

An introduction to the Linux command line

Prof. Andrew C.R. Martin, University College London

September, 2018
Updated April, 2020

This self-paced tutorial will take you through some of the key concepts of the BASH command line shell. There are also many tutorials available on the Web or in books. I use **this style** for commands you type, and *this style* for file-names. In the command summary, I use *this style* for things that should be substituted with an appropriate filename or word.

Contents

1	Why do I care about this?	3
2	Introduction — what is the Linux/BASH command line?	3
2.1	What is Linux?	3
2.2	What is the BASH command line?	4
3	Accessing the BASH command line	5
3.1	If you are using Linux...	5
3.2	If you are using a Mac...	5
3.3	If you are using Windows...	5
4	Directories and Folders	6
4.1	Listing files	6
4.2	Changing the current working directory	7
5	Finding out more about files	8
6	Using the shell	8
6.1	Recalling previous commands	8
6.2	Typing less	9
6.3	Cut and Paste	9
7	Organizing files into subdirectories	9
8	Grabbing files from the Internet and moving files around	10
8.1	Grabbing files	10
8.2	Moving files	11

9	File archives	11
10	Looking at the content of a file	12
10.1	Displaying and concatenating	12
10.2	Copying files	13
10.3	Looking for differences in files	14
10.4	Viewing big files	14
10.5	Heads or tails.	14
10.6	Counting lines	14
10.7	More or less.	15
11	Input and output: Redirection and pipes	16
11.1	Redirection	16
11.2	Pipes	17
12	Removing directories	18
13	Getting more clever	19
13.1	Extracting specific lines from a file: grep	19
14	Links or shortcuts	22
15	Printing on the screen and environment variables	22
16	Aliases	23
17	Doing more complex things with pipes	24
18	Programming in BASH	25
18.1	Creating multiple files	26
18.2	Renaming a batch of files	27
19	The ls long format and file permissions	27
20	Changing permissions	29
21	Creating a reusable script	29
22	Commands, programs and paths	30
23	Command Summary	31
24	Text Editors	33
24.1	Emacs	33
25	Getting help	34
26	Other Tutorials	34

1 Why do I care about this?

Most of the programs you are used to on Windows or Mac computers use a ‘graphical user interface’ (GUI). Software such as web browsers, word processors or image manipulation software (to name just a few) would be impossible to use without a GUI. However, writing your own software that supports a GUI can be pretty difficult — I’ve been programming for nearly 40 years and I still find it tough!

In many cases however, you don’t need a graphical interface and often a GUI can hinder what you are trying to achieve. If you are starting out programming in a language like Python, Perl or R, you don’t want to be bothered with writing code that provides menus where you specify data files to process or provide boxes to tweak parameters¹. This is where the ‘command line’ comes in: it lets you type commands (including the names of programs you want to run) at a prompt². Often those commands (or programs) will take parameters such as the name of a file on which the command needs to act. In addition, a command (or program) may allow you to specify optional parameters to override default settings or to modify the process it performs in some way. Often this is done via ‘switches’ — optional parameters introduced with a dash (e.g. `-v`).

In addition to avoiding the complexity of having to write a GUI for your programs, the command line has three great advantages:

1. it has dozens of little utilities built in (including searching for text in files, counting the number of lines or words in a file, extracting specific columns from a file, etc.) that would be much more tricky to use via a GUI;
2. it has the ability to take the output of one command (e.g. finding all the lines that contain a certain word) and sending it into another command (e.g. counting the number of lines) allowing you to do more complex things (in this case finding how many lines contain a given word);
3. third it has a built in mini-programming language, letting you combine commands (and programs) in much more complex ways.

2 Introduction — what is the Linux/BASH command line?

2.1 What is Linux?

An ‘operating system’ is a set of software that runs on the hardware of your computer to look after things like communicating with other computers, managing users and security, storing files on disk, loading other programs from disk storage into memory so that they can be run, displaying things on the screen, talking to printers, keyboards, mice, etc., etc.

¹Note that these languages (certain Python libraries and R in particular) provide you with graphical *output* such as graphs in a popup window, but this is not the same as having to write your Python or R script such that it provides you with menus, buttons, text boxes and so on.

