Contents

Introducing The Command Line

Primary Author(s):
Stephen Bent, Robinson Research Institute, University of Adelaide
Steve Pederson, Bioinformatics Hub, University of Adelaide
stephen.pederson@adelaide.edu.au

Contributor(s):
Dan Kortschak, Adelson Research Group, University of Adelaide
dan.kortschak@adelaide.edu.au

Initial Goals

1. Gain familiarity and confidence within the Linux command-line environment
2. Learn how to navigate directories, as well as to copy, move & delete the files within them
3. Look up the name of a command needed to perform a specified task

Background

Command-line tools are the mainstay of analysis of large biological data sets. Good candidate examples for command-line analysis are:

- Manual inspection of fastq, bam & sam files from NGS pipelines
- Automating similar analyses across multiple datasets
- Manipulations of data which are repetitive or laborious to perform manually
- Any analysis that is different from what is available in programs with graphical interfaces (i.e. GUIs).
- Job submission to HPC clusters

Recording all processes as simple scripts also makes an entire analysis more reproducible, as you will have *a record of every procedure you have performed*. It's amazing how often you'll need to revisit something you have performed months ago, and having records of what you've done is immensely useful. From the perspective of an "electronic lab book", this is also very important.

Today we'll explore a few commands to help you gain a little familiarity with some important ones, and to enable you to find help when you're working by yourself. We don't expect you to remember all the commands & options from today. The important thing is to become familiar with the basic syntax for commands, how to put them together, and where to look for help when you're unsure.

Finding your way around

Firstly we need to open a terminal, so **click on the terminal icon** at the bottom of the desktop. There are several variants of the shell, such as the *Bourne-again* shell (i.e. `bash`) and the *C-shell* (please feel free to make all the jokes you can think of). The terminal has a library of commands which are built into it at the time of installing the operating system, and which are part of the Bourne-again Shell, or *bash*. (Historically, it's a replacement for the earlier Bourne Shell, written by Stephen Bourne, so the name is actually a hilarious pun.) We'll explore a few of these commands below, and the words shell or bash will often be used interchangeably with the terminal window. Our apologies to any purists. If you've ever heard of the phrase *shell scripts*, this refers to a series of these commands strung together into a text file which is then able to be executed as a single command.

The tilde (~) represents a shorthand for your home directory, but we'll explore this more later.

Where are we?

When navigating through the folders on any computer, we are all very familiar with clicking through from one folder to another. The folder currently being displayed is usually given in the header of the folder view, so we know where we are currently looking. When using `bash`, we don't click on anything so we need to type commands to change directories, and inspect the contents.

The first thing we need to do is find where we are, when we open `bash`. Type the command `pwd` in the terminal and you will see the output `/home/hub` appear.

```
1 pwd
```

The command `pwd` is what we use for **p**rinting the current (i.e. **w**orking) **d**irectory. Printing in this context means to print some information to the terminal, as opposed to a physical printer. This style of printing harks back to the days when laser printers were not commonplace. Printing information to the terminal itself is what we refer to as printing to the *standard output* or `stdout`.

The directory path that appeared (`/home/hub`) is what will be referred to as your *home directory* for the remainder of the workshop. This is also the information that the tilde (~) represents as a shorthand version, so whenever you see the tilde in a directory path, this is interpreted as meaning `/home/hub`. The *working directory* simply refers to the directory on the computer where you are currently looking, and can be thought of as being the room you are in. We are very familiar with this concept graphically but instead of using a graphical view, we simply have a text-based view.

Looking at the Contents of a Directory

There is another built-in command "`ls`" that we can use to **l**i**s**t the contents of a directory. Enter the `ls` command as it is and it will list the contents of the current directory.

```
1 ls
```

Now open your home folder using the icon at the bottom to give the more familiar, graphical view and compare the contents.

