Contents

The Trainers

Ms Ramona Rogers
Computing Officer
IT Support
The University of Adelaide
South Australia
ramona.rogers@adelaide.edu.au

Mr Robert Qiao
Phoenix Support Team
The University of Adelaide
South Australia
robert.qiao@adelaide.edu.au

Mr Bowen Chen
Phoenix Support Team
The University of Adelaide
South Australia
bowen.chen@adelaide.edu.au

Mr Exequiel Sepúlveda
Phoenix Support Team
The University of Adelaide
South Australia
exequiel.sepulvedaescobedo@adelaide.edu.au

Welcome

Thank you for your attendance & welcome to the Introduction to Linux for Phoenix Users Workshop. This is an offering by the University of Adelaide, Reserach Computing team which is a centrally funded initiative, with the aim of assisting & enabling researchers in their work.

This workshop is largaly based on the "Introduction to Linux" workshop deliveried by The Bioinformatics Hub. The Hub has a web-page at http://www.adelaide.edu.au/bioinformatics-hub/, and **to be kept up to date on upcoming events and workshops, please join the internal Bioinformatics mailing list on** http://list.adelaide.edu.au/mailman/listinfo/bioinfo.

Phoenix has two main sources of information: http://www.adelaide.edu.au/phoenix/ and or wiki page: https://wiki.adelaide.edu.au/hpc/index.php/Main_Page.

Phoenix team has an active Slack team for discussing questions with the local community. Slack teams do require an invitation to join, so please email us the Hub on hpcsupport@adelaide.edu.au to join the community. All are welcome.

Today's workshop has been put together based on previous material and courses prepared by Dr Stephen Bent (*University of Queensland*), with generous technical support & advice provided by Dr Nathan Watson-Haigh (*ACPFG*) and Dr Dan Kortschak (*Adelaide University, Adelson Research Group*) and complemented by Phoenix team: Ramona Rogers, Robert Qiao, Bowen Chen and Exequiel Sepulveda.

We hope it will be useful in enabling you to continue and to advance your research.

Course Summary

In today's workshop, the morning session will be spent introducing you to the basic tools and concepts required for data handling. In the afternoon session we'll develop these skills to a more advanced level, with progress in both sessions being made at your own pace. Some people may finish early today, but the majority of you probably won't.

You will use Phoenix all the time.

The majority of data handling and analysis required for research uses the *command line*, alternatively known as the terminal or the *bash shell*. This is a text-based interface in which commands must be typed, as opposed to the Graphical User Interfaces (aka GUIs) that most of us have become accustomed to. Being able to access your computer using these tools enables you to more fully utilise the power & capabilities of your machine, for both Linux & Mac operating systems, and to a lesser extent will even enable you to dig

deeper on a Windows system.

Whilst some of the tools we cover today may appear trivial, they are used on a daily basis by those working in the field. These basic tools are essential for writing what are known as *shell scripts*, which we will begin to cover in the afternoon session. These are essentially simple programs that utilise the inbuilt functions of the shell, and are used to automate processes such as de-multiplexing read libraries, or aligning reads to the genome. A knowledge of this simple type of programming and navigation is also essential for accessing the high-performance computing resources such as `phoenix`.

Using the Post-it Notes

For today's session, you will be provided with 3 post-it notes of differing colours. Please use these to signal whether you need help or not by placing them on your monitors. We will interpret these as:

1. **Red** - Help! I can't make something work
2. **Yellow** - I'm working on something, but haven't made it yet
3. **Green** - I've finished the task I was working on

Providing Feedback

While we endeavour to deliver a workshop with quality content and documentation in a venue conducive to an exciting, well run hands-on workshop with a bunch of knowledgeable and likable trainers, we know there are things we could do better.

Whilst we want to know what didn't quite hit the mark for you, what would be most helpful and least depressing, would be for you to provide ways to improve the workshop. i.e. constructive feedback. After all, if we knew something wasn't going to work, we wouldn't have done it or put it into the workshop in the first place!

Clearly, we also want to know what we did well! This gives us that "feel good" factor which will see us through those long days and nights in the lead up to such hands-on workshops!

Document Structure

We have provided you with an electronic copy of the workshop's hands-on tutorial documents. We have done this for two reasons: 1) you will have something to take away with you at the end of the workshop, and 2) you can save time (mis)typing commands on the command line by using copy-and-paste.

We advise you to use Acrobat Reader to view the PDF. This is because it properly supports some features we have implemented to ensure that copy-and-paste of commands works as expected. This includes the appropriate copy-and-paste of special characters like tilde and hyphens as well as skipping line numbers for easy copy-and-past of whole code blocks.

