Simple Shell Scripting for Scientists

Day Three

Julian King

Bruce Beckles

University of Cambridge Computing Service

Introduction

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

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

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

What we don't cover

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

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

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

We will be using bash 4.0 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 4.1, which was released on 31 December, 2009.

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

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

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

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

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

Outline of Course

1. Recap of days one & two
2. The **`if`** statement
3. **`exit`**, standard error

SHORT BREAK

4. More tests
5. **`if`**…**`then`**…**`else`**
6. Better error handling, **`return`**
7. **`if`**…**`elif`**…**`elif`**…**`elif`**…**`else`**

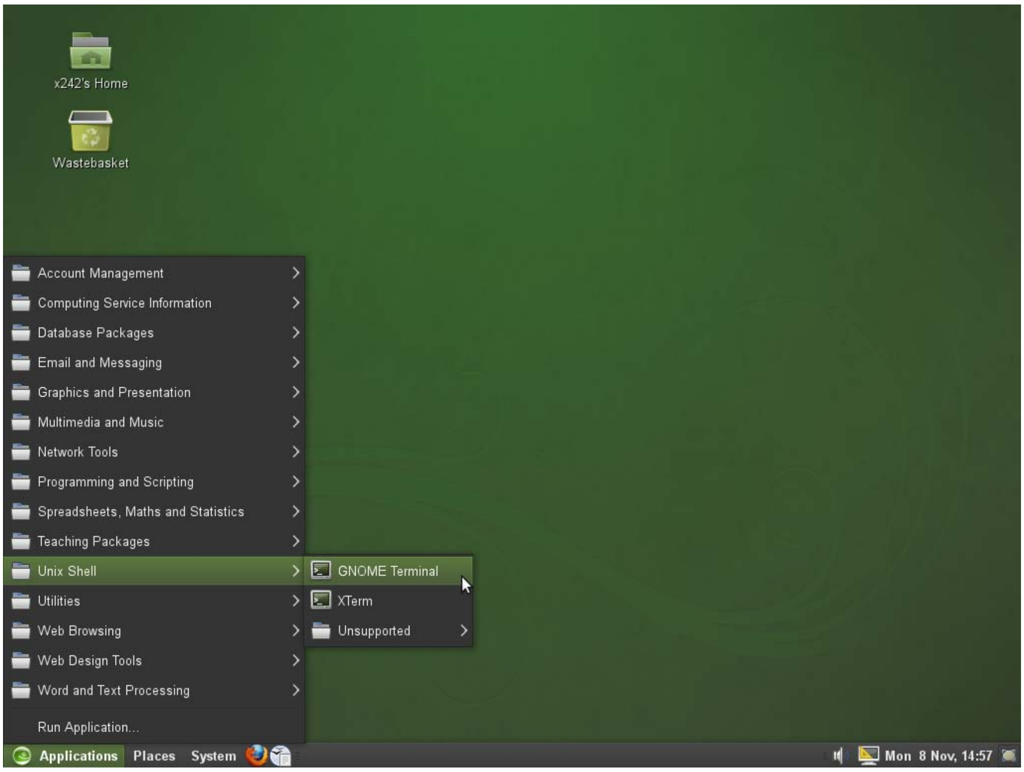
SHORT BREAK

8. Manipulating filenames
9. **`source`**

Exercise

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

Start a shell



Screenshot of newly started shell

Recap: Days One & Two

- 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
- (Integer) arithmetic tests
- Command substitution and (integer) arithmetic expansion
- The **`mktemp`** command

Recap: Shell functions (1)

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

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


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

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

`function` ***`function_name`*** **`()`**

`{`

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

`}`

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

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

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

Recap: Shell functions (2)

- "mini-shell scripts"
- Usually used for well-defined tasks (often called repeatedly)
- Specify arguments by listing them after function name when calling function

  `hello Dave`
- Positional parameters (and related special shell parameters) set to function's arguments within function

  In function `hello`, positional parameter **`1`** = `Dave`

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

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

main
```

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

Recap: Output redirection and pipes

- Commands normally send their output to *standard output* (which is usually the screen)
- Standard output can be ***redirected*** to a file
- A ***pipe*** takes the *output* of one command and supplies it to another command as *input*.

Recap: More input and output, and **`while`** loops

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

Note that even if we are using:

```
set -e
```

or the first line of our shell script is

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

the shell script will not exit if the "something" the **`while`** loop depends on gives a non-zero exit status (i.e. is false), since if it did, this would make **`while`** loops unusable(!).

Recap: Exit Status

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

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

You get the value of the special parameter **?** by using the construct **`${?}`**.

Recap: Tests

Test to see if something is true:

`[ <expression> ]`

or: `test <expression>`

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