²Actually pretty much every ‘command’ that you run is, in reality, a program — all will become clear later!

Linux is a ‘flavour’ of a group of operating systems known as ‘unix’. Unix is a true multi-tasking, multi-user operating system. ‘Multi-tasking’ means that it is able to run multiple programs at the same time — if the computer has multiple CPU cores, then it can actually do multiple things at once (otherwise it switches rapidly between different tasks); ‘multi-user’ means that many users can log onto the same computer at the same time from separate ‘terminals’. Each user is given an area of disk space in which they can create their own files (a ‘home directory’).

These days, if you are a Mac user, then you are using a flavour of unix (perhaps without knowing it) as OS/X is built on a version of unix called Darwin which, in turn, is based on BSD unix. So unix-based operating systems can have beautiful graphical user interfaces (GUIs).

The term ‘unix’ or ‘Linux’ actually refers to what is known as the ‘kernel’ of the operating system — in other words all the deep and complicated things that go on to allow a computer to run software, access files on disk or on a network, etc. Everything else (including the graphical interface, or ‘window manager’ and even the command line itself), is actually a program that runs on top of the operating system. This means that people can package the Linux kernel with different sets of tools to produce a different Linux ‘distribution’. There are several of these, some of the most popular being Ubuntu, Fedora, RedHat and CentOS.

The beauty of this arrangement is that people have a huge amount of flexibility to make a Linux-based system look and feel as they want it to. For example, you might want a flashy Mac or Windows style GUI with windows that slurp down into an icon when you shrink them or have 3D effects on your really powerful desktop computer, but you might want something much simpler on the laptop that you bought 10 years ago for £250.00. If you are running a server, then you probably don’t need a GUI at all so you don’t install one! You don’t get that flexibility with Windows or a Mac.

2.2 What is the BASH command line?

You are almost certainly familiar with using ‘Explorer’ on Windows or ‘Finder’ on a Mac. This is a graphical user interface that allows you to browse through the hierarchy of folders that you have on your computer, create new folders, drag files from one folder to another and view the contents of files by starting an appropriate program (such as Microsoft Word when you click on a Word document). Linux has a whole host of similar directory manager programs. However in all three environments (Windows, Mac, Linux) you can carry out all the same functions from the command line.

The command line is a prompt at which you type commands (or the names of programs) that you want to run, together with required and optional parameters to tell the program what to process and to tweak the details of how it should be processed.

Under Linux, even the default command line can be slightly different between distributions (but you can choose the one you want). The ‘shell’ is the program that provides a prompt and lets you type commands. It then interprets those commands and runs other programs as required. When these programs have finished, the shell shows the prompt again waiting for you to do something. Here we will introduce the most common command line shell which is known as BASH. The BASH shell is also the default command line on a Mac and can be used under Windows (as described below).

3 Accessing the BASH command line

3.1 If you are using Linux...

Search for an application called 'Terminal'. If you can't find it then look for 'xterm', 'Command Line' or 'Command Prompt'. The terminal is actually just a window that runs the 'shell'. It is most likely that your terminal will run the BASH shell and the prompt may look like:

```
bash-4.2$
```

in which case you know you are using the BASH shell. If you don't have a prompt containing the word 'bash', then:

1: Type the following:

```
echo $SHELL
```

You should find that it says something like:

```
/bin/bash
```

If this doesn't contain the word 'bash' then ask for some help in making BASH your default shell³.

3.2 If you are using a Mac...

You can access the command line using the 'Terminal' application which lives in the '/Applications/Utilities/' folder. To find it, go to your 'Applications' folder. Near the bottom, there is a folder called 'Utilities'. The terminal is actually just a window that runs the 'shell', but the default shell is BASH which is what we want.

3.3 If you are using Windows...

Things are a bit more difficult⁴. The easiest way to access a BASH shell is to install a package called 'git-bash'. First check to see if 'git-bash' is already installed: type **git-bash** into the search box. If not, you can download this from <https://git-for-windows.github.io/>

Task:

Install any necessary software and open a command line prompt.