While you could fly through the hands-on sessions doing copy-and-paste, you will learn more if you use the time saved from not having to type all those commands, to understand what each command is doing!

The commands to enter at a terminal look something like this:

```
1 tophat --solexa-quals -g 2 --library-type fr-unstranded -j \
    annotation/Danio_rerio.Zv9.66.spliceSites -o tophat/ZV9_2cells \
    genome/ZV9 data/2cells_1.fastq data/2cells_2.fastq
```

The following styled code is not to be entered at a terminal, it is simply to show you the syntax of the command. You must use your own judgement to substitute in the correct arguments, options, filenames etc

```
tophat [options]* <index_base> <reads_1> <reads_2>
```

The following icons are used in the margin, throughout the documentation to help you navigate around the document more easily:

 Important

 For reference

 Follow these steps

 Questions to answer

 Warning - STOP and read

 Bonus exercise for fast learners

 Advanced exercise for super-fast learners

Computer Setup

We will all be working on `Phoenix` directly, which is the University of Adelaide's High Performance Computing (HPC) system. The software client `Bitvise` or `putty` which you will have already installed, enables us to access these machines in a familiar Desktop style, even though the majority of our time will be spent within the terminal.

In case you need to install Bitvise, download the installer from here: Bitvise SSH Client

You will need to open a SSH session to `phoenix`. First, we need to create a session with the basic parameters

1. Hostname phoenix.adelaide.edu.au
2. Username your student or staff ID

Now we have created the session, you will be asked for your password.

Now that you are connected, you will notice we are now in the head node of `Phoenix`. Welcome to `Phoenix`.

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.

When you open the terminal on your VM, you'll see the phrase `hub@biohub:~`. This is simply username@computername:, and you may remember you logged on as `hub` during the setup. 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

```
1 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

```
1 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

```
1 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

Understanding file system on Phoenix

Primary Author(s):
Bowen Chen, Technology Service, University of Adelaide
bowen.chen@adelaide.edu.au

Contributor(s):
Bowen Chen, Technology Service, University of Adelaide
bowen.chen@adelaide.edu.au

Home directory

On Linux, each user has a personal directory, i.e. home directory, to store his own files and data, as well as directories. Your home directory on Phoenix is located at /home/⟨userid⟩. Here, /⟨userid⟩ means aXXXXXXX, your id.

When you login to Phoenix, your user shell is located in your home directory. The absolute pathname of your home directory, i.e. /home/⟨userid⟩, is also stored in the environmental variable, $HOME. Tilde symbol, ∼, also represent your home directory. You can use following commands to change into your home directory.

```
cd ∼
cd $HOME
cd /home/<userid>
```

The /home file system is backed up, and is solely intended for the files that define your user environment and irrecoverable data such as source code. The default quota of your home directory is 10 GB.

Don't use $HOME for launching jobs or active job data. The /home file system hardware is not designed to support the intensive file access generated by the many hundreds of jobs that run on the Phoenix compute cores, you should use the /fast file system for job input/output instead.

Fast directory

The /fast directory is designed to improve the data handling performance. It is supported by high performance Lustre filesystem and is intended for active job data and immediate data storage.

The /fast directory can handle the intensive file access created by the compute jobs that run on Phoenix. It is designed to launch jobs and store personal/research data. The default quota of your fast directory is 4 TB.

Your personal fast directory is located at /fast/users/⟨userid⟩ The absolute pathname of your fast directory is also stored in the environmental variable, $FASTDIR A symbolic link to that directory can be found in your home directory, i.e. ∼/fastdir. Thus, you can use following commands to change into your fast directory:

```
cd ∼/fastdir
cd $FASTDIR
cd /fast/users/<userid>
```

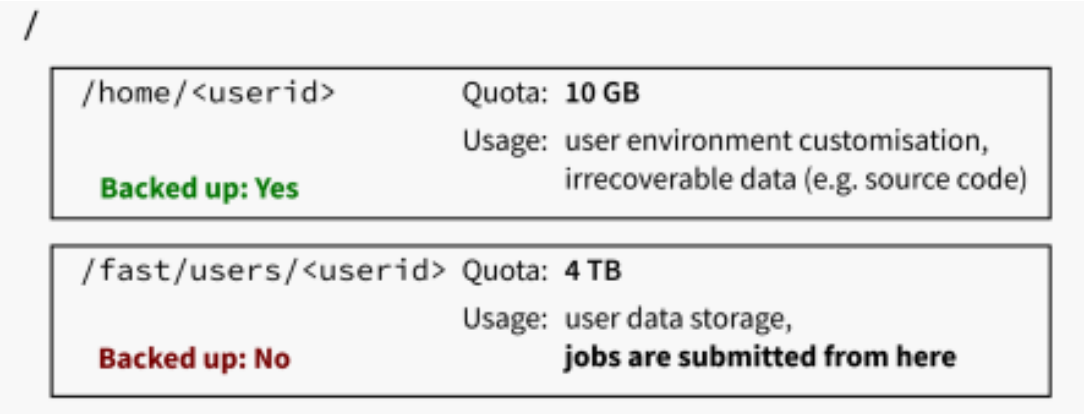

Figure 1: Home and fast directory on Phoenix

Mounting filesystems

"A file system or filesystem is used to control how data is stored and retrieved. Without a file system, information placed in a storage medium would be one large body of data with no way to tell where one piece of information stops and the next begins. By separating the data into pieces and giving each piece a name, the information is easily isolated and identified." (Wikipedia) Mounting a filesystem simply means making the particular filesystem accessible at a certain point in the Linux directory tree.

You can use following commands to observe the mounting point and other information about /home and /fast on Phoenix