```
[ "a" –eq "b" ]
[ "a" –le "b" ]
[ "a" –gt "b" ]
```


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

`"a" –lt "b"`	true if and only if the integer **a** is less than the integer **b**
`"a" –le "b"`	true if and only if the integer **a** is less than or equal to the integer **b**
`"a" –eq "b"`	true if and only if the integer **a** is equal to the integer **b**
`"a" –ne "b"`	true if and only if the integer **a** is not equal to the integer **b**
`"a" –ge "b"`	true if and only if the integer **a** is greater than or equal to the integer **b**
`"a" –gt "b"`	true if and only if the integer **a** is greater than the integer **b**

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

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

```
[ "${VAR}" –eq "5" ]
```

Or, equivalently:

```
test "${VAR}" –eq "5"
```

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

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

Recap: Shell arithmetic

- The shell can do *integer* arithmetic – this is known as ***arithmetic expansion***
- The shell can also perform arithmetic ***tests*** on integers (>, ≥, =, ≤, <)

Recap: Arithmetic Expansion **$(( ))**

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

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

Example:

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


The shell can also do (primitive) integer arithmetic, which can be very useful.

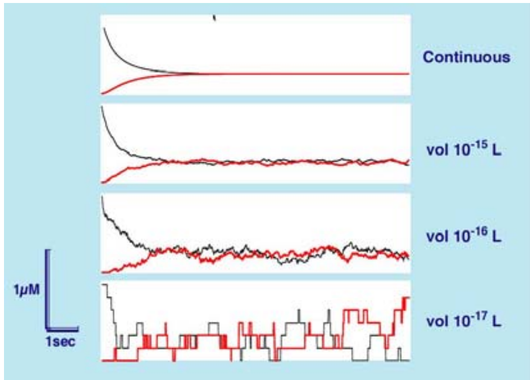
The construct **$((<*arithmetic-expression*>))** means replace **$((<*arithmetic-expression*>))** with the result of the ***integer*** arithmetic expression **<*arithmetic-expression*>**. This is known as *arithmetic expansion*. (The arithmetic expression is evaluated as integer arithmetic.)

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

Recap: What are we trying to do?

Scientific computing

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



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

As mentioned on the 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.

Sample program: **`iterator`**

```
$ ./iterator 100 100 1000 0.05
x dimension of grid:     100
y dimension of grid:     100
Number of iterations:    1000
Epsilon:                 0.050000

Output file:             output.dat

Iterations took 2.100 seconds
```


The **`iterator`** program is in your home directory. It is a program written specially for this course, but we'll be using it as an example program for pretty general tasks you might want to do with many different programs. Think of **`iterator`** 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 **`iterator`** program takes 4 numeric arguments on the command line: 3 positive integers and 1 floating-point number. It always writes its output to a file called `output.dat` in the current working directory, and also writes some informational messages to the screen.

The **`iterator`** 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 **`iterator`** is not well behaved is this: every time it runs it creates a file called `running` in the current directory, and it will not run if this file is already there (because it thinks that means it is already running). Unfortunately, it doesn't remove this file when it has finished running, so we have to do it manually if we want to run it multiple times in the same directory.

Exercise from Day Two (Part One)

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

The `multi-run-while.sh` shell script (in the `scripts` subdirectory of your home directory) runs the **iterator** program (via a shell function called **run_program**) once for each parameter set that it **read**s in from standard input. This exercise requires you to modify the **run_program** shell function of that script so that, as well as running the **iterator** program it also runs **gnuplot** to turn the output of the **iterator** program into a graph.

One sensible way of doing this would be as follows:

1. Figure out the full path of the `iterator.gplt` file. Store it a shell variable (maybe called something like **myGPLT_FILE**).

2. Immediately after running **iterator**, run **gnuplot**:

   ```
   gnuplot "${myGPLT_FILE}"
   ```

3. Rename the `output.png` file produced by **gnuplot** along the same lines as the `output.dat` file produced by **iterator** is renamed.

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

Solution to Part One

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

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

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

# Program to run: iterator
myPROG="$(pwd -P)/iterator"

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


If you examine the `multi-run-while.sh` script in the `scripts` subdirectory of your home directory, you will see that it has been modified as shown above to run **`gnuplot`** after running **`iterator`**.

You should be able to tell what all the highlighted parts of the shell script above do – 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.

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

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

You should see that there is a PNG file for each of the renamed `.dat` output files. You should also inspect `logfile` to see what it looks like now.

Exercise from Day Two (Part Two)

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

For a given parameter set *a b c d*, first your script should run **`iterator`** with the parameter set:

a b **10** *d*

…then with the parameter set:

a b **100** *d*

…and then with the parameter set:

a b **1000** *d*

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

10 10 50 0.5

…it should then run the **`iterator`** program 3 times, once for each of the following parameter sets:

10 10 **10** 0.5

10 10 **100** 0.5

10 10 **1000** 0.5

Now, currently the script will `read` in a parameter set and then call the **`run_program`** function to process that parameter set. Clearly, instead of passing all four parameters that the script `read`s in, the new script will now only be passing the first (**`myNX`**), second (**`myNY`**), and fourth (**`myEPSILON`**) parameters that it has `read` in. However, the **`iterator`** program *requires* 4 parameters (and it cares about the order in which you give them to it), so the new script still needs to give it 4 parameters, it is just going to <u>ignore</u> the third parameter it has `read` (**`myN_ITER`**) and substitute values of its own instead.

There are two obvious approaches you could have taken in performing this task. One would be to call the **`run_program`** function 3 times, once with 10 as the third parameter, once with 100 as the third parameter and once with 1000 as the third parameter. The other would be to use some sort of loop that calls the **`run_program`** function, using the appropriate value (10, 100 or 1000) for the third parameter on each pass of the loop. I wanted you to use the loop approach.

Solution to Part Two (1)