The Linux File System

In the above command, the home directory began with a slash, i.e. `/`. On a Linux-based system, this is considered to be the **root directory** of the file system. Windows users would be more familiar with seeing `C:\` as the root of the file system, and this is a very important difference in the two directory structures. Note also that whilst Windows uses the backslash (\) to indicate a new directory, a Linux-based system uses the forward slash (/), or more commonly just referred to simply as "slash", marking another but very important difference between the two.

Although we haven't directly discovered it yet, a Linux-based file system such as Ubuntu or Mac OS-X is also *case-sensitive*, whilst Windows is not. For example, the command `PWD` is completely different to `pwd` and if `PWD` is the name of a command which has been defined in your shell, you will get completely different results than from the intended `pwd` command.

Spaces are also highly important in Linux, so take note of them where-ever they appear in the given commands.

Changing Directories

The command `pwd` is an example of a *command* that is built into the shell. Another built-in command is `cd` which we use to **c**hange **d**irectory. This changes the directory we are looking in, just like clicking our way through a directory structure in the familiar graphical style we all know well. No matter where we are in a file system, we can move up a directory in the hierarchy by using the command

```
1 cd ..
```

The string "`..`" is the convention for "*one directory above*", whilst a single dot represents the current directory.

Enter the above command and notice that the location immediately to the left of the $ is now given as `/home`. This is also what will be given as the output if we enter the command `pwd`. Note that this given directory is because your personal home folder is within the folder `/home` which contains all of the home folders for all users on the computer. If we now enter `cd ..` one more time we will be in the root directory of the file system. Try this and print the working directory again. As detailed earlier, the output should be the root directory given as `/`.

We can change back to your personal home folder by entering one of either:

```
1 cd /home/hub
```

or

```
1 cd ~
```

or even just

```
1 cd
```

We can also move through multiple directories in one command by separating them with the forward slash "`/`". For example, we could also get to the root directory from our home directory by typing

```
1 cd ../../
```

Using the above process, return to your home directory `/home/hub`.

Viewing Directories Without Changing Where We Are

Alternatively, we can specify which directory we wish to view the contents of, without having to change into that directory. We simply type the `ls` command, followed by a space, then the directory we wish to view the contents of. To look at the contents of the root directory of the file system (i.e. `/`), we simply add that directory after the command `ls`.

```
1 ls /
```

Here you can see a whole raft of directories which contain the vital information for the computer's operating system. Among them should be the `/home` directory which is one level above your own home directory, and where the home directories for **all users** are

located, as mentioned earlier.

Try to think of two ways we could inspect the contents of the `/home` directory from your own home directory.

Notice that there are more entries in the `/home` directory. One is your home directory (`/home/hub`), whilst the other is the default home directory that came with the VM (`/home/ubuntu`) before we set up your home directory for today's workshop. Also note that if we ever refer to this higher-level directory, we will write it as `/home`, whereas your personal home directory is always referred to without the preceding forward-slash, and will also be written in normal text font instead of the font we specifically use for code.

Relative vs Absolute Paths

Also note that in your output from `pwd` the resulting path started with the slash, i.e. `/home/hub`. This indicates that it is an **absolute path** as it began with the root of the file system "/". If the slash was missing, it would refer to a sub-directory of the current working directory, and this is what we refer to as a **relative path**. This is an important point which will hopefully become more clear throughout the session.

Another simple example, is that from your home folder (`/home/hub`), the folder `Documents` can be called just using `Documents`. If you are in another folder, you'd probably have to use the full (or absolute) path, which is `/home/hub/Documents`.

A Handy Hint

When working in the terminal, you can scroll through your previous commands by using the up arrow to go backward, and the down arrow to move forward. This can be a big time saver if you've typed a long command with a simple typo, or if you have to do a series of similar commands.

Going Even Deeper

So far, the commands we have used were given either without the use of any subsequent arguments, e.g. `pwd` & `ls`, or with a specific directory as the second argument, e.g. `cd ../` & `ls /home`. Many commands have the additional capacity to specify different options as to how they perform, and these options are often specified between the command name, and the file being operated on. Options are commonly a single letter prefaced with a single dash, or a word prefaced with two dashes. The `ls` command can be given with the option `-l` specified between the command & the directory. This options gives the output in what is known as *long listing* format.