```
1 mount | egrep home
2 mount | egrep fast
```

File and folder permissions

Since Linux is a multi-user OS that is based on concepts of file ownership and permissions to provide security, at the file system level. Using the command, chmod, can help us to change permissions if you are the owner of the files or directories.
Before to use it, you need to remember several symbols and letters with the chmod.

Identities

- u - the user who owns the file, i.e. owner.
- g - the group to which the user belongs, i.e. group
- o - others (not the owner nor the owner's group)
- a - everyone or all (u, g, and o)

Permissions

- r - read access
- w - write access
- x - execute access

Actions

- + - adds the permission
- - - removes the permission
- = - makes it the only permission

The first thing we need to do is to create the file, foo.txt. Use the following commands to create the file and show its default permission.

```
1 touch foo.txt
2 ls -l foo.txt
```

When using ls -l command, you can see some thing like this:
-rw-r–r– 1 a1695820 CATS_STUDENTS 5 Sep 26 14:34 foo.txt

Let's focus on -rw-r–r–. To explain it, parentheses will be added.
In -(rw-)(r–)(r–), the first pair of parentheses means the users's permission; the second means the group's permissions; the last is for all the others.

Then using the echo command to write some words.

```
1 echo "On my way to supercomputing" > foo.txt
```

Using the following command to change permissions and show current permission of it. By typing u-r, you are going to remove read permission for the user, i.e. owner, and group from the file foo.txt.

```
1 chmod ug-r foo.txt
2 ls -l foo.txt
```

Now try to read the file with cat command

```
1 cat foo.txt
```

When you execute commands on Phoenix, you may also meet error message like: ... Permission denied. Considering the reason caused it and how should we fix it based on what we introduced.

You should know what went wrong. Now let's fix the issue.

```
ls -l foo.txt
cat foo.txt
chmod u+r foo.txt
ls -l foo.txt
cat foo.txt
```

Another way to change permissions uses numeric representation.
Each permission setting can be represented by a numerical value:

- r = 4
- w = 2
- x = 1
- - = 0

For example, if foo.txt has following permissions settings:
- (rw-) (rw-) (r–)

Let's compute the numeric representation. The numeric representation for the user is six(4+2+0), for the group is six(4+2+0), and for others is four(4+0+0). Thus, the permissions setting is 664.

Bellowing is a list of common settings, numerical and meanings:

Setting	**Numerical**	**Meaning**
`-rw-------`	600	Only owner can read and write
`-rw-r--r--`	644	Only owner can read and write; group and others can read only
`-rwx------`	700	Only owner can read, wirte and execute
`-rwxr-xr-x`	755	Owner can read, wirte and execute; group and others can read and execute
`-rwx--x--x`	711	Owner can read, wirte and execute; group and others can execute only
`-rw-rw-rw`	666	All can read and write. (Be careful with this setting)
`-rwxrwxrwx`	777	All can read, write and execute. (All can modify the file. Be careful.)

Bellowing is a list of common settings for directories:

Setting	Numerical	Meaning
`-rw-------`	600	Only owner can read and write in the directory
`-rwxr-xr-x`	755	Owner can read, wirte and execute in the directory; group and others can read and execute

Where are the applications on Phoenix?

The applications on Phoenix are located in the /apps folder. The directory stores applications, as well as related module files. We will talk about modules on Phoenix latter.

Let's observe the /apps directory. Firstly, list the directory's permission and what are in it.

```
ls -ld /apps
ls -l /apps
```

You should see the two directories, modules and software. Calculating the numerical permission of the two. Second, list what are in the directory, module

```
ls -l /apps/modules
```

Finally, list what are in the directory, software.

