1 Linux for Bioinformatics

1.1 Introduction

Unix and Unix-like operating systems are the standard operating system on most large computer systems in scientific research, in the same way that Microsoft Windows is the dominant operating system on desktop PCs. In this course we will use Linux, a Unix-like operating system which was originally created to provide a free open source operating system for PCs.

Linux and MS Windows both perform the important job of managing the computer's hardware (screen, keyboard, mouse, hard disks, network connections, etc…) on your behalf. They also provide you with tools to manage your files and to run application software. They both offer a graphical user interface (desktop). These desktop interfaces look different between the operating systems, use different names for things (e.g. directory versus folder) and have different images but they mostly offer the same functionality.

Linux is a powerful, secure, robust and stable operating system which allows dozens of people to run programs on the same computer at the same time. This is why it is the preferred operating system for large-scale scientific computing. It runs on all kinds of machines, from mobile phones (Android), desktop PCs… to supercomputers.

1.1.1 Why Linux?

Increasingly, the output of biological research exists as in silico data, usually in the form of large text files. Linux is particularly suitable for working with such files and has several powerful and flexible commands that can be used to process and analyse this data. One advantage of learning Linux is that many of the commands can be combined in an almost unlimited fashion. So if you can learn just six Linux commands, you will be able to do a lot more than just six things.

Linux contains hundreds of commands, but to conduct your analysis you will probably only need 10 or so to achieve most of what you want to do. In this tutorial we will introduce you to some useful Linux commands and provide examples of how they can be used in bioinformatics analyses.

1.2 Learning outcomes

By the end of the tutorial you can expect to be able to:

- Understand the Linux directory structure and navigate around this structure
- Extract information from large files
- Use regular expressions to search for particular patterns in a file
- Create a bash script to perform several tasks at once

1.3 Tutorial sections

This tutorial comprises the following sections:

1. Basic Linux
2. Commands grep and awk
3. Advanced Linux (loops and Bash scripts)
4. Bash scripting

Note: We do not expect you to get through all the material in the time allocated and a good target to aim for is the end of section 3 Advanced Linux (loops and Bash scripts). The remaining sections are optional and are for students who would like to expand their Linux skills and can be completed outside the course hours.

1.4 Authors and License

This tutorial was created by Jacqui Keane and Martin Hunt.

1.5 Running the commands in this tutorial

You can follow this tutorial by typing all the commands you see in a terminal window on your computer. This is similar to the "Command Prompt" window on MS Windows systems, which allows the user to type DOS commands to manage files.

To get started, open a terminal window and type the command below followed by the `Enter` key:

```
cd ~/course_data/linux/data
```

Now you can follow the instructions in the tutorial from here.

1.6 Cheat sheet

We've also included a cheat sheet at the end of this tutorial. It probably won't make a lot of sense now, but it might be a useful reminder of this module later in the tutorial.

1.7 Let's get started!

To get started with the tutorial, go to the next section: Basic Linux

2 Basic Linux

2.1 The Commandline

The commandline or 'terminal' is an interface you can use to run programs and analyse your data. If this is your first time using one it will seem pretty daunting at first but, with just a few commands, you'll start to see how it helps you to get things done more efficiently. You're probably more familiar with software which uses a graphical user interface, also known as a GUI.

2.2 Getting started

Let's check that you're in the right place. Type the command below in the terminal window followed by the `Enter` key:

```
pwd
```

It should display something similar to:

`/home/manager/course_data/linux/data`

Then continue through the course, entering any commands that you encounter (highlighted in a grey box with a keyboard symbol) into your terminal window. Let's start by listing the contents of the current directory:

```
ls
```

Before getting started there are some general points to remember that will make your life easier:

