` (also in the current directory).

Write a shell script that will read the first three parameters for **`infect.py`** from standard input and the fourth parameter from the command line. It should run the **`infect.py`** program, turning its output into a graph using **`gnuplot`**. The following should illustrate how to combine the parameters from these two sources – suppose you `read` the following values from standard input:

1.0 0.1 0.0005 70

2.0 0.1 0.0005 250

…and the values ***100 800*** from the command line, then your script should run:

```
./infect.py 1.0 0.1 0.0005 100
./infect.py 1.0 0.1 0.0005 800
./infect.py 2.0 0.1 0.0005 100
./infect.py 2.0 0.1 0.0005 800
```

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 21

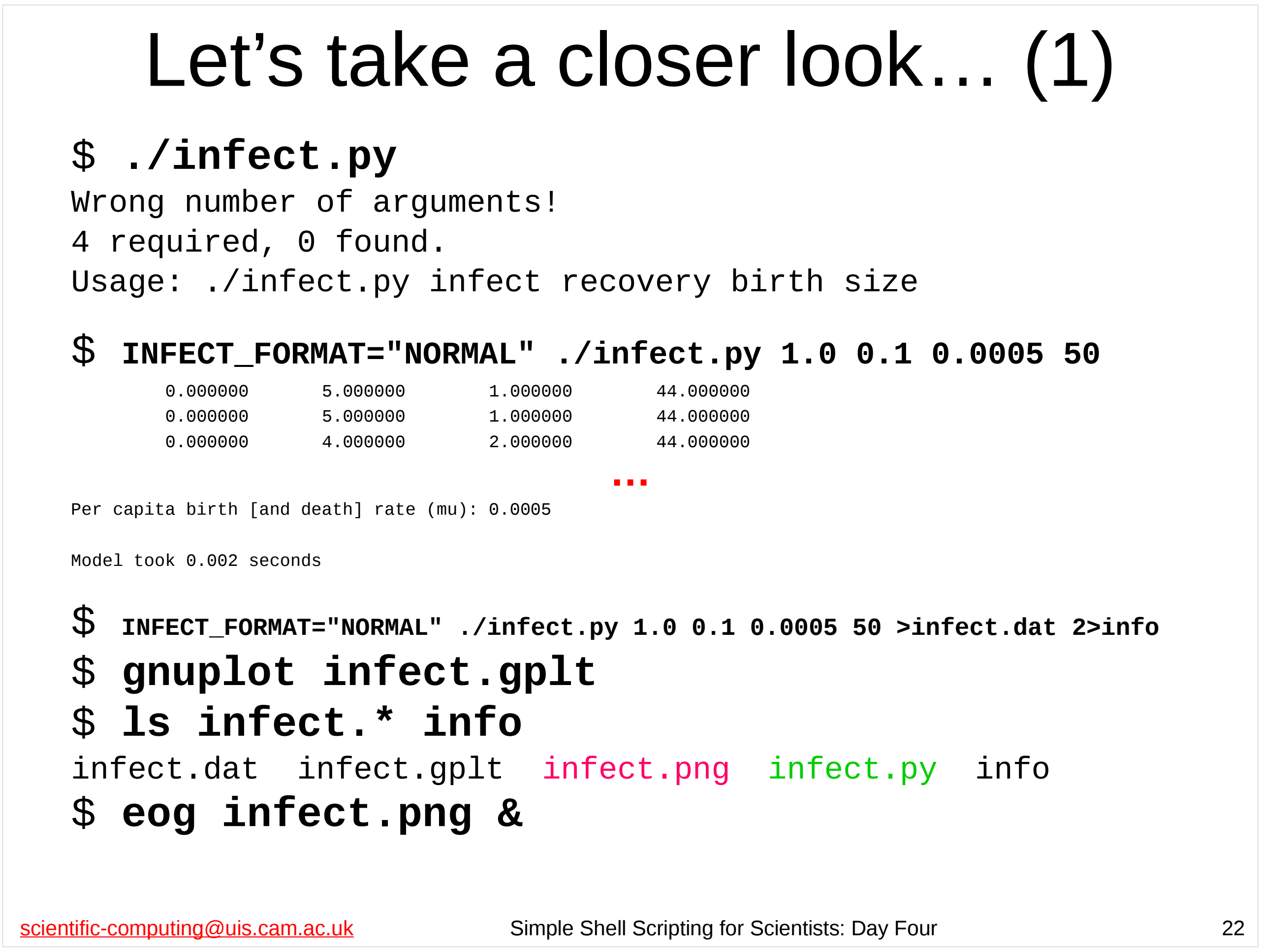
The point of this exercise was to consolidate everything you've learnt over the previous three days of this course. To that end you should have written your own shell script **FROM SCRATCH** for this exercise and not just taken one of the ones we'd already constructed over this course and changed the names of the programs it runs. Whilst you could certainly get an answer to this exercise that way, you wouldn't learn very much.

Also, you should have made your shell script ***as good a shell script as you could possibly make it*** – so it should:

- be well structured using shell functions,
- be fully commented,
- do some error handling,
- keep a log file of what it is doing,
- print its error messages on standard error,
- use a temporary directory for working in,
- do some checking of its input,
- etc

There is a file in the `scripts` subdirectory called `infect_params` that you can use as a source of parameters to `read` via standard input. It was suggested that for the command line arguments you use:

75 100 300 3000 50000



Let’s take a closer look… (1)

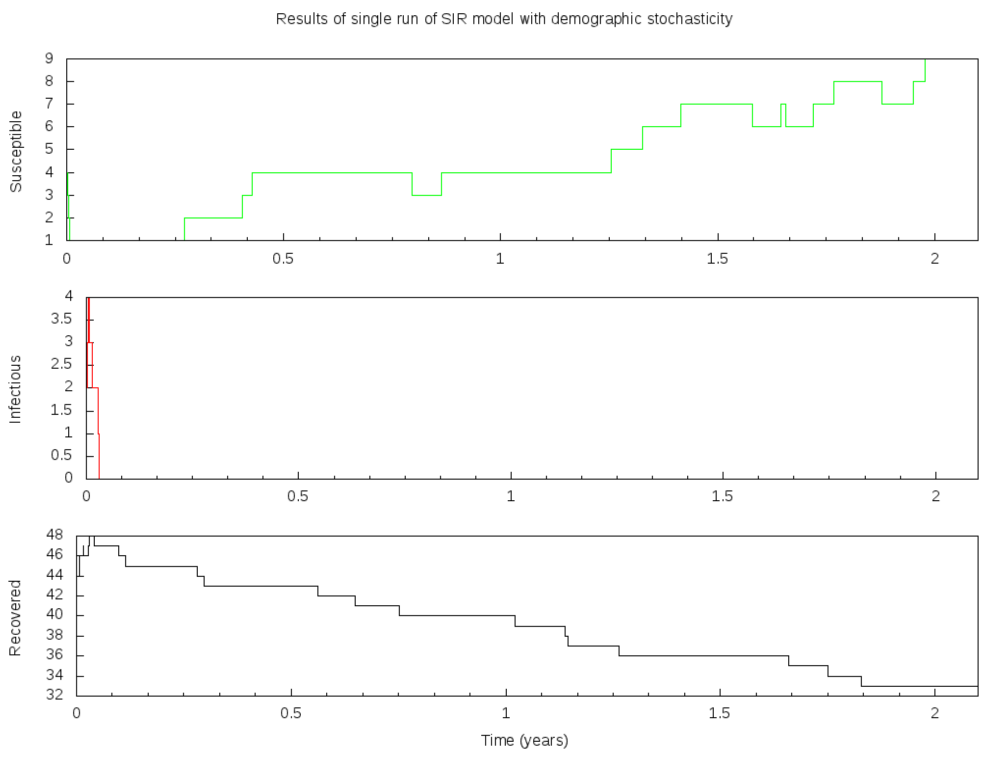
```
$ ./infect.py
Wrong number of arguments!
4 required, 0 found.
Usage: ./infect.py infect recovery birth size

$ INFECT_FORMAT="NORMAL" ./infect.py 1.0 0.1 0.0005 50
         0.000000       5.000000        1.000000        44.000000
         0.000000       5.000000        1.000000        44.000000
         0.000000       4.000000        2.000000        44.000000
                                     …
Per capita birth [and death] rate (mu): 0.0005

Model took 0.002 seconds

$ INFECT_FORMAT="NORMAL" ./infect.py 1.0 0.1 0.0005 50 >infect.dat 2>info
$ gnuplot infect.gplt
$ ls infect.* info
infect.dat  infect.gplt  infect.png  infect.py  info
$ eog infect.png &
```

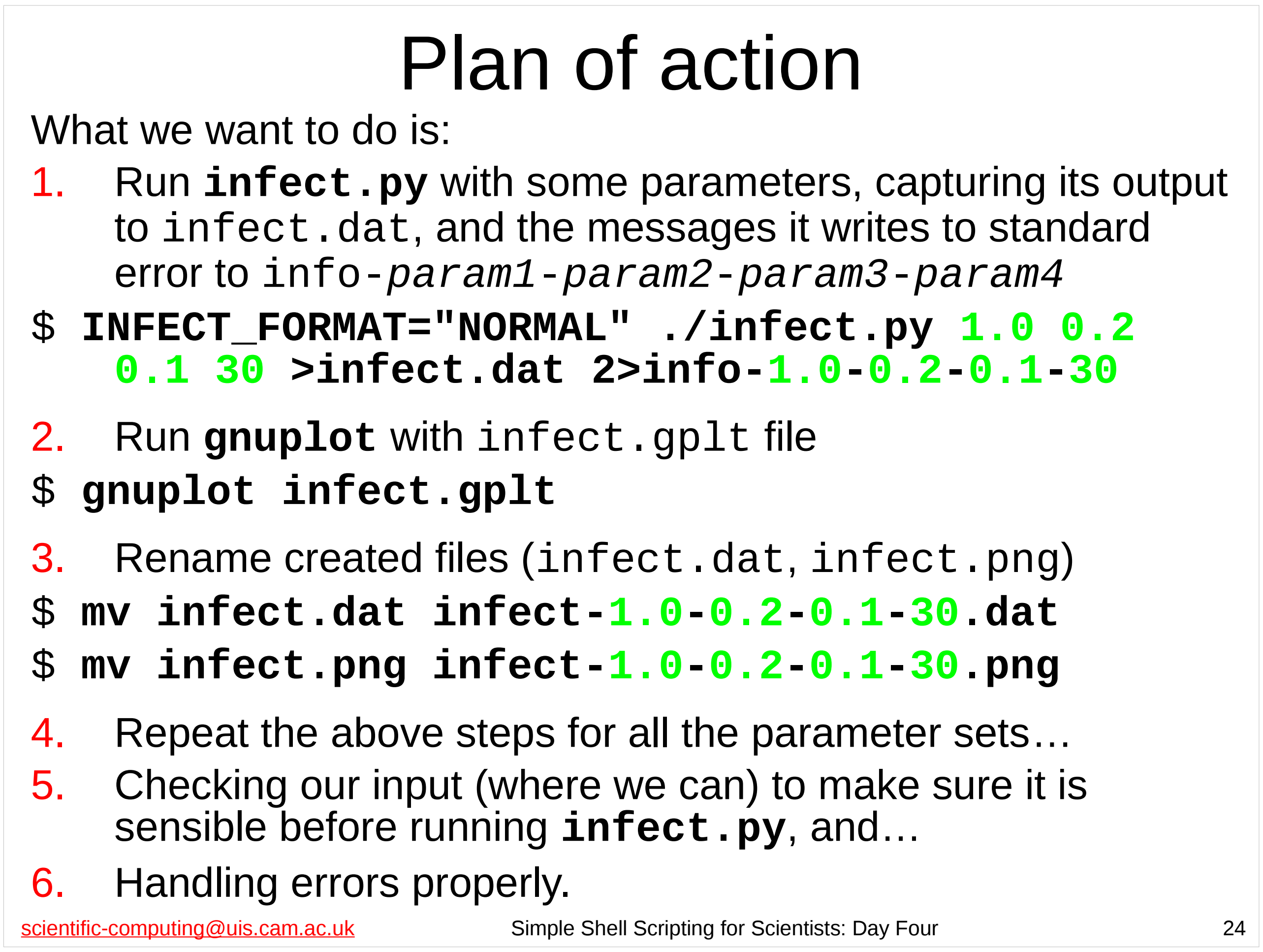
The **infect.py** program, which is located in your home directory, takes 4 numeric arguments (3 floating point numbers and 1 integer). **infect.py** always writes its output to standard output (which by default will be the screen) and some informational messages to standard error (which by default will also be the screen).

The instructions (in `infect.gplt` in your home directory) that we give to **gnuplot**, the program which we use to turn **infect.py**’s output into a graph, expect **infect.py**’s output to be in a file called `infect.dat`. So we need to arrange that **infect.py**’s output is redirected to a file called `infect.dat`. Running **gnuplot** will then produce a graphics file called `infect.png`. We also want to keep a copy of the informational messages **infect.py** writes to standard error, so we need to arrange that standard error is redirected to a file as well.

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.

Let’s take a closer look… (2)





Plan of action

What we want to do is:

1. Run **`infect.py`** with some parameters, capturing its output to `infect.dat`, and the messages it writes to standard error to `info-`*`param1`*`-`*`param2`*`-`*`param3`*`-`*`param4`*

```
$ INFECT_FORMAT="NORMAL" ./infect.py 1.0 0.2
  0.1 30 >infect.dat 2>info-1.0-0.2-0.1-30
```

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

```
$ gnuplot infect.gplt
```

3. Rename created files (`infect.dat`, `infect.png`)

```
$ mv infect.dat infect-1.0-0.2-0.1-30.dat
$ mv infect.png infect-1.0-0.2-0.1-30.png
```

4. Repeat the above steps for all the parameter sets…
5. Checking our input (where we can) to make sure it is sensible before running **`infect.py`**, and…
6. Handling errors properly.

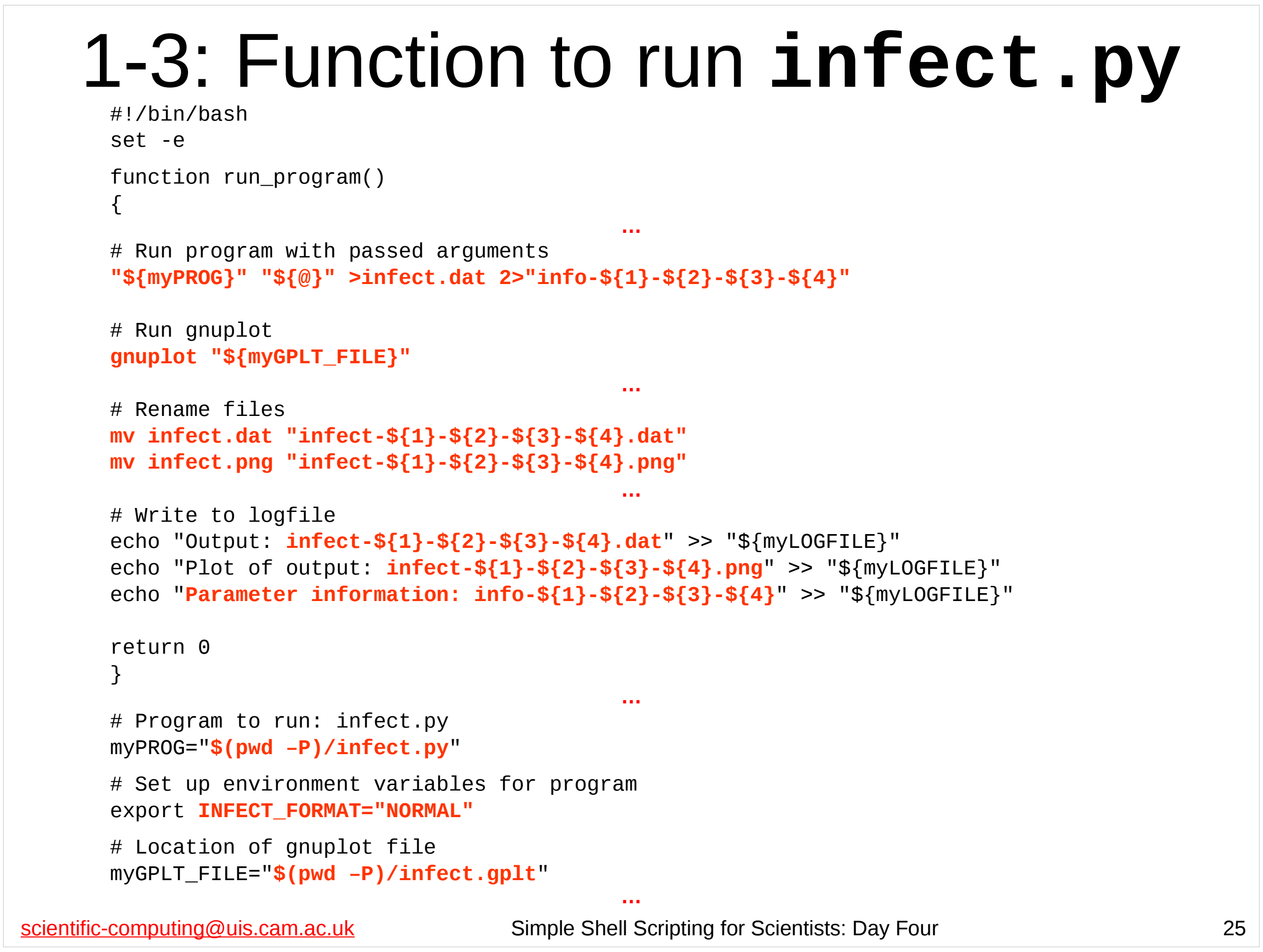
scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 24

So for this exercise you needed to create a shell script that basically did the above task. When writing a shell script that is at all complicated, it is best to first plan it out, and one way of doing that is to describe what the shell script should do as a numbered list.

Basically, we want to run the **`infect.py`** program several times with a different parameter set each time, plotting its output on a graph each time. After each run, we rename the files we've created so that they don't get overwritten.

Steps 1-3 can be straightforwardly achieved by writing a shell function that runs **`infect.py`**, then **`gnuplot`**, and then renames the files that have been created.

For step 4 we loop through the parameter sets. This is slightly more complicated than simply using a single loop since one of the parameters comes from standard input and the other from the command line. For step 5 we improve our shell script so that it checks its command line arguments. For step 6 we further improve our shell script to do some sensible error handling.



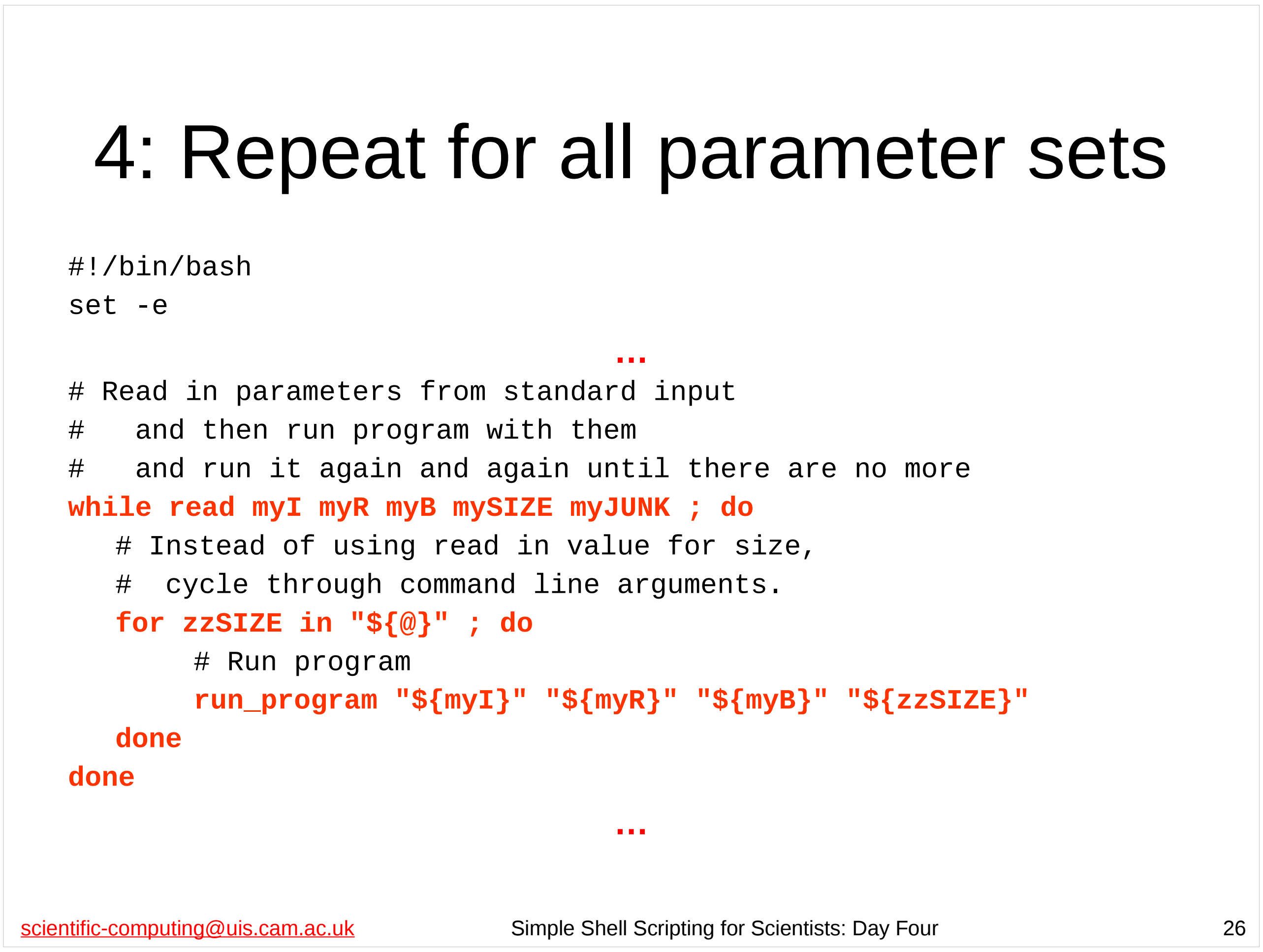
1-3: Function to run **infect.py**

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

function run_program()
{
                                         …
# Run program with passed arguments
"${myPROG}" "${@}" >infect.dat 2>"info-${1}-${2}-${3}-${4}"

# Run gnuplot
gnuplot "${myGPLT_FILE}"
                                         …
# Rename files
mv infect.dat "infect-${1}-${2}-${3}-${4}.dat"
mv infect.png "infect-${1}-${2}-${3}-${4}.png"
                                         …
# Write to logfile
echo "Output: infect-${1}-${2}-${3}-${4}.dat" >> "${myLOGFILE}"
echo "Plot of output: infect-${1}-${2}-${3}-${4}.png" >> "${myLOGFILE}"
echo "Parameter information: info-${1}-${2}-${3}-${4}" >> "${myLOGFILE}"

return 0
}
                                         …
# Program to run: infect.py
myPROG="$(pwd -P)/infect.py"

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

# Location of gnuplot file
myGPLT_FILE="$(pwd -P)/infect.gplt"
                                         …
```

If you examine the `multi-infect.sh` script in the `scripts` subdirectory of your home directory, you will see that it contains a **`run_program`** function that runs **`infect.py`**, capturing its output to `infect.dat` and its informational messages (written to standard error) to another file named after the parameters that were given to **`infect.py`**. It then runs **`gnuplot`**. Then it renames `infect.dat` and `infect.png` after the parameters that were given to **`infect.py`**. (i.e. **`run_program`** carries out steps 1-3 of our plan.)

You should be able to tell what the **`run_program`** function (some highlighted parts of which are shown above) does – if there is anything you don't understand, or if you had any difficulty with this part of the exercise, please let the course giver or demonstrator know.

(Although not shown on the slide above, inspection of the rest of the `multi-infect.sh` script should show you that the script defines the shell variable `myLOGFILE` before it calls the **`run_program`** function.)



4: Repeat for all parameter sets

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

                                  …
# Read in parameters from standard input
#   and then run program with them
#   and run it again and again until there are no more
while read myI myR myB mySIZE myJUNK ; do
   # Instead of using read in value for size,
   #  cycle through command line arguments.
   for zzSIZE in "${@}" ; do
        # Run program
        run_program "${myI}" "${myR}" "${myB}" "${zzSIZE}"
   done
done
                                  …
```

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 26

Now examine the main body of the `multi-infect.sh` script in the `scripts` subdirectory of your home directory, and you will see that it contains a **`while`** loop that **`read`**s in parameters from standard input. Inside the **`while`** loop is a **`for`** loop which cycles through the command line arguments.

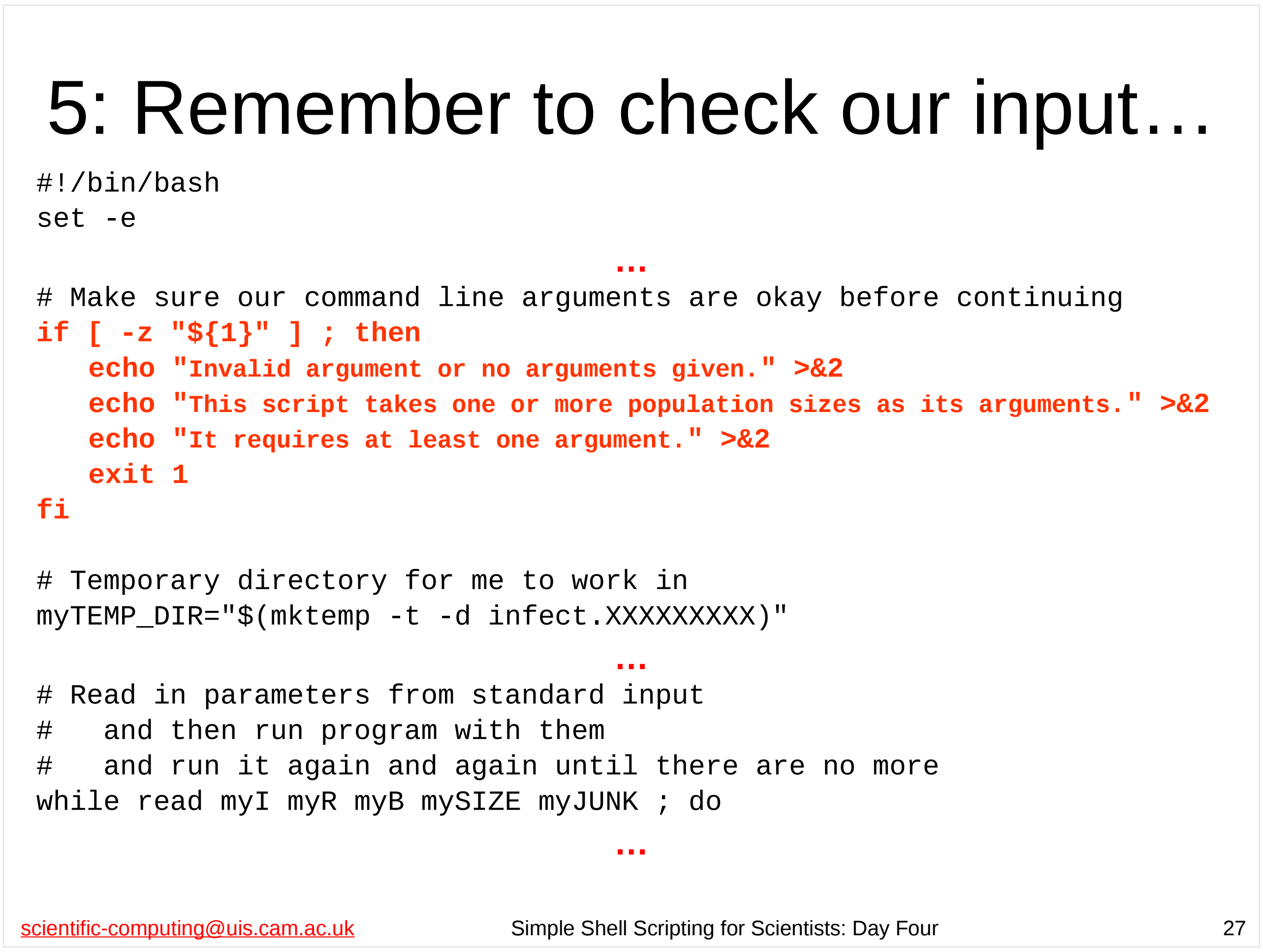
Recall that the exercise asked you to construct a shell script in which the first three parameters for **`infect.py`** came from standard input and the last parameter came from the command line, with the command line parameters being used with *each* of the parameters **`read`** in on standard input.

You should be able to tell what all the highlighted parts of the shell script above do, and you should be able to see why they work as a sequence of commands to carry out this step of the plan for our shell script – if there is anything you don't understand, or if you had any difficulty with this part of the exercise, please let the course giver or a demonstrator know.

You can test that this script works by doing the following:

```
$ cd
$ rm –f *.dat *.png info* logfile
$ cat scripts/infect_params | scripts/multi-infect.sh 75 100 300 3000 50000
$ ls
```

You should see that a number of PNG and `.dat` files have been produced. You could view some of the PNG files to make sure they were what was expected by using Eye of GNOME (**`eog`**) or another PNG viewer (such as Firefox).