³The command **chsh bash** may work for you; if not then you really do need to ask for help!

⁴If you are running Windows 10 with the 'Anniversary Update' then you can run the 'Windows Subsystem for Linux' which provides you with a Linux BASH environment running under Windows (see <https://www.howtogeek.com/265900/everything-you-can-do-with-windows-10s-new-bash-shell/>). However, we have found this to be unreliable and not to allow access to the internet or to files downloaded under the normal Windows environment. At the time of writing, this is *not* recommended!

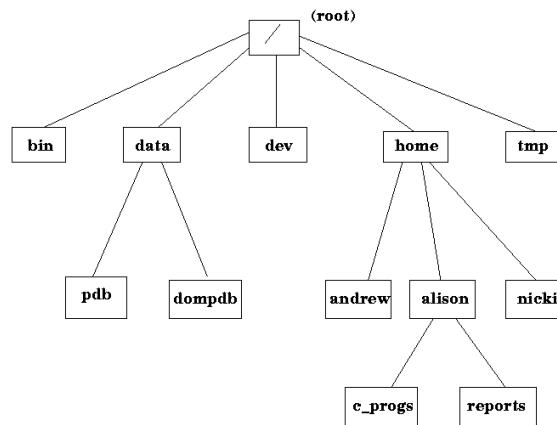


Figure 1: A typical unix filesystem.

4 Directories and Folders

In Linux, we tend to refer to ‘directories’ rather than ‘folders’, but it is exactly the same thing. Your disk space is organized into directories each of which can have sub-directories. You can also create sub-directories within the first level subdirectories and so-on to create a tree-like structure. Under unix, all files live in a single hierarchical directory tree (unlike Windows which has multiple trees beginning with C:, A:, D:, etc). The root (also called the top!) of that tree is called ‘/’ as shown in Figure 1.

4.1 Listing files

The first command we will introduce is `ls` which is short for ‘LiSt’.

2: Type the following:

```
ls
```

Note! That is a lower-case letter L (i.e. ‘l’) not the digit one (1).

In the BASH shell, we have the concept of a ‘current directory’. The `ls` command lists the files and directories in the current directory — it is like looking at a particular folder in the file explorer of a GUI.

3: Type the following:

```
ls -a
```

You should now find that you have some extra files listed — at a minimum, you will find one called ‘.’ and one called ‘..’. Any file that starts with a dot is a hidden file. The `-a` ‘flag’ (which stands for ‘all’) is placed after the `ls` command to see these files.

The directory called ‘.’, is simply the current directory so isn’t normally useful. The directory called ‘..’ is the directory above (i.e. closer to the root).

Optionally you can follow the `ls` command by a directory which you wish to view rather than the current directory.

4: Type the following:

```
ls /
```

In this example we are looking at the root (top) directory.

5: Type the following:

```
ls /etc
```

Now we are looking in the `etc` sub-directory of the root directory (this happens to be where system configuration files live). Specifying a directory or file starting with the root directory (`/`) is known as specifying the 'full path'. Each element of the full path of directories is separated with a `/`, so we could refer to a file such as `/etc/fstab` (see Figure 1).

4.2 Changing the current working directory

The `cd` ('Change Directory') command is used to change the current working directory.

6: Type the following:

```
cd /  
ls
```

This has the same effect as `ls /` but this time you have changed the current directory to the root directory and then obtained a listing.

To return to your home directory, type `cd` by itself.

7: Type the following:

```
cd  
ls
```

Note that the `~` is an abbreviation for your home directory so you can also:

8: Type the following:

```
cd ~  
ls
```

You will see why this is useful later.

You can find out which directory you are currently in by using the `pwd` command. This stands for 'Print Working Directory' and can be thought of as a 'where am I?' command:

9: Type the following:

```
pwd
```

The name of the home directory will depend on the system you are using. It might be something like `/home/andrew` or `/c/Users/andrew` or `/n/` if you are on a network drive.

Task:

Remembering that the directory above the current one is always available as `..`, how do you think you can move up one level in the tree of directories? Give it a go and use `pwd` to check where you are. Return to your home directory using `cd`.