```
ls -l /apps/modules
```

If too many things are listed, using pipe with less

```
ls -l /apps/modules | less
```

When using less, press f key for going forward and press b key for going backward. If you want to quit, press q.

Regular Expressions

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
Bowen Chen, Technology Service, University of Adelaide bowen.chen@adelaide.edu.au

Introduction

Regular expressions are a powerful & flexible way of searching for text strings amongst a large document or file. Most of us are familiar with searching for a word within a file using software such as MS Word or Excel, but regular expressions allow us to search for these with more power and flexibility, particularly in Linux files and directories management. Instead of searching strictly for a word or text string, we can search using less strict matching criteria. For example, we could search for a string that is either `slurm-3725` or `slurm-3825` by using the patterns `slurm-3[78]25` or `slurm-3(7|8)25`. These two patterns will search for an `slurm-3`, followed by either a `7` or `8`, then followed strictly by a `25`. Similarly a match to `slurm-37725` can be found by using the patterns `slurm-3[78][78]25` or `slurm-3[78]{2}25`.

Whilst the bash shell has a great capacity for searching a file to matches to regular expressions, this is where languages like *perl* and *python* offer a great degree more power, such as more complex data structure, object-oriented programming(OOP) and more libraries.

The command `grep`

The built-in command which searches using regular expressions in the terminal is `grep`. This function searches a file or input on a line-by-line basis, making patterns split across lines more difficult to find, which is one place that a programming language like Python or Perl would become preferable. The `man grep` page contains more detail on regular expressions under the `REGULAR EXPRESSIONS` header (scroll down a few pages). As can be seen in the `man` page, the command follows the form

```
grep [OPTIONS] 'pattern' filename
```

The option `-E` is preferable as it stands for Extended, which we can think of as "Easier". As well as the series of conventional numbers and characters that we are familiar with, we can match to characters with special meaning, as we saw above where enclosing the two letters in brackets gave the option of matching either. The `-E` option opens up the full set of wild-card characters, and can also be called simply by using `egrep` instead of `grep -E`. This is the default version that many of us use.

Special Character	Meaning
\w	match any letter or digit, i.e. a word character
\s	match any white space character, includes spaces, tabs & end-of-line marks
\d	match any digit from 0 to 9
.	matches any single character
+	matches one or more of the preceding character (or pattern)
*	matches zero or more of the preceding character (or pattern)
?	matches zero or one of the preceding character (or pattern)
{x} or {x,y}	matches x or between x and y instances of the preceding character
^	matches the beginning of a line (when not inside square brackets)
$	matches the end of a line
()	contents of the parentheses treated as a single pattern
[]	matches any one of the characters inside the brackets
[^]	matches anything other than any of the characters in the brackets
|	either the string before or the string after the ”pipe” (use parentheses)
\	don't treat the following character in the way you normally would. This is why the first three entries in this table started with a backslash, as this gives them their “special” properties, whereas placing a backslash before a ‘.’ symbol will enable it to function as an actual dot/full-stop.

Pattern Searching

In this section we'll learn the basics of using the `egrep` command & what forms the output can take. Using the command to download the file, regular_express.txt

```
cp /apps/examples/training_linux/regular_express.txt .
```

This is simply a text file with several words on every line.

Make sure the txt file in your working directory or else `egrep` won't be able to find the file.

Now let's try a few searches to get a feel for the basic syntax of the command. Using the previous table of special characters, try to describe what you're searching for on your notes **BEFORE** you enter the command. Do the results correspond with what you expected to see?

```
1 egrep -n 't[ae]st' regular_express.txt
```

```
1 egrep -n 'oo' regular_express.txt
```

```
1 egrep -n '[^g]oo' regular_express.txt
```

```
1 egrep -n '[^a-z]oo' regular_express.txt
```

```
1 egrep -n '[0-9]' regular_express.txt
```

```
1 egrep -n '[^[:lower:]]oo' regular_express.txt
```

```
1 egrep -n '^[a-z]' regular_express.txt
```

In the above, we were changing the pattern to extract different results from the files. Now we'll try a few different options to change the output, whilst leaving the pattern unchanged. If you're unsure about some of the options, don't forget to consult the `man` page.