5: Remember to check our input…

```
#!/bin/bash
set -e
                                   …
# Make sure our command line arguments are okay before continuing
if [ -z "${1}" ] ; then
   echo "Invalid argument or no arguments given." >&2
   echo "This script takes one or more population sizes as its arguments." >&2
   echo "It requires at least one argument." >&2
   exit 1
fi

# Temporary directory for me to work in
myTEMP_DIR="$(mktemp -t -d infect.XXXXXXXXX)"
                                   …
# Read in parameters from standard input
#   and then run program with them
#   and run it again and again until there are no more
while read myI myR myB mySIZE myJUNK ; do
                                   …
```

scientific-computing@uis.cam.ac.uk   Simple Shell Scripting for Scientists: Day Four   27

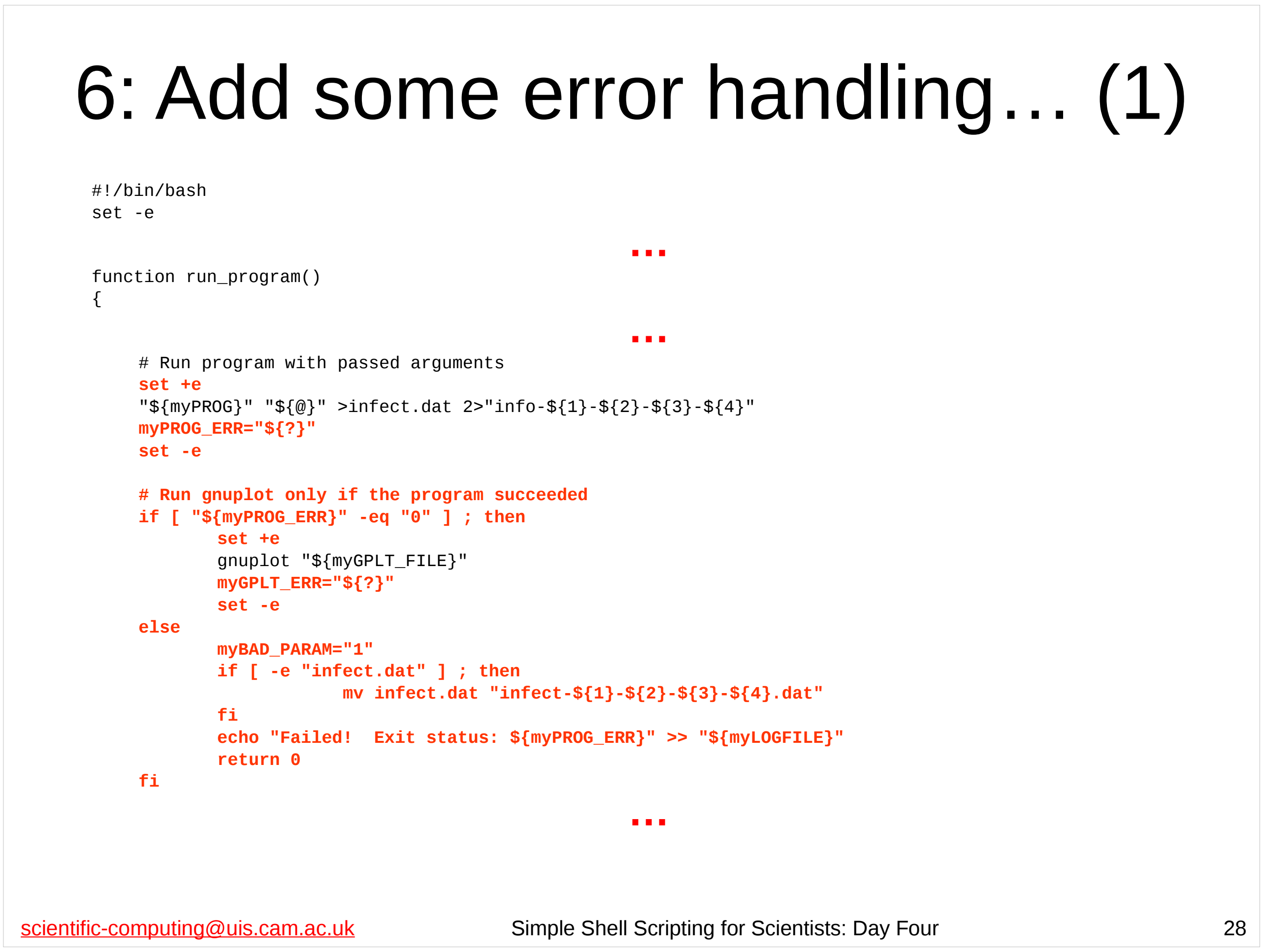
Recall that the exercise asked you to check your script's input. In this particular case, since we don't know what restrictions **`infect.py`** places on its parameters (other than the fourth one must be an integer), we'll just check that we have gotten some command line arguments, since if we haven't the script won't do anything.

You should be able to tell what all the highlighted part of the `multi-infect.sh` shell script (in the `scripts` subdirectory) above do, and you should be able to see why it will check whether or not the shell script got any command line arguments – if there is anything you don't understand, or if you had any difficulty with this part of the exercise, please let the course giver or a demonstrator know.

…and a reminder that you can test that this script works by doing the following:

```
$ cd
$ rm –f *.dat *.png info* logfile
$ cat scripts/info_params | scripts/multi-infect.sh 75 100 300 3000 50000
$ ls
```

…and examining the files produced.



6: Add some error handling… (1)

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

                                                …

function run_program()
{

                                                …

    # Run program with passed arguments
    set +e
    "${myPROG}" "${@}" >infect.dat 2>"info-${1}-${2}-${3}-${4}"
    myPROG_ERR="${?}"
    set -e

    # Run gnuplot only if the program succeeded
    if [ "${myPROG_ERR}" -eq "0" ] ; then
           set +e
           gnuplot "${myGPLT_FILE}"
           myGPLT_ERR="${?}"
           set -e
    else
           myBAD_PARAM="1"
           if [ -e "infect.dat" ] ; then
                      mv infect.dat "infect-${1}-${2}-${3}-${4}.dat"
           fi
           echo "Failed!  Exit status: ${myPROG_ERR}" >> "${myLOGFILE}"
           return 0
    fi

                                                …
```

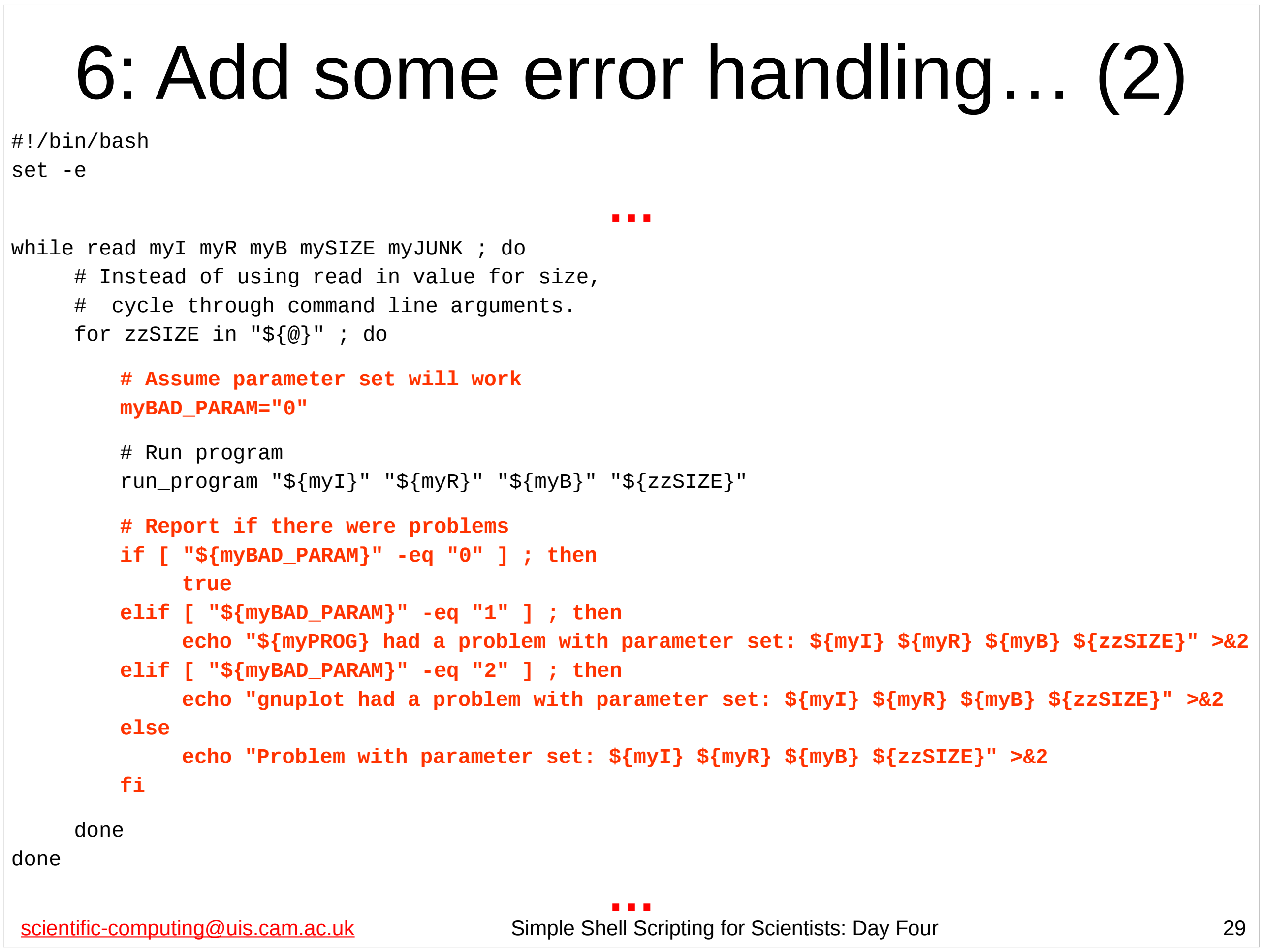
The exercise also wanted your script to do some error handling, which `multi-infect.sh` doesn't really do. Inspect the **`run_program`** function in the `multi-infect-errors.sh` script in the `scripts` subdirectory of your home directory, paying particular attention to the bits of the script highlighted above.

You should be able to work out what the highlighted bits (above) of the function are doing. (Recall that the exit status of the last command that ran is stored in the special shell parameter **`?`** and that the test **`-e "filename"`** returns true if and only if the file **`filename`** exists.)

Observe that the logic of this function is that if the **`infect.py`** program failed there's no point running **`gnuplot`** ("garbage in, garbage out"). We need to look a bit further down the function's definition (not shown above) to see what it does if **`gnuplot`** fails. Can you work out what it is doing (and why)?

Also note that we make sure we rename the `infect.dat` file (if it exists) even if **`infect.py`** fails, since we might well want to inspect the output of a failed run.

If you are not sure, or you have any questions, please ask the course giver or a demonstrator.



6: Add some error handling… (2)

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

                              ...

while read myI myR myB mySIZE myJUNK ; do
     # Instead of using read in value for size,
     #  cycle through command line arguments.
     for zzSIZE in "${@}" ; do

         # Assume parameter set will work
         myBAD_PARAM="0"

         # Run program
         run_program "${myI}" "${myR}" "${myB}" "${zzSIZE}"

         # Report if there were problems
         if [ "${myBAD_PARAM}" -eq "0" ] ; then
              true
         elif [ "${myBAD_PARAM}" -eq "1" ] ; then
              echo "${myPROG} had a problem with parameter set: ${myI} ${myR} ${myB} ${zzSIZE}" >&2
         elif [ "${myBAD_PARAM}" -eq "2" ] ; then
              echo "gnuplot had a problem with parameter set: ${myI} ${myR} ${myB} ${zzSIZE}" >&2
         else
              echo "Problem with parameter set: ${myI} ${myR} ${myB} ${zzSIZE}" >&2
         fi

     done
done

                              ...
```

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 29

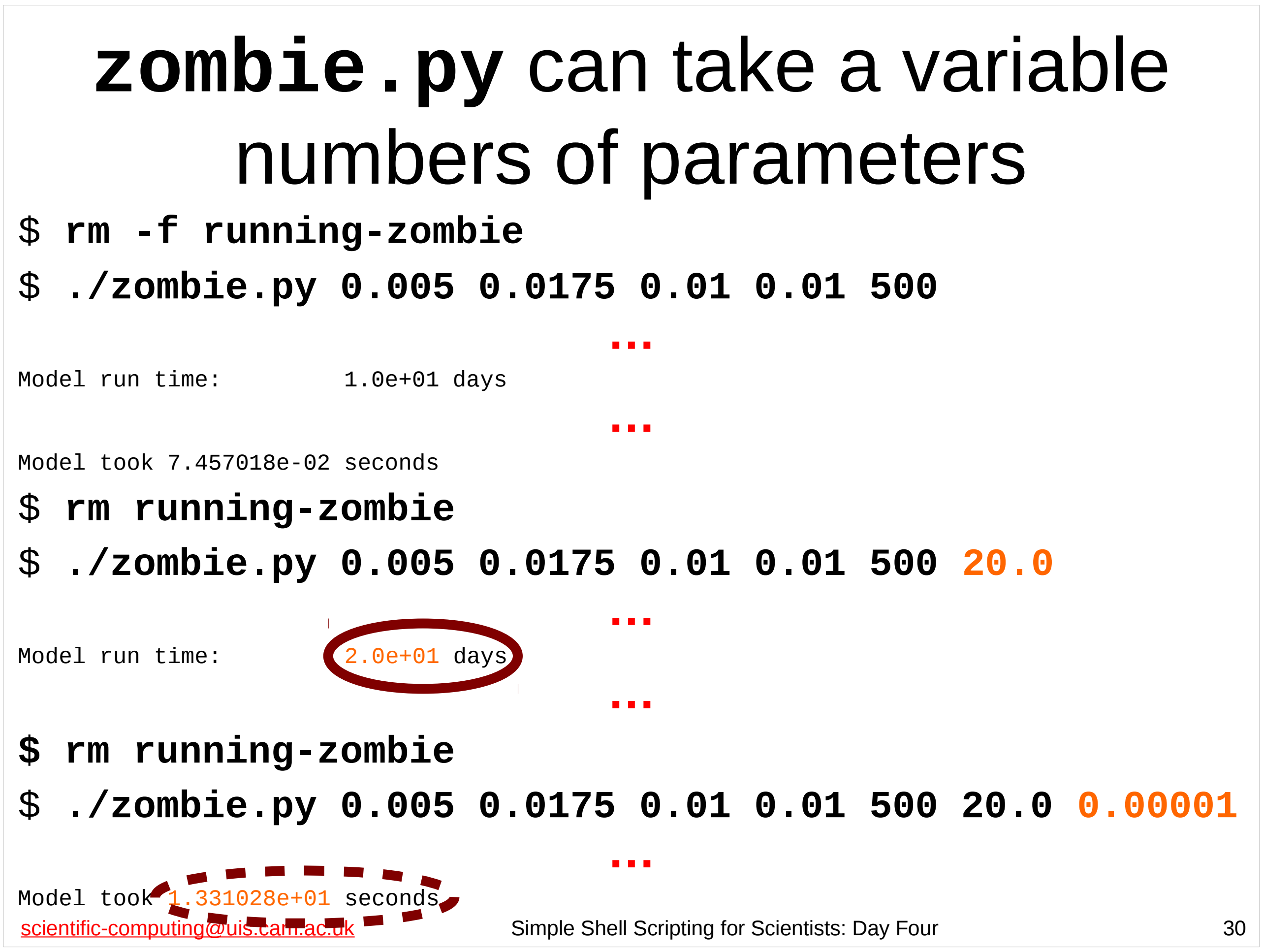
Now inspect the main body of the `multi-infect-errors.sh` script in the `scripts` subdirectory of your home directory, paying particular attention to the bits of the script highlighted above. You should be able to work out why we've changed it like this, what it does and why it works.

If you have any questions, or there's anything you don't understand, please ask the course giver or a demonstrator.

You can test this script works by doing the following:

```
$ cd
$ rm –f *.dat *.png info* logfile
$ cat scripts/infect_params | scripts/multi-infect-errors.sh 75 100 300 3000 50000
$ ls
```

…and examining the files produced.



zombie.py can take a variable numbers of parameters

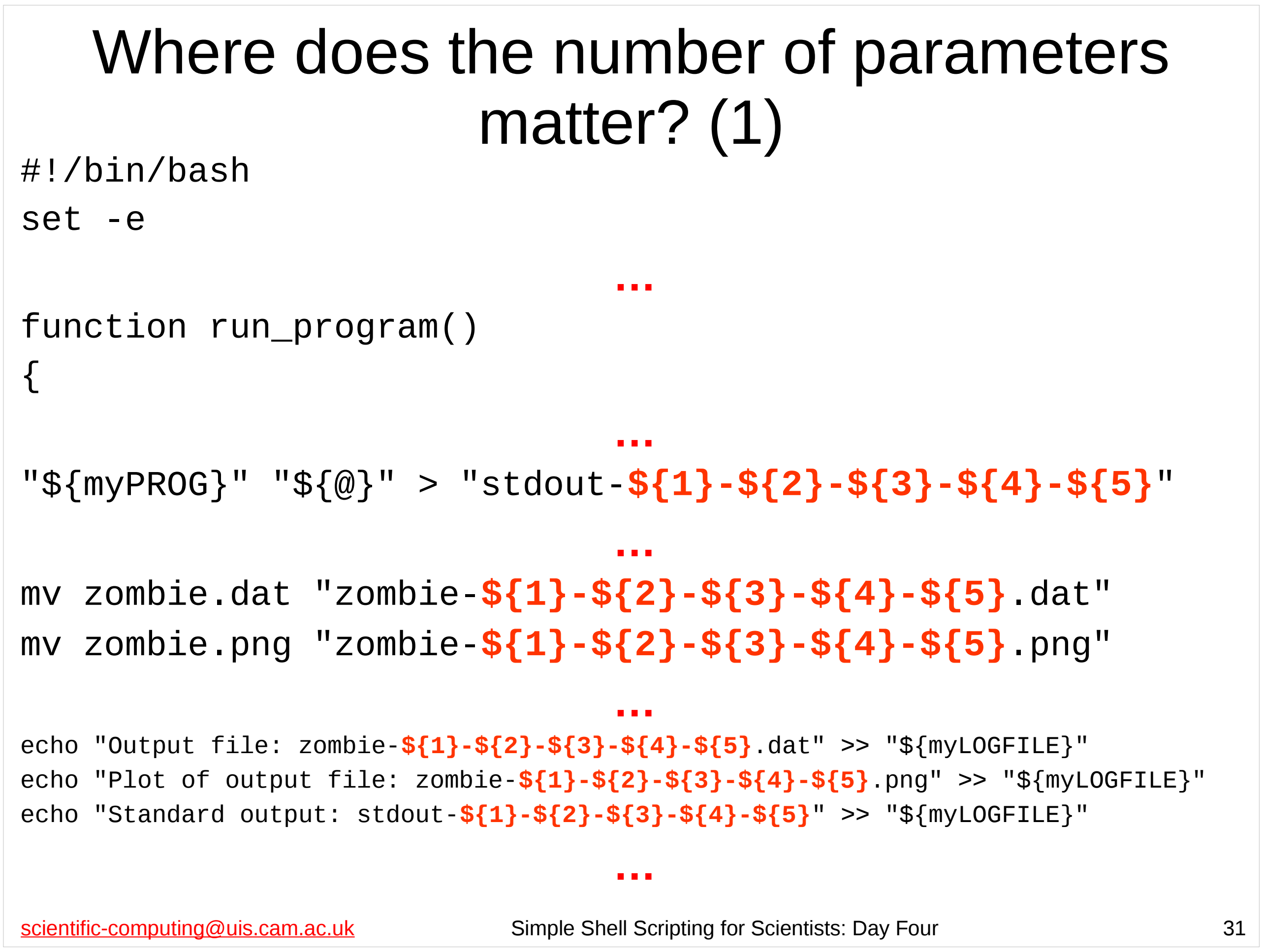
```
$ rm -f running-zombie
$ ./zombie.py 0.005 0.0175 0.01 0.01 500
                                                 ...
Model run time:          1.0e+01 days
                                                 ...
Model took 7.457018e-02 seconds
$ rm running-zombie
$ ./zombie.py 0.005 0.0175 0.01 0.01 500 20.0
                                                 ...
Model run time:          2.0e+01 days
                                                 ...
$ rm running-zombie
$ ./zombie.py 0.005 0.0175 0.01 0.01 500 20.0 0.00001
                                                 ...
Model took 1.331028e+01 seconds
```

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 30

Recall the **zombie.py** program is in your home directory that we have been using for most of our scripts on the previous days of the course. Previously, we've been giving the **zombie.py** program 5 numeric arguments on the command line: 4 positive floating-point numbers and 1 positive integer. However, the program can actually take *up to 7* numeric arguments on the command line. The last two arguments are optional (and, if specified, must be positive floating-point numbers). (It would be nice if the **zombie.py** program was well written enough to tell us it took optional arguments, but, alas, such poor documentation is sadly typical of many programs you will find yourself using.)

zombie.py's sixth command line argument (if specified) tells **zombie.py** the number of days for which we want to model the zombie outbreak. If this argument is not specified then it is assumed to be 10.0. If we specify this argument we can tell it has been accepted by looking at the value printed on the screen for the "Model run time".

zombie.py's seventh command line argument (if specified) tells **zombie.py** the size of the time step to use in the model. If this argument is not specified then it is set to the number of days divided by 10,000. Sadly, **zombie.py** does not tell us the size of the time step it is using, but we can tell that changing the size of the time step has had some effect by looking at how many points have been generated in the `zombie.dat` output file or observing how long the model takes to run – the smaller the time step, the more points will be generated and so the longer the model takes to run.



Where does the number of parameters matter? (1)

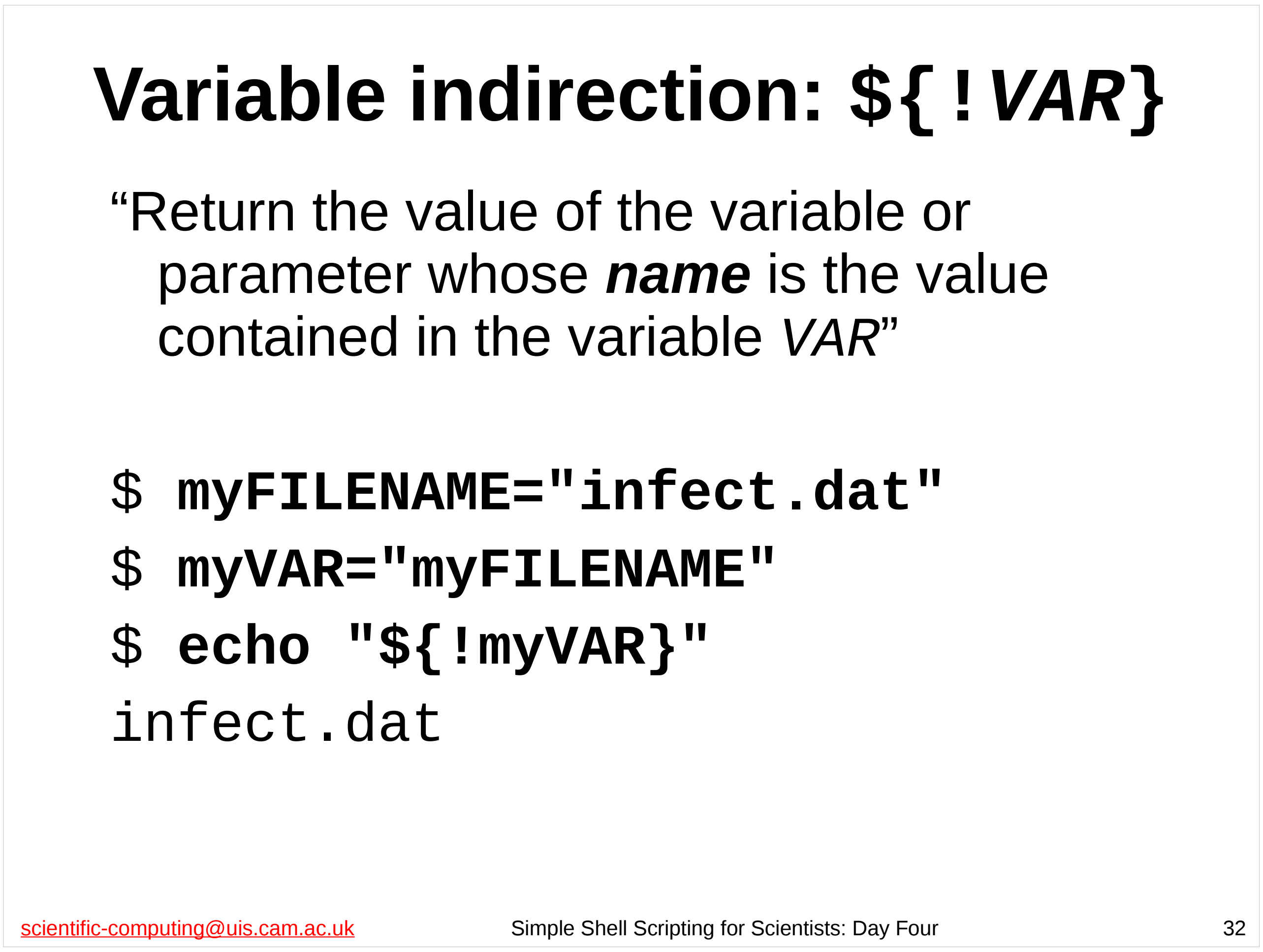
```
#!/bin/bash
set -e
                                    …
function run_program()
{
                                    …
"${myPROG}" "${@}" > "stdout-${1}-${2}-${3}-${4}-${5}"
                                    …
mv zombie.dat "zombie-${1}-${2}-${3}-${4}-${5}.dat"
mv zombie.png "zombie-${1}-${2}-${3}-${4}-${5}.png"
                                    …
echo "Output file: zombie-${1}-${2}-${3}-${4}-${5}.dat" >> "${myLOGFILE}"
echo "Plot of output file: zombie-${1}-${2}-${3}-${4}-${5}.png" >> "${myLOGFILE}"
echo "Standard output: stdout-${1}-${2}-${3}-${4}-${5}" >> "${myLOGFILE}"
                                    …
```

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 31

Obviously if we are going to handle variable numbers of parameters for **`zombie.py`**, we will need to make some changes to the script running it. If we were going to modify the `multi-run-while.sh` script (in the `scripts` directory), what would we need to change?

The first thing we need to do is to identify those parts of the script where the number of parameters being used by **`zombie.py`** matters to how the script works. One of the most obvious places is in the `run_program` function that actually runs **`zombie.py`**, since many of the files created or renamed by that function use the individual parameters for **`zombie.py`** in their names. If we inspect this function, we see that we use the string "`${1}-${2}-${3}-${4}-${5}`" in the names of several files. If **`zombie.py`** took six parameters, we would have to use "`${1}-${2}-${3}-${4}-${5}-${6}`", and if it took seven parameters, we would have to use "`${1}-${2}-${3}-${4}-${5}-${6}-${7}`".

Can we construct a generally useful function that can generate these sorts of strings for us?



Variable indirection: `${!VAR}`

“Return the value of the variable or parameter whose ***name*** is the value contained in the variable *VAR*”

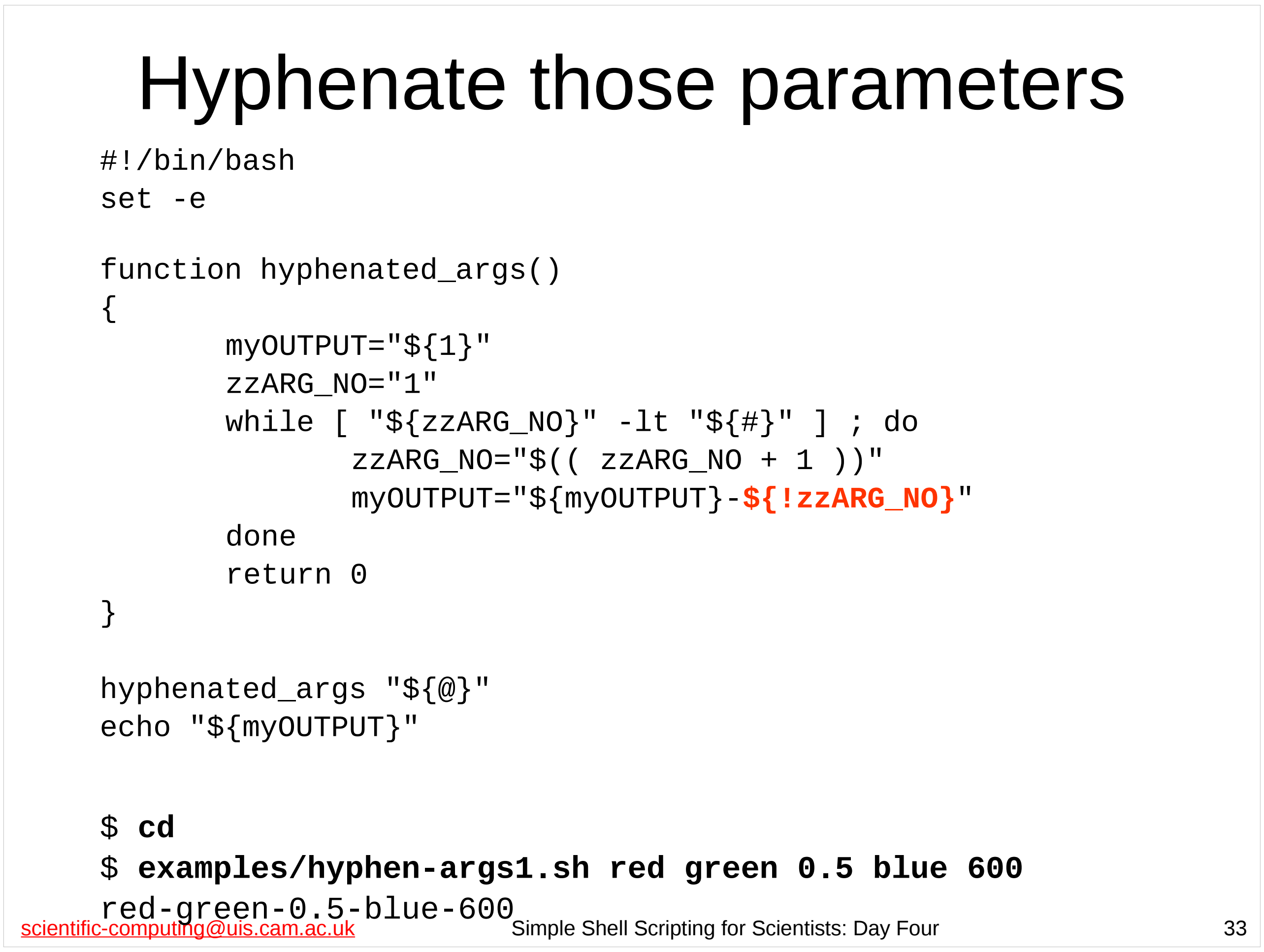
```
$ myFILENAME="infect.dat"
$ myVAR="myFILENAME"
$ echo "${!myVAR}"
infect.dat
```

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 32

Sometimes you might want to store the *name* of a variable in another variable. If you need to do this, you’ll need to use a special form of *parameter expansion* to actually get the value of the variable whose name you have stored: this is called *variable indirection*. (We’ve already met the simplest form of parameter expansion: **`${VARIABLE}`**, which just gives us the value of the environment variable, shell variable or parameter **`VARIABLE`**. We’ll meet some other sorts of parameter expansion later.)