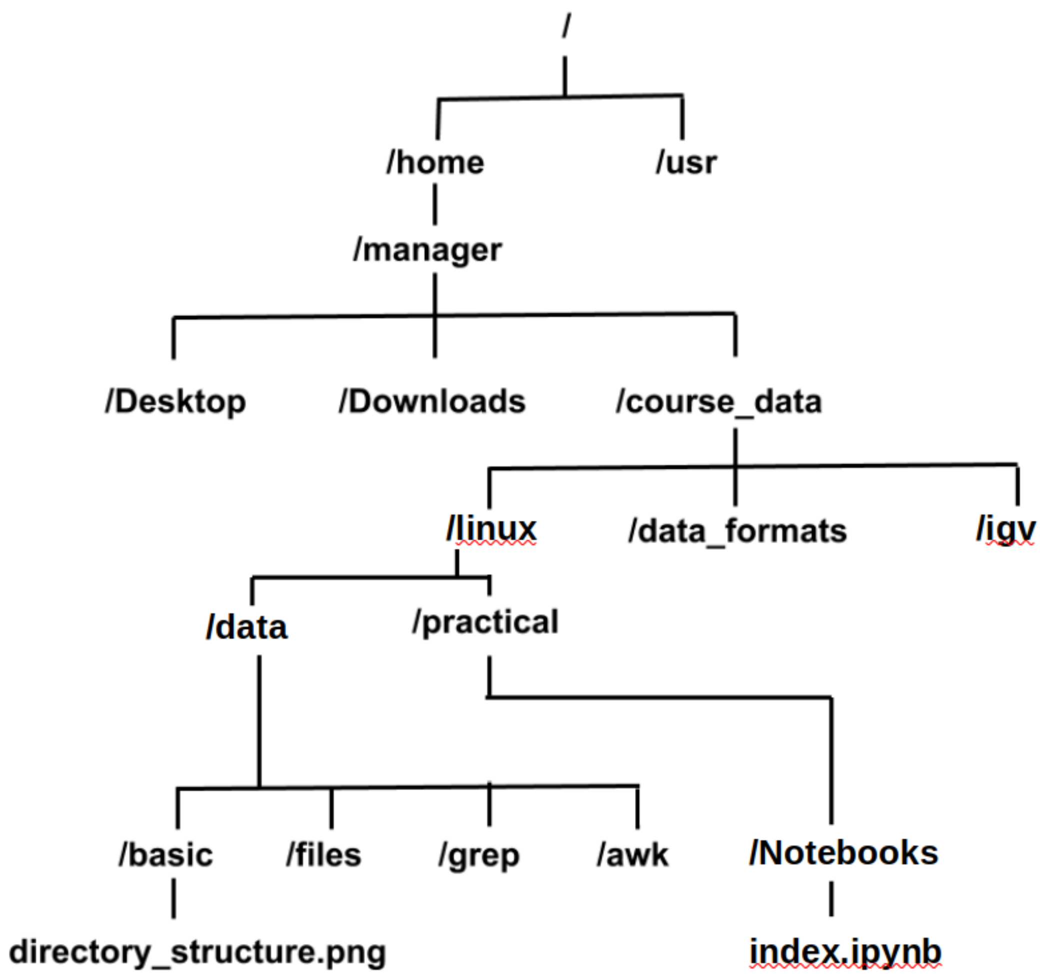
- Linux is case sensitive - typing `ls` is not the same as typing `LS`.
- Often when you have problems with Linux, it is due to a spelling mistake. Check that you have not missed or added a space. Pay careful attention when typing commands across a couple of lines.

2.3 Files and directories

Directories are the Linux equivalent of folders on a PC or Mac. They are organised in a hierarchy, so directories can have sub-directories and so on. Directories are very useful for organising your work and keeping your account tidy - for example, if you have more than one project, you can organise the files for each project into different directories to keep them separate. You can think of directories as rooms in a house. You can only be in one room (directory) at a time. When you are in a room you can see everything in that room easily. To see things in other rooms, you have to go to the appropriate door and crane your head around. Linux works in a similar manner, moving from directory to directory to access files. The location or directory that you are in is referred to as the current working directory.

For the file called `index.ipynb` under the `linux` directory, the location or *absolute path* can be expressed as:

```
ls /home/manager/course_data/linux/practical/Notebooks/index.ipynb
```



2.4 tree - display the directory hierarchy

The command `tree` can be used to recursively list or display the content of a directory in a tree-like format. It outputs the directory paths and files in each sub-directory and a summary of a total number of sub-directories and files.

To display the contents of the current directory, type:

```
tree
```

2.5 cd - change directory

The command `cd` stands for change directory. The `cd` command will move you from the current working directory to another directory.