```
#!/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 myNX myNY myN_ITER myEPSILON myJUNK ; do
   # Instead of using read in value for iterations,
   #  use 10, then 100, then 1000.
   for zzITER in 10 100 1000 ; do
        # Run program
        run_program "${myNX}" "${myNY}" "${zzITER}" "${myEPSILON}"
   done

                              …
```


If you examine the `multi-10-100-1000.sh` script in the `scripts` subdirectory of your home directory, you will see that it is a version of the `multi-run-while.sh` script that has been modified as shown above.

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 this is a solution to this part of the exercise – 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 stdout-* logfile
$ cat scripts/param_set | scripts/multi-10-100-1000.sh
$ 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).

Solution to Part Two (2)

```
#!/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 myNX myNY myN_ITER myEPSILON myJUNK ; do
   # Instead of using read in value for iterations,
   #  use 10, then 100, then 1000.
   zzITER=10
   while [ "${zzITER}" -le "1000" ]  ; do
        # Run program
        run_program "${myNX}" "${myNY}" "${zzITER}" "${myEPSILON}"
        zzITER=$(( ${zzITER} * 10 ))
   done
                              ...
```


There is another way you could have achieved the same thing, also using a loop, but this time using a **`while`** loop instead of a **`for`** loop. This may seem a somewhat perverse way of doing things, but if you had a parameter that was an integer that you wished to increase by some constant factor a large number of times, e.g. 2, 4, 8, 16, 32, 64, etc. then this would be a better way of doing it than trying to type them all out as a list of values for a **`for`** loop.

If you examine the `multi-10-100-1000-alternate.sh` script in the `scripts` subdirectory of your home directory, you will see that it is a version of the `multi-run-while.sh` script that has been modified as shown above.

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 this is a solution to this part of the exercise – 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 stdout-* logfile
$ cat scripts/param_set | scripts/multi-10-100-1000-alternate.sh
$ ls
```

…and examining the files produced.

Exercise from Day Two (Part Three)

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

Instead of running **`iterator`** three times for each parameter set it **`read`**s in, this script should accept a set of values on the command line, and use those instead of the hard-coded 10, 100, 1000 previously used.

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

So, if the script from the previous part of the exercise was called `multi-10-100-1000.sh`, and we called this new script `multi-iterations.sh` (and stored both in the `scripts` directory of our home directory), then running the new script like this:

```
$ cat ~/scripts/param_set | ~/scripts/multi-iterations.sh 10 100 1000
```

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

```
$ cat ~/scripts/param_set | ~/scripts/multi-10-100-1000.sh
```


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

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

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

Solution to Part Three

```
#!/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 myNX myNY myN_ITER myEPSILON myJUNK ; do
   # Instead of using read in value for iterations,
   #  cycle through command line arguments.
   for zzITER in "${@}" ; do
        # Run program
        run_program "${myNX}" "${myNY}" "${zzITER}" "${myEPSILON}"
   done
                              …
```


If you examine the `multi-iterations.sh` script in the `scripts` subdirectory of your home directory, you will see that it is a version of the `multi-10-100-1000.sh` script that has been modified as shown above.