As you can see from the example above, what happens is that the shell takes the name contained in the specified variable and treats it as the name of another variable or parameter whose value it returns.

We’ll see an example of where this is useful in a moment…



Hyphenate those parameters

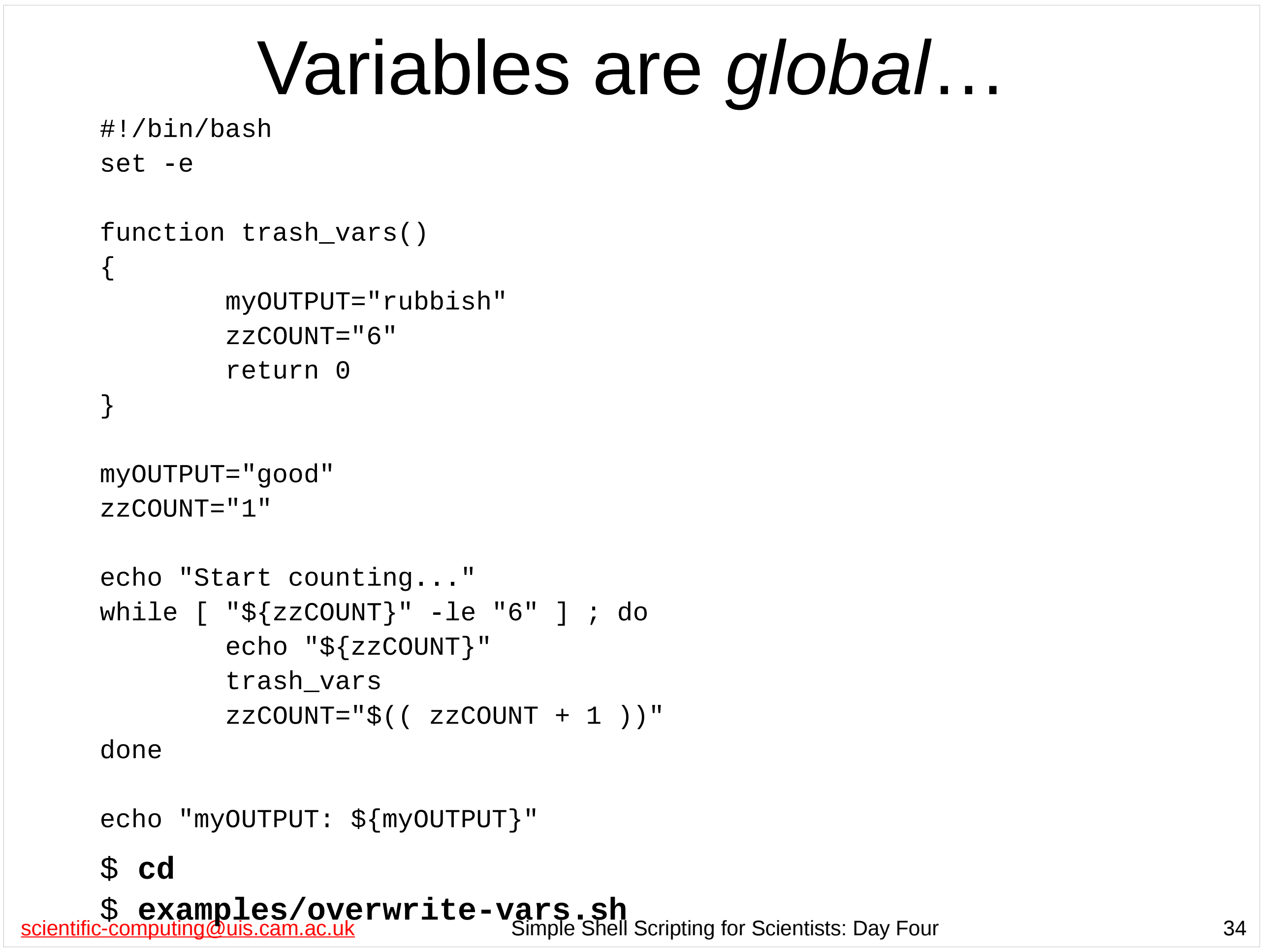
```
#!/bin/bash
set -e

function hyphenated_args()
{
        myOUTPUT="${1}"
        zzARG_NO="1"
        while [ "${zzARG_NO}" -lt "${#}" ] ; do
                zzARG_NO="$(( zzARG_NO + 1 ))"
                myOUTPUT="${myOUTPUT}-${!zzARG_NO}"
        done
        return 0
}

hyphenated_args "${@}"
echo "${myOUTPUT}"


$ cd
$ examples/hyphen-args1.sh red green 0.5 blue 600
red-green-0.5-blue-600
```

So here's a function (in the `hyphen-params1.sh` script in the `examples` directory) that, given any number of arguments, will put hyphens between the arguments and place the string thus constructed in a shell variable. (Recall that the number of arguments for a shell script or a function is contained in the special parameter `#`.)

Note the use of variable indirection to get, in turn, the individual arguments passed to the function. (There's another, arguably better/simpler, way we can do this using the **`shift`** shell builtin command that is briefly mentioned at the very end of this course.)

We might want to use a function like this whenever we have a program (like **`zombie.py`**) that takes a variable number of arguments and we want to use those arguments in the name of one or more files. This clearly is a function that might be useful in many different scripts we might write, so how can we easily make use of the same function in many different scripts?

Well, first there's a problem we need to address…



Variables are *global*…

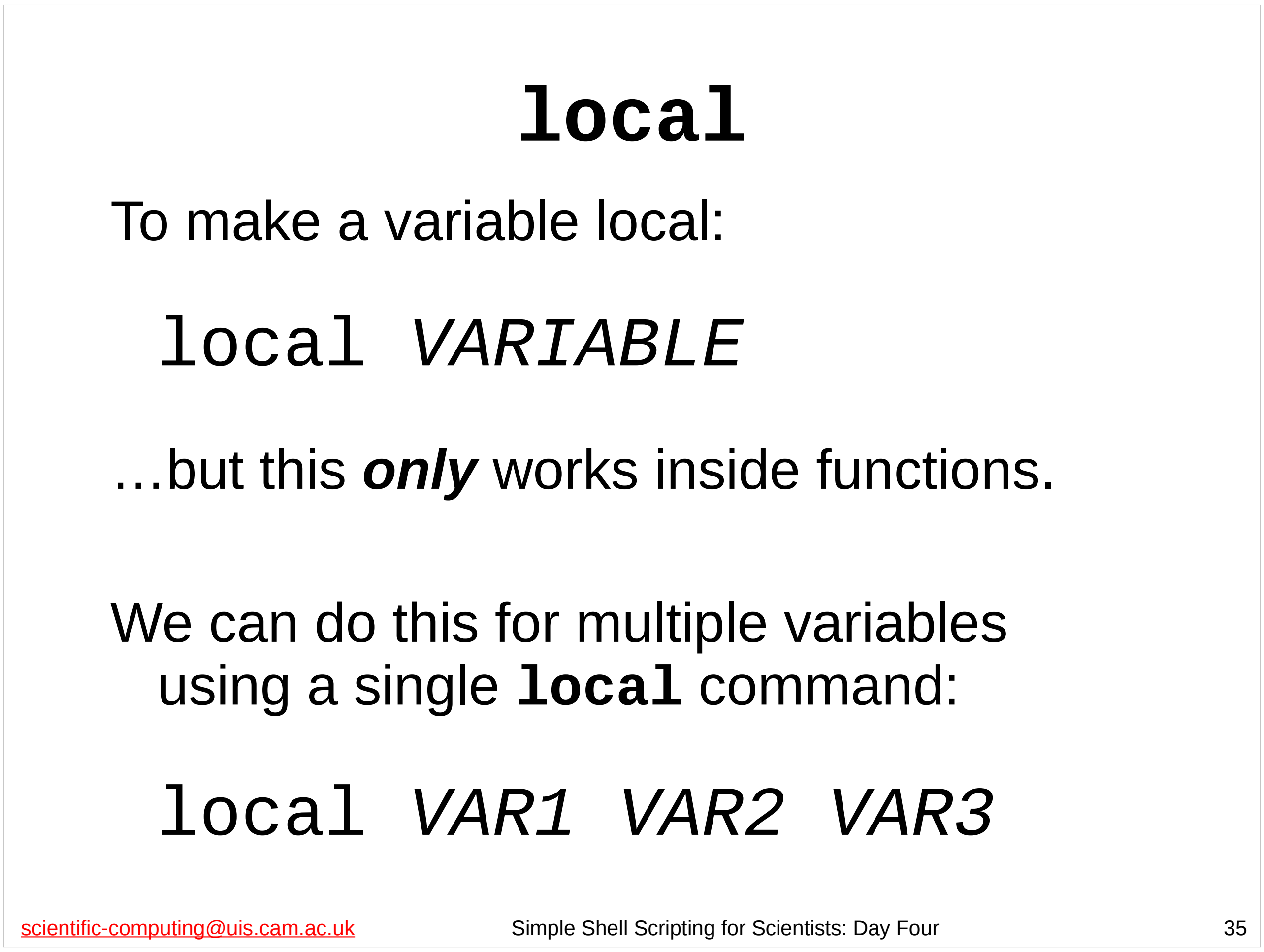
```
#!/bin/bash
set -e

function trash_vars()
{
        myOUTPUT="rubbish"
        zzCOUNT="6"
        return 0
}

myOUTPUT="good"
zzCOUNT="1"

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

echo "myOUTPUT: ${myOUTPUT}"
```

```
$ cd
$ examples/overwrite-vars.sh
```

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 34

By default, shell variables are *global*, i.e. we can read and modify there values both in our main script and in any functions we create. Whilst sometimes this can be useful, it can also cause problems if you forget this and accidentally re-use a variable name in a function that you are using elsewhere for something different. It is particularly a problem with loop variables, as we see in the example on the slide above.

So if we have a function that we want to use in lots of different scripts, we either need to make sure we keep careful track of what variables it uses and don't use them in ***any*** of our scripts, or we need to find a way to make variables *local* (i.e. only exist within the function and not affect the variables in the rest of the script, or in other functions).



`local`

To make a variable local:

```
local VARIABLE
```

…but this ***only*** works inside functions.

We can do this for multiple variables using a single **`local`** command:

```
local VAR1 VAR2 VAR3
```

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 35

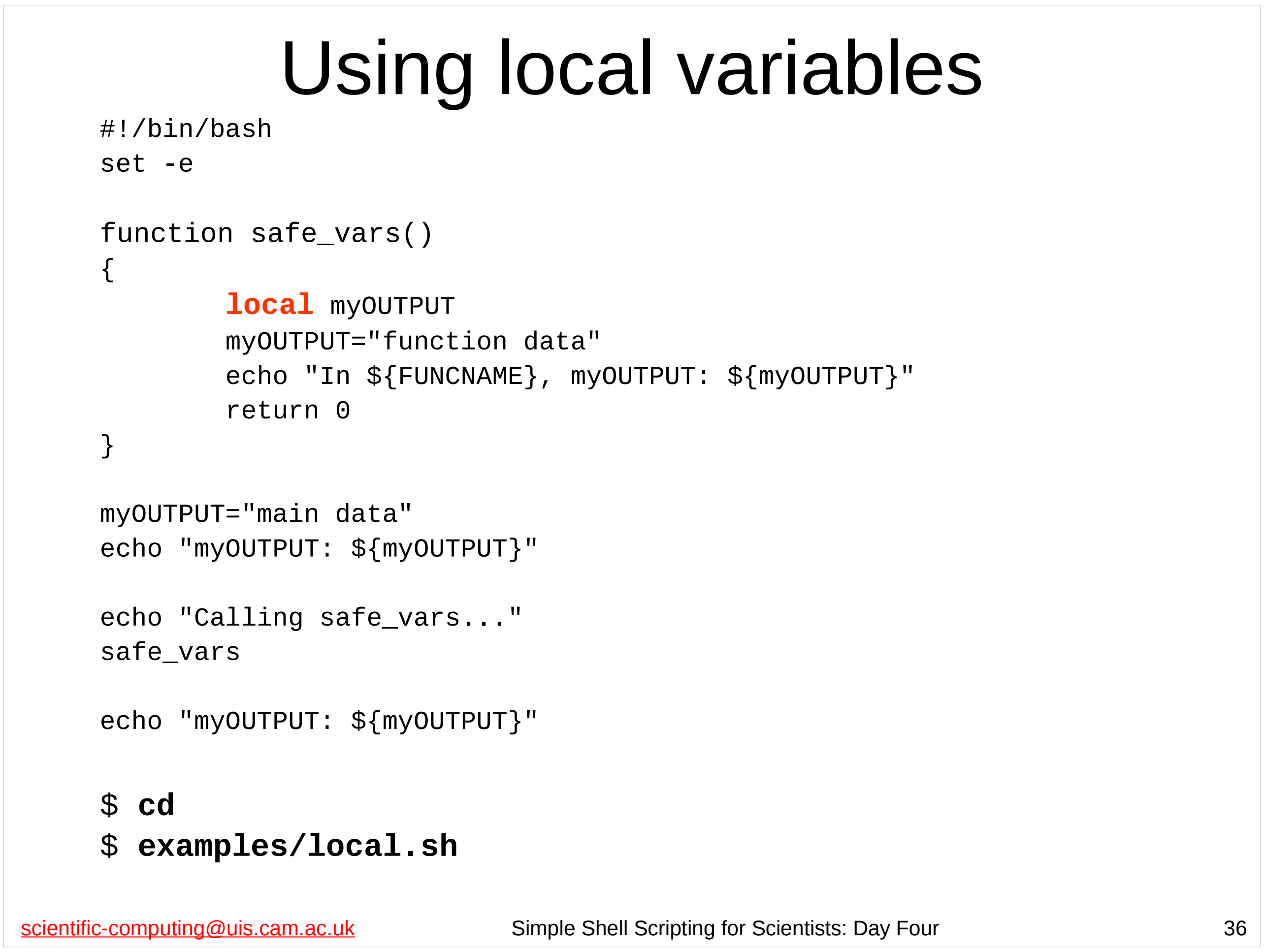
The **`local`** shell builtin command, when used within a function, makes a variable local to that function. A local variable doesn't affect any other variable that has the same name, and can only be read and modified in the function in which it has been created. When the function is finished any local variables it had are destroyed.

You cannot use **`local`** outside of a function.

You can create a local variable and set its value all at once if you want like this:

```
local VARIABLE=value
```

…where ***`VARIABLE`*** is the name of the variable and ***`value`*** is its value.



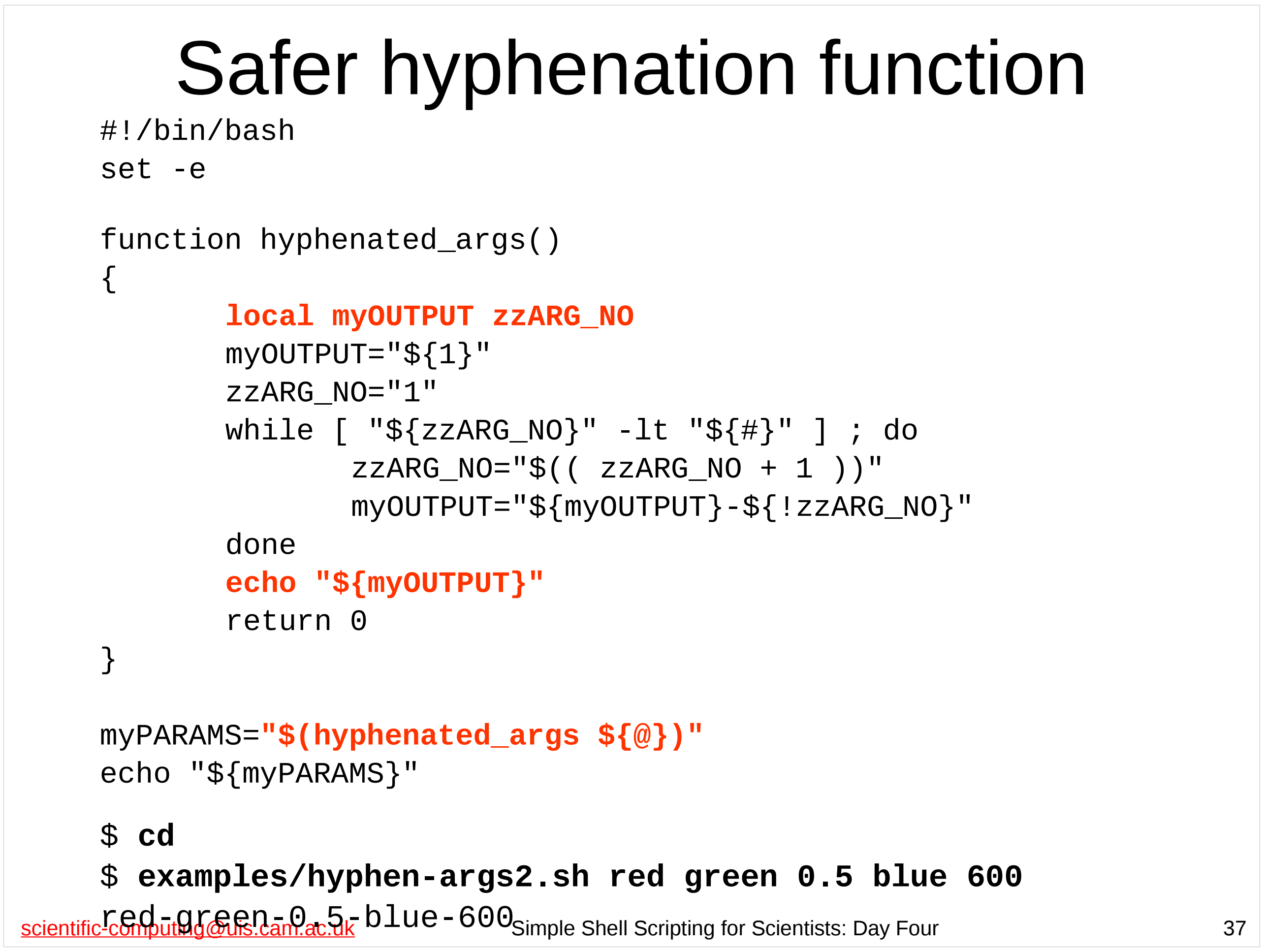
Using local variables

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

function safe_vars()
{
        local myOUTPUT
        myOUTPUT="function data"
        echo "In ${FUNCNAME}, myOUTPUT: ${myOUTPUT}"
        return 0
}

myOUTPUT="main data"
echo "myOUTPUT: ${myOUTPUT}"

echo "Calling safe_vars..."
safe_vars

echo "myOUTPUT: ${myOUTPUT}"
```

```
$ cd
$ examples/local.sh
```

By default, shell variables are *global*, i.e. we can read and modify there values both in our main script and in any functions we create. Whilst sometimes this can be useful, it can also cause problems if you forget this and accidentally re-use a variable name in a function that you are using elsewhere for something different. It is particularly a problem with loop variables, as we see in the example on the slide above.

So if we have a function that we want to use in lots of different scripts, we either need to make sure we keep careful track of what variables it uses and don't use them in ***any*** of our scripts, or we need to find a way to make variables *local* (i.e. only exist within the function and not affect the variables in the rest of the script, or in other functions).



Safer hyphenation function

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

function hyphenated_args()
{
        local myOUTPUT zzARG_NO
        myOUTPUT="${1}"
        zzARG_NO="1"
        while [ "${zzARG_NO}" -lt "${#}" ] ; do
                zzARG_NO="$(( zzARG_NO + 1 ))"
                myOUTPUT="${myOUTPUT}-${!zzARG_NO}"
        done
        echo "${myOUTPUT}"
        return 0
}

myPARAMS="$(hyphenated_args ${@})"
echo "${myPARAMS}"
```

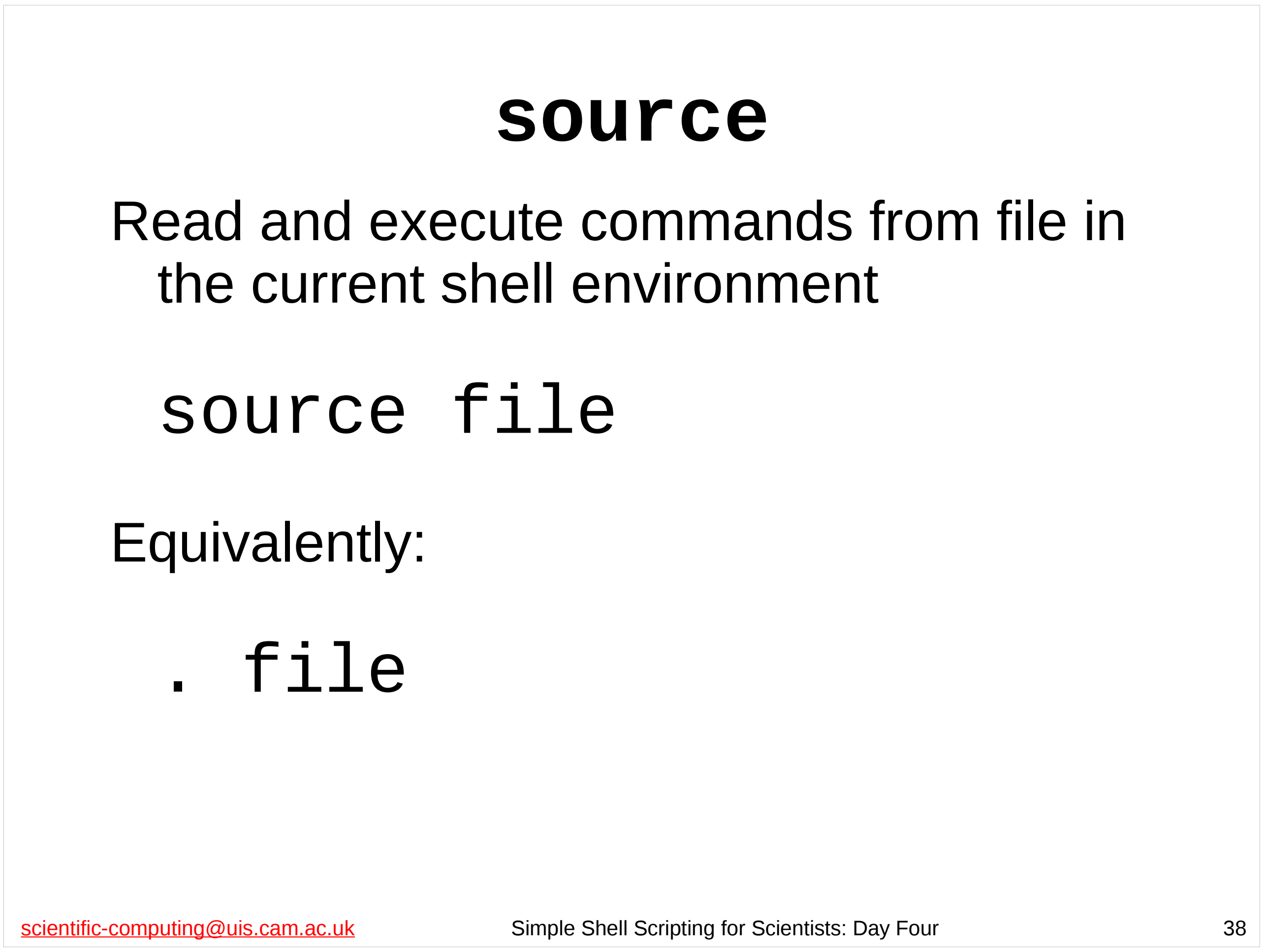
```
$ cd
$ examples/hyphen-args2.sh red green 0.5 blue 600
red-green-0.5-blue-600
```

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 37

So here’s a safer version of our hyphenation function (in the `hyphen-params2.sh` script in the `examples` directory) that, given any number of arguments, will put hyphens between the arguments and write the string thus constructed to standard output. (Note the use of the **`local`** shell builtin command.)

The most likely way that we would use this function is via command substitution, as shown in the `hyphen-params2.sh` script above. But regardless of whether or not we use command substitution, we can safely use this function in any script we like without worrying what variables are used in the script: even if they have the same names as the ones in this `hyphenated_args` function it won’t matter.

So, how do we use a function in lots of different scripts?



`source`

Read and execute commands from file in the current shell environment

```
source file
```

Equivalently:

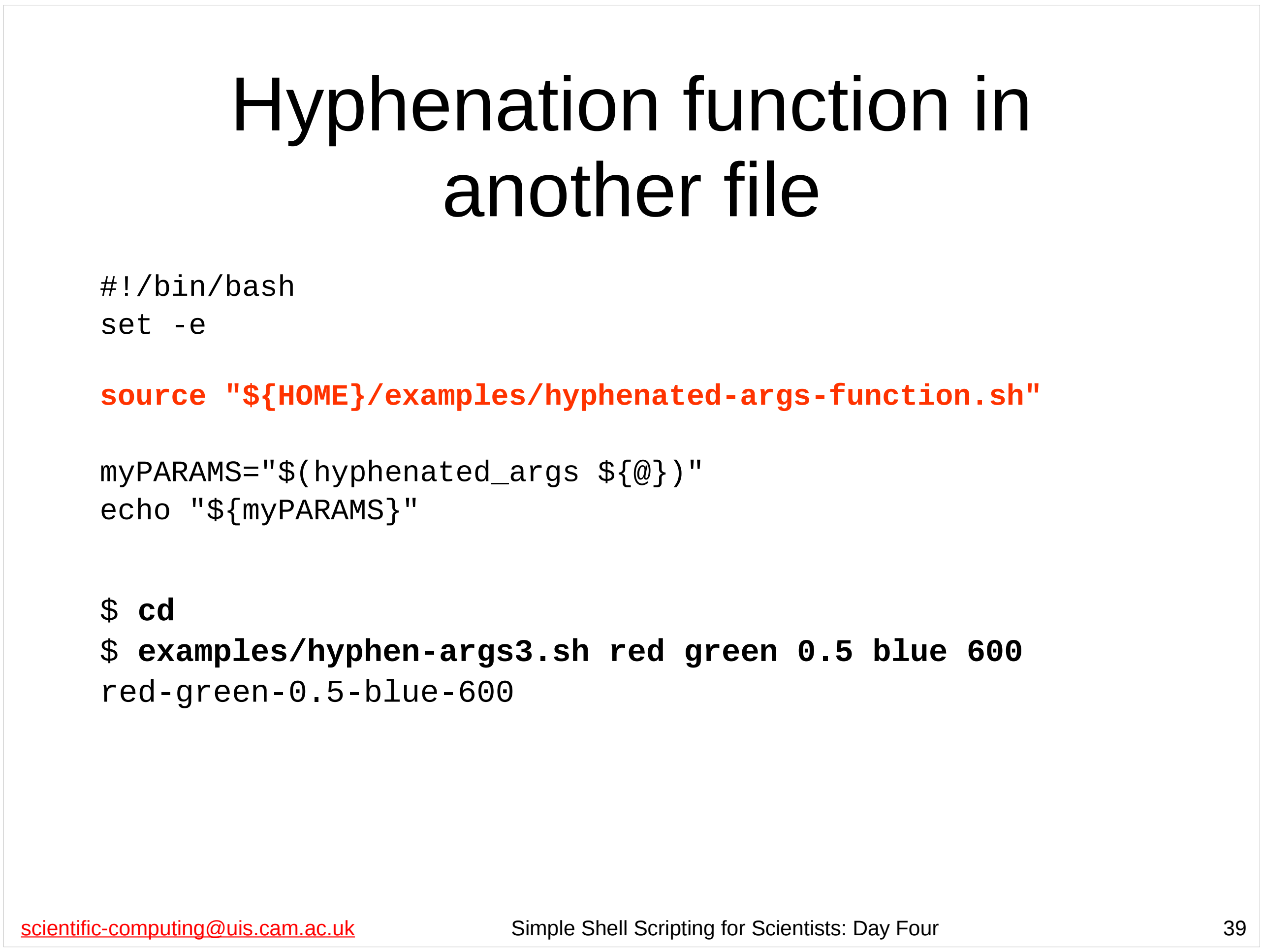
```
. file
```

`source` executes one shell script in the environment of the current shell script (or shell) – it is as though you had copied the shell script and pasted it into your current shell script. A synonym for `source` is “`.`”, i.e.

```
source filename
. filename
```

do the same thing – they both execute the contents of the file **`filename`** in the environment of the current shell script (or shell).

If your shell script just defines some functions, then using **`source`** on it will just define those functions for you in your current shell script (or shell). When used this way, you can think of the shell script containing the functions as a “library” of functions, and the **`source`** command as “loading” that library into the current script (or into the shell itself if you use it in an instance of the shell).