To move into the `basic` directory type the following command. Note, you'll remember this more easily if you type this rather than copying and pasting.

```
cd basic
```

2.6 ls - list the contents of a directory

The command `ls` stands for list. The `ls` command can be used to list the contents of a directory.

To list the contents of your current working directory type:

```
ls
```

You should see that there are 4 items in this directory.

To list all contents of a directory including hidden files and directories (hidden files and directories are not shown by default and are used to help prevent important data from being deleted) type:

```
ls -a -l
```

This is an example of a command which can take multiple options at the same time.

How many hidden files and directories are there?

Try the same command but with the `-h` option:

```
ls -alh
```

You'll also notice that we've combined `-a -l -h` into what appears to be a single `-alh` option. It's almost always ok to do this for options which are made up of a single dash followed by a single letter.

What does the `-h` option do?

To list the contents of the directory called Pfalciparum with extra information type:

```
ls -l Styphi/
```

Here we gave the command `ls` the *option* `-l` and the *argument* `Styphi`. Arguments are very similar to options, they are used to provide additional information to a command and often refer to things which are not prefixed with dashes.

In this example we have provided information about the directory that we want to list the contents of, this argument `Styphi` is the *relative path* to the directory `Styphi` from our current working directory.

How many files are there in this directory?

2.7 Getting help man

It is not possible to remember all the options and arguments for all Linux commands! A useful command to obtain further information on any of Linux command is the `man` command. For example, to get a full description and examples of how to use the `find` command type the following command in a terminal window.

```
man find
```

2.8 find - find a file

The `find` command can be used to find files matching a given pattern. It can be used to recursively search the directory tree for a specified name, seeking files and directories that match the given name.

To find all files in the current directory and all its subdirectories that end with the suffix gff:

```
find . -name *.gff
```

How many gff files did you find?

To find all the subdirectories contained in the current directory type:

```
find . -type d
```

How many subdirectories did you find?

2.9 Tab completion

Typing out directory or file names is really boring and you're likely to make typos which will at best make your command fail with a strange error and at worst overwrite some of your carefully crafted analysis. *Tab completion* is a trick which normally reduces this risk significantly.

Instead of typing out `ls Styphi/`, try typing `ls S` and then press the `tab` key (instead of `Enter`). The rest of the folder name should just appear. If you have two files of directories with simiar names (e.g. `directory_structure.png` and `directory_structure2.png`) then you might need to give your terminal a bit of a hand to work out which one you want. In this case you would type `ls -l d`, when you press `tab` the terminal would read `ls -l directory_structure`, you could then type 2 followed by another `tab` and it would work out that you meant `directory_structure2.png`

2.10 Tips

There are some short cuts for referring to directories:

- . Current directory (one full stop)
- .. Directory above in the hierarchy (two full stops)
- ~ Home directory (tilde)
- / Root of the file system (like C: in Windows)

Try the following commands, what do they do?

```
ls .
```

```
ls ..
```

```
ls ~
```

2.11 Exercises

Many people panic when they are confronted with a Linux prompt! Don't! All the commands you need to solve these exercises are provided above and don't be afraid to make a mistake. If you get lost ask an instructor. If you are a person skilled at Linux, be patient this is only a short exercise.

To begin, open a terminal window and navigate to the `basic` directory under the the `linux` directory (remember use the command `cd`) and then complete the exercise below.

1. Use the `ls` command to show the contents of the `basic` directory.
2. How many files are there in the `Pfalciparum` directory?
3. What is the largest file in the `Pfalciparum` directory?
4. Move into the `Pfalciparum` directory.
5. How many files are there in the `fasta` directory?
6. Copy the file `Pfalciparum.bed` in the `Pfalciparum` directory into the `annotation` directory.
7. Move all the fasta files in the directory `Pfalciparum` to the `fasta` directory.
8. How many files are there in the `fasta` directory?
9. Use the `find` command to find all gff files in the `linux` directory, how many files did you find?
10. Use the `find` command to find all the fasta files in the `linux` directory, how many files did you find?

When you have completed these exercises move on to the next part of the tutorial, grep and awk.