5 Finding out more about files

We have already seen the `ls` command to list the files in a directory and the `-a` option (also referred to as a ‘flag’ or ‘switch’) that allows us to see hidden files. There are also other options to give more information.

- l Long format — give information about file sizes, permissions, etc. For the moment, don’t worry about the details other than the file size (in the 5th column), the date and time (in the 6th-8th columns) and the filename (at the end).
- t Sort by time — newest files first
- r Reverse the sort, so `-t -r` will give the newest file last which is often more useful
- h Give file sizes in a more human readable (kilobytes, gigabytes etc.)

These one-letter options can be combined into one as shown in this example:

10: Type the following:

```
ls -ltrh
```

Task:

Work out what the previous command has done. Play with `cd` and `ls` and its options to explore the files on your computer.

6 Using the shell


6.1 Recalling previous commands

You have probably found yourself typing the same, or very similar commands several times. Wouldn’t it be nice if you could recall those previous commands and modify them if required? Well you can.

- You can use the up-arrow on the keyboard to recall previous commands (and the down-arrow to step forward through previous commands),

- You can navigate through previous commands to edit them using the left- and right-arrow keys,
- You can use **Ctrl-a** (hold down the **Ctrl** key and press 'a') to move to the beginning of a line you have recalled or **Ctrl-e** to move to the end of a line.⁵

6.2 Typing less

The BASH shell has a feature known as 'command and file completion'. When you start to type a command, you can press the Tab key (typically an arrow with a vertical bar: ) to complete the command or filename. If you just type 'l' followed by Tab, nothing will happen as there are probably lots of commands starting with 'l'. If you press Tab a second time, BASH will list all the commands that start with 'l'. It may tell you that there are hundreds of possibilities and ask if you really want to see them. If there are so many that it can't display them all in one screen, it may say `--More--` at the bottom of the page; if so, press the spacebar to see the next page. You can add letters until you arrive at a unique start for a command. Obviously this isn't very useful for a 2-letter command like **ls** or **cd**, but with longer command names it can save a lot of typing.

The same applies to file and directory names — we will see an example in a moment.

6.3 Cut and Paste

Text can very easily be cut and paste between windows. Highlight a region of text by pressing the left mouse button at the start of the region of text, holding it down and dragging to the end of the region before releasing the button. To paste the text into another window, move the pointer to that window and click the right mouse button under Windows (git-bash) or the middle button on other systems. If your mouse (or trackpad) has only two buttons, pressing them both at once generally has the same effect as pressing the middle button.

7 Organizing files into subdirectories

Just as you probably have folders in your GUI file browser to organize your files, you can (and should) do the same from the command line creating directories and sub-directories.

Create a sub-directory called 'cli' in your home directory. The **mkdir** command is short for⁶ 'MaKe DIRectory':

11: Type the following:

```
cd
mkdir cli
```

⁵Using git-bash under Windows, these may not work correctly. Right-click in the top of the git-bash window to see the menu. Select **Properties** and then **Options**. Uncheck the **Enable Ctrl key shortcuts** option.

⁶When I say 'short for', I do **not** mean that you can also type the full piece of text.

Task:

Use `ls` to see your new sub-directory within your home directory. Navigate to your new directory using `cd` but try specifying the sub-directory name by just typing `c` and using the Tab key. Then look to see what is in there using `ls`. Are there any hidden files? Finally move back up to your home directory. What are the different ways of getting back to your home directory?

You may remember that `~` is an abbreviation for your home directory, so to get to the `cli` directory, rather than typing `cd` followed by `cd cli`, you can simply:

12: Type the following:

```
cd ~/cli
```

8 Grabbing files from the Internet and moving files around

8.1 Grabbing files

Let's grab a file to look at from the Internet. Note that this file (`pdbbs.tgz`) is only an example file. There are three ways of grabbing a file:

1. **We do not recommend this method**, but you can use your web browser to navigate to www.bioinf.org.uk/teaching/splats/ and then download the file `pdbbs.tgz` by clicking the `Example files` button and save it to the `cli` directory that you created from the command line — you should be able to navigate to this in the usual way in the file browser that appears when saving a file,
2. Using `curl`,
3. Using `wget`.