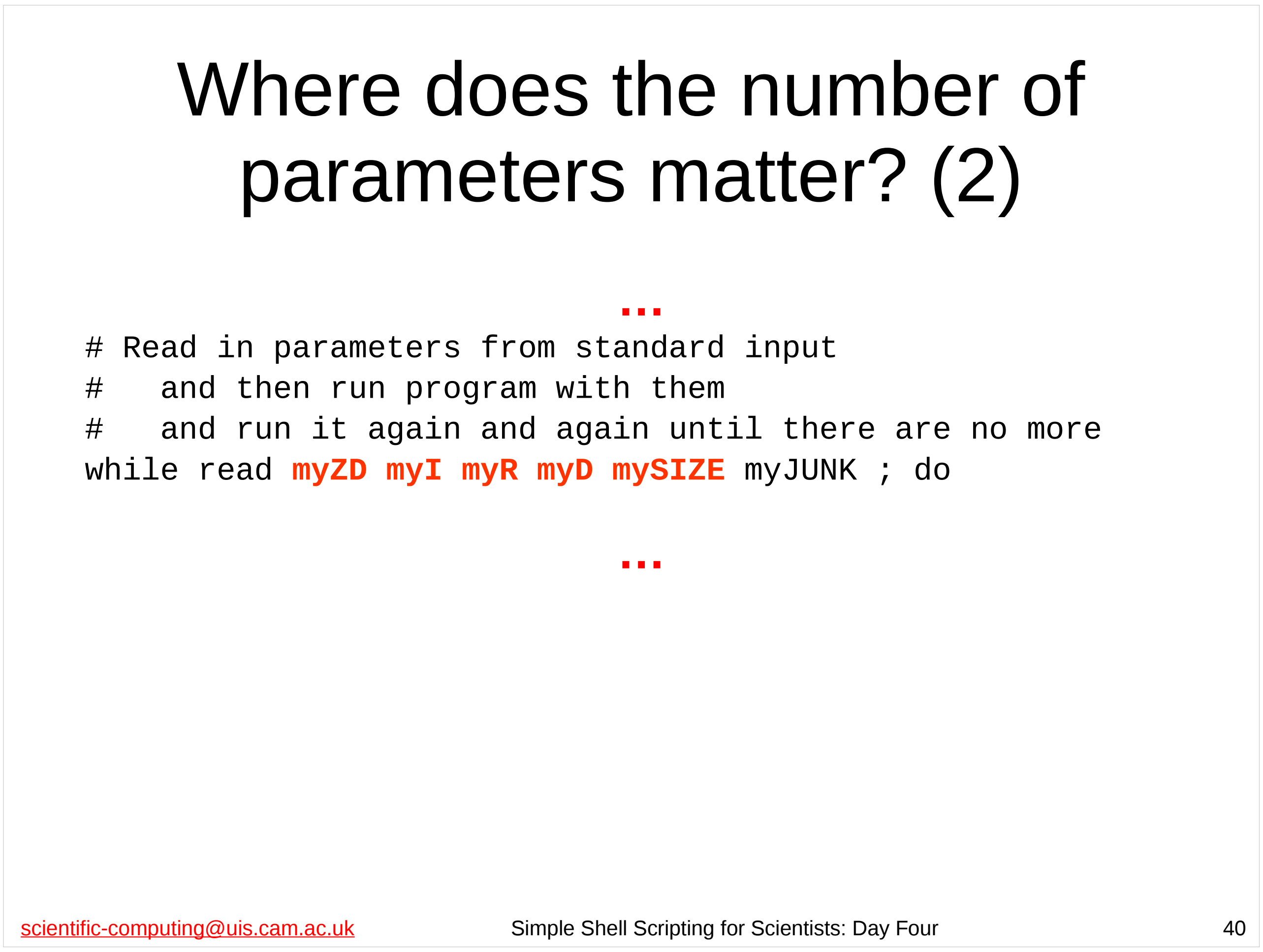
Hyphenation function in another file

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

source "${HOME}/examples/hyphenated-args-function.sh"

myPARAMS="$(hyphenated_args ${@})"
echo "${myPARAMS}"
```

```
$ cd
$ examples/hyphen-args3.sh red green 0.5 blue 600
red-green-0.5-blue-600
```

So here's a script (`hyphen-params3.sh` in the `examples` directory) that uses the same hyphenation function as before, but the function is defined in a separate script. (Note the use of the **`source`** shell builtin command.)

A common practice is to put a lot of useful functions in a shell script that *only* defines functions and then use **`source`** to read that file in to lots of different shell scripts that one writes.



Where does the number of parameters matter? (2)

...

```
# Read in parameters from standard input
#   and then run program with them
#   and run it again and again until there are no more
while read myZD myI myR myD mySIZE myJUNK ; do
```

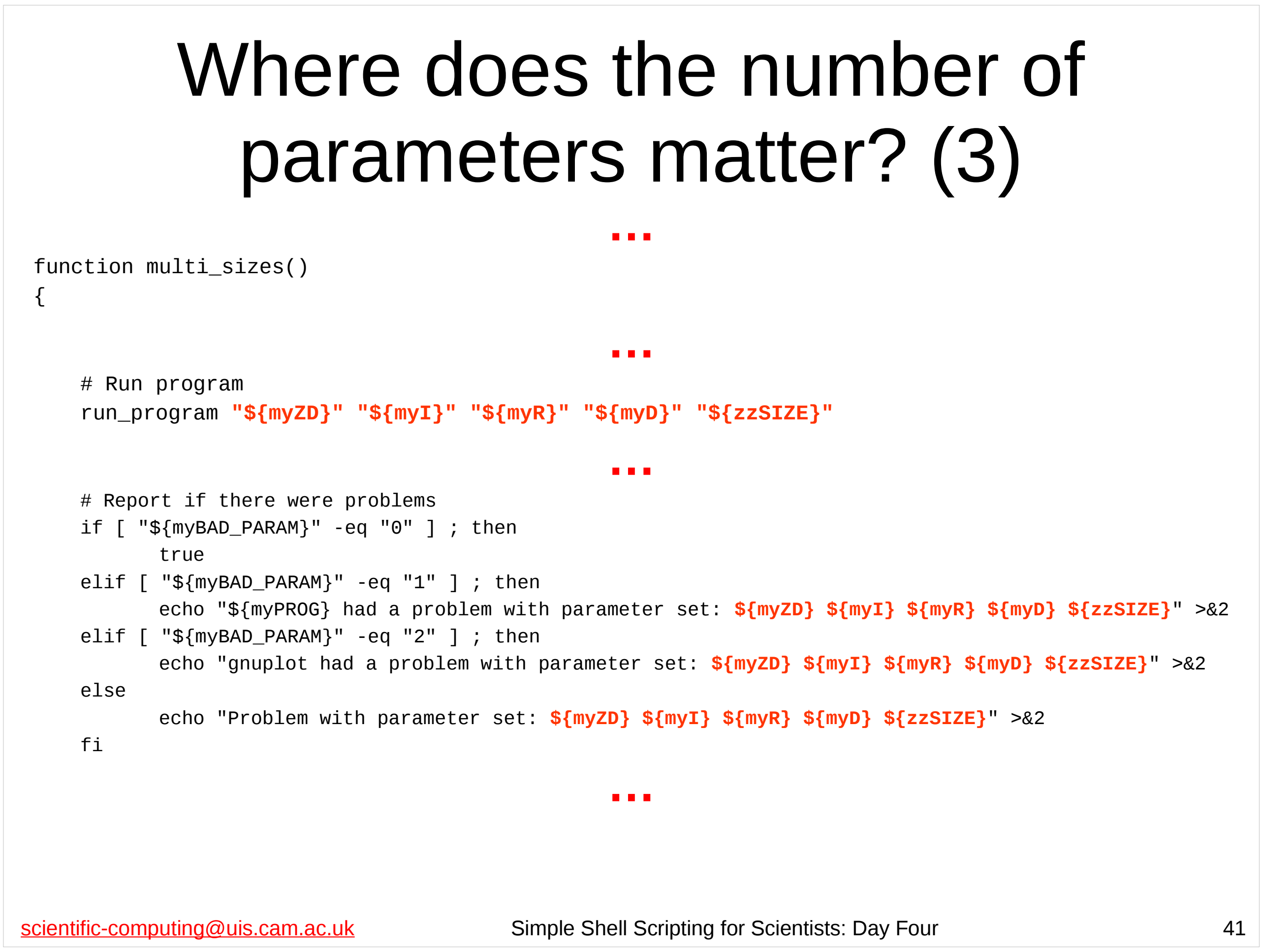
...

We've seen one function (the `run_program` function) that we would need to modify if we are going to handle variable numbers of parameters for **`zombie.py`**. (We looked at this function in the `multi-run-while.sh` script (in the `scripts` directory), but we'd need to make the same sort of modifications to any of the `run_program` functions in any of our scripts that run **`zombie.py`**.)

But what else would we need to change?

If we look at the `multi-sizes-errors.sh` script (in the `scripts` directory) we note that this script, like several of the others we've created for running **`zombie.py`**, `reads` in the parameters for **`zombie.py`** from standard input and expects to `read` in *five* parameters. So if **`zombie.py`** could take up to seven parameters, then we need to modify this part of the script to read in up to seven parameters, e.g. by changing the `while read` line to:

```
while read myZD myI myR myD mySIZE myTIME myTSTEP myJUNK ; do
```



Where does the number of parameters matter? (3)

...

```
function multi_sizes()
{
```

...

```
    # Run program
    run_program "${myZD}" "${myI}" "${myR}" "${myD}" "${zzSIZE}"
```

...

```
    # Report if there were problems
    if [ "${myBAD_PARAM}" -eq "0" ] ; then
           true
    elif [ "${myBAD_PARAM}" -eq "1" ] ; then
           echo "${myPROG} had a problem with parameter set: ${myZD} ${myI} ${myR} ${myD} ${zzSIZE}" >&2
    elif [ "${myBAD_PARAM}" -eq "2" ] ; then
           echo "gnuplot had a problem with parameter set: ${myZD} ${myI} ${myR} ${myD} ${zzSIZE}" >&2
    else
           echo "Problem with parameter set: ${myZD} ${myI} ${myR} ${myD} ${zzSIZE}" >&2
    fi
```

...

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 41

We would also need to change the way we call the `run_program` function, since it is when we call this function that we give it the command line arguments that **`zombie.py`** will actually use. In the `multi-sizes-errors.sh` script (in the `scripts` directory) this happens in the `multi_sizes` function, so we'll need to modify this function as well.

Can you guess how we'll have to change this function?

First exercise

Copy the `multi-sizes-errors.sh` script to `multi-variable-params.sh` and modify that so that it can handle variable numbers of parameters for **`zombie.py`**:

1) Modify the `run_program` function to use the `hyphenate_args` function in the `my-functions.sh` file in the `scripts` directory.
2) Modify the `while read` line in the main body of the script so that it `read`s in seven parameters instead of five.
3) Modify the `multi_sizes` functions so that if the script has `read` seven arguments for **`zombie.py`**, it calls `run_program` with seven arguments; if it has `read` six arguments, it calls `run_program` with six arguments, and if it has `read` five arguments, it calls `run_program` with five arguments.

15 minutes

The purpose of this exercise is to copy the `multi-sizes-errors.sh` script (in the `scripts` subdirectory) to a script called multi-variable-params.sh and then modify that script so that it can cope with parameter sets that have five, six or seven parameters in them for **`zombie.py`** to take as its arguments. Everything you need to do this exercise we've either just covered or was covered on previous days of the course.

To read in the definition of the `hyphenate_args` function (which is in the `my-functions.sh` script in the `scripts` directory) use the **`source`** shell builtin command.

When modifying the `multi_sizes` function, one approach you might consider is to do some sort of test on the seventh parameter `read` in by the script to see whether or not it contains any value; if so, you would then call the `run_program` function with seven parameters. You could then do something similar for the sixth parameter, and then, if nothing was `read` in for either of those parameters, you would call the `run_program` function with five parameters (as is currently done in the `multi-sizes-errors.sh` script).

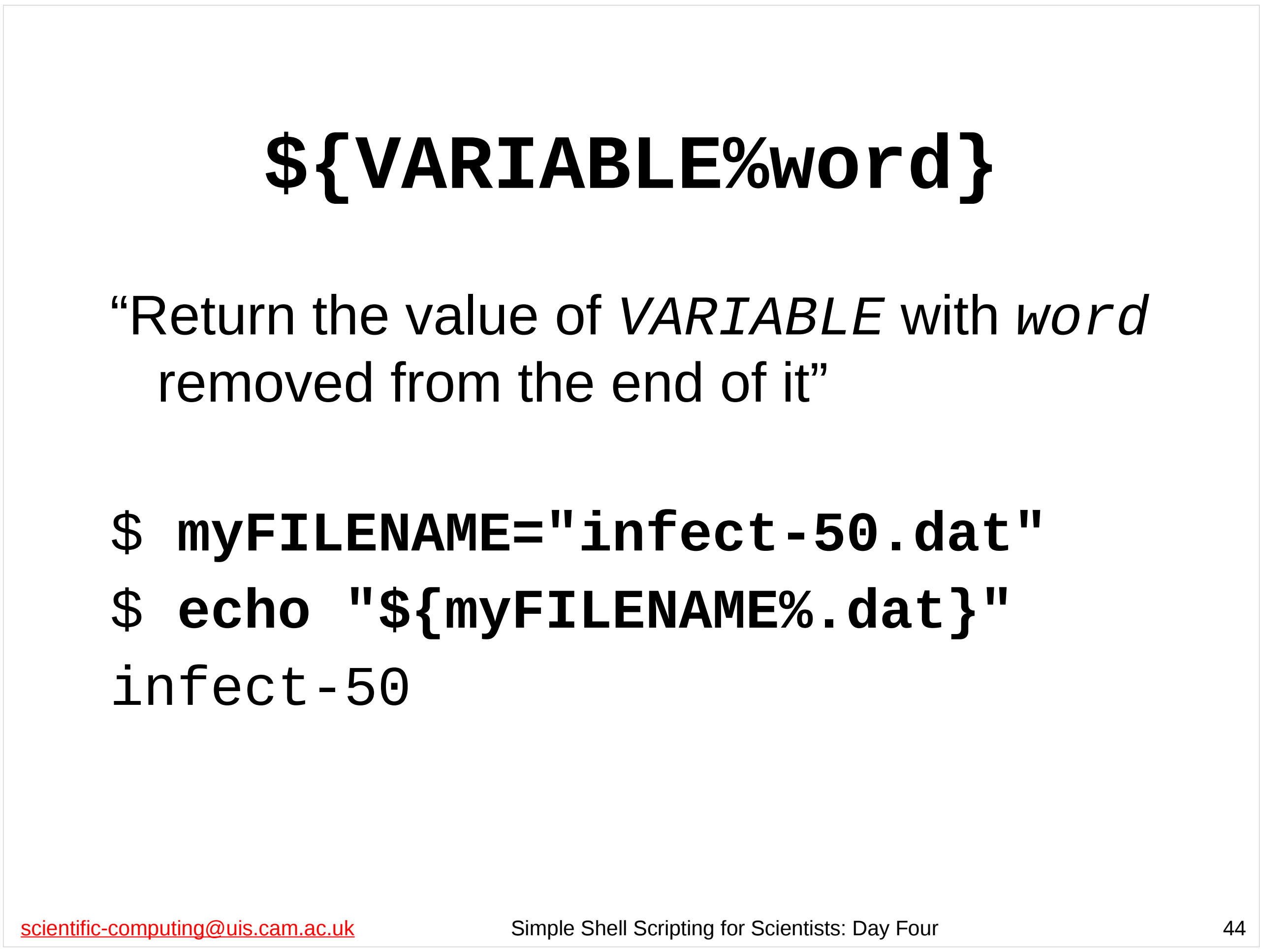
You can find a file with parameter sets with variable numbers of parameters in them in the `variable_params` file in the `scripts` directory.

When you finish this exercise, take a short break and then we'll start again with the solution. (I really ***do*** mean take a break – sitting in front of computers for long periods of time is very bad for you. Move around, go for a jog, do some aerobics, whatever…)

Manipulating filenames (1)

On the second day of this course we met some scripts that, given a series of `.dat` files produced by **`infect.py`**, would run **`gnuplot`** on each of these files.

These scripts would take a file called `infect-50.dat` and produce a file called `infect-50.dat.png`. Can we make these scripts use better names for the files they create (e.g. `infect-50.png`)?



${VARIABLE%word}

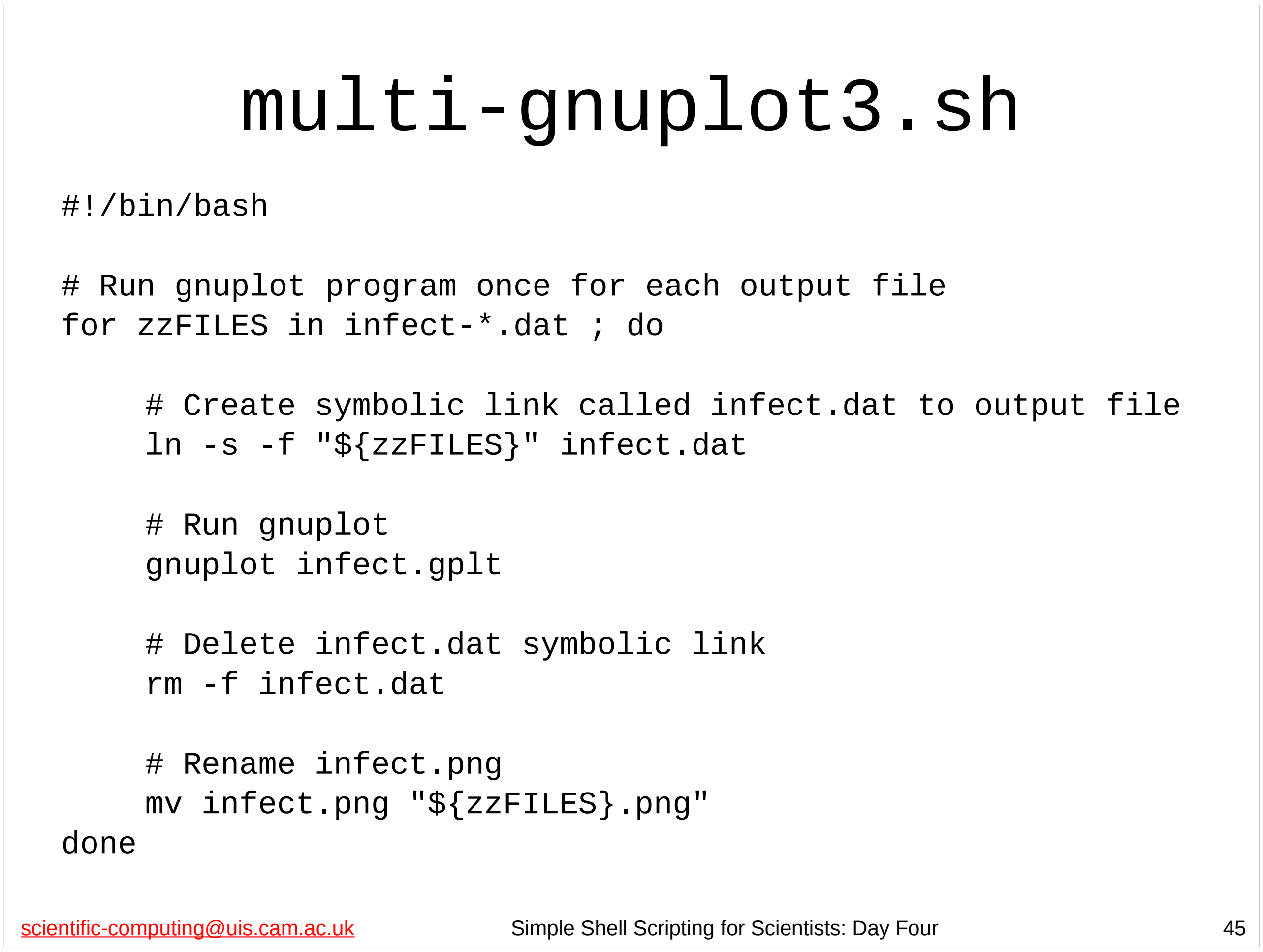
"Return the value of *VARIABLE* with *word* removed from the end of it"

```
$ myFILENAME="infect-50.dat"
$ echo "${myFILENAME%.dat}"
infect-50
```

This strange looking operation is a form of what is known as *parameter expansion*. We've already met the simplest form of parameter expansion: **`${VARIABLE}`**, which just gives us the value of the environment variable, shell variable or parameter **`VARIABLE`**. There are many minor variants like the one above, but we're not going to cover most of them in this course. See the Parameter Expansion section of `bash`'s man page for further details on the other forms.

As you can see from the example above, this form of parameter expansion just removes the specified characters from the end of the variable's value and then returns that to us – it is important to realise that it doesn't directly modify the variable itself.

In the context we've just been looking at, we can make use of this form of expansion to remove the common ending from our filenames – we can then produce more sensibly named files.



multi-gnuplot3.sh

```
#!/bin/bash

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

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

    # Run gnuplot
    gnuplot infect.gplt

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

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

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 45

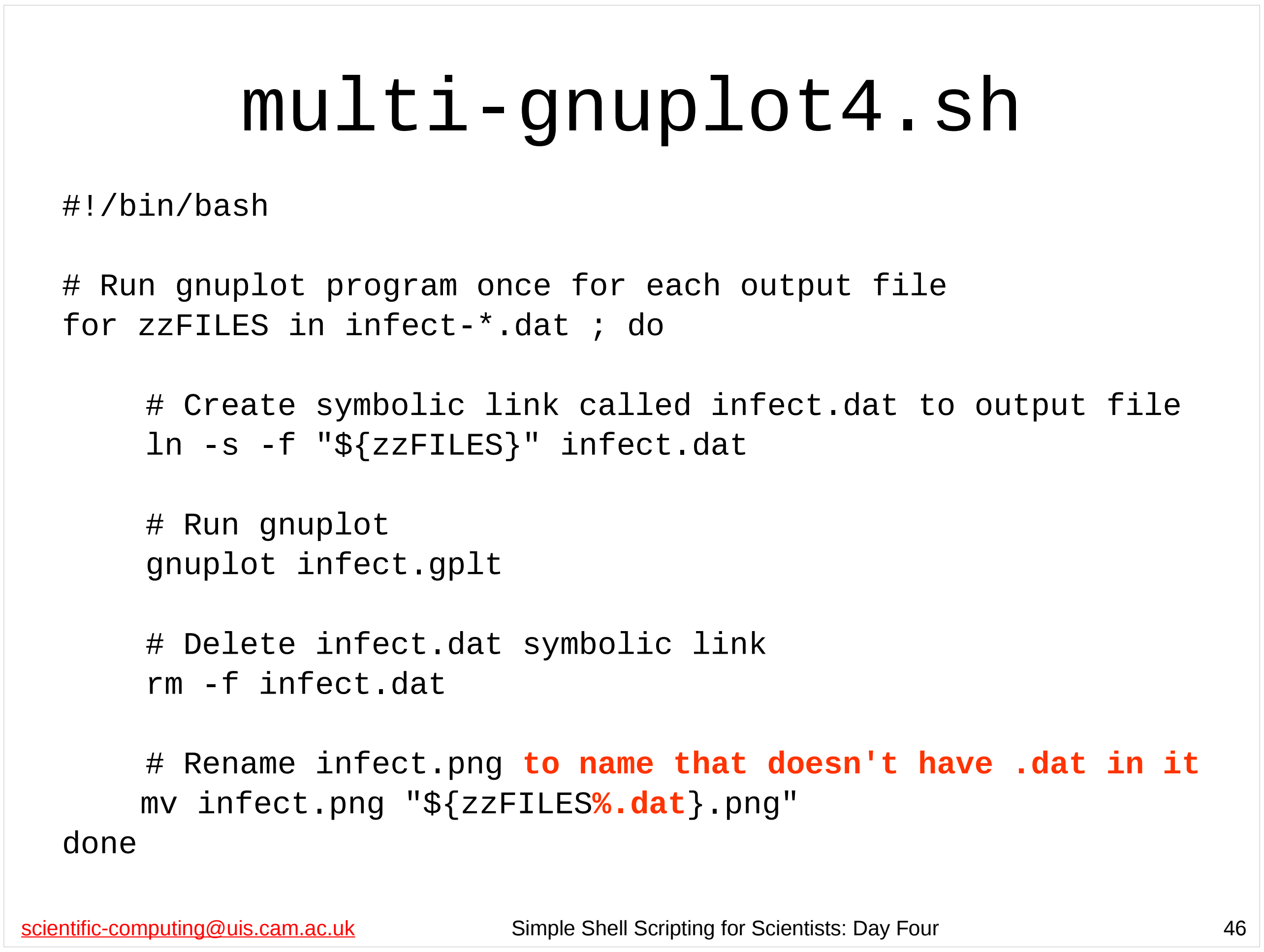
This file (`multi-gnuplot3.sh`) is in the `gnuplot` subdirectory of your course home directory.

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

To try out this script first create some files for it to process and then run it:

```
$ cd
$ rm –f *.dat *.png stdout-* logfile
$ scripts/multi-run.sh 50 100 500 1000 3000 5000 10000 50000
$ gnuplot/multi-gnuplot3.sh
```

Now do an **`ls`** to see what files have been created.



multi-gnuplot4.sh

```
#!/bin/bash

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

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

    # Run gnuplot
    gnuplot infect.gplt

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

    # Rename infect.png to name that doesn't have .dat in it
   mv infect.png "${zzFILES%.dat}.png"
done
```

This file (`multi-gnuplot4.sh`) is a modified version of `multi-gnuplot3.sh`, also in the `gnuplot` subdirectory of your course home directory.

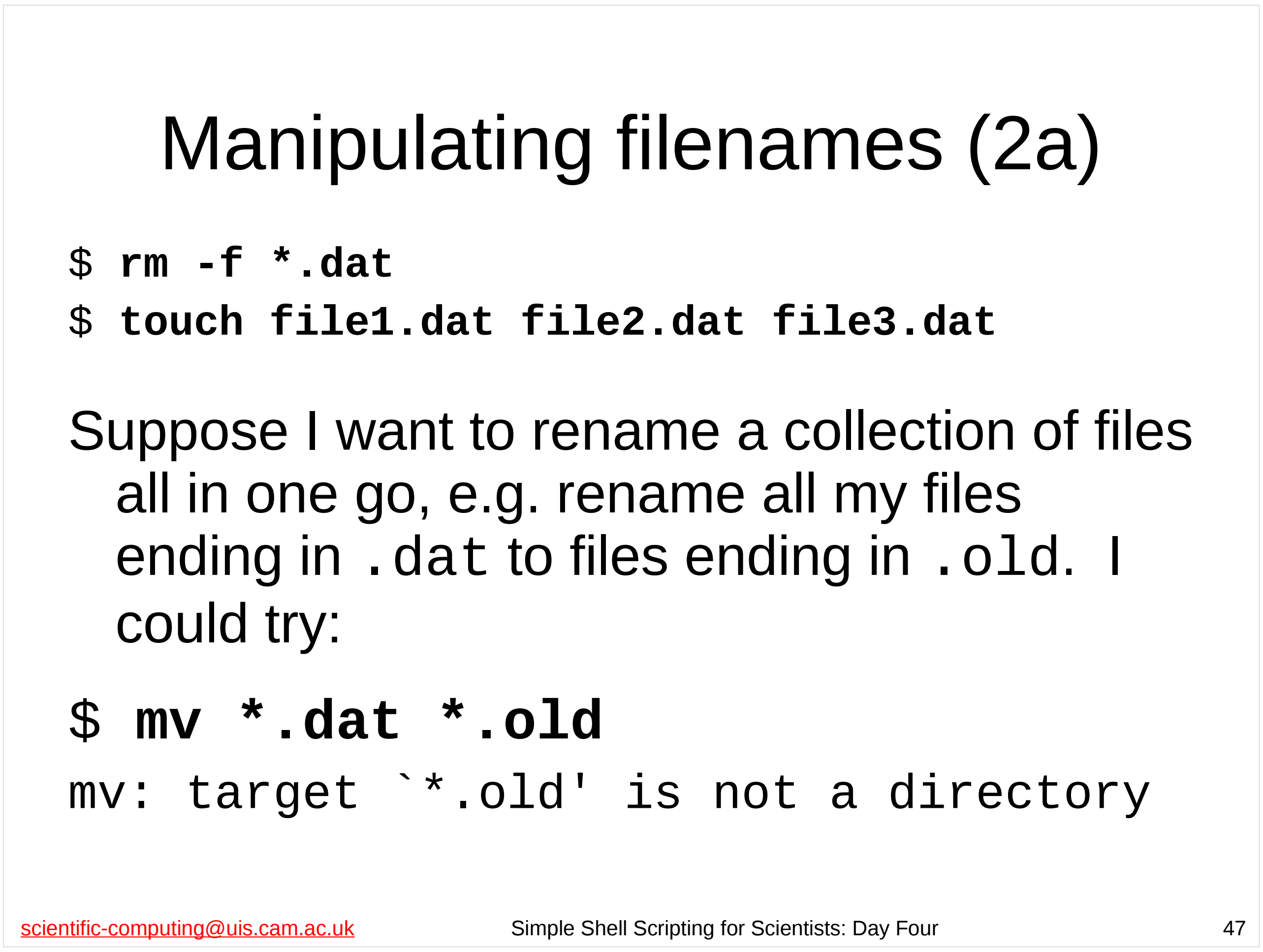
It takes each file whose name is of the form `infect-<`*`something`*`>.dat` (where the <*`something`*> can be any set of characters that can appear in a filename) in turn and creates a *symbolic link* to it called `infect.dat`, runs **`gnuplot`**, then deletes the symbolic link (*not* the original file),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

It uses the special form of parameter expansion we've just met to strip off the "`.dat`" from `infect-<`*`something`*`>.dat` so that it can rename the file produced by **`gnuplot`** to `infect-<`*`something`*`>.png`.

To try out this script first create some files for it to process and then run it:

```
$ cd
$ rm –f *.dat *.png stdout-* logfile
$ scripts/multi-run.sh 50 100 500 1000 3000 5000 10000 50000
$ gnuplot/multi-gnuplot4.sh
```

Now do an **`ls`** to see what files have been created.