3 The commands `grep` and `awk`

In this section we will introduce the commands `grep` and `awk`.

3.1 Searching inside files with `grep`

A common task is to extract information from large files. This can be achieved using the Linux command `grep`, which stands for "Globally search for a Regular Expression and Print". The meaning of this acronym will become clear later, when we discuss Regular Expressions. First, we will consider simpler examples.

Before we start, change into the `grep` directory:

```
cd ../grep
```

3.1.1 Simple pattern matching

We will use a small example file (in "BED" format), which contains information about gene features in a genome. A bed file is a column-based file, each column contains information about the feature and is seperated by a tab character. There can be more than 10 columns, but only the first three are required to be a valid file. The file format is described in full here: http://genome.ucsc.edu/FAQ/FAQformat#format1. We will use the first 5 columns:

1. Sequence name
2. start position (starting from 0, not 1)
3. end position (starting from 0, not 1)
4. feature name
5. score (which is used to store the gene expression level in our examples).

Here is the contents of the first example BED file used in this section:

```
cat gene_expression.bed
```

In reality, such a file could contain 100,000s of lines, so that it is not practical to read manually. Suppose we are interested in all the genes from chromosome 2. We can find all these lines using grep:

```
grep chr2 gene_expression.bed
```

This has shown us all the lines that contain the text or string "chr2".

We can use a pipe to then just extract the genes that are on the positive strand, using grep a second time:

```
grep chr2 gene_expression.bed | grep +
```

However, since `grep` is reporting a match to a string *anywhere* on a line, such simple searches can have undesired consequences. For example, consider the result of doing a similar search for all the genes in chromosome 1:

```
grep chr1 gene_expression.bed
```

Oops! We found genes in chromosome 10, because "chr1" is a substring (subset) of "chr10".

Or consider the following file, where the genes have unpredictable names (which is not unusual for bioinformatics data).

```
cat gene_expression_sneaky.bed
```

Now we try to find genes on chromosome 1 that are on the negative strand. We put the minus sign in quotes, to stop Linux interpreting this as an option to `grep`, as opposed to the string we are searching for:

```
grep chr1 gene_expression_sneaky.bed | grep '-'
```

The extra lines are found by `grep` because of matches in columns we were not expecting to match. Remember, `grep` is reporting these lines because they each contain the strings "chr1" and "-" *somewhere*.

We need a way to make searching with `grep` more specific.

3.1.2 Regular expressions

Regular expressions provide the solution to the above problems. They are ways of defining more specific patterns to search for.

Matching the start and end of lines First, we can specify that a match must be at the start of a line using the symbol "^", which means "start of line". Without the ^, we find any match to "chr1":

```
grep chr1 gene_expression_sneaky.bed
```

```
grep '^chr1' gene_expression_sneaky.bed
```

Good! We have removed the match to the badly-named gene "chr11.gene1", which is on chromosome 8. Now we want to avoid matching chromosomes 10 and 11. This can be done by also looking for a "tab" character, which is represented by writing `\t`. For technical reasons, which are beyond the scope of this course, we must also put a dollar sign before the quotes to make any search involving a tab character work.

```
grep $'^chr1\t' gene_expression_sneaky.bed
```

To find the genes on the negative strand, all that remains is to match a minus sign at the *end* of the line (so that we do not find "sneaky-gene3"). We can do this using the dollar "$", which means "end of line".

```
grep $'^chr1\t' gene_expression_sneaky.bed | grep '\-$'
```

Wildcards and alphabets Another special character in regular expressions is the dot: ".". This stands for any single character. For example, this finds all matches to chromosomes 1-9, and chromosomes X and Y:

```
grep $'^chr.\t' gene_expression.bed
```

In fact, the earlier command that found all genes on chromosome 1 that are on the negative strand, could be found with a single call to grep instead of two calls piped together. To do this, we need a regular expression that finds lines that:

- start with chr1, then a tab character
- end with a minus
- have arbitrary characters between.