Inspect the contents of your home directory using the long listing format. Please make sure you can tell the difference between the characters `l & 1`.

```
1 ls -l ~
```

The above will give a few lines of output & the first line should be something similar to

```
drwxr-xr-x 2 hub hub 4096 mmm dd hh:mm Desktop
```

where `mmm dd hh:mm` are time and date information.

The important thing to notice is that the word `Desktop` at the end of the line will be coloured **blue**, indicating that it is a directory. The letter '`d`' at the beginning of the initial string of codes `drwxr-xr-x` also indicates this fact. Formally, these letters are known as flags which identify key attributes about each file or directory. We can ignore the fine detail in the rest of these flags until this afternoon (or see the bonus section), but the values `rwx` simply refer to who is able to **r**ead, **w**rite or e**x**ecute the contents of the file or directory. These are very helpful attributes for data security & protection against malicious software.

The entries `hub hub` respectively refer to who is the owner of the directory (or file) & to which group they belong. Again, this information won't be particularly relevant to us today, so we can ignore this until later in our programming careers. In brief, on an Ubuntu system you could have every member of your lab group as an individual user, whilst making every member part of a group. File access permissions can then be applied to any file based on who created it, or whether they are a member of your lab group. Finally, the value `4096` is the size of the directory structure in bytes, whilst the date & time refer to when the directory was created.

The flags we saw at the beginning of the entries above have a clearly defined structure. The first entry shows the file type and for most common files, this entry will be the "-" seen above. The next entries are three triplets which refer to 1) the file's owner, 2) the group they belong to & 3) all users.

In some other entries you'll come across, the name of the file will be in **green**, indicating that it is a file not a directory. There will also be a '`-`' instead of a '`d`' at the beginning of the initial string of flags. The remainder of the information is essentially the same as for the directories we've already seen.

There are many more options that we could specify to give a slightly different output from the `ls` command. Two particularly helpful ones are the options `-h` and `-R`. We could have specified the previous command as

```
ls -l -h ~
```

This will change the file size to *"human-readable"* format, whilst leaving the remainder of the output unchanged. Try it & you will notice that where we initially saw `4096` bytes, the size is now given as `4.0K`. This can be particularly helpful for larger files, as most NGS files are very large indeed and seeing a file size in gigabytes will be much more informative.

The option `-R` tells the `ls` command to look through each directory recursively. If we enter

```
ls -l -R ~
```

the output will be given in two sections. The first is what we have seen previously, but following that will be the contents of the directory `/home/hub/Desktop`. It should become immediately clear that the output from setting this option can get very large & long depending on which directory you start from. It's probably not a good idea to enter `ls -l -R /` as this will print out the entire contents of your file system.

In the case of the `ls` command we can actually specify all the above options together in the command

```
ls -lhR ~
```

This can often save some time, but it is worth noting that not all programmers write their commands in such a way that this convention can be followed. The built-in

shell commands are usually fine with this, but many NGS data processing functions do not accept this convention.

Don't Panic!!!

It's easy for things to go wrong when working in the command-line, but if you've accidentally set something running which you need to exit or if you can't see the command prompt, there are some simple options for stopping a process & getting you back on track. Some options to try are:

`Ctrl-c`	kill the current job. This is usually the first port of call when things go wrong. Also see `man kill` or `man killall` for details on how to kill a process.
`Ctrl-d`	end of input. Sometimes `Ctrl-c` doesn't work but this does.

Exploring Commands In More Detail

Most commands we wish to use have a series of options (sometimes called flags) we are able to set. In reality no-one can remember the full suite of available options, so we need to find out how to get this information. In order to help us find what options are able to be specified, every command built-in to the shell has a manual, or a help page which can take some time to get familiar with. These help pages are displayed using the pager known as `less` which essentially turns the terminal window into a text viewer so *we can display text in the terminal window*, but with no capacity for us to edit the text.