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 this is a solution to this part of the exercise – 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 stdout-* logfile
$ cat scripts/param_set | scripts/multi-iterations.sh 10 100 1000
$ ls
```

You should see that a number of PNG and `.dat` files have been produced.

What else are tests good for?

We have seen that we can use tests in **`while`** loops. What else are they good for?

Suppose we know some (valid) parameters for our program produce no interesting output. Could we use some tests to filter these out?

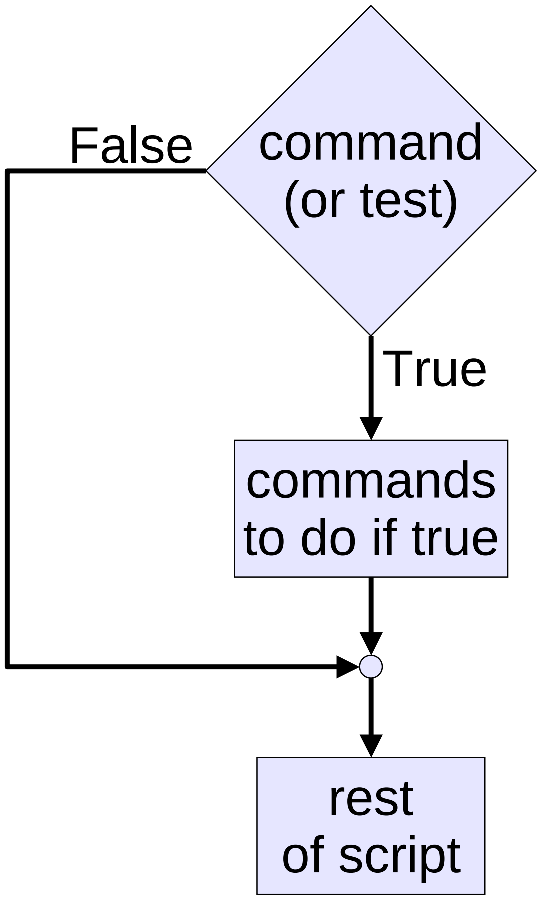
Using tests (1)

We've met (integer) arithmetic tests.

Suppose we'd like to test to see whether some of our parameters are within a certain range (say 1 to 10000). If they are not, we shouldn't do anything, i.e.

If parameter < 1 or parameter > 10000 stop executing the script…

How do we do this?



`if` statement

Do something only *if* some command (or test) is *true*

scientific-computing@ucs.cam.ac.uk Simple Shell Scripting for Scientists: Day Three 28

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)).

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(!).

if

Do something **only** *if* some command is *true*

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


We use an **`if`** statement like this:

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

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

In a similar manner to **`for`** and **`while`** loops, you can put the **`then`** on a separate line, in which case you can omit the semi-colon (**`;`**), i.e.

```
if <command>
then
        <some commands>
fi
```

Now, we just need to know how to tell our script to stop executing and we will have all the pieces we need to modify our script to behave the way we want…

`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.)

Using **if** (and tests)

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

                                    …
while read myNX myNY myN_ITER myEPSILON myJUNK ; do
   # Instead of using read in value for iterations,
   #  cycle through command line arguments.
   for zzITER in "${@}" ; do

        if [ "${zzITER}" -lt "1" ] ; then
                echo "Number of iterations (${zzITER}) must be positive!"
                exit 1
        fi
        if [ "${zzITER}" -gt "10000" ] ; then
                echo "Too many iterations (${zzITER})!"
                exit 1
        fi

        # Run program

                                    …
```