The asterisk "*" has a special meaning: it says to match any number (including zero) of whatever character is before the *. For example, the regular expression 'AC*G' will match AG, ACG, ACCG, etc. The simpler, improved command is:

```
grep $'^chr1\t.*-$' gene_expression_sneaky.bed
```

As well as matching any character using a dot, we can define any list of characters to match, using square brackets. For example, [12X] means match a 1, 2, or an X. This can be used to find all genes from chromosomes 1, 2 and X:

```
grep $'^chr[12X]\t' gene_expression.bed
```

Or just the autosomes may be of interest. To do this we introduce two new features:

- Ranges can be given in square brackets, for example [1-5] will match 1, 2, 3, 4 or 5.
- The plus sign "+" has a special meaning that is similar to "*". Instead of any number of matches (including zero), it looks for at least one match. To avoid simply matching a plus sign, it must be preceded by a backslash: "\+". For example, the regular expression 'AC\+G' will match ACG, ACCG, ACCCG etc (but will not match AG).

Warning: Adding a backslash is often called *escaping* (e.g. *escape the plus symbol*). Depending on the software you're using (and the options you give it), you may need to escape the symbol to indicate that you want its special regex meaning (e.g. multiple copies of the last character please) or its literal meaning (e.g. give me a '+' symbol please). If your command isn't working as you expect, try playing with these options and always test your regular expression before assuming it gave you the right answer.

The command to find the autosomes is:

```
grep $'^chr[0-9]\+\t' gene_expression.bed
```

3.1.3 Other `grep` options

The Linux command grep and regular expressions are extremely powerful and we have only scratched the surface of what they can do. Take a look at the manual (by typing `man grep`) to get an idea. A few particularly useful options are discussed below.

```
man grep
```

Counting matches A common scenario is counting matches within files. Instead of output each matching line, the option "-c" tells grep to report the number of lines that matched. For example, the number of genes in the autosomes in the above example can be found by simply adding -c to the command.

```
grep -c $'^chr[0-9]\+\t' gene_expression.bed
```

Case sensitivity By default, grep is case-sensitive. It can be useful to ignore the distinction between upper and lower case using the option "-i". Suppose we have a file of sequences, and want to find the sequences that contain the string ACGT. It is not unusual to come across files that have a mix of upper and lower case nucleotides. Consider this FASTA file:

```
cat sequences.fasta
```

A simple search for ACGT will not return all the results:

```
grep ACGT sequences.fasta
```

However, making the search case-insensitive solves the problem.

```
grep -i ACGT sequences.fasta
```

Inverting matches By default, grep reports all lines that do match the regular expression. Sometimes it is useful to filter a file, by reporting lines that *do not* match the regular expression. Using the option "-v" makes grep "invert" the output. For example, we could exclude genes from autosomes in the BED file from earlier.

```
grep -v $'^chr[0-9]\+\t' gene_expression.bed
```

3.1.4 Replacing matches to regular expressions

Finally, we show how to replace every match to a regular expression with something else, using the command "sed". The general form of this is:

```
sed 's/regular expression/new string/' input_file
```

This will output a new version of the input file, with each match to the regular expression replaced with "new string". For example, try:

```
sed 's/^chr/chromosome/' gene_expression.bed
```

This will replace every occurence of the text "chr" that appears at the start of a line in the file gene_expression.bed with the text "chromosome".

3.1.5 Exercises

The following exercises all use the FASTA file `exercises.fasta`. Before starting the exercises, open a new terminal and navigate to the `linux/data/grep` directory, which contains `exercises.fasta`.

Use `grep` to find the answers. Hint: some questions require you to use `grep` twice, and possibly some other Linux commands.

1. Make a `grep` command that outputs just the lines with the sequence names.
2. How many sequences are in the file?
3. Do any sequence names have spaces in them? What are their names?
4. Make a `grep` command that outputs just the lines with the sequences, not the names.
5. How many sequences contain unknown bases (an "n" or "N")?
6. Are there any sequences that contain non-nucleotides (something other than A, C, G, T or N)?
7. How many sequences contain the 5' cut site GCWGC (where W can be an A or T) for the restriction enzyme AceI?
8. Are there any sequences that have the same name? You do not need to find the actual repeated names, just whether any names are repeated. (Hint: it may be easier to first discover how many unique names there are).