To display the help page for `ls` enter the command

```
1 man ls
```

As beforehand, the space between the two is important & in the first word we are invoking the command `man` which then looks for the *manual* associated with the command `ls`. To navigate through the manual page, we need to know a few shortcuts which are part of the `less` pager..

Although we can navigate through the `less` pager using up & down arrows on our keyboards, some helpful shortcuts are:

<`enter`>	go down one line
<`spacebar`>	go down one page (i.e. a screenful)
b	go up (i.e. **b**ackwards one page)
<	go to the beginning of the document
>	go to the end of the document
q	exit (i.e. **q**uit the page)

Look through the manual page for the `ls` command. How could we give the directory contents in long listing format, sorted by file size?

Many software tools create "*hidden*" files by starting their name with a dot. By default, these files won't be displayed using `ls` How could we make `ls` display these files as well as the main set of visible files.

We can actually find out more about the `less` pager by calling it's own `man` page. Type the command

```
1 man less
```

and the complete page will appear. This can look a little overwhelming, so try pressing `h` which will take you to a summary of the shortcut keys within `less`. There are a lot of them, so try out a few to jump through the file.

A good one to experiment with would be to search for patterns within the displayed text by prefacing the pattern with a slash. Try searching for a common word like *"the"* or *"to"* to see how the function behaves, then try searching for something a bit more useful, like the word *"move"*.

Accessing Manuals or Help Pages

As well as entering the command `man` before the name of a command you need help with, you can often just enter the name of the command with the options `-h` or `--help` specified. Note the convention of a single hyphen which indicates an individual letter will follow, or a double-hyphen which indicates that a word will follow. Unfortunately the methods can vary a little from command to command, so if one method doesn't get you the manual, just try one of the others.

Sometimes it can take a little bit of looking to find something and it's important to be realise we won't break the computer or accidentally launch a nuclear bomb when we look around. It's very much like picking up a piece of paper to see what's under it. If you don't find something at first, just keep looking and you'll find it eventually.

Try accessing the manual for the command `man` all three ways. Was there a difference in the output depending on how we asked to view the manual?

Could we access the help page for the command `ls` all three ways?

Some Useful Commands

So far we have explored the commands `pwd`, `cd`, `ls` & `man` as well as the pager `less`. Inspect the `man` pages for the commands in the following table & fill in the appropriate fields. Have a look at the useful options & try to understand what they will do if specified when invoking the command.

Command	Description of function	Useful options
`man`	Display on-line manual	-k (search for keywords if you don't know the command)
`pwd`	Print working directory, i.e show where you are	none commonly used
`ls`	List contents of a directory	-a, -h, -l, -S, -t, -R
`cd`	Change directory	(scroll down in `man builtins` to find `cd`)
`mv`		-b, -f, -u
`cp`		-b, -f, -u
`rm`		-r (careful...)
`rmdir`		
`mkdir`		-p
`cat`		
`wc`		-l
`head`		-n
`tail`		-n
`echo`		-e
`cut`		-d, -f, -s
`sort`		
`uniq`		-c

Tab auto-complete

A very helpful & time-saving tool in the command line is the ability to automatically complete a command, file or directory name using the <`tab` > key. Try typing

```
1 ls /home/h <tab>
```

where <`tab` > represents the tab key.

Notice how the word `hub` is completed automatically! This functionality will automatically fill as far as it can until conflicting options are reached. In this case, there was only one option so it was able to complete all the way to the end of the file path. This enables us to quickly enter long file paths without the risk of typos. Using this trick will save you an

enormous amount of time trying to find why something doesn't work. The most common error we'll see today will be mistakes in file paths caused by people not taking advantage of this trick.

Now enter

```
ls ~/Do <tab>
```

and it will look like the auto-complete is not working. This is because there are two possibilities & it doesn't know which you want. Hit the tab twice and both will appear in the terminal, then choose one. As well as directory paths, you can use this to auto-complete filenames.