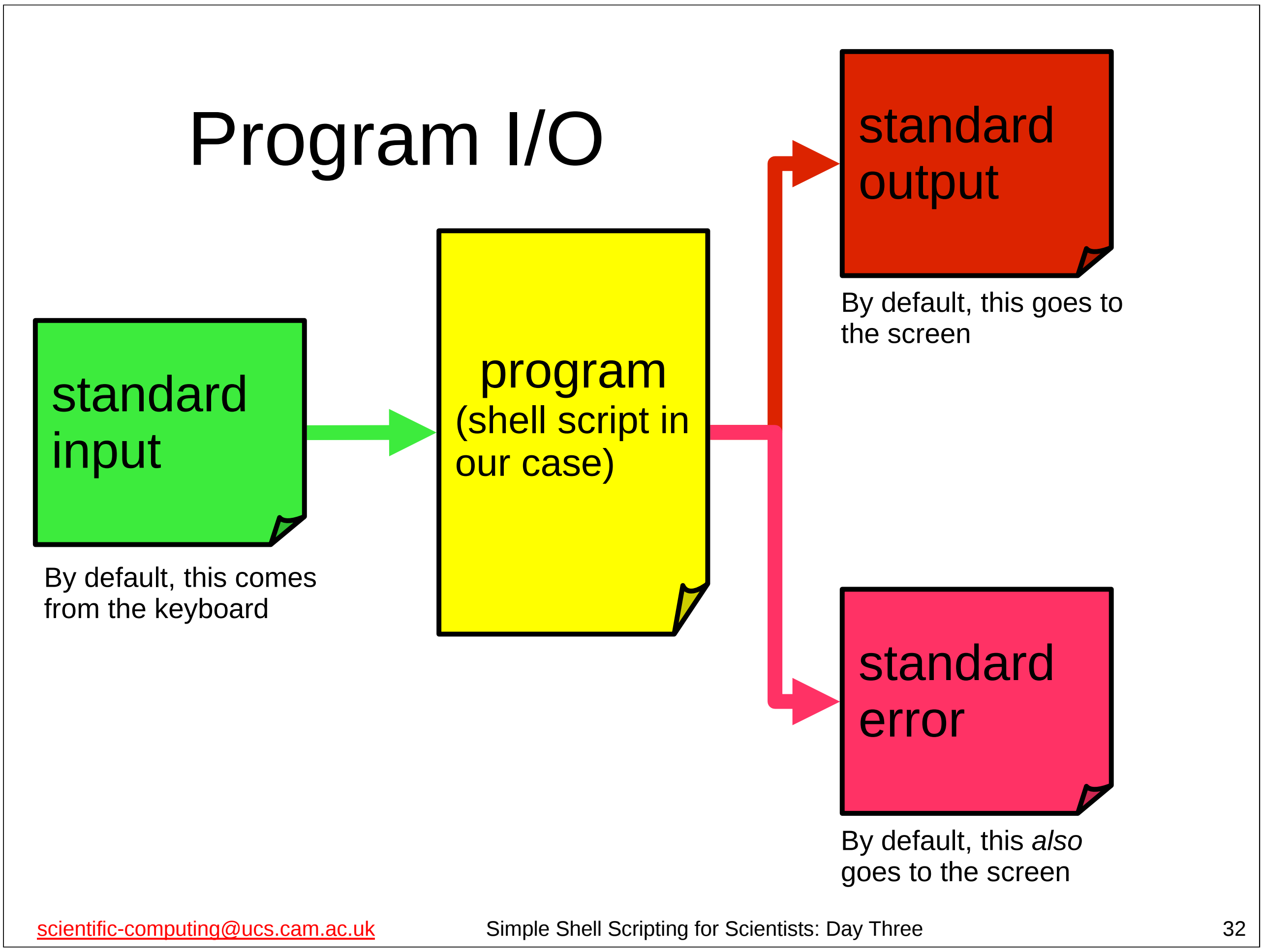

Modify the `multi-iterations.sh` script in the `scripts` subdirectory of your home directory as shown above. (Make sure to save it after you've modified it.)

What do you think these modifications do?

Note that if we **`exit`** the script because one of the command line arguments is incorrect, then we need to indicate that there was a problem running the script, so we set our exit status to a non-zero value (1 in this case, which is the conventional value to use if we don't set different values for the exit status for different types of error).

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

```
$ cd
$ rm –f *.dat *.png stdout-* logfile
$ cat scripts/param_set | scripts/multi-iterations.sh 0
Number of iterations (0) must be positive!
$ cat scripts/param_set | scripts/multi-iterations.sh 20000
Too many iterations (20000)!
```



We are already familiar with *standard output* as a "channel" along which our program or shell script's output is sent to somewhere. By default, this "somewhere" will be the screen, unless we *redirect* it to somewhere else (like a file).

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.

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

Standard Error (1)

```
$ ls iterator
iterator

$ ls iterator zzzzfred
/bin/ls: zzzzfred: No such file or directory
iterator
$ ls iterator zzzzfred > stdout-ls
/bin/ls: zzzzfred: No such file or directory

$ cat stdout-ls
iterator
```


If we look at what happens when a standard Unix command, such as **`ls`**, encounters an error, the way standard error works may become clearer.

When we ask **`ls`** to list a non-existent file, it prints out an error message. If we redirect the (standard) output of **`ls`** to a file, we see that the error message still goes to the screen. This is because the error message does not go to standard output, but to standard error. If we wanted to send the error message to file we would need to redirect *standard error* to that file.

So how do we manipulate standard error?

Please note that the output of the **`ls`** command may not exactly match what is shown on this slide – in particular, the colours may be slightly different shades.

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
```


Note that 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.

Using standard error

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

                              …
while read myNX myNY myN_ITER myEPSILON myJUNK ; do
   # Instead of using read in value for iterations,
   #  cycle through command line arguments.
   for zzITER in "${@}" ; do

        if [ "${zzITER}" -lt "1" ] ; then
                echo "Number of iterations (${zzITER}) must be positive!" >&2
                exit 1
        fi
        if [ "${zzITER}" -gt "10000" ] ; then
                echo "Too many iterations (${zzITER})!" >&2
                exit 1
        fi

        # Run program

                              …
```


Modify the `multi-iterations.sh` script in the `scripts` subdirectory of your home directory as shown above. (Remember to save it after you've made the above changes or they won't take effect.)