```
1 egrep -n 'the' regular_express.txt
```

```
1 egrep -vn 'the' regular_express.txt
```

```
1 egrep -in 'the' regular_express.txt
```

The Tools sed & awk

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

sed: The Stream Editor

One very useful command in the terminal is `sed`, which is short for *stream editor*. Instead of the `man` page for `sed` the `info sed` page is larger but a little easier to digest. This is a very powerful command which can be a little overwhelming at first. If using this for your own scripts & you can't figure something out, remember "Google is your friend" & sites like www.stackoverflow.com are full of people wrestling with similar problems to you. You can be certain you're not the first person to be stumped by a problem & these are great places to start looking for help. Even advanced programmers use Google & Stack Overflow to find solutions.

For today, there are two key `sed` functionalities that we want to introduce.

1. Using `sed` to alter the contents of a file/input;
2. Using `sed` to print regions of a file

Altering a file or other input

`sed` uses *regular expressions* that we have come across under the `grep` section, and we can use these to replace strings or characters within a text string. The command works in the form

```
sed SCRIPT INPUT
```

and the script section is where all the action happens. Input can be given to `sed` as either a file, or just as a text stream via `stdin` using the *pipe* symbol that we have already introduced.

In the following example the script section begins with an 's' to indicate that we are going to make a substitution. The beginning of the first pattern (i.e. the *regexp* we are searching for) is denoted with the backslash, with the identical delimiter indicating the replacement pattern, and this is in turn completed with the same delimiter. Try this simple example from the link http://www.grymoire.com/Unix/Sed.html which is a very detailed & helpful resource about the usage `sed`. Here we are sending the input to the command via the pipe, so no 'INPUT' section is required:

```
1 echo Sunday | sed 's/day/night/'
```

Here you are passing `sed` the string Sunday, and `sed` takes day and turns it into night. `sed` will only replace the first instance of the string on any line, so try:

```
1 echo Sundayday | sed 's/day/night/'
```

Note that it only replaced the first instance of day and left the second. However, you can make it 'global', where it switches every instance by using the 'g' option at the end of the pattern like this:

```
1 echo Sundayday | sed 's/day/night/g'
```

You can also 'capture' parts of the pattern in parentheses and access that in the second part of the regular expression (what you are switching to) using \1, \2, etc., to denoted the number of the captured string, in the order they were captured. If you want to match 'ATGNNNTGA', where N is any base, and just output these three bases you could try the following:

```
1 echo 'ATGCCAGTA' | sed -r 's/ATG(.{3})GTA/\1/g'
```

Clearly, we have just given this command a sequence so we know exactly what to expect. However, hopefully this demonstrates the concept of extracting a subset of the sequence.

Or if we needed to replace those three bases with an expanded repeat of them, you could do the following where we capture the undefined string between `ATG` & `GTA`, and expand it:

```
1 echo 'ATGCCAGTA' | sed -r 's/ATG(.{3})GTA/ATG\1\1\1GTA/g'
```

The \1 take the contents of the first parenthesis and uses it in the substitution, even though you don't know what the bases are. Note that the '-r' option was set for these operations, which turns on extended regular expression capabilities. This can be a powerful tool & multiple parentheses can also be used:

```
1 echo 'ATGCCAGTA' | sed -r 's/(ATG)(.{3})(GTA)/\3\2\2\1/g'
```

In this last command we switched the order of the first & last triplet, and expanded the middle unknown string twice. Note how quickly this starts to look confusing though! Taking care to be clear when writing these types of procedures can be an important idea when you have to go back & re-read your code a year or two later. (Yes this will happen a lot!!!)

The use of backslashes to delineate each section of the script used by `sed` is the most common convention, but we are not restricted to it. We could have uses any 'wild-card' type character to follow the 's', such as '*' ':' or '%' although this must be consistent across all sections of the script, and should only be used if the backslash itself is part of the search or replacement string.

Displaying a region from a file

The command `sed` can also be used to replicate some of the functionality of the `head` & grep commands, but with a little more power at your fingertips. By default `sed` will print the entire input stream it receives, but setting the option '-n' will turn this off. Try this by adding an 'n' immediately after the '-r' in one of the above lines & you will notice you receive no output. This is useful if we wish to restrict our output to a subset of lines within a file, and by including a 'p' at the end of the script section, only the section matching the results of the script will be printed.

Make sure you are in the correct directory & we can look through the `regular_express.txt` file again.

```
sed -n '1,10p' regular_express.txt
```

This will print the first 10 lines, like the `head` command will by default. However, we could now print any range of lines we choose. Try this by changing the script to something interesting like '15,21p'.

We could also restrict the range to specific lines by using the `sed` increment operator '~'.

```
sed -n '1~5p' regular_express.txt
```

This will print every 5th line, beginning at the first.

We can also make sed operate like `grep` by making it only print the lines which match a pattern.

```
sed -rn '/g[o]*d/ p' regular_express.txt
```

Note however, that the line numbers are not present in this output.

Some Important Programming Concepts

Before moving on to awk, we need to quickly recap two of the most widely used techniques in programming:

1. The `for` loop
2. Logical tests using an `if` statement

For Loops

A `for` loop is what we use to cycle through an input one item at a time. As a simple example, we could print each number from a set of numbers.