Depending on the system you are using you will have one (or both) of `curl` and `wget` installed — they are equivalent command line tools that allow you to grab files from the internet.

To use `curl`:

13: Type the following:

```
cd
curl -O http://www.bioinf.org.uk/teaching/splats/pdbbs.tgz
```

Note! That is a capital letter 'O', not the digit zero (0).

To use `wget`:

14: Type the following:

```
cd
wget http://www.bioinf.org.uk/teaching/splats/pdbbs.tgz
```

Task:

Having downloaded the file, do a directory listing to check that it is there and how big it is.

8.2 Moving files

When you looked at the directory listing you should have realized we made a (deliberate) mistake. We should have put the file in our `cli` directory not in our home directory.

We can use the `mv` (short for 'MoVe') command to move the file into that directory:

15: Type the following:

```
mv pdbc.tgz cli
```

The `mv` command can also be used to rename files — if the second parameter is a name that isn't a directory, the file (given in the first parameter) will be renamed to the name you specified. **Be careful!** If this second parameter is the name of another file that exists already, then that file will be over-written!

9 File archives

The file you have downloaded is in a format known as a 'gzipped tar file'. This is very similar to a ZIP file with which you are probably familiar. It is a compressed archive containing several other files all bundled together.

`tar` stands for 'Tape ARchiver' — it was originally used for performing backups to tape, but can also create (or unpack) archive files on disk.

Navigate to your `cli` directory and unpack the archive file.

16: Type the following:

```
cd ~/cli
tar xvzf pdbc.tgz
```

The four letters after the `tar` command tell it what to do:

x Extract files

v Be verbose (the program tells you what it's doing)

z The tar file has been compressed with a program called **gzip**

f The archive is a file (the name of which follows) rather than coming from a tape

These options can appear in any order except that the '**f**' must come last.

Task:

Find out what files or directories you have now. You should find a new directory called `pdbc` has been created, so change to that directory and see what is in there.

10 Looking at the content of a file

Make sure you are in the new directory that you have created by unpacking the archive. If you aren't there already:

17: Type the following:

```
cd ~/cli/pdbs
```

10.1 Displaying and concatenating

You should have a number of PDB files in the directory that you have just created. PDB files are used to store the structures of proteins and DNA, generally derived from X-ray crystallography or NMR.

The command to view a file is called **cat** — not the most obvious name, but there is a good reason for it as it can be used to 'conCATenate' (or join) two or more files together.

18: Type the following:

```
cat pdb9aat.ent
```

The content of the file should have been displayed (very quickly) on the screen. If we give more than one filename then it will display each file on the screen.

19: Type the following:

```
cat pdb9aat.ent pdb9abp.ent
```

This happens so fast that you don't really notice the two files.

If we wanted to join the two files together into a new file, we can use a **>** symbol to 'redirect' the output that went to the screen so it goes into a file instead:

20: Type the following:

```
cat pdb9aat.ent pdb9abp.ent > both.pdb
ls
```

When you do **ls** you should find that you now have a new file called `both.pdb` which will contain the content of the two files concatenated together.

Task:

Use the appropriate option to **ls** to obtain the sizes of the files and verify that the `both.pdb` file is the size of the other two files added together. (The file sizes may not add up exactly as the values may be rounded — particularly if you are using the human-readable format.)

Now we have finished with the `both.pdb` file, so we can delete or 'ReMove' it with the **rm** command:

21: Type the following:

```
rm both.pdb
```

Be very careful with the **rm** command — unlike a graphical file browser it doesn't put the file into a 'trashcan' from which it can be recovered. Once it's gone, it's gone!

If we wished to concatenate all the files that end in `.ent` we can use a `*` which matches any characters to save typing all the filenames⁷.

22: Type the following:

```
cat *.ent > all.pdb
```

Task:

Use **ls** to verify that the `all.pdb` file is of an appropriate size that it includes all the files and then remove the `all.pdb` file.