3.2 File processing with AWK

AWK is a programming language named after the initials of its three inventors: Alfred **A**ho, Peter **W**einberger, and Brian **K**ernighan. AWK is incredibly powerful at processing files, particularly column-based files, which are commonplace in Bioinformatics. For example, BED, GFF, and SAM files.

Although long programs, put into a separate file, can be written using AWK, we will use it directly on the command line. Effectively, these are very short AWK programs, often called "one-liners".

Before we start, change into the `awk` directory:

```
cd ../awk
```

3.2.1 Extracting columns from files

The command `awk` reads a file line-by-line, splitting each line into columns. This makes it easy to do simple things like extract a column from a file. We will use the following GFF file for our examples.

```
cat genes.gff
```

A GFF file is similar to a BED file in that it contains information about gene features in a genome. It is a column-based file, each column contains information about the feature. The columns in the GFF file are separated by tabs and each column has the following meaning:

1. Sequence name
2. Source - the name of the program that made the feature
3. Feature - the type of feature, for example gene or CDS
4. Start position
5. Stop position

6. Score
7. Strand (+ or -)
8. Frame (0, 1, or 2)
9. Optional extra information, in the form key1=value1;key2=value2;...

The score, strand, and frame can be set to '.' if it is not relevant for that feature. The final column 9 may or may not be present and could contain any number of key, value pairs.

We can use `awk` to just print the first column of the file. `awk` calls the columns `$1`, `$2`, ... etc, and the complete line is called `$0`. Try

```
awk -F"\t" '{print $1}' genes.gff
```

A little explanation is needed.

- The option `-F"\t"` was needed to tell `awk` that the columns are separated by tabs (more on this later).
- For each line of the file, `awk` does what is inside the curly brackets. In this case, we simply print the first column ($1).

The repeated chromosome names are not nice. It is more likely to want to know just the unique names, which can be found by piping into the Linux command `sort -u`.

```
awk -F"\t" '{print $1}' genes.gff | sort -u
```

In this command the `|` symbol is known as the *pipe* symbol. This *pipes* (sends) the output of the `awk` command into the input of `sort -u`. The `sort` will sort the contents of the input. When you sort the input, lines with identical content end up next to each other in the output. The `-u` option of the sort command will select only the unique values in the output.

You can connect as many commands as you want. For example, type:

```
awk -F"\t" '{print $1}' genes.gff | sort -u | wc -l
```

This will count the number of lines in the output.

3.2.2 Filtering the input file

Similarly to `grep`, `awk` can be used to filter out lines of a file. However, since awk is column-based, it makes it easy to filter based on properties of any columns of interest. The filtering criteria can be added before the braces. For example, the following extracts just chromosome 1 from the file.

```
awk -F"\t" '$1=="chr1" {print $0}' genes.gff
```

There are two important things to note from the above command:

1. `$1=="chr1"` means that column 1 must be *exactly* equal to "chr1". This means that "chr10" is not found.
2. The "`{print $0}`" part only happens when the first column is equal to "chr1", otherwise `awk` does nothing (the line gets ignored).

Awk commands are made up of two parts, a *pattern* (e.g. `$1=="chr1"`) and an *action* (e.g. `print $0`) which is contained in curly braces. The *pattern* defines which lines the *action* is applied to.

In fact, the action (the part in curly braces) can be omitted in this example. `awk` assumes that you want to print the whole line, unless it is told otherwise. This gives a simple method of filtering based on columns.

3.2.3 Exercises

The following exercises all use the BED file `exercises.bed`. Before starting the exercises, open a new terminal and navigate to the `awk` directory, which contains `exercises.bed`.

Use `awk` to find the answers to the following questions about the file `exercises.bed`. Many questions will require using pipes (eg "`awk ... | sort -u`" for question 1).

1. What are the names of the contigs in the file?
2. How many contigs are there?
3. How many features are on the positive strand?
4. How many features are on the negative strand?
5. How many repeat features are there?

When you have completed these exercises move on to the next part of the tutorial, Advanced Linux.

4 Advanced Linux

So far, we have run single commands in a terminal. Now we will look at ways to run more than a single command and how to automate tasks.