Since when we exit the script because we don't like one of the parameters, we consider this an error, the message we print out telling the user what the problem is is an error message, and so should go to standard error rather than standard output. This is what adding ">&2" to those `echo` shell builtin commands does.

This is the conventional behaviour for shell scripts (or indeed any other program) – ordinary output goes to standard output, error messages go to standard error. It is *very important* that you follow this convention when writing your own shell scripts as this is what anyone else using them will expect them to do.

First exercise

The problem with the checking we've added to the `multi-iterations.sh` script is that it will only stop as and when it encounters a bad parameter, so that it may start a run and then abort it part way through.

Write a function called **`check_args`** to check that each of its arguments is between 1 and 10000. (You can assume that each argument is an integer.) Modify the script to use this function on *all* the command line arguments before it enters its **`while`** loop.

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

                                  …

function check_args()
{
# This function checks all the arguments it has been given
                                        What goes here?
}

                                  …

# Location of log file
myLOGFILE="${myDIR}/logfile"

# Make sure our command line arguments are okay before continuing
check_args "${@}"

                                  …
```


The `multi-iterations.sh` shell script is in the `scripts` directory of your home directory. Your task is to add a shell function to this script that the script can use to check *all* the command line parameters it has been given to ensure they are between 1 and 10000 (you can assume the parameters are integers), and then to modify the script to call that function before it does anything significant. Above I've given you the skeleton of what the modified script should look like. You should be able to fill in the rest. *Make sure you save your script after you've modified it.*

Note that you need to (re)move the **`if`** statements that we've added to the shell script as once we use the **`check_args`** function we will have already checked the command line arguments by the time we enter the **`while`** loop, and there is no point in checking them twice.

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…)

Note that in the skeleton above I call the **`check_args`** function *before* I use the **`mktemp`** command – there's no point in creating a temporary directory if I've been given bad parameters and am going to abort my script…

Hint: We've actually already written most of the function – so you can cut-and-paste those lines of the current shell script into the function. You then need to somehow ***loop*** through all the function's arguments, checking each in turn.

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 have already met, 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).

Remember 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 ! ./iterator ; then
          echo "Unable to run ./iterator successfully"
fi
```

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

Using tests (2)

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

function check_args()
{
# This function checks all the arguments it has been given

# Make sure our first argument is not nothing; this also makes sure we are not
#  given no arguments at all.
if [ -z "${1}" ] ; then
   echo "No valid arguments given." >&2
   echo "This script takes one or more number of iterations as its arguments." >&2
   echo "It requires at least one argument." >&2
   exit 1
fi
```

...

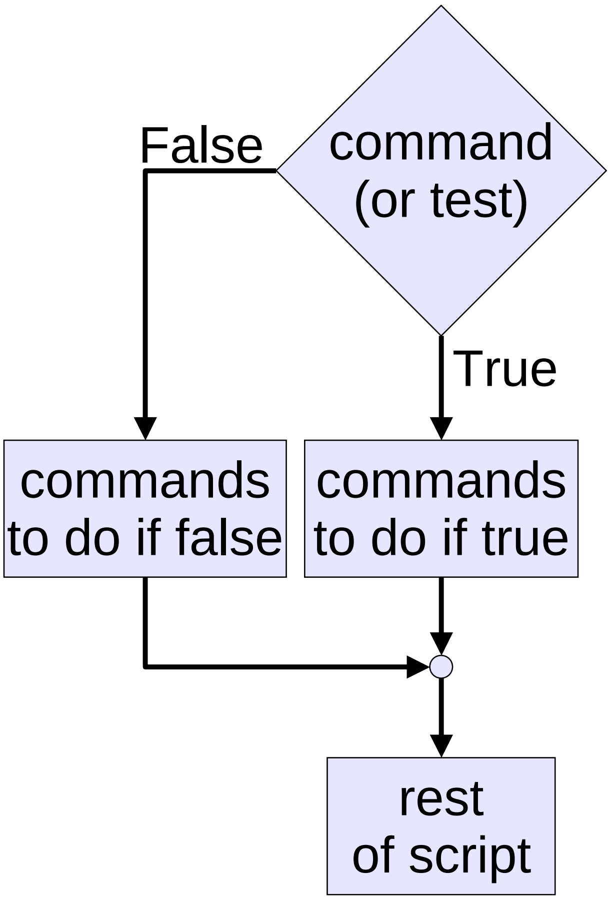
Modify the `multi-iterations.sh` script in the `scripts` subdirectory of your home directory as shown above. (Remember to save it after you've made the above changes or they won't take effect.)

Now we not only complain if we have arguments that are out of range, we also complain if we have no arguments at all (and also if our first argument is an empty string). Try this script out now and see what happens:

```
$ cd
$ cat scripts/param_set | scripts/multi-iterations.sh
No valid arguments given.
This script takes one or more number of iterations as its arguments.
It requires at least one argument.
```

Note also that we are once again making use of the fact that we have separated some functionality from our script and put it in a function. We can easily change the function without complicating the rest of the script or affecting its structure.

`if`…`then`…`else`



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

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 command (or test) we are evaluating is ***false***.

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
```


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.