We won't look at it now, but you can also append to an existing file using redirection. In this case we simply use `>>` rather than `>`.

10.2 Copying files

Once you have a file, you often need to copy it — perhaps so you can keep a version before you make some changes. You may not realise it, but you have already seen a way you can do this using **cat** and redirection.

Task:

Think about how you could use **cat** to copy the `pdb9aat.ent` file to another name and give it a try. Delete the copied file once you have finished.

However there is an easier way of copying a file using the **cp** ('CoPy') command:

23: Type the following:

```
cp pdb9aat.ent 9aat.pdb
ls
```

Note that file extensions are not as important in Linux as they are in Windows. More precisely, file extensions are not so important when you are using the command line rather than a GUI to access files. When you click on a Word document in Windows or on a Mac, the GUI looks at the extension of the file (`.doc` or `.docx`) and knows to use the Word software to open the document. The file extension doesn't really say anything about the content of the file; rather it tells the GUI which program to use to open the file. In contrast, when you use the command line you always have to specify which program to use to access a file.

⁷We could also do something like `pdb*a*.ent` if we wanted only those files that have an 'a' in the PDB code and more complex patterns are also possible.

10.3 Looking for differences in files

Now we have two copies of the same file: one called `pdb9aat.ent` and one called `9aat.pdb`. If we have two files and want to know if they are the same, or perhaps we have modified one of them and want to know what the differences between the two versions are, we can use the **diff** ('DIFFerences') command to compare them.

24: Type the following:

```
diff pdb9aat.ent 9aat.pdb
```

This command should return nothing. This indicates that there are no differences between the files. If there had been differences, these would have been displayed; we will see an example in a moment.

10.4 Viewing big files

cat is all very well, but it's not very useful for big files as the content zooms past on the screen too quickly to see.

10.5 Heads or tails...

Often it is useful just to be able to see the top of a file where there might be some comments or identifying information. Suppose we are interested in the first 20 lines of the file `pdb9aat.ent` or the 'head' of the file:

25: Type the following:

```
head -20 pdb9aat.ent
```

The `-20` part says that we want the first 20 lines.

Similarly to see the end of the file, or its 'tail', we use the **tail** command:

26: Type the following:

```
tail -20 pdb9aat.ent
```

Task:

Remembering what you learned about using `>` for redirection, create a file called `pdb9aat.top` containing just the top 25 lines of `pdb9aat.ent`. Use **cat** to verify that this file contains what you expect.

10.6 Counting lines

Let's verify that there really are 25 lines in the file you just created. There is a command for doing a 'Word Count' with the rather unfortunate name **wc**.

27: Type the following:

```
wc pdb9aat.top
```

This provides three numbers representing the number of lines, number of words and number of characters. If you are only interested in the number of lines you can use the `-l` flag:

28: Type the following:

```
wc -l pdb9aat.top
```

There are equivalent `-c` and `-w` flags to display only the number of characters or words.

Task:

Extract the top 25 lines from `pdb9abp.ent` into a file. Use the `diff` command to look at the differences compared with the first 25 lines of `pdb9aat.ent` that you already stored in a file.

You should find a block of lines starting with a `<` — this means that these lines must be deleted from the first file you specified to create the second file; you will then see another block starting with a `>` which means that these lines must be added to the first file to create the second file.

10.7 More or less...

Even more useful would be to see a page of the file and then see some more by pressing a key. Surprisingly enough the command to do this is called `more`:

29: Type the following:

```
more pdb9aat.ent
```

(note that if you are using 'git-bash' on Windows, this may not work — the `more` command isn't available.)

Press the spacebar to move through the file and press the `'q'` key to quit.

The problem with `more` is that you can only move forward through the file. A better command is `less` which allows you to move in both directions:

30: Type the following:

```
less pdb9aat.ent
```

Press the spacebar to move forwards through the file a page at a time and `'b'` to move back up a page. Press the `'>'` sign to move to the end of the file or the `'<'` sign to move to the start.

You can also search for the next line containing some text using the `'/'` key. For example, `/ATOM` will take you to a line containing the word 'ATOM'. You can repeat the search and move to the next line containing the word of interest just by pressing the `'/'` key again.