Manipulating filenames (2a)

```
$ rm -f *.dat
$ touch file1.dat file2.dat file3.dat
```

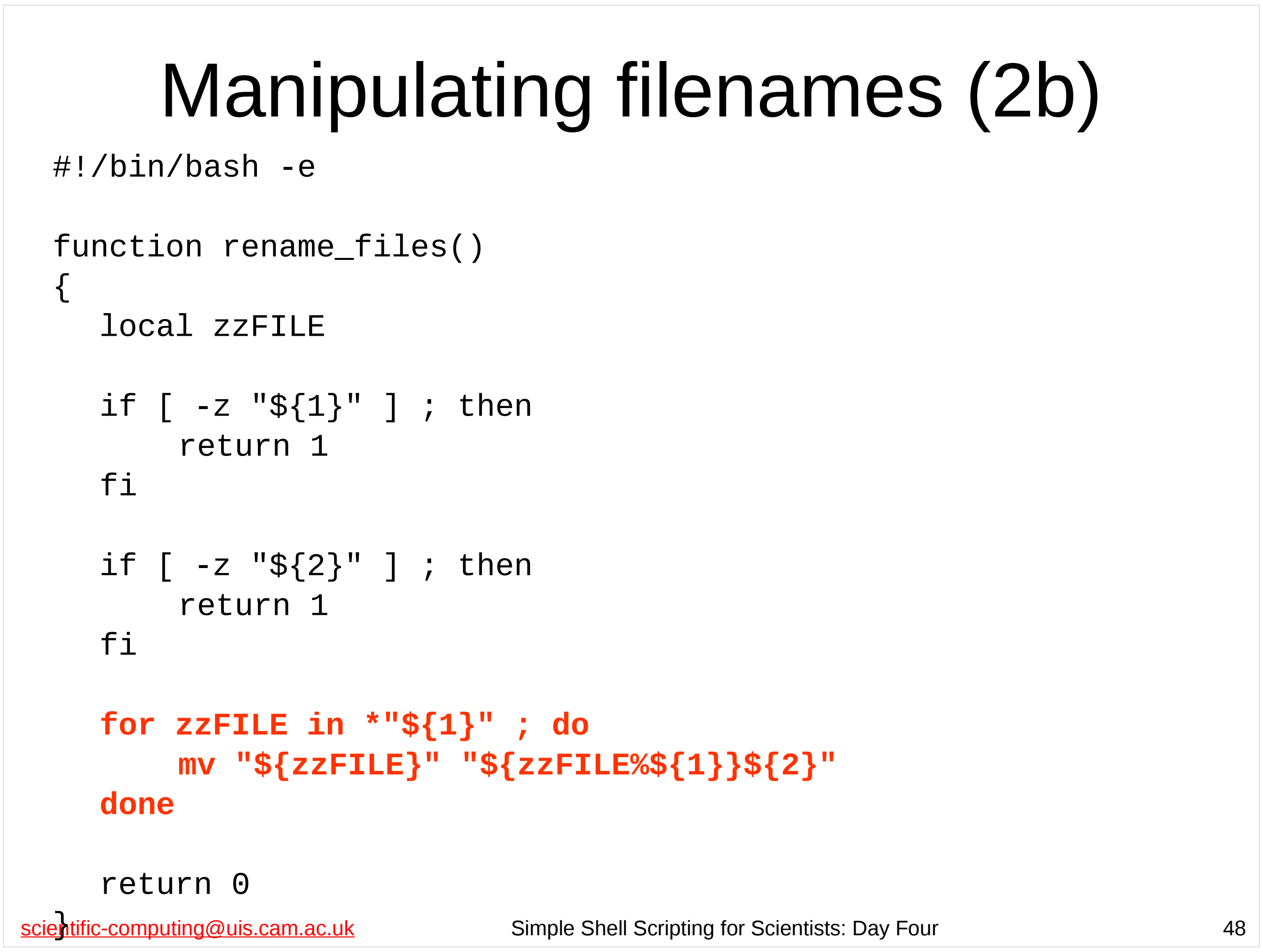
Suppose I want to rename a collection of files all in one go, e.g. rename all my files ending in `.dat` to files ending in `.old`. I could try:

```
$ mv *.dat *.old
mv: target `*.old' is not a directory
```

Here's another example where this form of parameter expansion comes in handy.

A common issue you'll probably run into on a Unix/Linux platform is trying to rename groups of files whose names all end in the same characters.

For example, let's suppose that you have a collection of data files all ending in `.dat` from the previous time you ran your program. You want to run the program again, but don't want to overwrite the old files, so you want to rename them so they all end in `.old`. Other than manually renaming each file, how can we do this?



Manipulating filenames (2b)

```
#!/bin/bash -e

function rename_files()
{
   local zzFILE

   if [ -z "${1}" ] ; then
        return 1
   fi

   if [ -z "${2}" ] ; then
        return 1
   fi

   for zzFILE in *"${1}" ; do
        mv "${zzFILE}" "${zzFILE%${1}}${2}"
   done

   return 0
}
```

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 48

In the `scripts` subdirectory there is a file called `my-functions.sh` that contains the **`rename_files`** function shown above. You can inspect it with your favourite editor or by just using the **`more`** command.

Note that because this function is designed to be used in an *interactive* shell ***only***, i.e. not in a shell script, we can reasonably safely use **`return`** with a non-zero return value (to indicate that something has gone wrong).

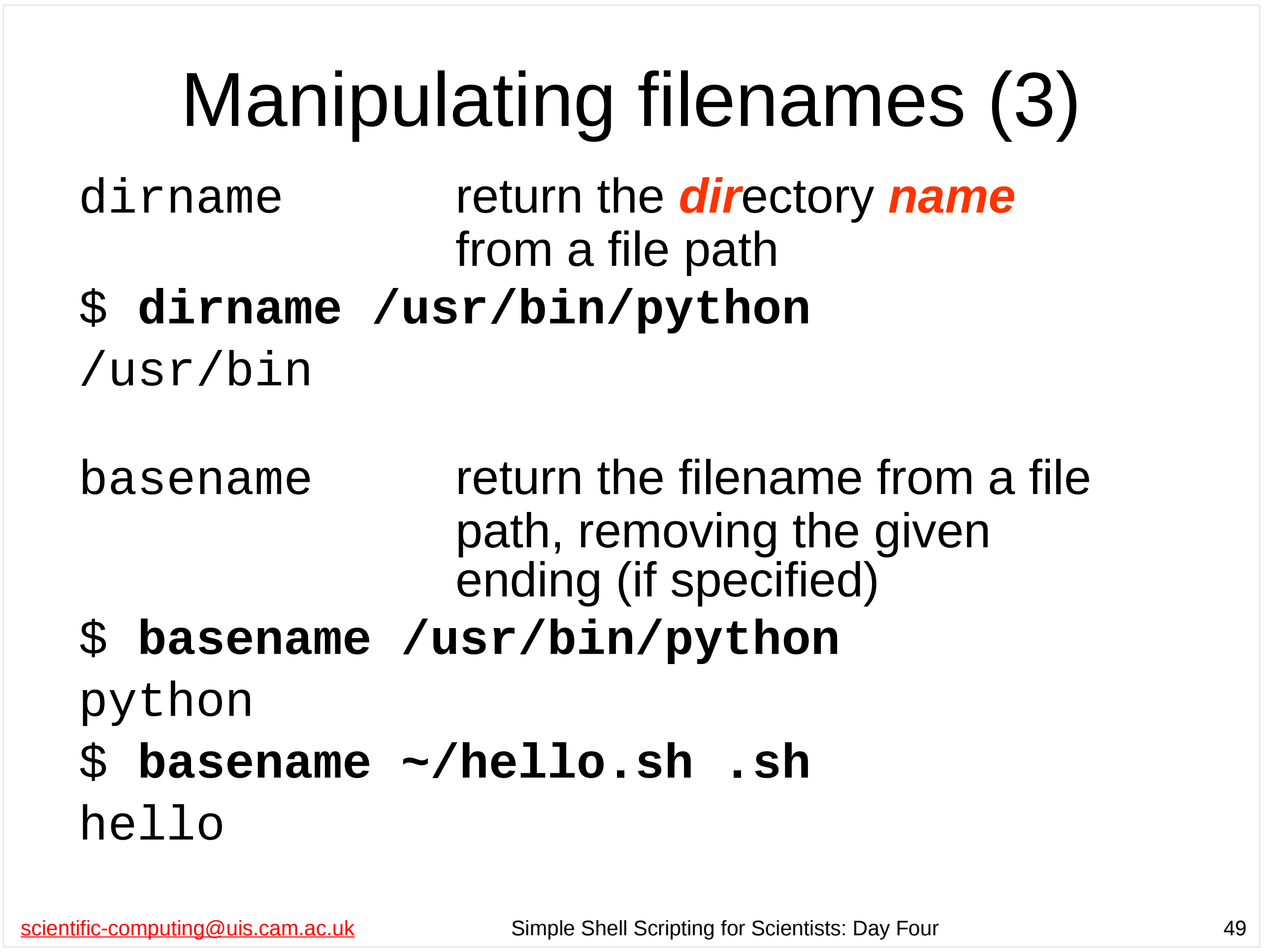
The heart of this function is the highlighted portion above: **`for`** each file ending with the first argument the function has been given, it renames the file to the same name with a different ending. So if we called this function like this:

`rename_files .dat .old`

…then it would change the name of any files ending in `.dat` to end in `.old`.

We can try this function out like this (remembering that the **`source`** shell builtin command “loads” the functions from `my-function.sh` into the running instance of the shell):

```
$ cd
$ source scripts/my-functions.sh
$ rm –f *.dat *.old
$ touch file1.dat file2.dat file3.dat
$ ls *.dat *.old
/bin/ls: *.old: No such file or directory
file1.dat  file2.dat  file3.dat
$ rename_files .dat .old
$ ls *.dat *.old
/bin/ls: *.dat: No such file or directory
file1.old  file2.old  file3.old
```



Manipulating filenames (3)

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

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

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

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

Before we move on, just a quick note of a couple of Unix/Linux commands that can help with manipulating files. 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 files, which means we could have used command substitution and the **`basename`** command in the **`rename_files`** function we just looked at as an alternative way of implementing it.

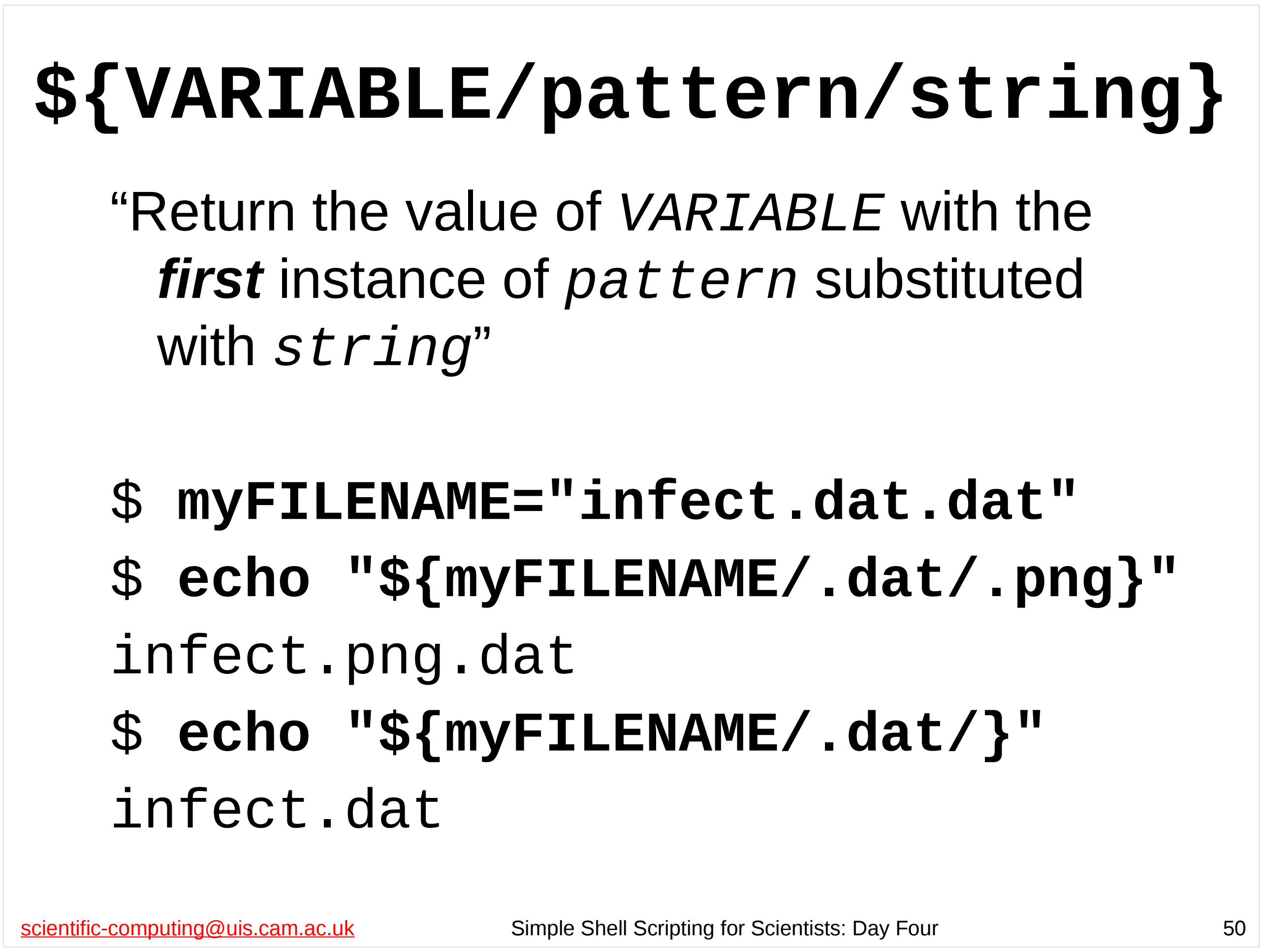
If you need to do more advanced filename (or file) manipulation, then you should look at the **`find`** and **`xargs`** commands. The **`find`** command is covered in the “Unix Systems: Further Commands” course, the notes for which are available here:

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



`${VARIABLE/pattern/string}`

“Return the value of *`VARIABLE`* with the ***first*** instance of *`pattern`* substituted with *`string`*”

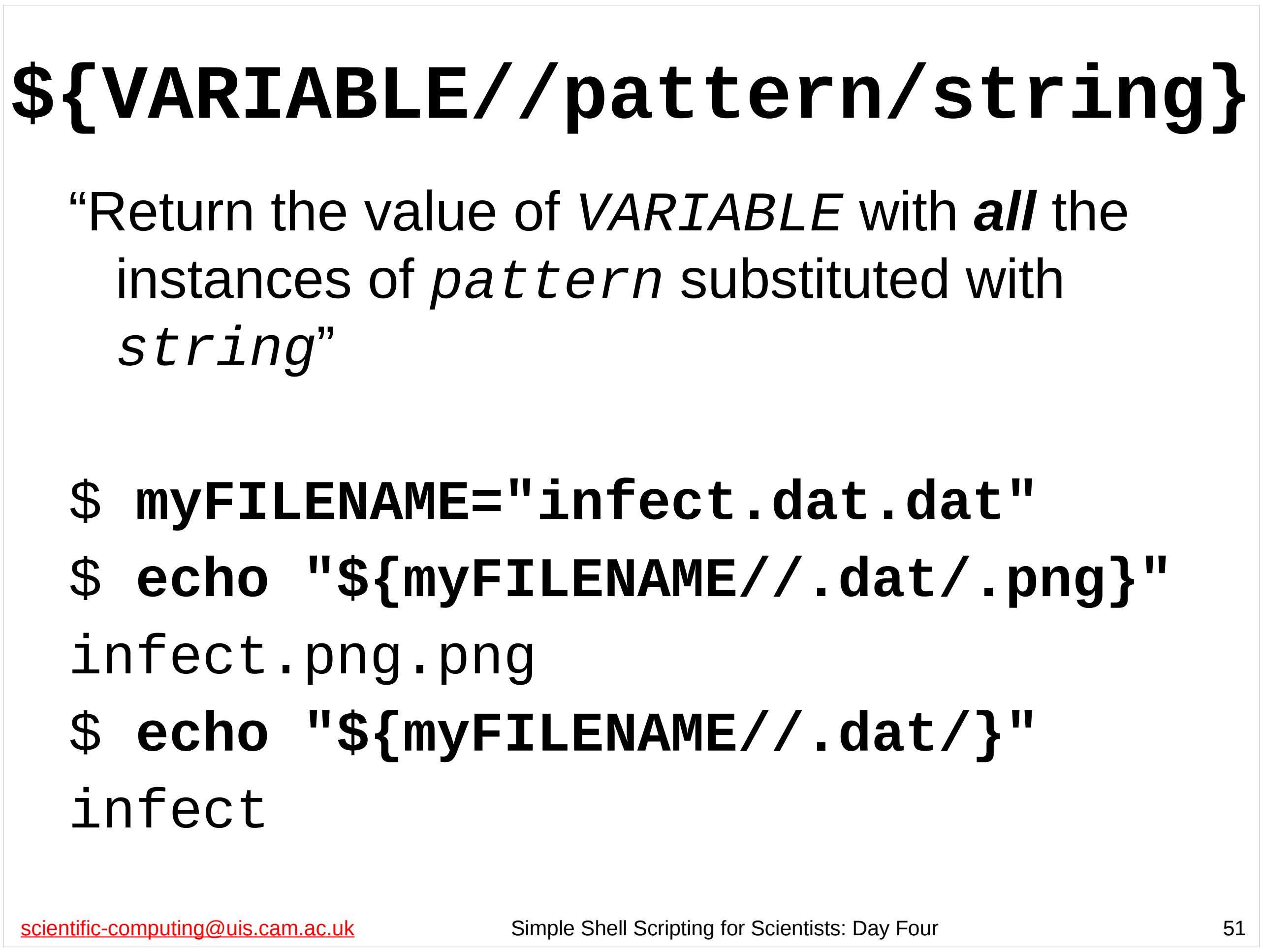
```
$ myFILENAME="infect.dat.dat"
$ echo "${myFILENAME/.dat/.png}"
infect.png.dat
$ echo "${myFILENAME/.dat/}"
infect.dat
```

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 50

This is another form of *parameter expansion*. As you can see from the example above, this form of parameter expansion just replaces the *first* instance of the specified pattern with the specified string in the variable’s value and then returns that to us. (Once again, note that it doesn’t directly modify the variable itself.)

The specified string can be the empty string, (e.g. **`${myFILENAME/.dat/}`**), in which case the specified pattern is simply removed. (In this case you can omit the final `/`, so **`${myFILENAME/.dat}`** is the same as **`${myFILENAME/.dat/}`**.)

As you can probably imagine, this can be used for manipulating filenames, but it can also be used for many other things, some of which we will see shortly.



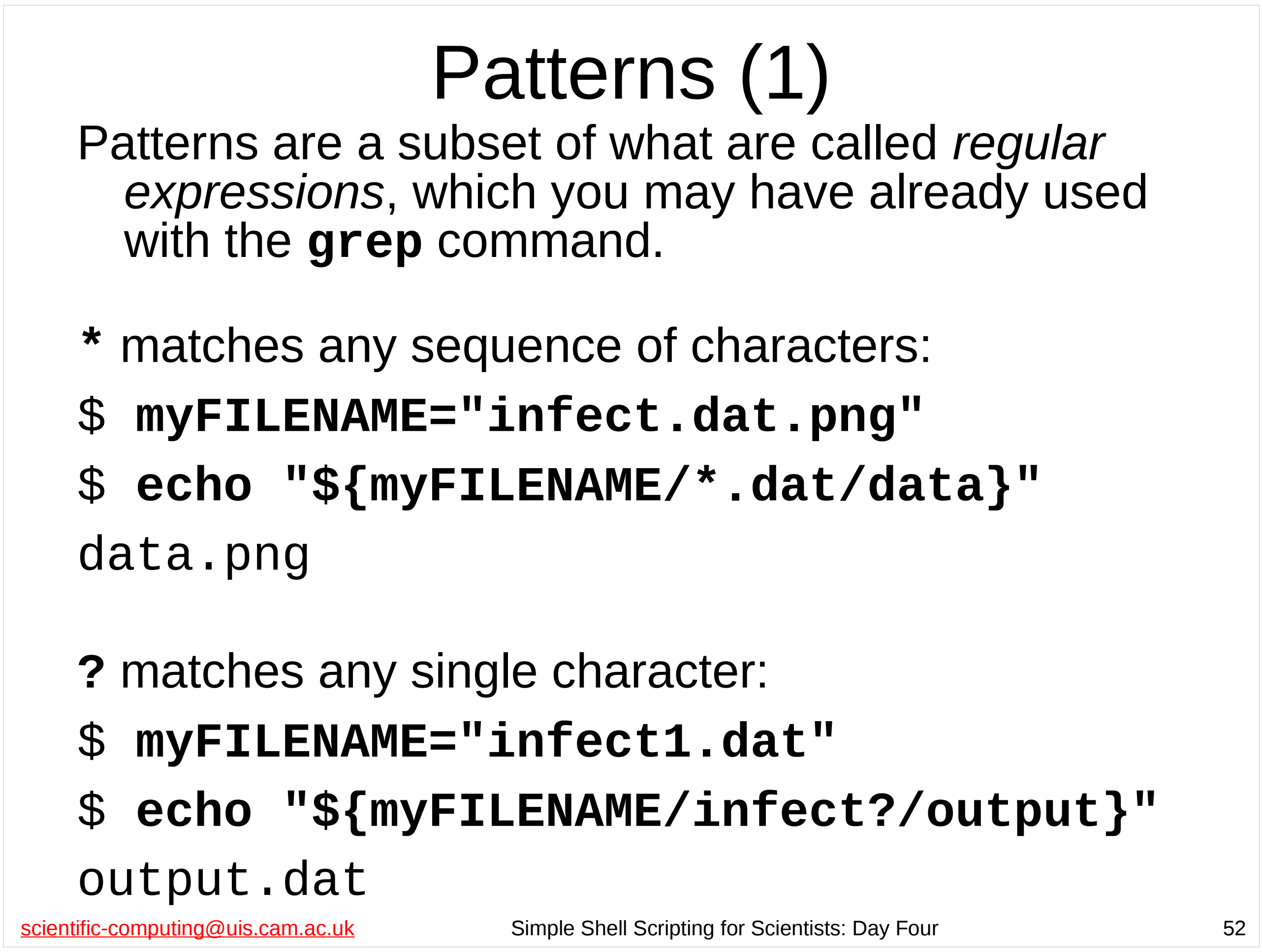
`${VARIABLE//pattern/string}`

“Return the value of *`VARIABLE`* with ***all*** the instances of *`pattern`* substituted with *`string`*”

```
$ myFILENAME="infect.dat.dat"
$ echo "${myFILENAME//.dat/.png}"
infect.png.png
$ echo "${myFILENAME//.dat/}"
infect
```

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 51

If, instead of **`${VARIABLE/pattern/string/}`**, we use **`${VARIABLE//pattern/string/}`** then ***all*** instances of the specified pattern are replaced with the specified string in the variable’s value and then returned to us.



Patterns (1)

Patterns are a subset of what are called *regular expressions*, which you may have already used with the **`grep`** command.

`*` matches any sequence of characters:

```
$ myFILENAME="infect.dat.png"
$ echo "${myFILENAME/*.dat/data}"
data.png
```

`?` matches any single character:

```
$ myFILENAME="infect1.dat"
$ echo "${myFILENAME/infect?/output}"
output.dat
```

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 52

The permissible patterns are the same as the ones that are used in “file name globbing” as covered in the CS “Unix: Introduction to the Command Line Interface” course. For details of this course, see:

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

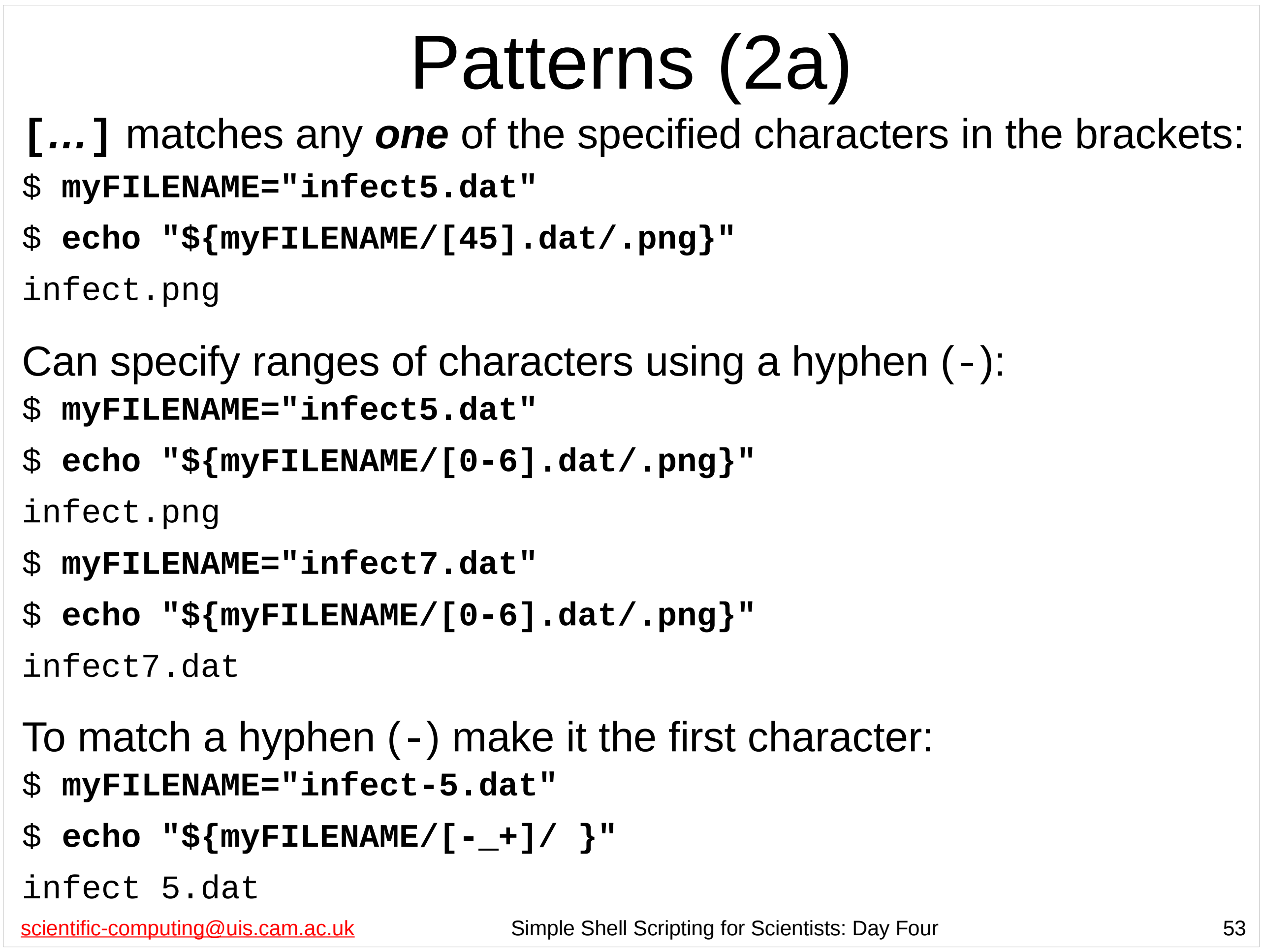
The notes from this course are available on-line at:

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

Note that **`*`** matches *any* sequence of characters, including a sequence that consists of *no* characters, i.e. the empty string.

We won’t be covering regular expressions in their full complexity 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:

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



Patterns (2a)

[…] matches any ***one*** of the specified characters in the brackets:

```
$ myFILENAME="infect5.dat"
$ echo "${myFILENAME/[45].dat/.png}"
infect.png
```

Can specify ranges of characters using a hyphen (-):

```
$ myFILENAME="infect5.dat"
$ echo "${myFILENAME/[0-6].dat/.png}"
infect.png
$ myFILENAME="infect7.dat"
$ echo "${myFILENAME/[0-6].dat/.png}"
infect7.dat
```

To match a hyphen (-) make it the first character:

```
$ myFILENAME="infect-5.dat"
$ echo "${myFILENAME/[-_+]/ }"
infect 5.dat
```

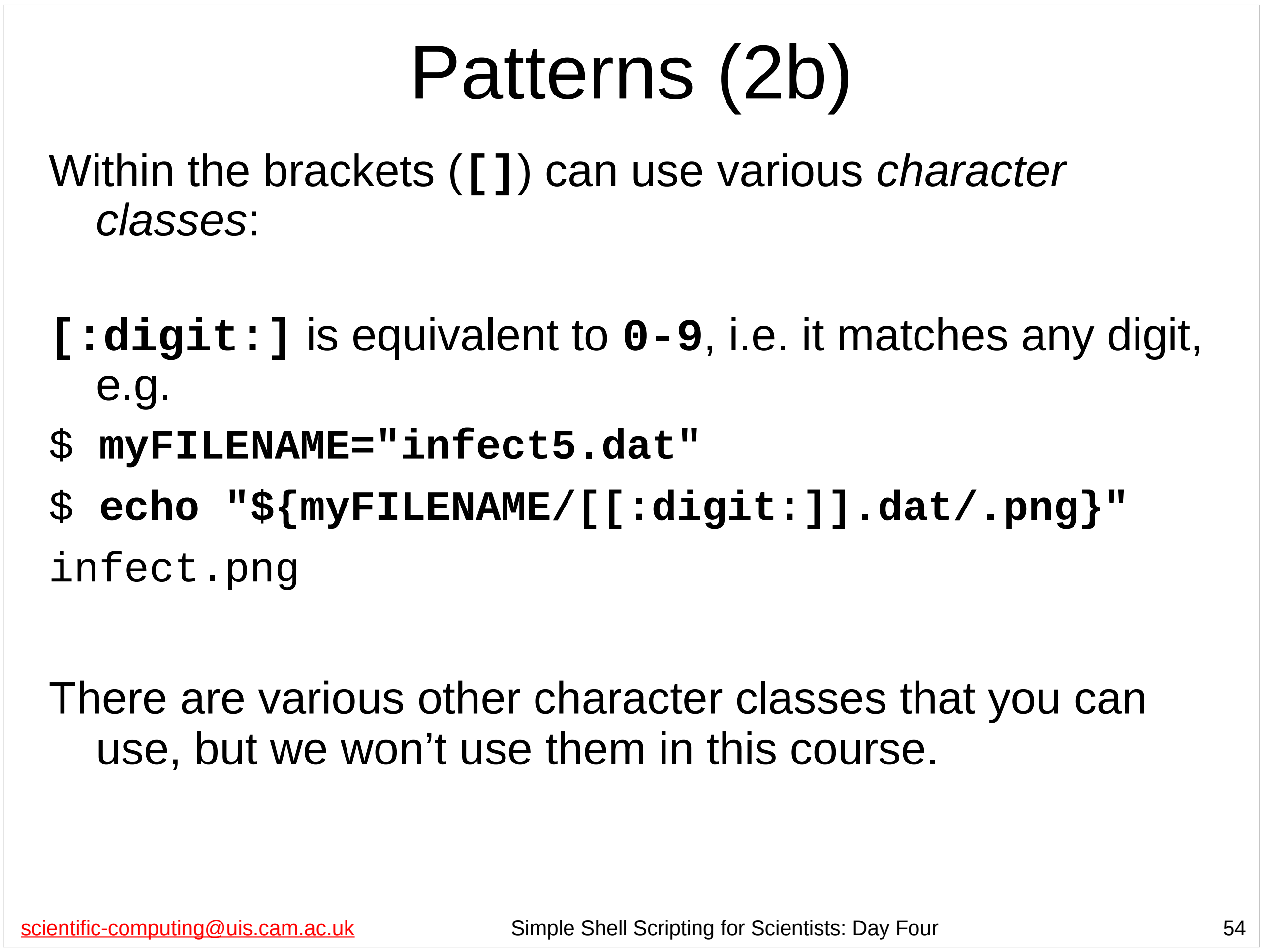
Note that at most only ***one*** of the specified characters in the brackets will match, so:

```
$ myFILENAME="infect45.dat"
$ echo "${myFILENAME/[45].dat/.png}"
infect4.png
```

Ranges of characters, such as **[c$_1$-c$_2$]**, mean match any ***single*** character in the range **c**$_1$ to **c**$_2$. Thus **[0-100]** means "match any character in the range **0-1**, or match **0**, or match **0**". So it only matches the characters "0" or "1". It does not match the integers 0 to 100. The range of characters is determined by the current sort order, which is language dependent. Thus the range **[a-z]** may give different things depending on the sort order in use – for this reason we often use character classes (see next slide) so we can be sure what characters we are matching against.