This technique can be used to also find command names. Type in `he` followed by two strike of the <`tab` > key and it will show you all of the commands that being with the string `he`, such as `head`, `help` or any others that may be installed on your computer. If we'd hit the <`tab` > key after typing `hea`, then the command `head` would have auto-completed, although clearly this wouldn't have saved you any typing.

Putting It All Together

Now we can use some the above commands to perform something useful.

Creating and Viewing a File

First, let's create a personal directory under `/home/hub` with your first name as the directory name. **Where you see *firstname* below, use your actual firstname.**

```
cd ~
mkdir firstname
```

Now we can change into this directory.

```
cd firstname
```

Create an empty text file. Check the `man` page for `touch` if you're not sure about this line.

```
touch hello.txt
```

We can read it, but it won't have anything in it yet.

```
cat hello.txt
```

Obviously nothing was printed to your terminal in the previous line because the file is

empty. Let's write something to the file, then try reading it again. In the following line, the symbol `>>` places the text **at the end** of whatever is already in the file. As the file is empty, this will just write a single line.

```
1 echo "Hello" >> hello.txt
2 cat hello.txt
```

We can find a whole lot of information about the file.

```
1 wc hello.txt
```

In the previous line, what do the three numbers represent?

When we added the word `Hello` to our file, we used the symbol >> which actually wrote the word to the end of the file. As the file was completely empty, this placed the word in the first line. This is a VERY handy short-cut for writing information to the end of a file

Now let's add more to the file.

```
1 echo "It's me" >> hello.txt
2 cat hello.txt
3 wc hello.txt
```

For most of the above commands we could have used the auto-complete feature of the bash terminal. Did you remember this trick?

Copying and Renaming a File

Later today, we're going to look through a file containing a list of words. It's currently on your VM in the folder `/usr/share/dict` and has the name `cracklib-small`. Let's copy this to your `firstname` folder, in your home directory.

```
1 cp /usr/share/dict/cracklib-small ~/firstname
```

Next we will rename the file. Note, that in bash there is no "rename" tool. Instead we "move" from it's current location to any location we choose, with whatever name we choose. In the following, the location is the same so we are effectively just giving the file a new name.

```
mv ~/firstname/cracklib-small ~/firstname/words
```

We can look at the first 5 lines of the file using

```
head -n5 ~/firstname/words
```

Or we can look at the last 10 lines of the file using

```
tail -n10 ~/firstname/words
```

We could even page through the file using `less`. (Remember to hit `q` to exit the pager.)

```
less ~/firstname/words
```

We can even find how many lines there are in the file by using

```
wc -l ~/firstname/words
```

Did we need to enter the full file path in the above commands, or could we have saved ourselves some effort by changing into the `~/firstname` directory?

Summary

Now we have experience in several key tasks which are some of the most common tasks needed for any bioinformatics analysis, or for any general HPC usage.

1. How to access a manual

2. How to create and navigate through directories

3. How to create and view files

4. How to copy, move and rename files

Writing Scripts

Primary Author(s):
Stephen Bent, Robinson Research Institute, University of Adelaide
Steve Pederson, Bioinformatics Hub, University of Adelaide
stephen.pederson@adelaide.edu.au
Robert Qiao, Research Services Phoenix Team, University of Adelaide
robert.qiao@adelaide.edu.au

Contributor(s):
Dan Kortschak, Adelson Research Group, University of Adelaide
dan.kortschak@adelaide.edu.au
Jimmy Breen, Robinson Research Institute & Bioinformatics Hub, University of Adelaide
jimmy.breen@adelaide.edu.au

We often need to perform repetitive tasks, or need to perform complex series of procedures. Rather than typing instruction receptively and stearing the screen waiting for each process to finish before providing next instruction, writing the set of instructions into a script and interpreter/compiler does the waiting and stearing for us is a very powerful way of liberating our time. They are also an excellent way of ensuring the commands you have used in your research are retained for future reference. Keeping copies of all electronic processes to ensure reproducibility is a very important component of any research. Writing scripts requires an understanding of several key concepts which form the foundation of much computer programming, so let's walk our way through a few of them.