Task:

Play with `less` to explore the file. Once you are finished, press `'q'` to quit.

11 Input and output: Redirection and pipes

11.1 Redirection

When you did the `diff` between the two files that you created each containing the top 20 lines of one of the PDB files, you should again have found that the results rushed past and you couldn't see the top of the file. You should already know one solution to that problem.

Task:

Run the `diff` command again, but use the `>` symbol to redirect the output to a file called `differences.txt`. Now use `less` to look at the file.

However this is not altogether convenient as you end up creating files that you will then delete almost immediately. Unix-like operating systems provide a concept known as 'pipes' that allow the output of one program (such as `diff`) to be fed directly into another program (such as `less`).

We have already seen the idea of 'redirection' of the output from a program. Many programs default to sending their output to what is known as 'standard output' (or `stdout`) and, by default this is the screen. In examples you have seen previously, you used `>` to redirect standard output to a file instead.

Many commands such as `less` will take input from a file specified on the command line, but if no file is specified, input will be taken from 'standard input' (`stdin`). By default, standard input is the keyboard, but we can use the `<` operator to redirect standard input to be a file.

For example, we previously saw the command `cat` to display a whole file on the screen:

31: Type the following:

```
cat pdb9aat.ent
```

The `cat` program has seen that there is a filename specified on the command line so knows that it should take its input from there. If you try typing `cat` by itself, it will expect input from standard input (the keyboard) instead.

32: Type the following:

```
cat
```

The `cat` program is now sitting waiting for input from the keyboard. So let's type a couple of lines and finish with `Ctrl-d` (press and hold the `Ctrl` key, press and release `'d'`, then release `Ctrl`) which is used to indicate the end of a file (i.e. no more text is coming from standard input).

33: Type the following:

```
hello
world
Ctrl-d
```


(Remember **Ctrl-d** does *not* mean that you type the letters **C,t,r,l,-,d** !)

Note what happened: **cat** reads from standard input (the keyboard) and writes to standard output (the screen).

A little variation of that is a very quick way of creating a file. Let's create a file called `helloworld.txt` containing the two lines: `Hello` and `World`:

34: Type the following:

```
cat > helloworld.txt
Hello
World
Ctrl-d
```

Task:

Check the file has been created and check its content with **cat**. Once you are done, remove the `helloworld.txt` file.

This is actually a very useful way of creating very small files quickly. To create larger files or modify the content of files, you use what is known as a 'text editor' which is like a word processor, but just creates simple plain text files without being able to change fonts, sizes, bold, italics, etc. We will not look at text editors in this session although Section 24.1 gives you a brief introduction to **emacs**, one of the most powerful text editors.

We can use the `<` operator to redirect standard input so that it comes from a file instead of from the keyboard.

35: Type the following:

```
cat < pdb9aat.ent
```

In practice this is exactly the same as typing `cat pdb9aat.ent` (without the `<`), but underneath something slightly different is happening.

11.2 Pipes

You can probably see how redirection to a file is very useful, but are probably wondering why redirection from a file is useful. In general it isn't used very much, but the concept of standard input is very important. Wouldn't it be wonderful if we could take the output of our **diff** program that created too much to see on the screen in one go and send it to the **less** program directly to be able to page through it rather than having to go via an intermediate file. In other words, we want to send the standard output of one program (which would normally go the screen) directly to the standard input (which would normally be the keyboard) of another program. Pipes let us do that. A pipe is created using the symbol `|` (which is probably bottom left on your keyboard, next to the left shift button).

Let's take a simple example first:

36: Type the following:

```
cat pdb9aat.ent | less
```

Here we are taking the standard output of `cat` and sending it to the standard input of `less`. This has exactly the same effect as typing `less pdb9aat.ent` or `less < pdb9aat.ent`, so is not particularly useful.

Here is a much more useful example where we have solved the problem of looking at the differences between the two files a page at a time without having to create an intermediate file:

37: Type the following:

```
diff pdb9aat.top pdb9abp.top | less
```

12 Removing directories

We have already seen how to create a directory using the `mkdir` command. How do we remove a directory?