Using **if**…**then**…**else**

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

                                    …

function multi_iterate()
{
# Instead of using read in value for iterations,
#  cycle through arguments passed to function.

    for zzITER in "${@}" ; do

           # Run program
           run_program "${myNX}" "${myNY}" "${zzITER}" "${myEPSILON}"

    done
}

                                    …
while read myNX myNY myN_ITER myEPSILON myJUNK ; do

    if [ -z "${1}" ] ; then
           # If no first command line argument given,
           #  use these defaults.
           echo "Using default number of iterations: 10, 100, 1000"
           multi_iterate "10" "100" "1000"
    else
           # Use the command line arguments
           multi_iterate "${@}"
    fi

done
```


Open up the `multi-iterations-default.sh` script in the `scripts` subdirectory of your home directory in your favourite editor (or gedit) and have a look at it.

Notice that the **`check_args`** function in this script doesn't complain if there are no command line arguments. This is because this script will use some *default* parameters if it hasn't been given any on the command line. (And note that we print a message on the screen so the person using our script knows its using default values and *what those values are*.)

Pay particular attention to the bits of the script highlighted above. Can you work out what we've changed and how the shell script will now behave? If not, please tell the course giver or a demonstrator what part of the script you don't understand.

Try out this script and see what happens:

```
$ cd
$ rm –f *.dat *.png stdout-* logfile
$ cat scripts/param_set | scripts/multi-iterations-default.sh
$ ls
```

Note that we didn't *need* to create a separate **`multi_iterate`** function – we could have just typed out very similar lines of shell script twice. This would have been a mistake – just like with real programming languages, repetition of parts of our script (program) are almost *always* to be avoided.

Better error handling (1)

At the moment, any errors stop our script dead. Often, that's better than letting it carry on regardless, but sometimes we want to be a bit more sophisticated.

For instance, supposing a few parameter sets we read in are corrupt and cause errors in **`iterator`** or **`gnuplot`** – we might want to note which ones these were and continue with the others.

How can we do this?

`return`

Just like programs and shell scripts have an exit status, so too do shell functions. We can set the exit status of a function using the **`return`** shell builtin command.

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

```
return
```

…or we can explicitly set an exit status as we exit the function thus:

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 (if desired).

As with ordinary programs and shell scripts themselves, the exit status of a shell function is an integer between 0 and 255, and should be 0 ***only*** if the function was successful in what it was trying to do. If the function encounters an error it should **`return`** with a non-zero exit status.

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.

(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.)

Better error handling (2)

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

...

```
function multi_iterate()
{
# Instead of using read in value for iterations,
#  cycle through arguments passed to function.

for zzITER in "${@}" ; do

    # Run program and report if there were problems
    if ! run_program "${myNX}" "${myNY}" "${zzITER}" "${myEPSILON}" ; then
          echo "Problem with parameter set: ${myNX} ${myNY} ${zzITER} ${myEPSILON}" >&2
    fi

done
}
```

...

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

Open up the `multi-iterations-errors.sh` script in the `scripts` subdirectory of your home directory in your favourite editor (or gedit) and have a look at it.

First have a look at the `multi_iterate` function, paying particular attention to the bits of the script highlighted above. Can you work out why we've changed this function like this? Recall that shell functions should exit with an exit status of 0 only if they were successful, and that **`if ! command`** will do something only if **`command`** *failed* (exited with a non-zero exit status) – **`command`** can be a shell function as well as a program or shell script.

To be sure that this really is behaving the way we expect, we need to look at the **`run_program`** function and see how that's been changed. First though, we need to learn how to toggle the shell's "quit on any error" behaviour on and off at will…

`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.

Better error handling (3)

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

                              …

function run_program()
{

                              …

    # Run program with passed arguments
    set +e
    "${myPROG}" "${@}" > "stdout-${1}-${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
            rm -f running
            echo "Failed!  Exit status: ${myPROG_ERR}" >> "${myLOGFILE}"
            return 1
    fi

                              …
```


Now look at the **`run_program`** function in the `multi-iterations-errors.sh` script, paying particular attention to the bits of the script highlighted above.

Can you work out what the highlighted bits are doing? Recall that the exit status of the last command that ran is stored in the special shell parameter **`?`**.

We observe that the logic of this function is that if the **`iterator`** 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)?