If you want to match a hyphen (-), it must be the *first* character that appears in the brackets, so **[-_+]** means match any one of the following characters: -, _ or +.

```
$ myFILENAME="infect-5.dat"
$ echo "${myFILENAME/[-_+]/ }"
infect 5.dat
$ myFILENAME="infect+5.dat"
$ echo "${myFILENAME/[-_+]/ }"
infect 5.dat
$ myFILENAME="infect_5.dat"
$ echo "${myFILENAME/[-_+]/ }"
infect 5.dat
```



Patterns (2b)

Within the brackets (**[]**) can use various *character classes*:

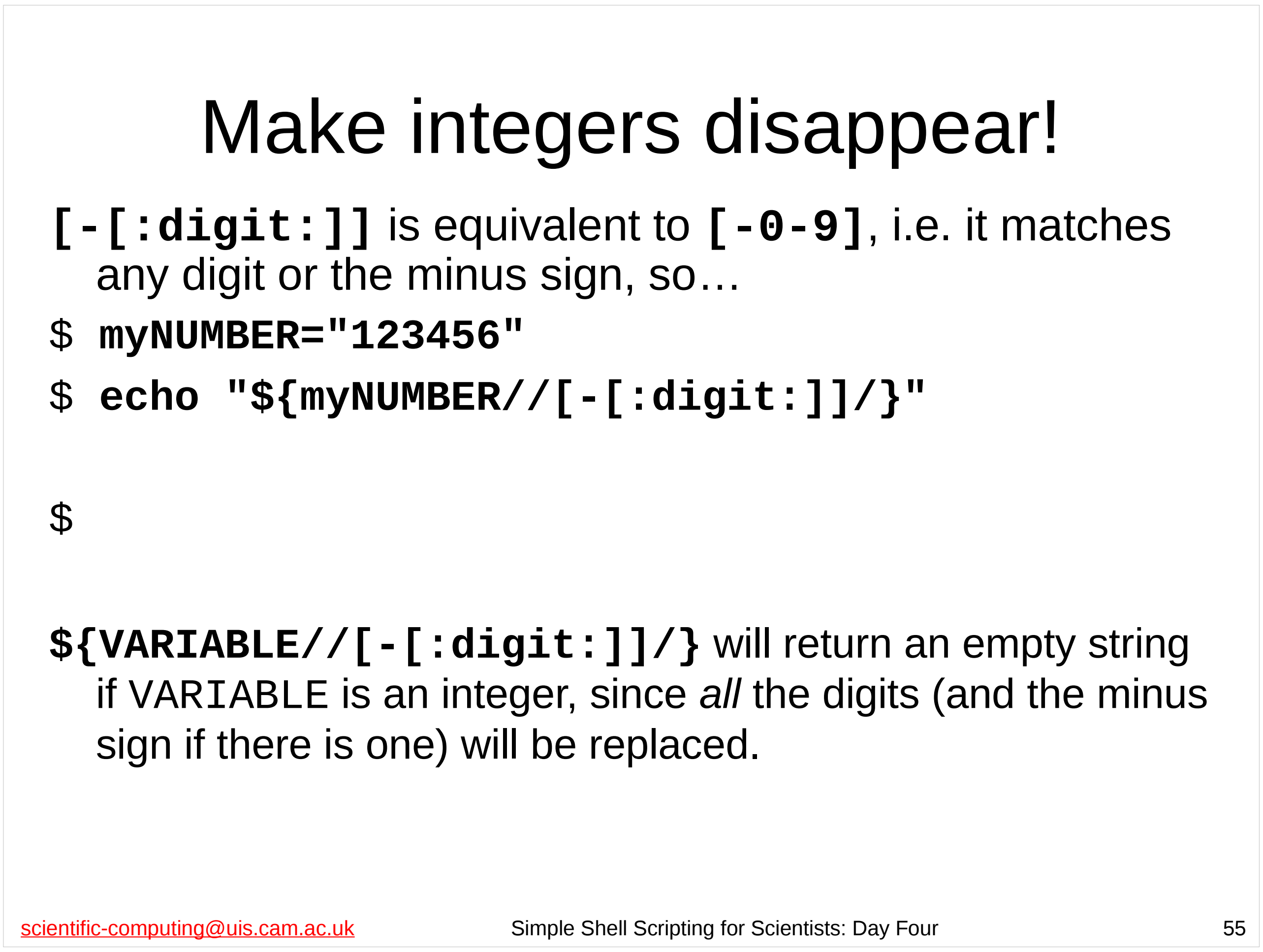
[:digit:] is equivalent to **0-9**, i.e. it matches any digit, e.g.

```
$ myFILENAME="infect5.dat"
$ echo "${myFILENAME/[[:digit:]].dat/.png}"
infect.png
```

There are various other character classes that you can use, but we won’t use them in this course.

As mentioned on the previous slide, the range of characters is determined by the current sort order, which is language dependent. Thus the range **`[a-z]`** may give different things depending on the sort order in use, for instance it might mean **`[abcdefghijklmnopqrstuvwxyz]`** or it might mean **`[aBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz]`**. For this reason we often use character classes, e.g. **`[[:lower:]]`** matches any on of the lower case alphabetic characters ‘a’, ‘b’, ‘c’, … ‘z’. so we can be sure what characters we are matching against. You can find a list of the allowed character classes in the the Pattern Matching subsection (in the Pathname Expansion section) of `bash`’s `man` page, but here are some of the most useful ones:

`[:alnum:]`	Any alphabetic character (upper or lower case) or any digit.
`[:alpha:]`	Any alphabetic character (upper or lower case).
`[:blank:]`	Any horizontal white space (space or tab, essentially).
`[:digit:]`	Any of the ten digits (0-9).
`[:lower:]`	Any lower case alphabetic character.
`[:space:]`	Any white space (space, tab, newline, etc).
`[:upper:]`	Any upper case alphabetic character.
`[:xdigit:]`	Any hexadecimal digit (i.e. 0-9 and A-F (upper or lower case)).



Make integers disappear!

`[-[:digit:]]` is equivalent to **`[-0-9]`**, i.e. it matches any digit or the minus sign, so…

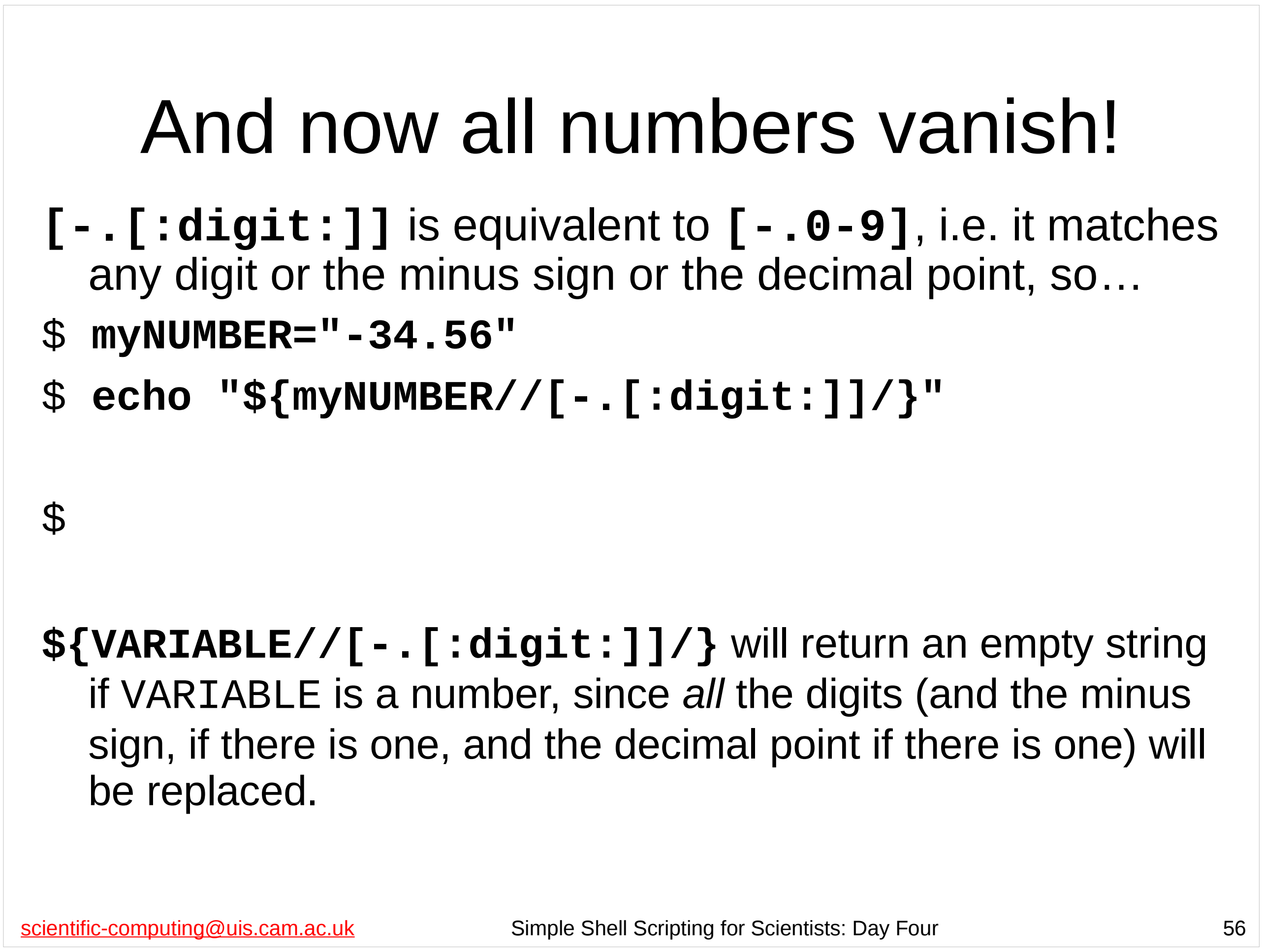
```
$ myNUMBER="123456"
$ echo "${myNUMBER//[-[:digit:]]/}"

$
```

`${VARIABLE//[-[:digit:]]/}` will return an empty string if `VARIABLE` is an integer, since *all* the digits (and the minus sign if there is one) will be replaced.

You may be wondering why we would do this: in a few minutes we'll see how we can use this to test for integers.

Note that we need to use the **`${VARIABLE//pattern/}`** form to make sure we replace *all* instances of the pattern (digits or a minus sign).

Also, note that this gives false positives since it will return the empty string for things like "`12---35`" as well as "`-1235`".



And now all numbers vanish!

`[-.[:digit:]]` is equivalent to **`[-.0-9]`**, i.e. it matches any digit or the minus sign or the decimal point, so…

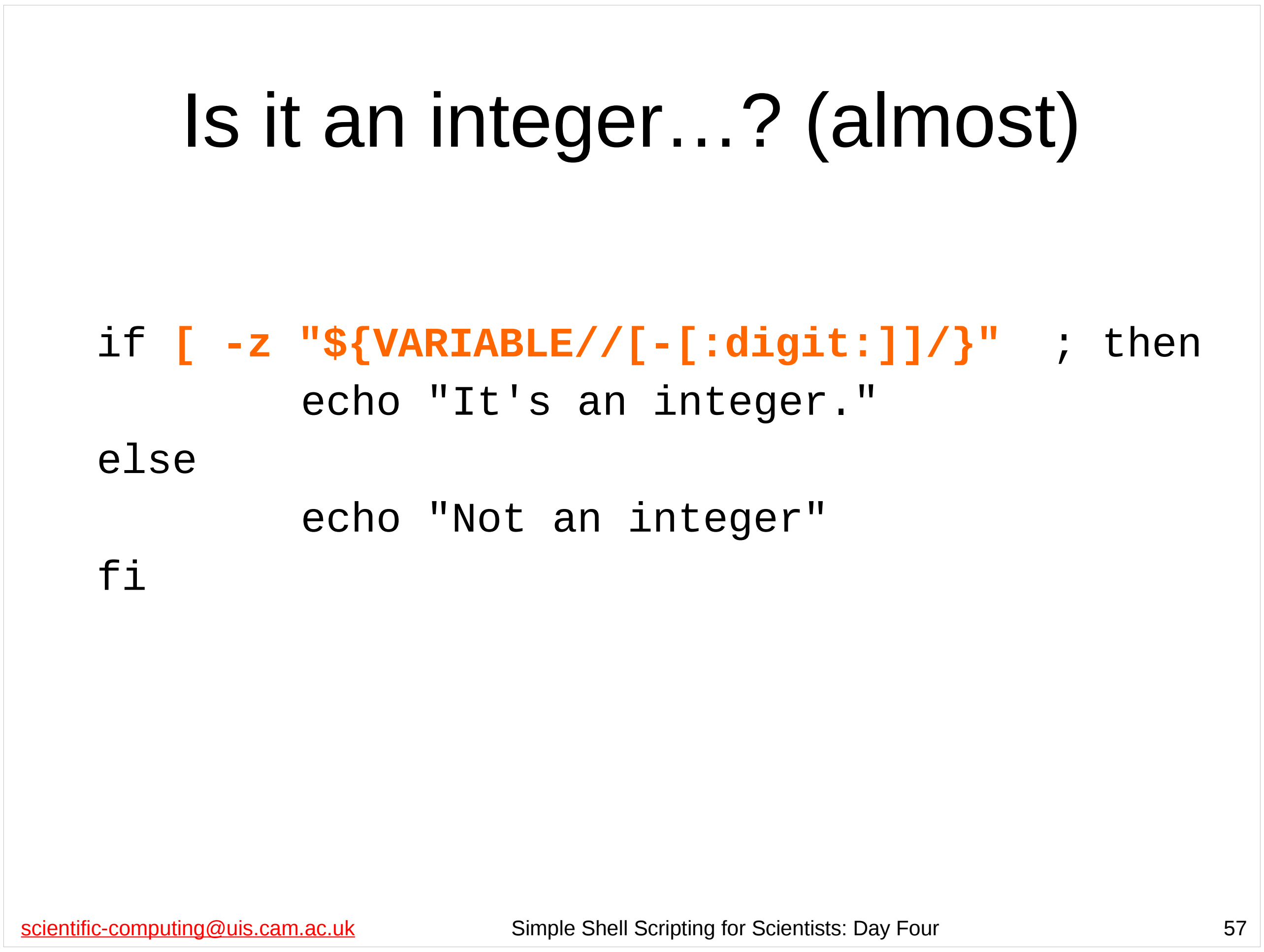
```
$ myNUMBER="-34.56"
$ echo "${myNUMBER//[-.[:digit:]]/}"


$
```

`${VARIABLE//[-.[:digit:]]/}` will return an empty string if `VARIABLE` is a number, since *all* the digits (and the minus sign, if there is one, and the decimal point if there is one) will be replaced.

Again, you may be wondering what use this is: in a few minutes we'll see how we can use this to test for numbers.

Note that we need to use the **`${VARIABLE//pattern/}`** form to make sure we replace *all* instances of the pattern (digits or a minus sign or a decimal point).

Also, note that this gives false positives since it will return the empty string for things like "`12.-.35`" as well as "`-12.35`".



Is it an integer…? (almost)

```
if [ -z "${VARIABLE//[-[:digit:]]/}"  ; then
        echo "It's an integer."
else
        echo "Not an integer"
fi
```

So now we can test and see whether an environment variable, shell variable or parameter is an integer or not (well, almost).

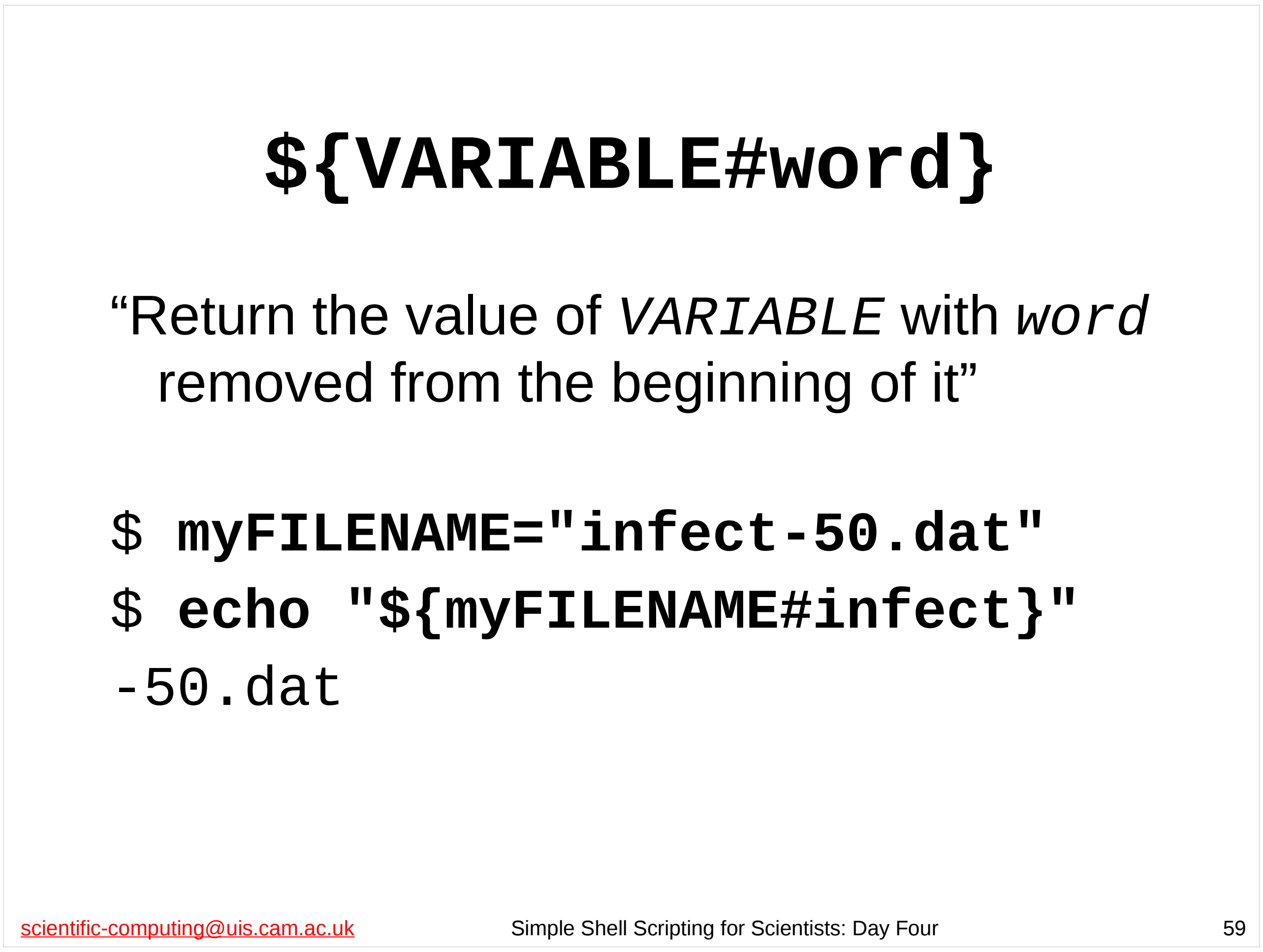
Note that this test does give false positives since it gives true for things like “`12---35`” (which is ***not*** an integer) as well as “`-1235`” (which is an integer).

Is it a number…? (almost)

```
if [ -z "${VARIABLE//[-.[:digit:]]/}"  ; then
        echo "It's a number."
else
        echo "Not a number"
fi
```

So now we can test and see whether an environment variable, shell variable or parameter is a (decimal) number or not (again, almost).

Note that this test does give false positives since it gives true for things like “`12-.-35`” (which is ***not*** a number) as well as “`-12.35`” (which is a number).



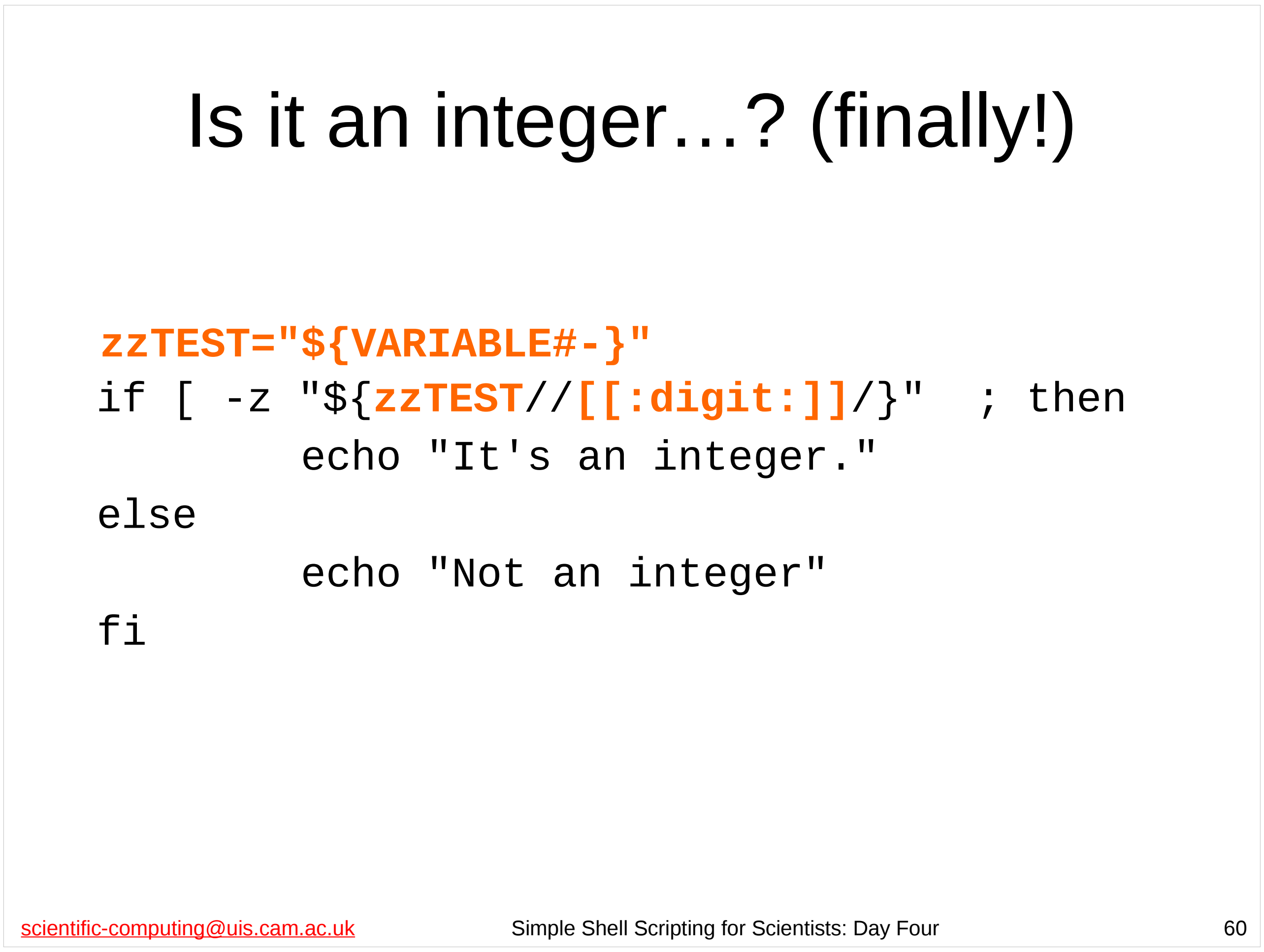
${VARIABLE#word}

“Return the value of *`VARIABLE`* with *`word`* removed from the beginning of it”

```
$ myFILENAME="infect-50.dat"
$ echo "${myFILENAME#infect}"
-50.dat
```

This is another form of *parameter expansion*. As you can see from the example above, this form of parameter expansion just removes the specified characters from the beginning of the variable’s value and then returns that to us.

In the context we’ve just been looking at, we can make use of this form of expansion to remove any leading minus sign from our value, which will make it easier to test whether or not it is a number.

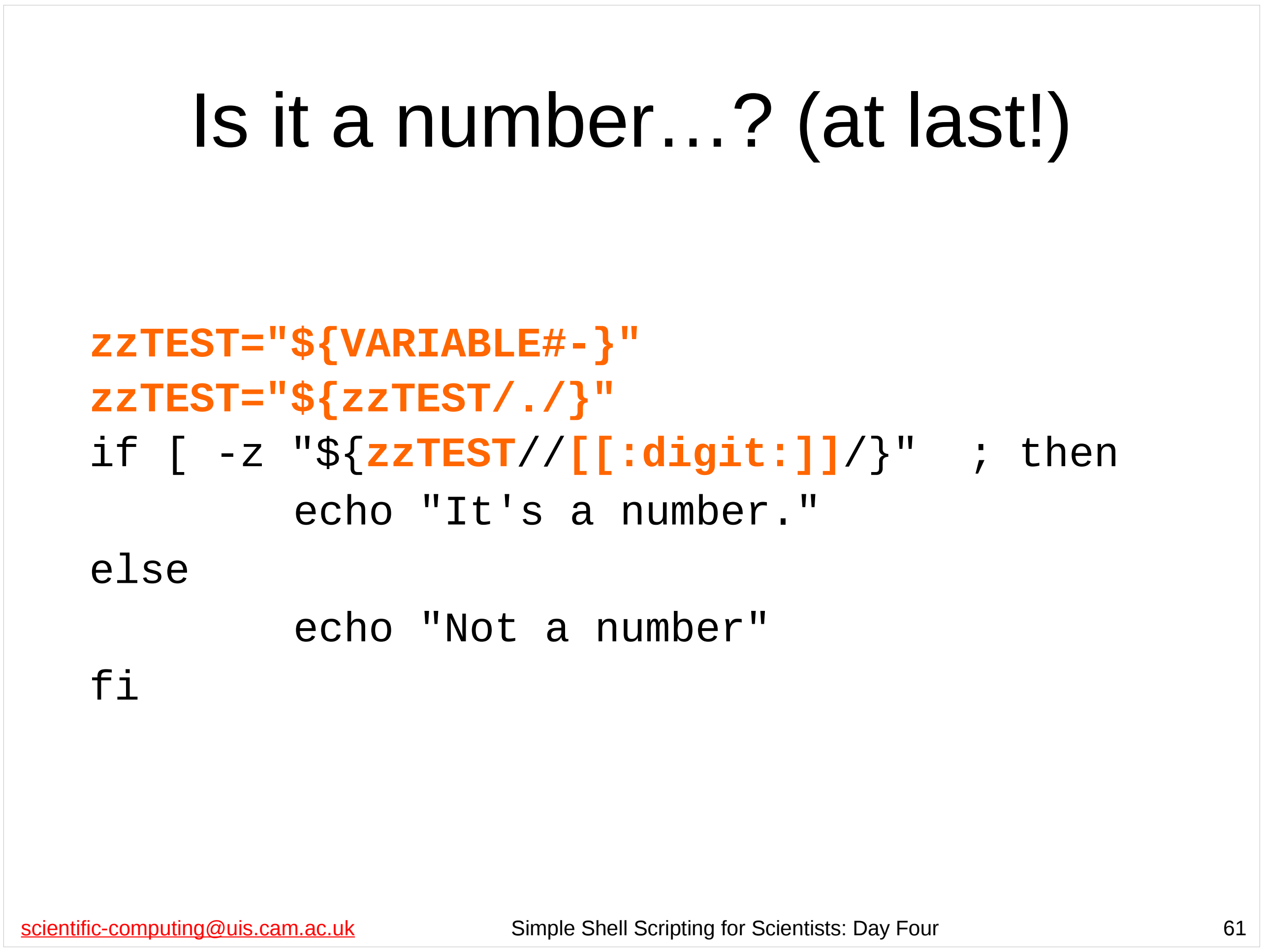


Is it an integer…? (finally!)

```
zzTEST="${VARIABLE#-}"
if [ -z "${zzTEST//[[:digit:]]/}"  ; then
        echo "It's an integer."
else
        echo "Not an integer"
fi
```

So now we can test and see whether an environment variable, shell variable or parameter is an integer or not (with no false positives).

First we strip off any leading minus sign, and then we make sure that all that is left are digits (0-9).