Before you start this section change into the `advanced_linux` directory:

```
cd ../advanced_linux
```

4.1 Repeating analysis with loops

It is common in Bioinformatics to run the same analysis on many files. Suppose we had a command that ran one type of analysis, and we wanted to repeat the same analysis on 100 different files. It would be tedious, and error-prone, to write the same command 100 times. Instead we can use a loop.

As an example, say we wanted to run `wc` on each file in the directory `loop_files`.

First let's look at the contents of the `loop_files` directory:

```
ls loop_files/
```

To run `wc` on each of the files found in the directory `loop_files/` use the following command:

```
for filename in loop_files/*; do wc $filename; done
```

However, it is useful to be able to run multiple commands that process some data and produce some output. These commands can be put into a file (i.e. a script), and run on input data. This has the advantage of saving time and reproducibility, so that the same analysis can be run on many input data sets.

4.2 Introduction to BASH scripting

It is traditional when learning a new language (in this case BASH), to write a script that says "Hello World!". Open a terminal and make a new directory in your home called `scripts`, by typing

```
cd ~
mkdir ~/scripts
```

Next open a text editor, which you will use to write the script. What text editors are available will depend on your system. For example, gedit in Linux. Do not try to use a word processor, such as Word! If you don't already have a favorite, try nano by running the following command:

```
nano scripts/hello.sh
```

Type this into the text editor:

```
echo Hello World!
```

and save this to a file called `hello.sh` in your new `scripts` directory. This script will print `Hello World!` to the screen when we run it. First, check that the script is saved in the correct place.

```
ls scripts/hello.sh
```

Now try to run the script. For now, we need to tell Linux that this is a BASH script and where it is (inside the scripts directory):

```
bash scripts/hello.sh
```

4.3 Setting up a scripts directory

It would be nice if our scripts could be run from anywhere in the filesystem, without having to tell Linux where the script is, or that it is a BASH script. This is how built-in commands work, like `cd` or `ls`.

To tell Linux that the script is a BASH script, edit the file and add this line as the first line of the script:

```
#!/usr/bin/env bash
```

and remember to save the script again. This special line at the start of the file tells Linux that the file is a bash script, so that it expects bash commands throughout the file. There is one more change to be made to the file to tell Linux that it is a program to be run (it is "executable"). This is done with the command `chmod`. Type this into the terminal to make the file executable:

```
chmod +x scripts/hello.sh
```

Now, the script can be run, but we must still tell Linux where the script is in the filesystem. In this case, it is in a directory called scripts in the current working directory, "`./scripts`".

```
./scripts/hello.sh
```

We need to change our setup so that Linux can find the script without us having to explicitly say where it is. Whenever a command is typed into Linux, it has a list of directories that it searches through to look for the command. To see the list of directories type:

```
echo $PATH
```

It returns a list of directories, which are all the places Linux will look for a command. First, check what happens if we try to run the script without telling Linux where it is:

```
hello.sh
bash: hello.sh: command not found
```

Linux did not find it! The command to run to add the scripts directory to `$PATH` is:

```
export PATH=$PATH:~/scripts/
```

If you want this change to be permanent, ie so that Linux finds your scripts after you restart your computer or logout and login, add that line to the end of a file called `~/.bashrc`. If you are using a Mac, then the file should instead be `~/.bash_profile`. If the file does not already exist, then create it and put that line into it.

Now the script works, no matter where we are in the filesystem. Linux will check the scripts directory and find the file `hello.sh`. You can be *anywhere* in your filesystem, and simply running

```
hello.sh
```

will always work. Try it now.

```
hello.sh
```

In general, when making a new script, you can now copy and edit an existing script, or make a new one like this:

```
cd ~/scripts
touch my_new_script.sh
chmod +x my_new_script.sh
```

and then open `my_new_script.sh` in a text editor.

If you would like to learn more advanced bash scripting we have provided some further optional material in BASH Scripting.

5 BASH scripting

This section is additional material and provided for anyone who would like to learn more advanced bash scripting.