```
1 for i in 1 2 3; do (echo -e $i); done
```

In the above code, the fragment before the semi-colon asked the program to cycle through the values 1, 2 & 3, letting the variable 'i' take each value in order of appearance. First i = 1, then i = 2 & finally i = 3. If you're wondering why we chose 'i', it just seemed like a sensible choice for an integer. We simply needed to choose a name for a *variable* which we would pass the values to.

After assigning each value to `i`, was the instruction on what to do for each value. Note that the value of the variable 'i' was *prefaced by the dollar sign ($). This is how the bash shell knows it is a variable, not the letter 'i'.* The command `done` then finished the `do` command. All commands like `do`, `if` or `case` have completing statements, which respectively are `done`, `fi` & `esac`.

Another important concept which was glossed over in the previous paragraph is that of a *variable*. These are essentially just '*placeholders*' which have a value that can change (hence the name). In the above loop, the same operation was performed on the variable `i`, but the value changed from 1 to 2 to 3. Variables in shell scripts can hold numbers or text strings and don't have to be formally defined as in some other languages.

An alternate approach could be to make a breathtaking claim about some files. Here we'll use the variable called 'f', which seems sensible for a filename.

```
1 cd /apps/examples/training_linux/scripting/
2 for f in $(ls); do (echo -e "I can see the file $f"); done
3 cd -
```

Note, that we've also assigned the output from the command `ls` to this variable, by using the $() syntax. The use of double quotes for the `echo` command also allows us to refer to the values held by f. Single quotes at this point would only return the characters '$f'.

If Statements

If statements are those which only have a binary 'yes' or 'no' response, or more correctly a `TRUE/FALSE` response. For example, we could specify things like:

- `if` (i> 1) then `do` something, or
- `if` (fileName==bob.txt) then `do` something else

Notice that in the second if statement, there was a double equals sign. This is the programmers way of saying *compare* the first argument with the second argument. A single equals sign is generally interpreted by a program as *assign* the value of the first argument to be the second argument. This use of 'double operators' is very common, notably you will see && to represent the command '*and*', and || to represent '*or*.' A final useful trick to be aware of is the use of an exclamation mark to reverse a command. A good example of this is the use of the command '!=' as the representation of *not equal to* in a logical test.

awk: A command and a language

Moving on to `awk`, this is a very useful tool which can be used either as a command, as well as functioning as it's own language. We'll just use it as a command today, and it is extremely useful for dealing with tab- or comma-separated files, such as we often see in biological data.

The basic structure of an `awk` command is:

```
awk '/<pattern>/' file
```

`awk` will then search the file and output any line containing the regular expression pattern (kind of like `grep`). With `awk`, you can also do:

```
awk '\{<code>\}' file
```

where you can put a program, or set of instructions in the curly braces. In the code, you can specify values from different columns of the file by using the numbers $1, $2, etc., (or you can use $0 for the whole line). Values can also be returned in the output by using the command `print` followed by the field number.

We have that regular_express.txt file and we've already looked at it at little, so let's pull out some particular features! Make your terminal as wide as the screen, then change into the appropriate directory & enter

```
1 awk '{if ($3=="is") print $0}' regular_express.txt
```

Here we've specified that the third field must be is.

We could make it a little more complex and just look for genes in a given region. **In the following line, the symbol '\' has been placed here to indicate it is a single line, extending beyond the width of the page. Do not enter this character!**

```
awk '{if ( ($3=="gene") && ($4 > 10000) && ($4 < 20000) ) print $0}' \
    regular_express.txt
```

In the above code, `$3==''gene''` asks for the entry in the third field to be "gene." The next two fragments request for the values in the fourth field (i.e. $4) to be between 10000 & 20000. Notice that each these three commands were enclosed in a pair of brackets within an outer pair of brackets. This gave a command of the form:
`( (Condition1) && (Condition2) && (Condition3) )`
After this came the fragment `print $0` which asked `awk` to print the entire line if the 3 conditions are true. You've just written (& hopefully understood) a computer program!

Another example (what does this do?):

```
awk '{if (($5 - $4 > 1000) && ($3 == "gene")) print $0}' \
    regular_express.txt
```

If you don't want to output all of the columns, you can specify which ones to output. While we're at it, let's save the output as a file:

```
awk '{if (($5 - $4 > 1000) && ($3 == "gene")) print $1, $2, $4, $5, \
    $9}' regular_express.txt > awkout.txt
```

The command `awk` has a pretty serious set of in-built commands which can be used in the code sections as above. Although it looks a little overwhelming, there is a detailed page http://www.grymoire.com/Unix/Awk.html which gives a rundown on the full capabilities of the language. One command that we may find helpful is `length`, which counts the number of characters in a line.

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
```

```
#SBATCH -N 1
#SBATCH -n 8
#SBATCH --time=20:00:00
#SBATCH --mem=20GB

# Execution code going below
<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.

Using Modules

Primary Author(s):
Exequiel Sepulveda, Research Services, University of Adelaide
exequiel.sepulvedaescobedo@adelaide.edu.au

Contributor(s):
Robert Quiao, Research Services, University of Adelaide
robert.quiao@adelaide.edu.au

There are many applications that need to change or define configurations to work properly. For example, they need to add the folder where binaries are located to the PATH environmental variable. If an application has many binaries located in /apps/MYAPP/bin, setting the PATH variable can be done by:

```
1 export PATH=$PATH:/apps/MYAPP/bin
```

Doing manually these modifications may be very tedious. The situation worsen if there are many applications that need to do the same with many potential conflicts and side effects. To keep environmental variables under control, Phoenix has an program that manages applications in a safe way. This program is called "modules"

Basic commands of modules

Phoenix has installed hundreds of applications for users to use. By default, all applications are located in /apps/software and each one has a module configuration located in /apps/modules/all. Modules program has an unique executable named module.

Try to get the help from module program:

```
module --help
```

The relevant options of module command are:

Option	Description of function	Useful options
`avail [name]`	Display the list of modules	-r for using regular expressions, -d for listing only the default version of each module
`spider [name]`	Explore and list modules	-r for using regular expressions, -d for listing only the default version of each module
`load name`	Load the specific module name	
`unload name`	Unload the specific module name	
`list`	List all loaded modules	
`show name`	Show the information of the module name	
`purge`	Unload all loaded modules to have a fresh starts	

The `avail/spider` option should be always used with a text to serach for, because listing all modules installed may be very slow and useless. The correct way to use avail/spider is using a text after, for example, to search Perl modules:

```
module avail Perl
module spider Perl
```

Spider and Avail are similiar but the output format is different. As an exercise, try to find all matlab modules using spider and avail options.

```
module spider matlab
module avail matlab
```

Repeat the same exercise to search for Python modules installed on Phoenix.

In general, most of modules follow this pattern: MODULE/VERSION[-TOOLCHAIN]. For example, for the module Python/3.6.1-foss-2016b: MODULE=Python, VERSION=3.6.1 and TOOLCHAIN=foss-2016b. Which toolchain should be selected is critical to understand.

Toolchain is a set of compiler, utilities and libraries (all they are modules!). The general rule is to use the newest official toolchain, which is foss-2016b. On Phoenix there are two main toolchains, foss and intel. The foss toolchain includes GNU compilers, whereas intel includes Intel compilers. You should use modules with the same toolchain. If you need to

use modules with a different toolchain, a best practice is to use the purge option before changing toolchain.

Loading modules

Once you have found the module you want to load, the next step is loading it.

For example, search for R modules and load the newest one:

```
module spider R

    Versions
R/3.2.1-foss-2015b
R/3.3.0-foss-2016uofa
R/3.4.0-foss-2016b
```

```
module load R/3.4.0-foss-2016b
```

To see what happens after the module is loaded, use the option list:

```
module list
```

Why are there many loaded modules? It is because R/3.4.0-foss-2016b module has many prerequisites that are automatically loaded as well.

In most the cases, module configurations include automatically all modules needed. In few cases, because a precise control is necessary, this should be manually done.

Using modules

Once you have loaded modules, the current session will be correctly configured to use those modules. For example, you could load any Python module and check if the version is the correct one:

```
module load Python

python --version
```

Be careful with modules containing programs that are part of the operating system. For example, C/C++ compiler, Python, Perl, among others. If you don't load the right module, you may end using the wrong version.

What is the Python version included in the operating system?

A complete exercise of modules

In this section you are asked to complete a full exercise to reach a master level of modules. Run the traditional "Hello World!" program in four different languages and versions.

Python 2.X:

```
print "Hello World, from Python 2.x"
```

Python 3.X:

```
print("Hello World, from Python 3.x")
```

R:

```
print("Hello World, from R")
```

Matlab:

```
display("Hello World, from matlab");
```

And these are the examples of how to run programs on those languages:
R:

```
echo 'COMMAND' | R --no-save
```

Matlab:

```
matlab -noFigureWindows -nodisplay -r 'COMMAND; exit;'
```

Python:

```
python -c COMMAND
```

The exercise is to find the modules to run each program and execute them. Replace COMMAND accordingly to each language.

Writing Basic Slurm Scripts

Primary Author(s):
Exequiel Sepulveda, Research Services, University of Adelaide
exequiel.sepulvedaescobedo@adelaide.edu.au

Contributor(s):
Robert Quiao, Research Services, University of Adelaide
robert.quiao@adelaide.edu.au

Phoenix is a shared HPC facility, therefore, all its resources are not for exclusive use of a particular user. The Phoenix architecture includes a head node and many computing nodes. The head node is where you are placed, after connect to Phoenix and is not designed for real computing.
The computing nodes are only available to our scheduler Slurm, which receives many scripts from all users and decides when and where to run those scripts according to the resources availables and account priorities. When Slurm receives a now job (or script) to run, he evaluates when is the sooner available time to run the job and assigns one or many computing nodes. The job will be ran by computing nodes, but not by the head node.

Slurm accounting, resources and priorities are complex enough. We have a workshop dedicated to this.

Therefore, to use the real power of Phoenix you need to write a slurm script. The script template is very simple: it is a normal script as you have written many, but with several Slurm meta-parameters at the beginning of the script.

Basic template of a Slurm script

A useful template for any slurm job is like this:

```
#!/bin/bash
#SBATCH -p batch
#SBATCH -N 1
#SBATCH -n 1
#SBATCH --time=00:05:00
#SBATCH --mem=1GB

#If you want to get feedback by email
#SBATCH --mail-type=ALL
#SBATCH --mail-user=firstname.lastname@adelaide.edu.au

#LOAD HERE ALL MODULES YOU NEED
#module load MODULE1
#module load MODULE2
#module load MODULE3

#EXECUTE HERE WHATEVER YOU WANT
#hostname
```

Let's have a look to each Slurm meta-parameter:

`#SBATCH -p batch` indicates the queue where the job will be submited to.

Phoenix has several queues, for example batch, test and bigmem. The batch queue is the default one for almost all kind of jobs. The queue test is useful to test jobs since the waiting times are reduced but also resources are limited just for testing.

`#SBATCH -N 1` indicates the number of computing nodes needed to run the job. Usually should be set to 1. `#SBATCH -n 1` indicates the number of cores needed to run the job. Usually should be set to 1.

Most of programs are serial and do not have parallel capabilities. If you do not know the capability of a program, assume the program is serial and, therefore, nodes and cores should be set to 1. If the program can use multicores, nodes should be set to 1 and cores to the desired number of cores, but less or equals to 32, which is the current maximum. If the program can use Message Passing Interface (MPI) and multicores, nodes should be set to equals or greater than 1, and cores to the desired number of cores, but less or equals to nodes*32.

`#SBATCH --time=00:05:00` indicates the maximum walltime of the job. In this example the maximum time will be 5 minutes.
`#SBATCH --mem=1GB` indicates the maximum amount of memory. In this example the maximum memory will be 1 Gigabytes.

Both time and mem meta-parameters are hard limits parameters. This means, if the job spends more time or consumes more memory, the jos will be cancelled. There is a trade-off. You would use conservative values, but if you overestimate them, the scheduler will take longer to executed it.

Getting notifications by e-mail is optional, but it is very recommended to know the progress of a job.
`#SBATCH --mail-type=ALL` indicates that Slurm will notify any state change, such as Cancelation, Success and Failure. `#SBATCH --mail-user=firstname.lastname@adelaide.edu.au` indicates the email address to send those notifications.

Submitting Scripts

Once you have a script ready to submit, you need to "submit" the script to the scheduler by the use of the command `sbatch`. Let's assume there is a script *job_script.sh*. To submit it to the schedule, just use the following command:

```
$ sbatch jobs_script.sh
Submitted batch job 3853482
```

In return you get a job number or *JOBID*, 3853482 is the example above, which is very important to record. Also, a new file with name slurm-*JOBID*.out is generated with the output of the script.

Create and submit a script that shows the hostname. Hint: the command for asking the hostanme is `hostname`.

Managing Scripts

Finally, after a job submission, you would need to manage its execution and get the information of completition. There are four basic commands to manage jobs:

Command	**Description of function**	**Options**
`squeue`	Display the list of queued jobs	-u USER
`scancel`	Cancel the execution of JOBID	JOBID
`scontrol show job`	Show useful information of a queued or running job	JOBID
`rcstat`	Show useful informartion of a job, specially for a completed job	JOBID

Now, from the last exercise in the Modules chapter (the execution of Hello World example in four languages), the final exercise:

Adapt that script as a Slurm script and submit it to the scheduler.

You can monitor its execution.

The command `rcstat` is useful to get the job statistics back, such as the real time spent, cpu and memory used, among others. You should use this information to adjust those values for the next submissions in order to request the right resources as closer as possible to the real ones.

Space for Personal Notes or Feedback