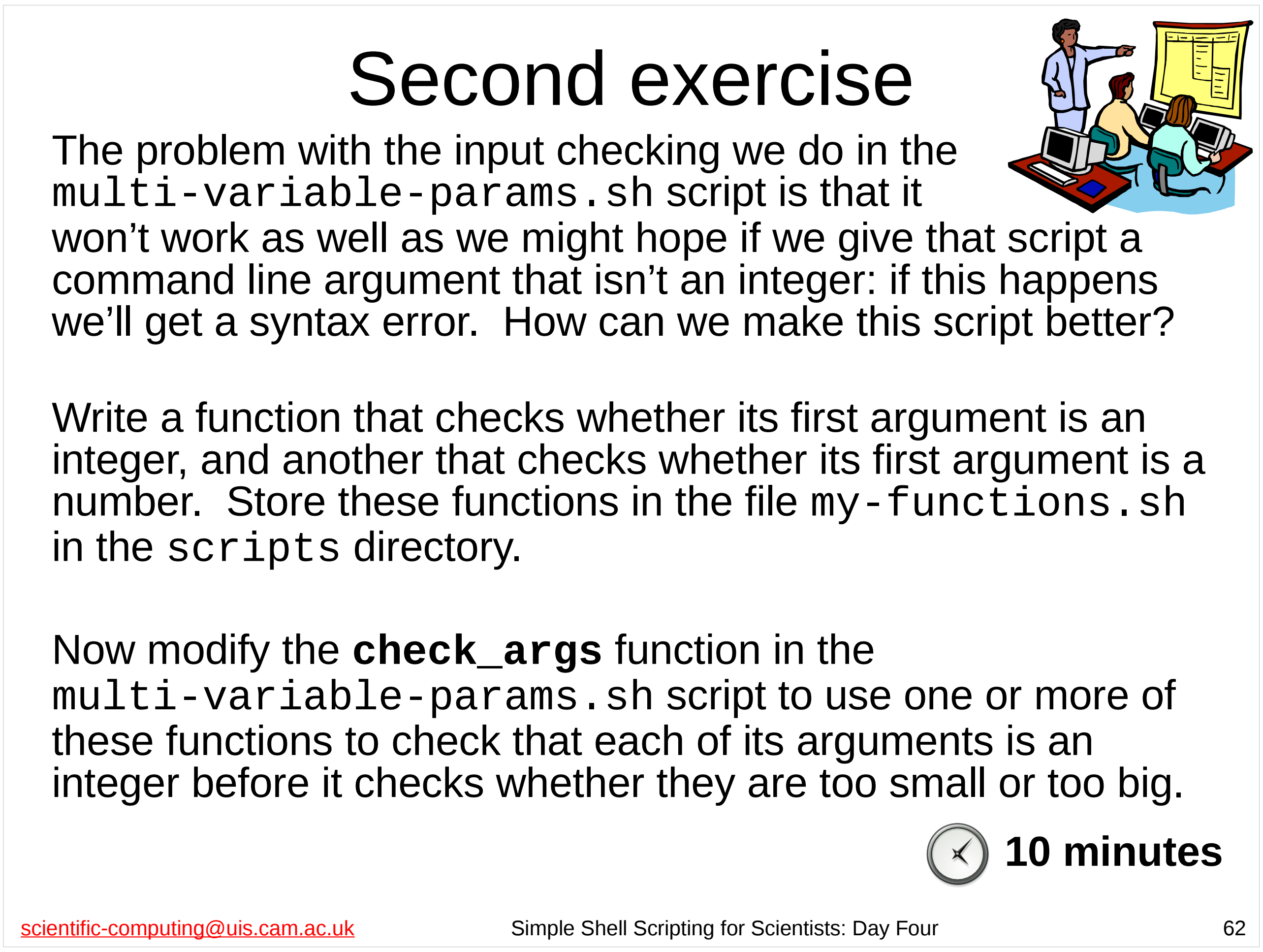


Is it a number…? (at last!)

```
zzTEST="${VARIABLE#-}"
zzTEST="${zzTEST/./}"
if [ -z "${zzTEST//[[:digit:]]/}"  ; then
        echo "It's a number."
else
        echo "Not a number"
fi
```

So now we can test and see whether an environment variable, shell variable or parameter is a (decimal) number or not (again, without any false positives).

First we remove any leading minus sign. Then we remove *at most* one decimal point. Finally we make sure that all that is left are the digits (0-9).



Second exercise

The problem with the input checking we do in the `multi-variable-params.sh` script is that it won't work as well as we might hope if we give that script a command line argument that isn't an integer: if this happens we'll get a syntax error. How can we make this script better?

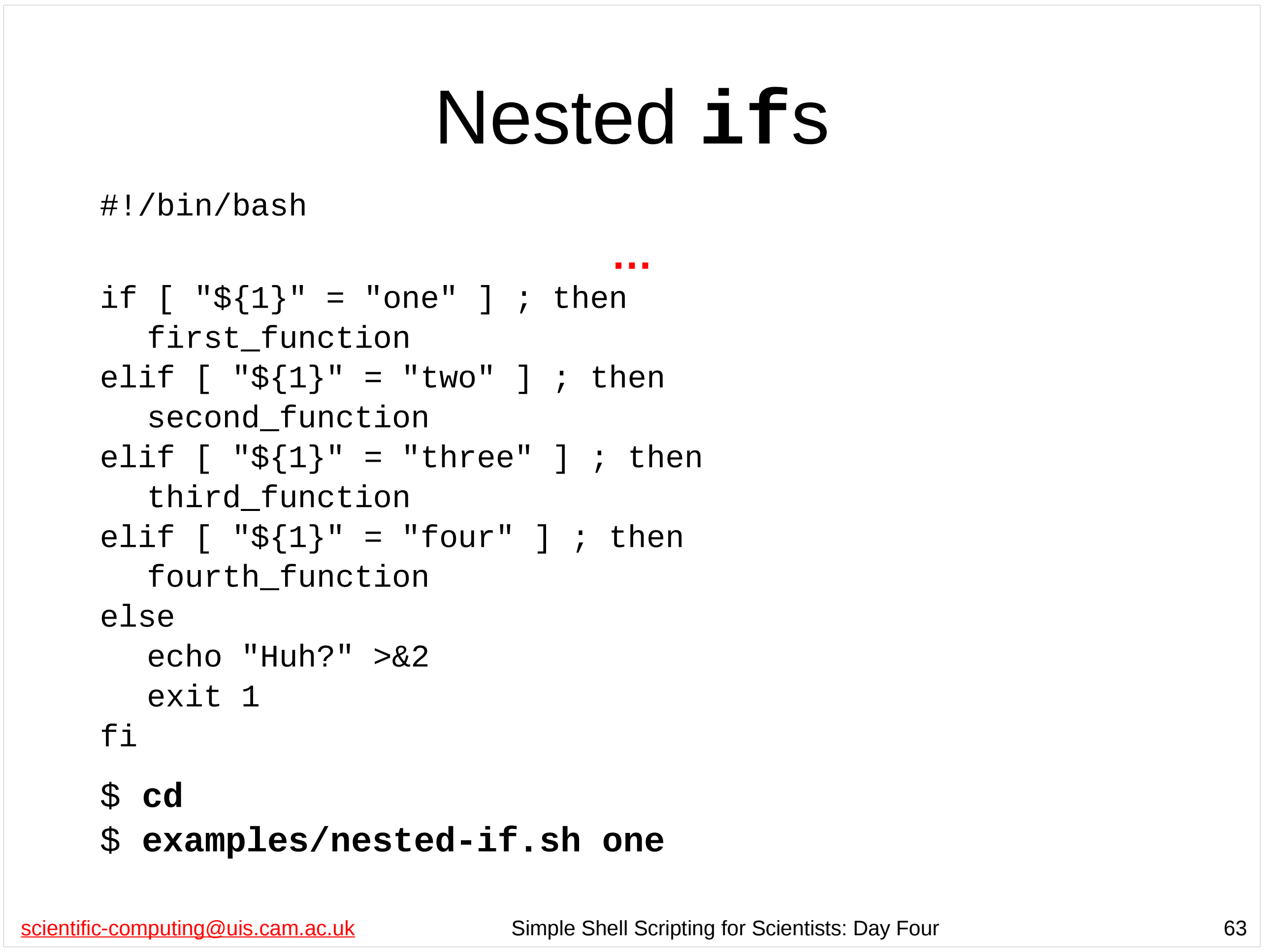
Write a function that checks whether its first argument is an integer, and another that checks whether its first argument is a number. Store these functions in the file `my-functions.sh` in the `scripts` directory.

Now modify the **`check_args`** function in the `multi-variable-params.sh` script to use one or more of these functions to check that each of its arguments is an integer before it checks whether they are too small or too big.

10 minutes

You should have created the `multi-variable-params.sh` shell script in the previous exercise. If you didn't, use the `multi-sizes-errors.sh` script in the `scripts` directory of your home directory instead.

When you finish this exercise, take a short break and then we'll start again with the solution. (I really ***do*** mean take a break – sitting in front of computers for long periods of time is very bad for you. Move around, go for a jog, do some aerobics, whatever…)



Nested **`if`**s

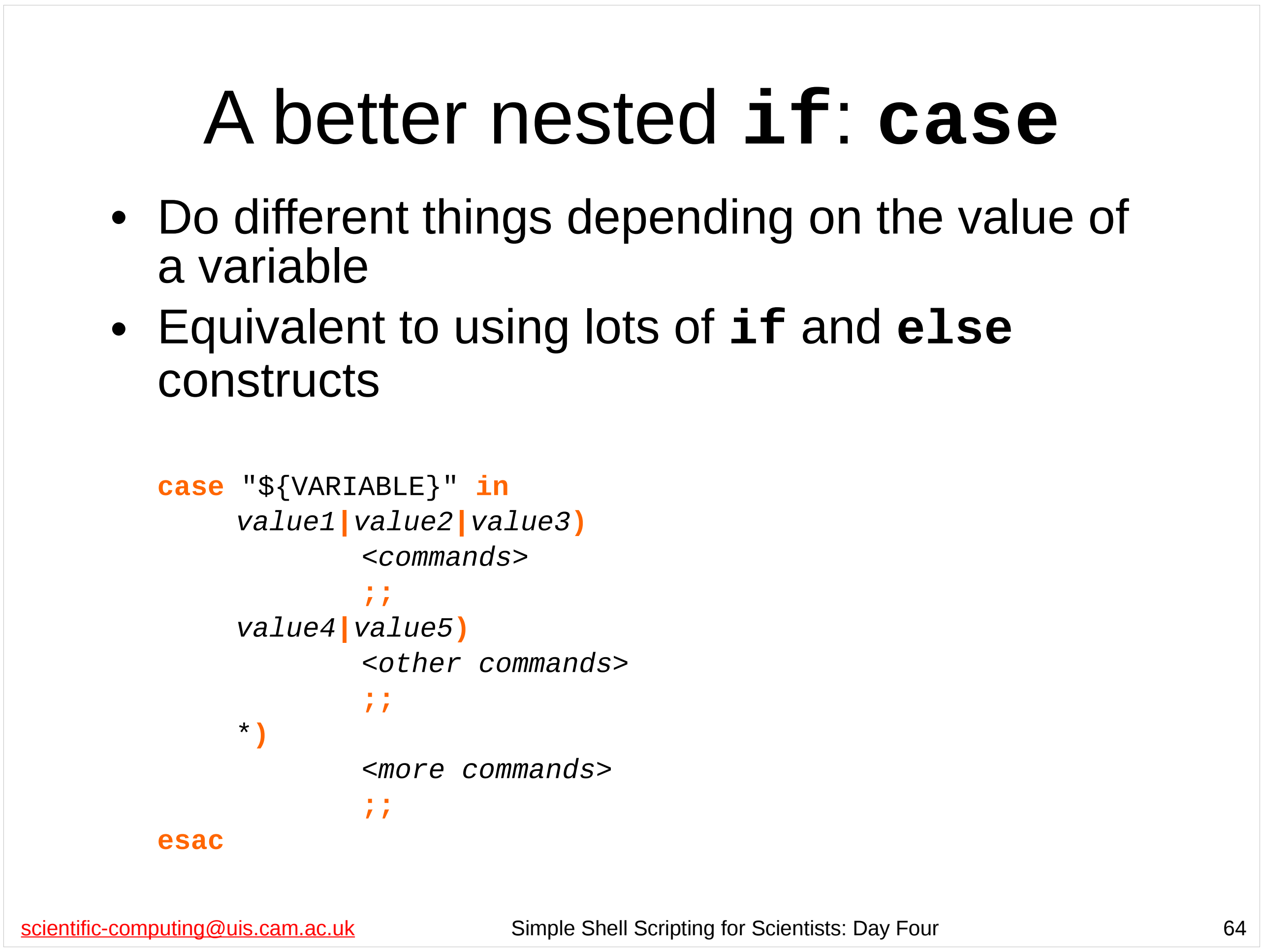
```
#!/bin/bash
                                                ...
if [ "${1}" = "one" ] ; then
   first_function
elif [ "${1}" = "two" ] ; then
   second_function
elif [ "${1}" = "three" ] ; then
   third_function
elif [ "${1}" = "four" ] ; then
   fourth_function
else
   echo "Huh?" >&2
   exit 1
fi
```

```
$ cd
$ examples/nested-if.sh one
```

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 63

On the previous day of the course we saw that we could use “nest” **`if`** statements. In the `examples` subdirectory there is a silly shell script called `nested-if.sh` that illustrates the nested `if` construct. The heart of the script is shown above – **`first_function`**, **`second_function`**, **`third_function`** and **`fourth_function`** are all shell functions defined in the script.

Try the script out and remind yourself what it does. Although it’s a silly example, it should give you an idea of the sort of useful things for which you can use such scripts.



A better nested **`if`**: **`case`**

- Do different things depending on the value of a variable
- Equivalent to using lots of **`if`** and **`else`** constructs

```
case "${VARIABLE}" in
     value1|value2|value3)
             <commands>
             ;;
     value4|value5)
             <other commands>
             ;;
     *)
             <more commands>
             ;;
esac
```

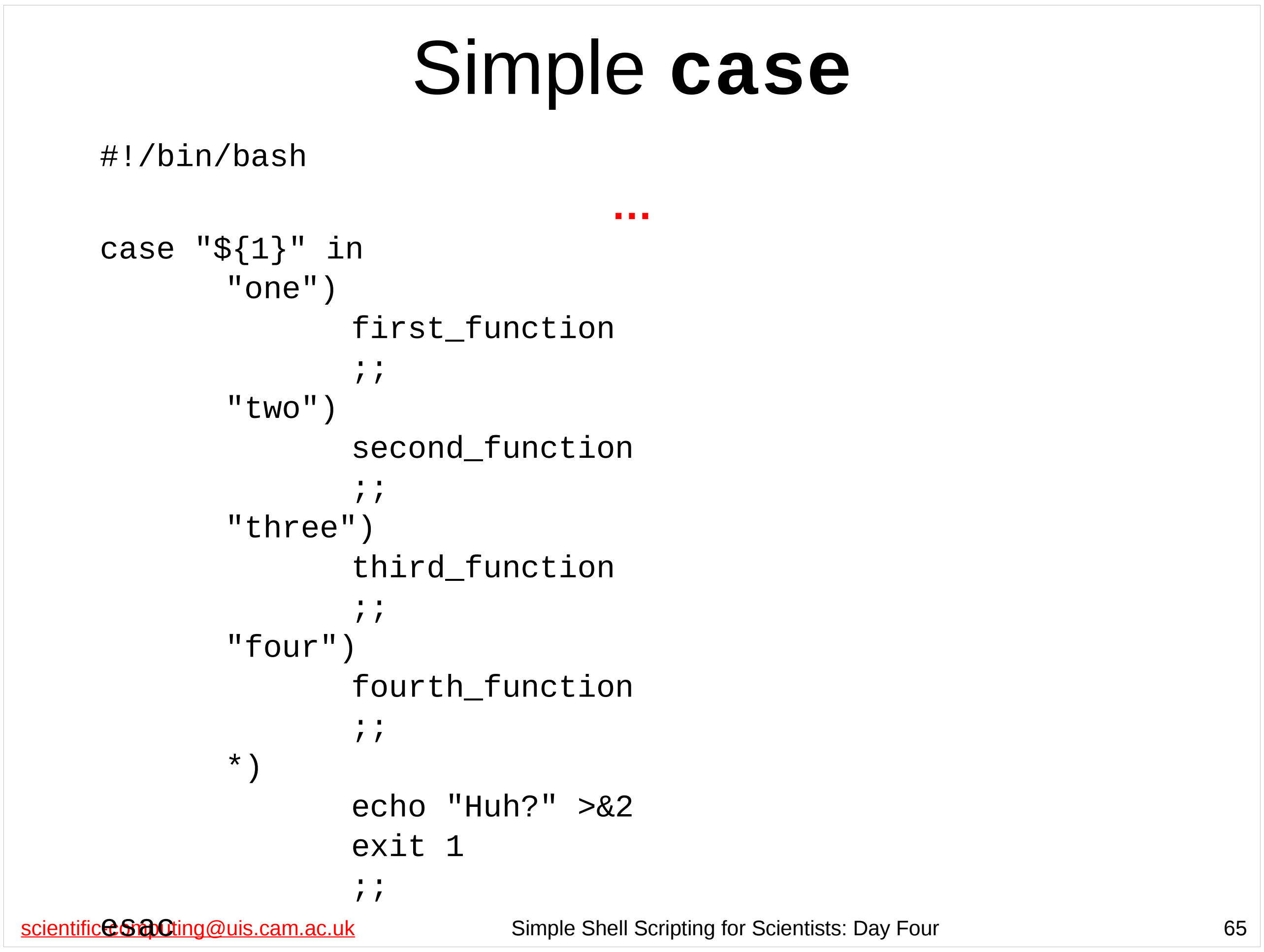
Some programming languages have a construct which does the same sort of thing as the shell's **`case`** construct. In many of these languages it is known as the **`switch`** statement.

There are some examples of how to use it in the following files in the `examples` directory:

`case1.sh`

`case2.sh`

…and we shall now look at how to use the **`case`** construct in more detail.

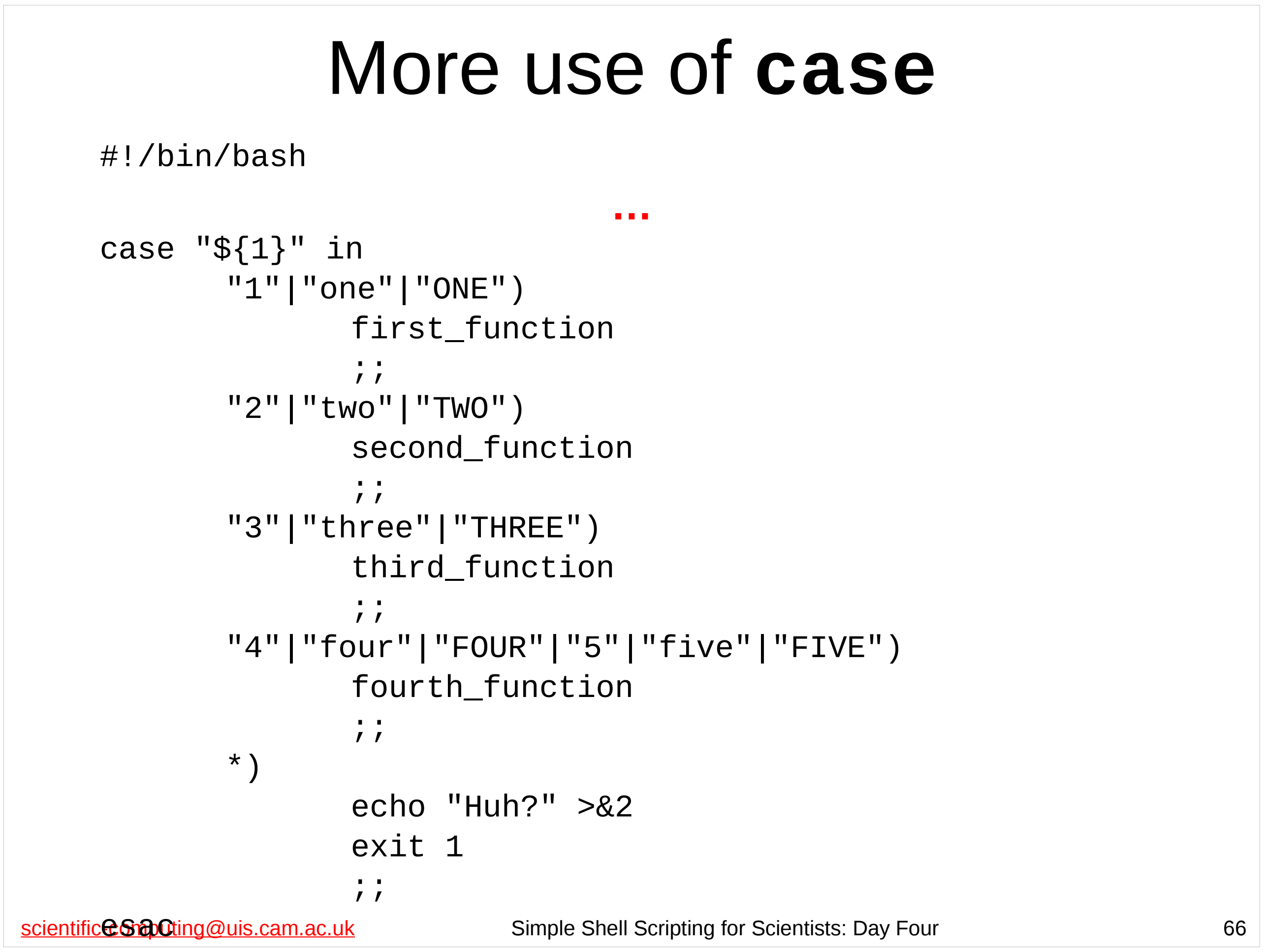


Simple **case**

```
#!/bin/bash
                                    ...
case "${1}" in
       "one")
              first_function
              ;;
       "two")
              second_function
              ;;
       "three")
              third_function
              ;;
       "four")
              fourth_function
              ;;
       *)
              echo "Huh?" >&2
              exit 1
              ;;
esac
```

In the `examples` subdirectory there is a shell script called `case-equivalent.sh` that implements the logic of the `nested-if.sh` shell script (also in the `examples` subdirectory) using **`case`**. The heart of the script is shown above – **`first_function`**, **`second_function`**, **`third_function`** and **`fourth_function`** are all shell functions defined in the script.

If you want you can try the script out and see for yourself that it does the same thing as `nested-if.sh`.

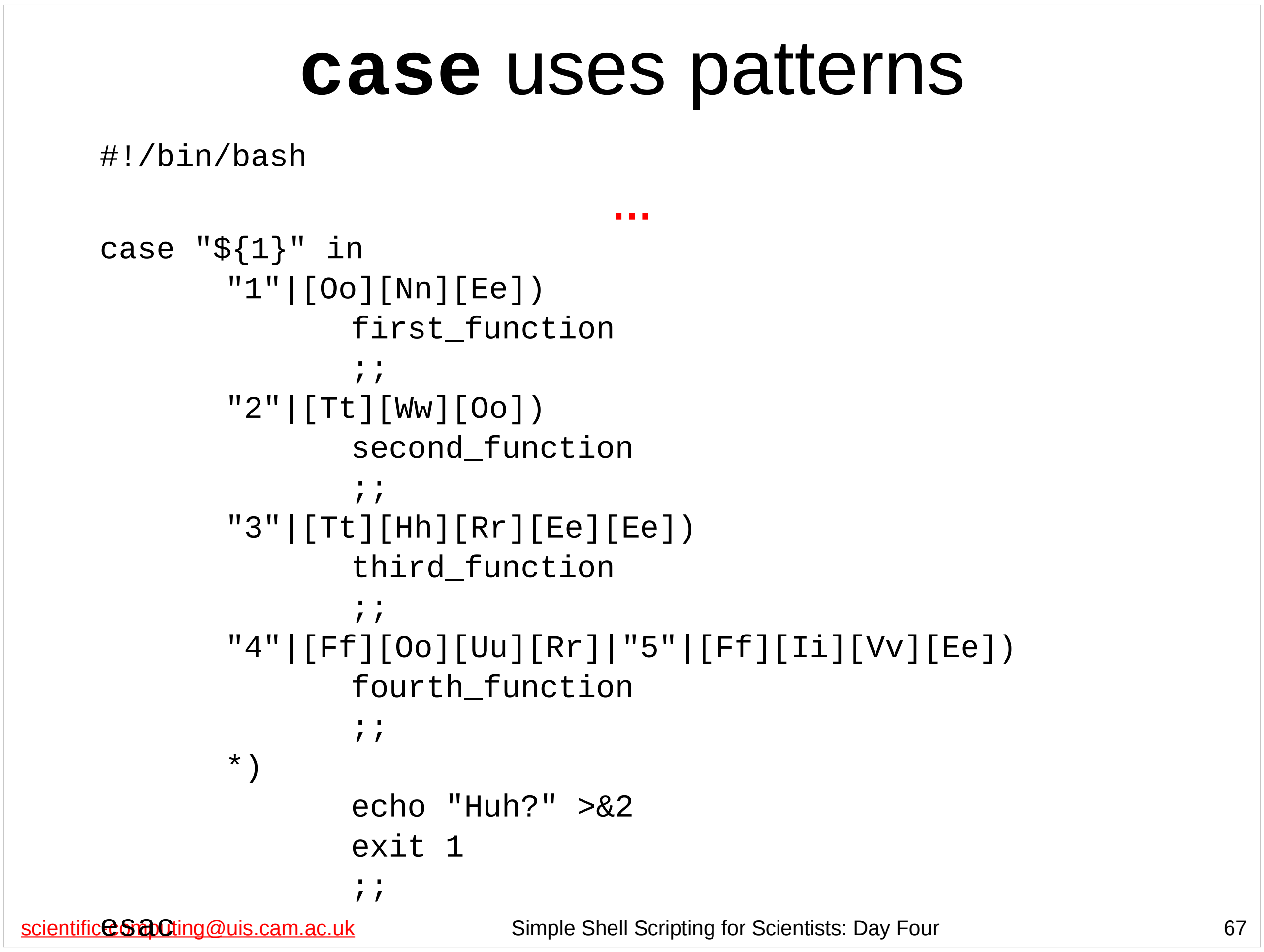


More use of **case**

```
#!/bin/bash
                                   ...
case "${1}" in
       "1"|"one"|"ONE")
              first_function
              ;;
       "2"|"two"|"TWO")
              second_function
              ;;
       "3"|"three"|"THREE")
              third_function
              ;;
       "4"|"four"|"FOUR"|"5"|"five"|"FIVE")
              fourth_function
              ;;
       *)
              echo "Huh?" >&2
              exit 1
              ;;
esac
```

In the `examples` subdirectory there is a shell script called `case-is-better.sh` that implements an expanded version of the `nested-if.sh` shell script (also in the `examples` subdirectory) using **`case`**. The heart of the script is shown above – **`first_function`**, **`second_function`**, **`third_function`** and **`fourth_function`** are all shell functions defined in the script.

Try the script out, giving it the uppercase words “`ONE`”, “`TWO`”, and the integers, `1`, `2`, etc as arguments. Then consider how many extra lines you would have to add to the original `nested-if.sh` script to get the same functionality if you used **`if`** instead.



case uses patterns

```
#!/bin/bash
                              ...
case "${1}" in
       "1"|[Oo][Nn][Ee])
              first_function
              ;;
       "2"|[Tt][Ww][Oo])
              second_function
              ;;
       "3"|[Tt][Hh][Rr][Ee][Ee])
              third_function
              ;;
       "4"|[Ff][Oo][Uu][Rr]|"5"|[Ff][Ii][Vv][Ee])
              fourth_function
              ;;
       *)
              echo "Huh?" >&2
              exit 1
              ;;
esac
```

You may have noticed that **case** uses ***** to mean “match anything”. We already met ***** when we looked at patterns earlier. **case** can use any of the patterns that are used in parameter expansion, so we can use **[…]** to mean match one of a range of the specified characters as above.

However, when using patterns like this for case, the pattern must **not** be enclosed in quotes or it won’t be treated as a pattern.

In the `examples` subdirectory there is a shell script called `case-is-great.sh` that demonstrates the use of patterns with **case**. Try the script out, giving it the mixed case words “OnE”, “tWO”, etc as arguments. Now consider how many extra lines you would have to add to get the same functionality in the original `nested-if.sh` script if you used **if** instead.

Third exercise

Convert the nested **if** statements in this script to **case** constructs.

...

```
# Report if there were problems
if [ "${myBAD_PARAM}" -eq "0" ] ; then
     true
elif [ "${myBAD_PARAM}" -eq "1" ] ; then
     echo "${myPROG} had a problem with parameter set: ${myPARAMS}" >&2
elif [ "${myBAD_PARAM}" -eq "2" ] ; then
     echo "gnuplot had a problem with parameter set: ${myPARAMS}" >&2
else
     echo "Problem with parameter set: ${myPARAMS}" >&2
fi
```

...

7 minutes

You should have created the `multi-variable-params.sh` script as a solution to the first exercise. (If you didn't, use the `multi-sizes-error.sh` script in the `scripts` subdirectory of your home directory instead, although note that that only has one nested **if** statement and the `echo` shell builtin commands in it will be slightly different.) This script will have at least one nested **if** statement (which will look something like the one shown on the slide above) – it may have more than one. Convert all its nested **if** statements into **case** statements.

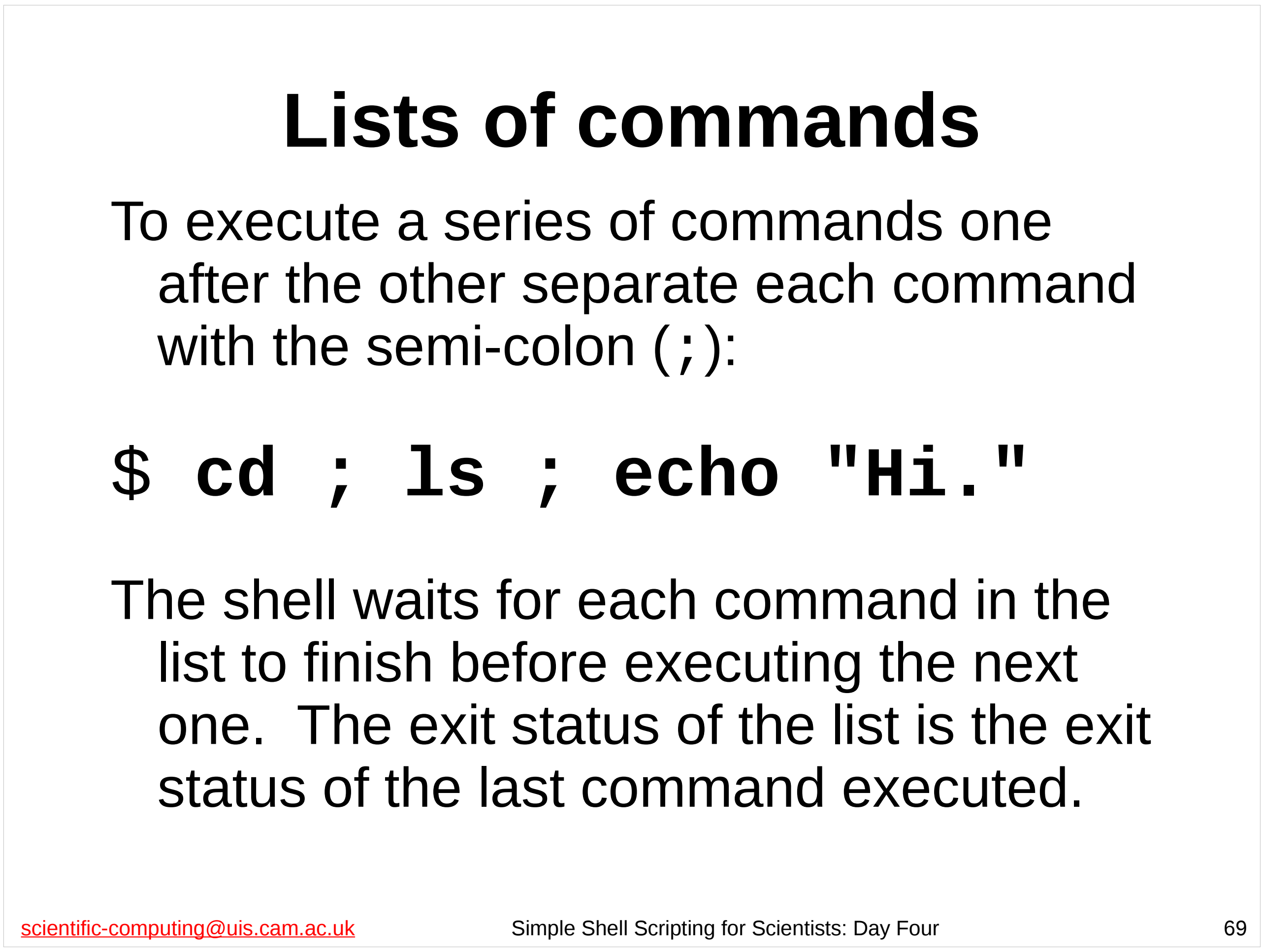
If you have any questions, or there's anything you don't understand, please ask the course giver or a demonstrator.