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

You should try out this script and see what it does:

```
$ cd
$ rm –f *.dat *.png stdout-* logfile
$ cat scripts/bad_param_set | scripts/multi-iterations-errors.sh
Nx must be positive
Problem with parameter set: Z00 100 10 0.1
Nx must be positive
Problem with parameter set: Z00 100 100 0.1
Nx must be positive
Problem with parameter set: Z00 100 1000 0.1
$ ls
```

The file `bad_param_set` contains one bad parameter set mixed in amongst some good ones, as you can see by inspecting it.

Nested **`if`**s (1)

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
```


We can have even more complicated **`if`** statements than those we have met. 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, but they involve a construct (**`case`**) and a shell builtin command (**`shift`**) that we don't cover on this course – brief notes on these are given on the "Advanced techniques" slides at the end of this course.

Nested **ifs** (2)

```
#!/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
```


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 see 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.

Second exercise

The `multi-iterations-errors.sh` script is reasonably robust at dealing with bad parameter sets. However, it would be nice if it told us whether it was **`iterator`** or **`gnuplot`** which failed.

Modify this script so that in its `multi_iterate` function it prints different messages depending on whether it was **`gnuplot`** or **`iterator`** that failed. (You may also need to modify other parts of the script as well.)

When you've finished this exercise, take a short break (break = "not still in front of the computer") and then we'll look at the answer.

The `multi-iterations-errors.sh` shell script is in the `scripts` directory of your home directory. Your task is to modify this script – mainly the **`multi_iterate`** function – so that the **`multi_iterate`** function prints out different messages on standard error depending on whether it was **`iterator`** or **`gnuplot`** that failed. *Make sure you save your script after you've modified it.*

Some of you may be tempted to just dispense with `bash`'s "exit the shell script on any error" feature for this exercise. **<u>Don't</u>** – part of the purpose of this exercise is to get used to how the shell handles errors and how you work with this.

Remember that this shell script attempts to change directory – ***a very dangerous thing to do in a shell script***, so you must make sure that if the script fails to change directory that it exits and doesn't try to do things in the wrong directory. The easiest way to do that is to have **`set -e`** in effect.

When you finish this exercise, take a short break and then we'll start again with the solution. (Yes, I really ***do*** mean "a break from the computer".)

Hint: One approach is to get the **`run_program`** function to return a different exit status depending on whether it was **`iterator`** or **`gnuplot`** that failed. You could then test for this in the **`multi_iterate`** function.

Another hint: You may wish to use nested **`if`** statements, although they aren't the only way to do this exercise.

Manipulating filenames (1)

```
$ 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
```


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?

`${VARIABLE%word}`

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

```
$ myFILENAME="output.dat"
$ echo "${myFILENAME%.dat}"
output
```


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 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 variables 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 more easily rename the files.

Manipulating filenames (2)

```
#!/bin/bash -e

function rename_files()
{
   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
}
```


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.

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 (for the moment accept that the **`source`** shell builtin command "loads" the functions from `my-function.sh` into the running instance of the shell – we'll look at it in more detail in a minute):

```
$ 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
```

`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).

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
```


Finally 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 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-uxsup.csx.cam.ac.uk/courses/Commands/

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

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

`man xargs`

Final exercise

In your home directory is a program called **`lissajous.py`**, which produces points on a Lissajous curve that it prints to standard output. **`lissajous.py`** takes two floating point command line arguments, although we'll restrict ourselves to using only integer arguments for it.

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

Write a shell script that will read the first parameter for **`lissajous.py`** from standard input and the second parameter from the command line. It should run the **`lissajous.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:

12

23

…and the values ***5 9 32*** from the command line, then your script should run:

```
./lissajous.py 12 5
./lissajous.py 12 9
./lissajous.py 12 32
./lissajous.py 23 5
./lissajous.py 23 9
./lissajous.py 23 32
```


Please read this BEFORE you start on this exercise!

The point of this exercise is to consolidate everything you've learnt over all three days of this course. To that end I want you to write your own shell script **FROM SCRATCH** to do this exercise – do *not* just take one of the ones we've constructed over this course and change 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, I want your shell script to be ***as good a shell script as you can possibly make it*** – 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 `lissajous_params` that you can use as a source of parameters to `read` via standard input. I suggest that for the command line arguments you use:

2 4 6 8

The files you need to do this exercise are available on-line at:

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

Final exercise – Files

All the files (scripts, the **`lissajous.py`** and **`iterator`** programs, etc) used in this course are available on-line at:

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

Advanced Techniques

The following slides outline some more advanced shell scripting techniques that, whilst beyond the scope of this course, may be of interest.

Advanced techniques: **`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`

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}`

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

An example of its use is given in the file `shift.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`