Shell Scripts

Now that we've been through just some of the concepts & tools we can use when writing scripts, it's time to tackle one of our own where we can bring it all together.

Every bash script begins with what is known as a *shebang*, which we would commonly recognise as a hash sign followed by an exclamation mark, i.e `#!`. This is immediately followed by `/bin/bash`, which tells the interpreter to run the command `bash` in the directory `/bin`. (This is actually where the program `bash` lives on a Linux system.) This opening sequence is vital & tells the computer how to respond to all of the following commands. As a string this looks like:

```
#!/bin/bash
```

The hash symbol generally functions as a comment character in scripts. Sometimes we can include lines in a script to remind ourselves what we're trying to do, and we can preface these with the hash to ensure the interpreter doesn't try to run them. It's presence as a comment here, followed by the exclamation mark, is specifically looked for by the interpreter but beyond this specific occurrence, comment lines are generally ignored by scripts & programs.

Some Example Scripts

Let's now look at some simple scripts. These are really just examples of some useful things you can do & may not really be the best scripts from a technical perspective. Hopefully they give you some pointers so you can get going

To setup this part of the course, please login to your Phoenix account and download the example file to your /fastdir directory now

```
1 cp -r /apps/examples/training_linux/ ~/fastdir/
```

please check to make sure you command blow contains valid 4 files. If you see error messages, please indicate to tutors, you should get this step fixed before carry on further

```
1 ls -al ~/fastdir/training_linux
```

A Simple Example to Start

Don't try to enter these commands directly in the terminal!!! They are designed to be placed in a script which we will do after we've inspected the contents of the script. First, just have a look through the script & make sure you understand what the script is doing.

Also remember that any long lines of code may be automatically broken into new lines on your page by the '\' character. We don't want to enter this character when we create our script. Note that the line numbers on the left of the code don't change when this happens, e.g. line 9.

Before we go any further, have a look at the following script.

```
#!/bin/bash
#
# First we'll declare some variables with some text strings
ME='Put your name here'
MESSAGE='This is your first script'

# Now well place these variables into a command to get some output
echo -e "Hello ${ME}\n${MESSAGE}\nWell Done!"
```

Firstly, you may notice some lines that begin with the # character. These are *comments* which have no impact on the execution of the script, but are written so you can understand what you were thinking when you wrote it. If you look at your code 6 months from now, there is a very strong chance that you won't recall exactly what you were thinking, so these comments can be a good place just to explain something to the future version of

yourself. There is a school of thought which says that you write code primarily for humans to read, not for the computer to understand.

In the above script, there are two variables. Although we have initially set them to be one value, they are still variables. What are their names?

First we'll create an empty file which will become our script. We'll give it the suffix `.sh` as that is the common convention for bash scripts.

```
1 cd ~/firstname
2 touch wellDone.sh
```

Now using the text editor *nano*, enter the above code into this file *setting your actual name as the ME variable*, and save it by using `Ctrl+o`, which is indicated as `^O` in the nano screen.

```
1 nano wellDone.sh
```

Once you're finished, you can exit the `nano` editor by hitting `Ctrl+x`.

Another coding style which can be helpful is the enclosing of each variable name in curly braces every time the value is called. Whilst not being strictly required, this can make it easy for you to follow in the future when you're looking back. Variables have also been names using strictly upper-case letters. This is another optional coding style, but can also make things clear for you as you look back through your work. Most command line tools use strictly lower-case names, so this is another reason the upper-case variable names can be helpful.