Check that the script still works by doing the following:

```
$ cd
$ rm –f stdout-* *.dat *.png logfile
$ cat scripts/variable_params | scripts/multi-variable-params.sh
$ ls
```

…and examining the files produced.

When you finish this exercise, take a short break and then we'll start again with the solution. (I really ***do*** mean take a break – sitting in front of computers for long periods of time is very bad for you. Move around, go for a jog, do some aerobics, whatever…)



Lists of commands

To execute a series of commands one after the other separate each command with the semi-colon (`;`):

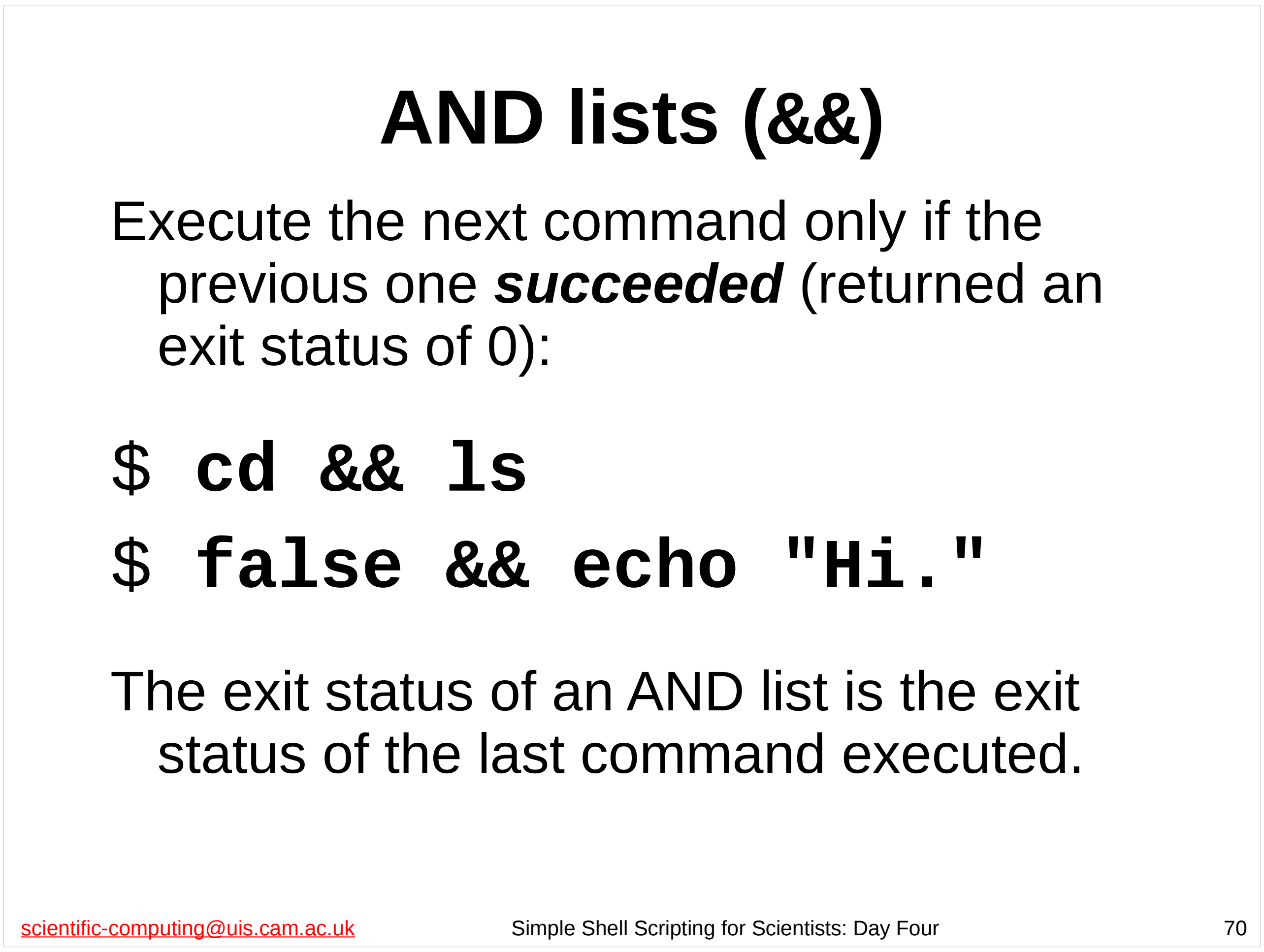
```
$ cd ; ls ; echo "Hi."
```

The shell waits for each command in the list to finish before executing the next one. The exit status of the list is the exit status of the last command executed.

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 69

A list of commands is an ordered sequence of commands. There are different types of lists. The simplest type of list uses the semi-colon (`;`) to separate commands: each command is executed in turn, and the shell waits for the each command to finish before executing the next one. The exit status of a list of commands is the exit status of the last command executed.

This type of list is most frequently used when you want to give a sequence of commands to the interactive shell to execute all at once rather than typing each command, waiting for the shell prompt, typing the next command, etc. This type of list is not used that often in shell scripts, although you may occasionally come across it.



AND lists (&&)

Execute the next command only if the previous one ***succeeded*** (returned an exit status of 0):

```
$ cd && ls
$ false && echo "Hi."
```

The exit status of an AND list is the exit status of the last command executed.

Note that an AND list uses two ampersands with no spaces between them (**&&**). A *single* ampersand (&) is used to run the command preceding it in the background.

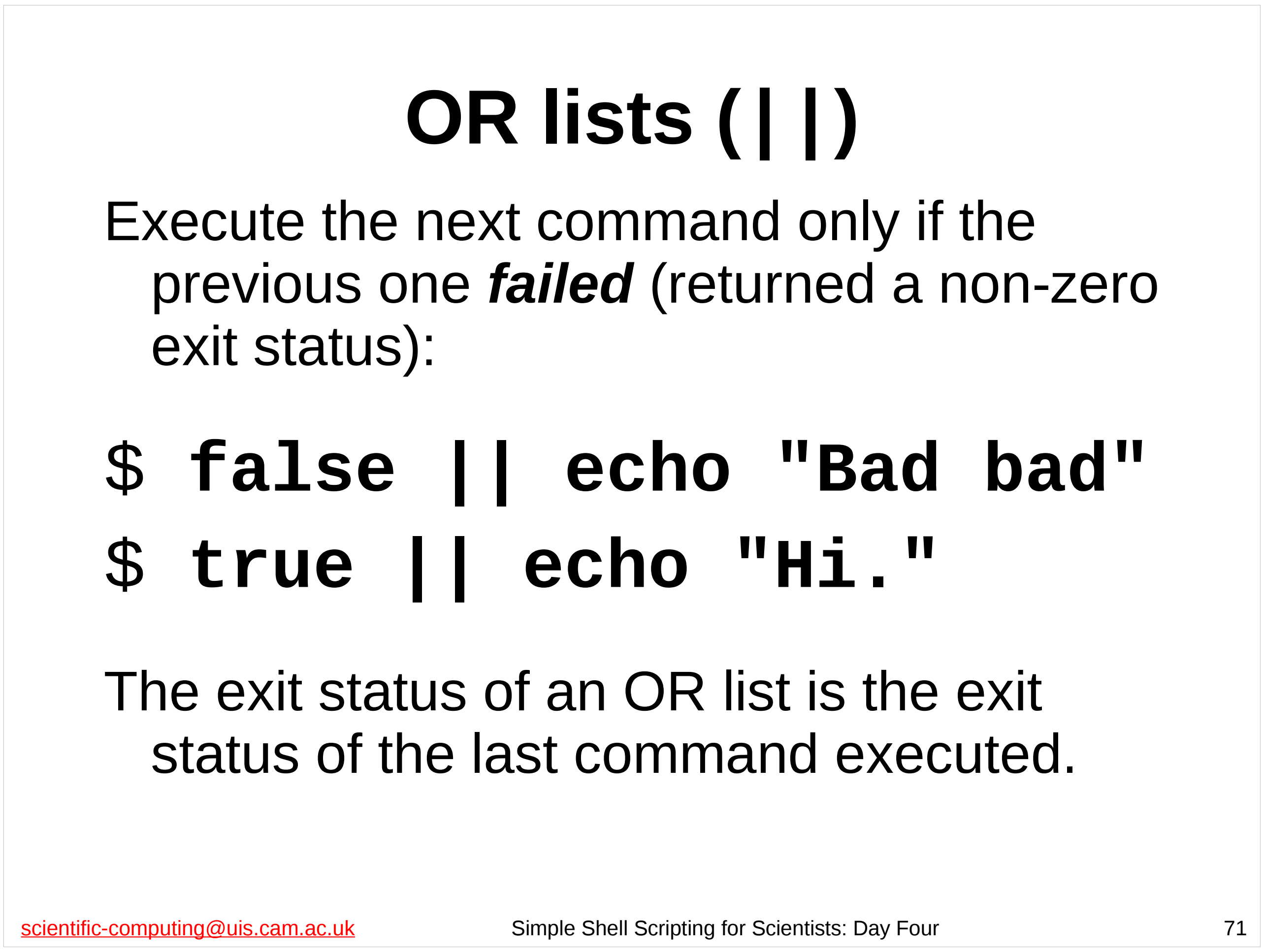
In an AND list, the next command is only executed if the previous command succeeded (i.e. it returned an exit status of 0). This type of list can be extremely useful for ensuring that one command is only executed if another one has succeeded regardless of whether or not **`set -e`** is in effect.

A common use of AND lists is to change directory to somewhere and, ***if and only if***, the change directory succeeded then do something, e.g.

```
cd /tmp && rm -Rf *
```

An AND list can have many commands in it, e.g.

```
cd && ls && echo "It worked."
```



OR lists (||)

Execute the next command only if the previous one ***failed*** (returned a non-zero exit status):

```
$ false || echo "Bad bad"
$ true || echo "Hi."
```

The exit status of an OR list is the exit status of the last command executed.

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 71

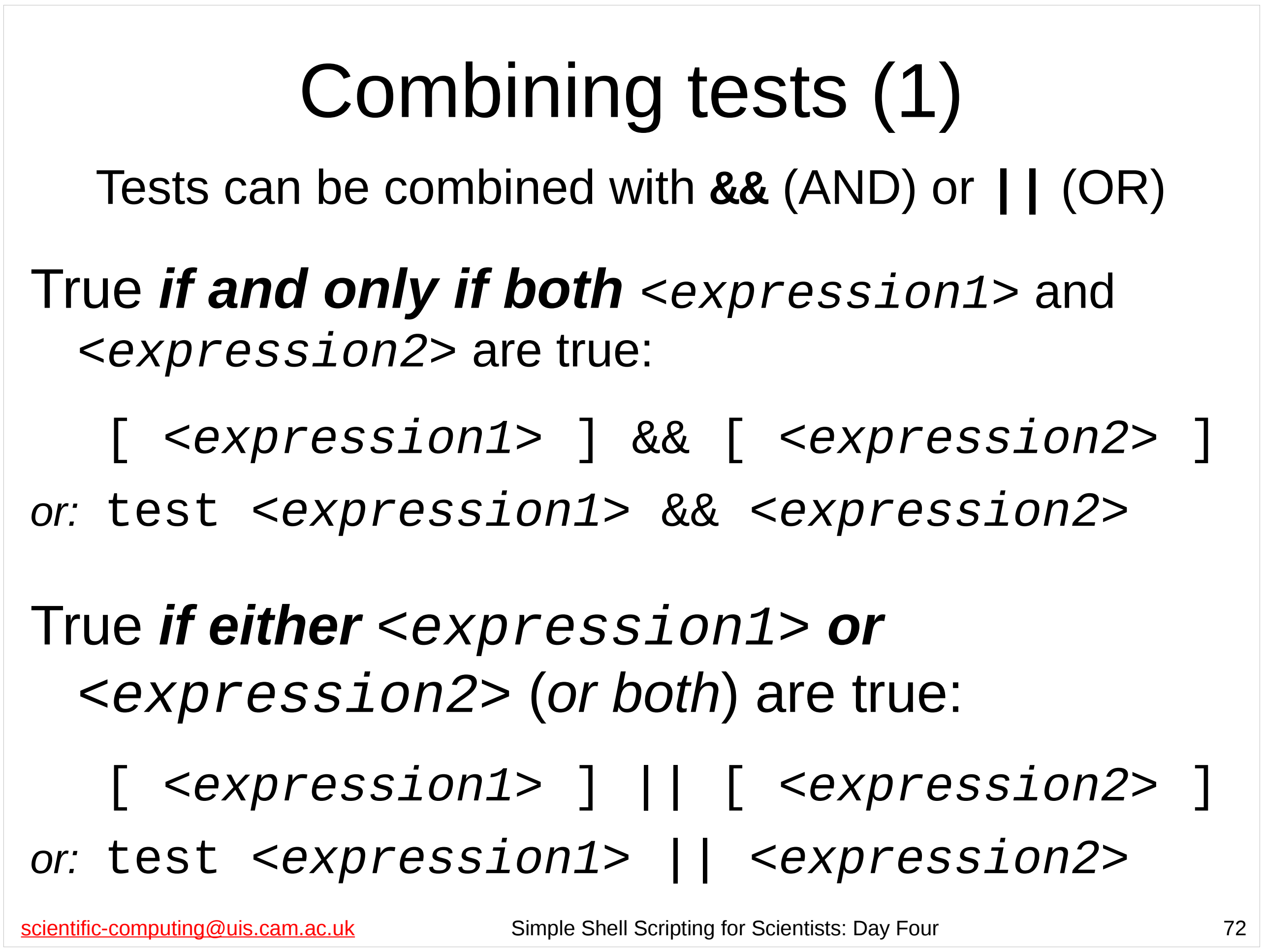
Note that an OR list uses two vertical bars with no spaces between them (**||**). A *single* vertical bar (|) is used to create a pipe (the standard output of the command before the vertical bar is sent to the standard input of the command after the vertical bar).

In an OR list, the next command is only executed if the previous command failed (i.e. it returned with a non-zero exit status). This type of list can be extremely useful when **`set -e`** is in effect to ensure that the shell does not quit even if a given command fails. Thus, a common use of an OR list is to try and run a command and, if the command fails, to then run **`true`** so that the exit status of the OR list is 0, thus ensuring that the shell does not quit even if **`set -e`** is in effect, e.g.

```
        program-that-often-crashes || true
```

An OR list can have many commands in it, e.g.

```
        false || cd /NOWHERE || echo "Nothing worked."
```



Combining tests (1)

Tests can be combined with **&&** (AND) or **||** (OR)

True ***if and only if both*** `<expression1>` and `<expression2>` are true:

```
[ <expression1> ] && [ <expression2> ]
```

or: `test <expression1> && <expression2>`

True ***if either*** `<expression1>` ***or*** `<expression2>` (*or both*) are true:

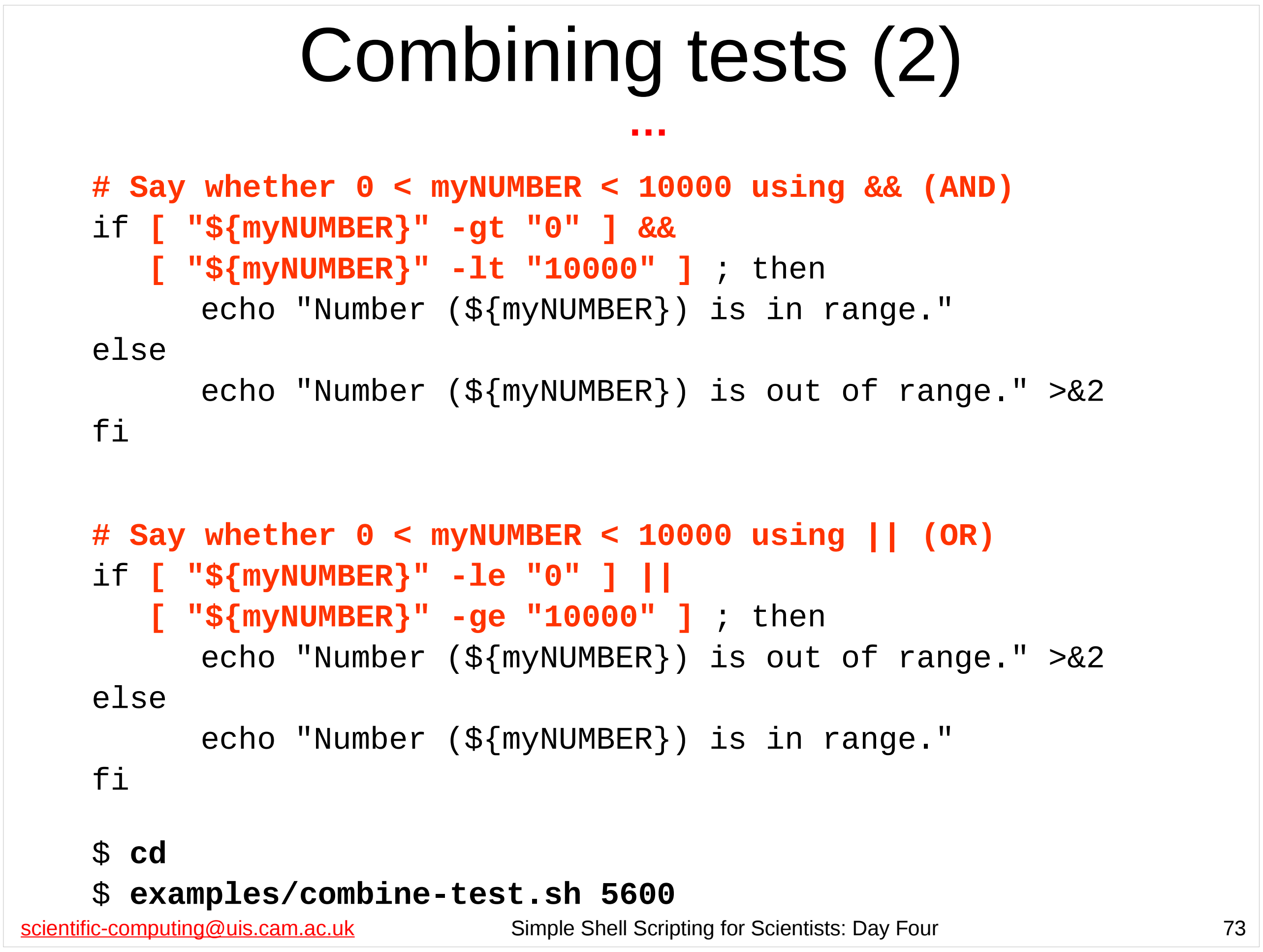
```
[ <expression1> ] || [ <expression2> ]
```

or: `test <expression1> || <expression2>`

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 72

All the tests we have already met can be combined with **&&** (AND) and **||** (OR). We can then use such combined tests in **`if`** statements.

See the next slide for an example.



Combining tests (2)

```
…

# Say whether 0 < myNUMBER < 10000 using && (AND)
if [ "${myNUMBER}" -gt "0" ] &&
   [ "${myNUMBER}" -lt "10000" ] ; then
      echo "Number (${myNUMBER}) is in range."
else
      echo "Number (${myNUMBER}) is out of range." >&2
fi


# Say whether 0 < myNUMBER < 10000 using || (OR)
if [ "${myNUMBER}" -le "0" ] ||
   [ "${myNUMBER}" -ge "10000" ] ; then
      echo "Number (${myNUMBER}) is out of range." >&2
else
      echo "Number (${myNUMBER}) is in range."
fi

$ cd
$ examples/combine-test.sh 5600
```

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 73

The `combine-tests.sh` script in the `examples` subdirectory of your home directory (partially shown on the slide above) demonstrates combining tests with **&&** (AND) and **||** (OR) in **`if`** statements.

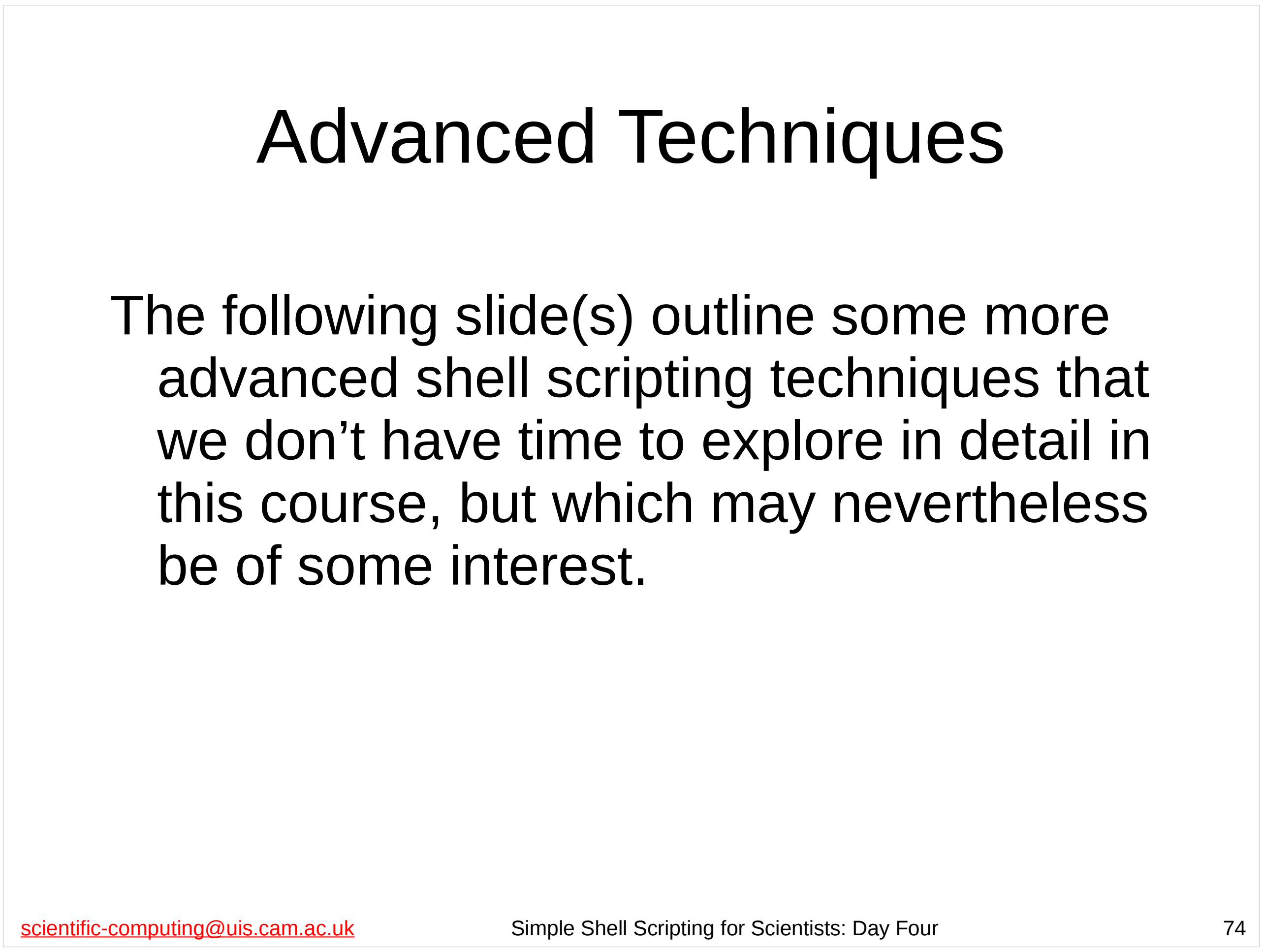
Note that the tests in the **`if`** statement have been split across several lines. This is not compulsory – you can put them all on the same line if you wish – but putting them on separate lines often improves readability.

Try the script by doing the following:

```
$ cd
$ examples/combine-test.sh 5600
```

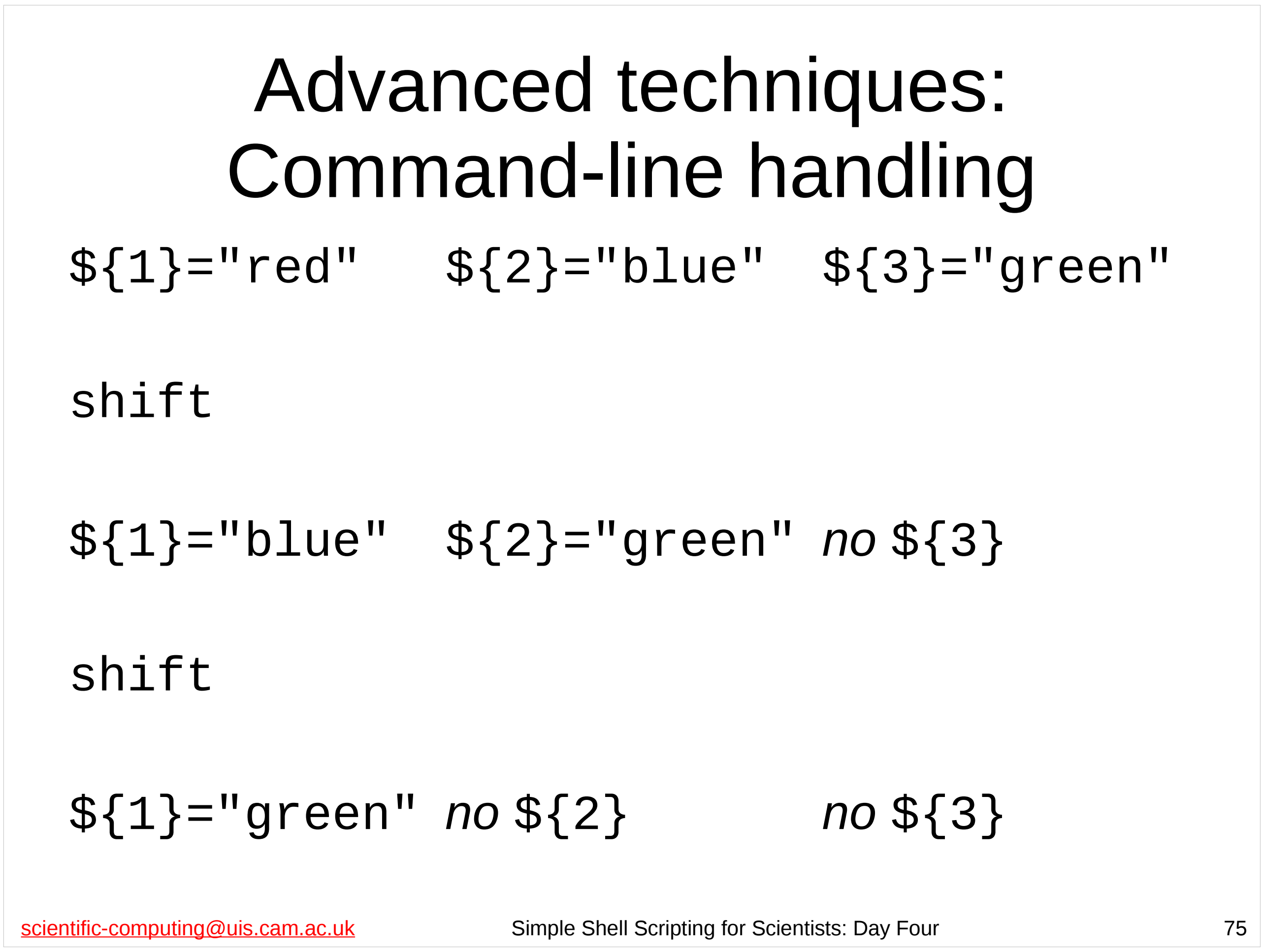
…and then try giving the script some other integers as its first argument until you are sure that it works the way you expect.

Please ask the course giver or a demonstrator if you have any questions or there’s anything you don’t understand.



Advanced Techniques

The following slide(s) outline some more advanced shell scripting techniques that we don’t have time to explore in detail in this course, but which may nevertheless be of some interest.



Advanced techniques: Command-line handling

`${1}="red"` `${2}="blue"` `${3}="green"`

`shift`

`${1}="blue"` `${2}="green"` *no* `${3}`

`shift`

`${1}="green"` *no* `${2}` *no* `${3}`

scientific-computing@uis.cam.ac.uk Simple Shell Scripting for Scientists: Day Four 75

The **`shift`** shell builtin command moves command-line parameters “along one to the left”.

Examples of its use are given in the files `shift1.sh` and `shift2.sh` in the `examples` directory.

In conjunction with the **`case`** construct we can use it to do some reasonably sophisticated command-line handling. The following files in the `examples` directory give some examples of how to do this:

```
params1.sh
params2.sh
```

Give us Feedback!

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/