Before you start this section change into the `bash_scripts` directory:

```
cd ../bash_scripts
```

5.1 Getting options from the terminal and printing a help message

Usually, we would like a script to read in options from the user, such as the name of an input file. This would mean a script can be run like this:

```
my_script.sh input_file
```

Inside the script, the parameters provided by the user are given the names `$1`, `$2`, `$3` etc (do not confuse these with column names used by `awk`!). Here is a simple example that expects the user to provide a filename and a number. The script simply prints the filename to the screen, and then the first few lines of the file (the number of lines is determined by the number given by the user).

```
cat options_example.sh
```

```
options_example.sh test_file 2
```

The options have been used by the script, but the script itself is not very readable. It is better to use names instead of `$1` and `$2`. Here is an improved version of the script that does exactly the same as the previous script, but is more readable.

```
cat options_example.2.sh
```

5.2 Checking options from the user

The previous scripts will have strange behaviour if the input is not as expected by the script. Many things could go wrong. For example:

- The wrong number of options are given by the user
- The input file does not exist.

Try running the script with different options and see what happens.

A convention with scripts is that it should output a help message if it is not run correctly. This shows anyone how the script should be run (including you!) without having to look at the code inside the script.

A basic check for this script would be to verify that two options were supplied, and if not then print a help message. The code looks like this:

```
if [ $# -ne 2 ]
then
        echo "usage: options_example.3.sh filename number_of_lines"
```

```
        echo
        echo "Prints the filename, and the given first number of lines of the file"
        exit
fi
```

You can copy this code into the start of any of your scripts, and easily modify it to work for that script. A little explanation:

- A special variable `$#` has been used, which is the number of options that were given by the user.
- The whole block of code has the form "`if [ $# -ne 2 ] then .... fi`". This only runs the code between the `then` and `fi`, if `$#` (the number of options) is not 2.
- The line `exit` simply makes the script end, so that no more code is run.

```
options_example.3.sh
```

Another check is that the input file really does exist. If it does not exist, then there is no point in trying to run any more code. This can be checked with another `if ... then ... fi` block of code:

```
if [ ! -f $filename ]

then
    echo "File '$filename' not found! Cannot continue"
    exit
fi
```

Putting this all together, the script now looks like this:

```
cat options_example.3.sh
```

Two new features have also been introduced in this file:

1. The second line is "`set -eu`". Without this line, if any line produces an error, the script will carry on regardless to the end of the script. Using the `-e` option, an error anywhere in the file will result in the script stopping at the line that produced the error, instead of continuing. In general, it is best that the script stops at any error. The `-u` creates an error if you try to use a variable which doesn't exist. This helps to stop typos doing bad things to your analysis.
2. There are several lines starting with a hash `#`. These lines are "comment lines" that are not run. They are used to document the code, containing explanations of what is happening. It is good practice to comment your scripts!

The above script provides a template for writing your own scripts. The general method is:

1. Tell Linux that this is a BASH script, and to stop at the first error.
2. Check if the user ran the script correctly. If not, output a message telling the user how to run the script.
3. Check the input looks OK (in this case, that the input file exists).
4. Process the input.

5.3 Using variables to store output from commands

It can be useful to run a command and put the results into a variable. Recall that we stored the input from the user in sensibly named variables:

```
filename=$1
```

The part after the equals sign could actually be any command that returns some output. For example, running this in Linux

```
wc -l filename | awk '{print $1}'
```

returns the number of lines. In case you are wondering why the command includes `| awk '{print $1}'`, check what happens with and without the pipe to `awk`:

```
wc -l options_example.3.sh
```

```
wc -l options_example.3.sh | awk '{print $1}'
```

With a small change, this can be stored in a variable and then used later.

```
filename=options_example.3.sh
line_count=$(wc -l $filename | awk '{print $1}')
echo There are $line_count lines in the file $filename
```

5.4 Exercises

1. Write a script that gets a filename from the user. If the file exists, it prints a nice human-readable message telling the user how many lines are in the file.
2. Use a loop to run the script from Exercise 1 on the files in the directory `loop_files/`.
3. Write a script that takes a GFF filename as input. Make the script produce a summary of various properties of the file. There is an example input file provided called `bash_scripts/exercise_3.gff`. Use your imagination! You could have a look back at the `awk` section of the course for inspiration. Here are some ideas you may wish to try:

- Does the file exist?
- How many records (ie lines) are in the file?
- How many genes are in the file?
- Is the file badly formatted in any way (eg wrong number of columns, do the coordinates look like numbers)?