Unfortunately, this script cannot be executed yet but we can easily enable execution of the code inside the script. If you recall the flags from earlier which denoted the read/write/execute permissions of a file, all we need to do is set the execute permission for this file. First we'll look at the files in the folder using `ls -l` and note these triplets should be `rw-` for the user & the group you belong to. To make this script executable, enter the following in your terminal.

```
1 cd ~/firstname
2 chmod +x wellDone.sh
3 ls -l
```

Notice that the third flag in the triplet has now become an `x`. This indicates that we can now execute the file in the terminal. As a security measure, Linux doesn't allow you to execute a script from within the same directory so to execute it enter the following:

```
1 ./wellDone.sh
```

Making a Small Change

Now let's change the variable `ME` in the script to read as

```
1 ME=$1
```

and save this as `wellDone2.sh`. (You may like to create this first using `cp`) You'll now need to set the execute permission again.

```
1 chmod +x wellDone2.sh
```

This time we have set the script to *receive input from stdin* (i.e. the terminal), and we will need to supply a value, which will then be placed in the variable `ME`. Choose whichever random name you want and enter the following

```
1 ./wellDone2.sh Boris
```

As you can imagine, this style of scripting can be useful for iterating over multiple objects. A trivial example, which builds on a now familiar concept would be to try the following.

```
1 for n in Boris Fred; do (./wellDone2.sh $n); done
```

A more complicated script

Here's a more complicated script with some more formal procedures. This is a script which will extract only the ProbeFeature features from the .gff file we have been working with, and export them to a separate file. Look through each line carefully & write down your understanding of what each line is asking the program to do.

```
#!/bin/bash
# Declare some helpful variables
FILEDIR=~/fastdir/training_linux/scripting
FILENAME=NC_015214.gff
OUTFILE=NC_015214_CDS.txt
# Make sure the directory exists
if [ -d ${FILEDIR} ]; then
  echo Changing to ${FILEDIR}
  cd ${FILEDIR}
else
  echo Cannot find directory ${FILEDIR}
  exit 1
fi

# If the file exists, extract the important ProbeFeature data
if [ -a ${FILENAME} ]; then
  echo Extracting ProbeFeature data from ${FILEDIR}/${FILENAME}
  echo "SeqID Source Start Stop Strand Tags" > ${OUTFILE}
  awk '{if (($3=="ProbeFeature")) print $1, $2, $4, $5, $7, $9}' \
      ${FILENAME} >> \
  ${OUTFILE}
else
  echo Cannot find ${FILENAME}
  exit 1
fi
```

Notice that this time we didn't require a file to be given to the script. We defined it within the script, as we did for the output file.

The directory & file checking stages were of the form if [...]. This is a curious command that checks for the presence of something. The options -d & -a specify a directory or file respectively.

Will the above script generate a tab, comma or space delimited text file?

Open the `gedit` text editor & save the blank file in your directory as *extract_CDS.sh*. Now write this above script into the editor, but *taking care to use the directory where you have the .gff file stored in the appropriate place.* Once you have written the script, save it & close it. Now make it executable and run it.

Moving towards High Performance Computing

High Performance Computing

In current genomics era, where we regularly work with large datasets, the amount of resources available on desktop computers are often insufficient to enable your script finish in a reasonable time. For these datasets, it maybe useful to work on a high performance computing system, which will enable your data or command to be run simultaneously on > 8 threads, i.e. in parallel. It is possible to gain access to large computing resources through the University's Phoenix HPC https://www.phoenix.adelaide.edu.au, enabling analysis of large datasets efficiently.

Having many users on one machine at one time also means that there needs to be a system which determines who runs what and when. Phoenix uses a schedular "SLURM", whereby users submit jobs to a queue, and then executed when the appropriate resources on the machine become available.

A typical slurm job script contains extra parameters such as:

- `-n`: The number of threads to allow
- `--time`: The maximum time it is allowed to takes to completion
- `--mem`: The amount of memory to allocate to the job

```
1 #!/bin/bash
2 #SBATCH -p batch
```

```
3 #SBATCH -N 1
4 #SBATCH -n 8
5 #SBATCH --time=20:00:00
6 #SBATCH --mem=20GB
7
8 # Execution code going below
9 <execution code>
```

We don't need to write this script. It is included here as a simple example of a real world script as used by Phoenix users. This script may look a little intimidating at first, but slowly work your way through each line & try to understand what each line is specifying.

More detailed info will be included in the last chapter of this course.

Space for Personal Notes or Feedback