First we will ensure we are in our `cli` directory and create a directory called `foo`⁸:

38: Type the following:

```
cd ~/cli
mkdir foo
ls
```

You should see your `foo` directory has been created.

To delete the directory we use `rmdir` ('ReMove DIRectory'):

39: Type the following:

```
rmdir foo
```

Now let's do the same but put a file in the directory:

40: Type the following:

```
cd ~/cli
mkdir foo
cp pdbc/pdb9aat.ent foo
rmdir foo
```

This time you will find that the system will refuse to delete a directory that has something in it. If you want to delete a directory and its contents, you use `rm` with the flag `-r` which means to delete recursively — i.e. work all the way down the directory tree from the `foo` directory deleting any files that are found and deleting the directories once they are empty. Obviously this is a **very dangerous command** so you must **use with caution!**

41: Type the following:

```
rm -r foo
```

⁸`foo` and `bar` are commonly used in the computing world as random names for junk or test files that we are going to delete quickly. If you want to know why, Google 'fubar'.

13 Getting more clever

In the associated lecture we talked about a basic algorithm for counting the number of amino acids in a PDB file. If we understand the power of BASH we don't actually need a program to do this; we can do it all using standard BASH commands. This is the real power of unix-like command-line shells rather than GUIs.

13.1 Extracting specific lines from a file: `grep`

The slightly oddly named `grep` command ('Globally search a Regular Expression and Print') allows us to search a file for lines that match a specified pattern and print them.

Let's look at one of our PDB files and see what it contains:

42: Type the following:

```
cd ~/cli/pdbs
less pdb9aat.ent
```

Page through the file using the space-bar and press 'q' to quit when you are done. You will see that each record (line) starts with a field (e.g. `HEADER. REMARK. SEQRES`) that says what the record contains. As you move through the file, you will find that the records that start with `ATOM` are the atomic coordinates. These are the records we need if we want to count the amino acids.

Using `grep` we can find all the records that contain the word `ATOM`:

43: Type the following:

```
grep ATOM pdb9aat.ent
```

Task:

Use your knowledge of 'pipes' to send the output of this `grep` command into `less` or into `head` to view the first 20 lines.

Looking at the beginning of the output from `grep`, you should notice that the first few lines do indeed contain `ATOM`, but not at the start of the line. We are only interested in lines that *start* with `ATOM`, so we tell `grep` to anchor the search to the start of a line using the 'caret' or 'hat' character (^):

44: Type the following:

```
grep ^ATOM pdb9aat.ent
```

Task:

Again, use your knowledge of pipes to send the output of this `grep` command into `less` or `head`

You should find that the command is now doing the right thing and only returning lines that start with `ATOM`.

Sending this output into `less` is all well and good, but what we eventually want to know is the number of amino acids. As a step towards that we can use the word count program that we used before (`wc`) to obtain the number of `ATOM` records which will be the total number of atoms in the file.

Task:

Use a pipe to send the output of this `grep` command into `wc` to count the lines.

You should find there are 6322 atoms.

Remembering from the lecture that we can count the amino acids by looking for the $C\alpha$ atom (which is called `CA` in the PDB files), we can send the output of our first `grep` command (that found the word `ATOM`) via a pipe into another `grep` command that looks for `CA` and then send that via another pipe into `wc` to count the number of amino acids. `grep` is another of these programs that will take input from standard input if a filename isn't specified, so your second `grep` command will simply be `grep CA` (with a pipe symbol either side of it).

Task:

Give it a try:

- Use the `grep` command to extract lines that start with `ATOM`
- Use a pipe to send the output into another `grep` command that finds lines containing `CA`
- Use a pipe to send the output into the `wc` command to count the lines.

You should find you have 802 amino acids.

If you wanted to find out the total number of amino acids in *all* the PDB files, you could simply replace the `pdb9aat.ent` with `*.ent`

Task:

Give it a try — you should find there are 2117 amino acids in total.

More advanced topics

14 Links or shortcuts

You are probably familiar with the concept of ‘shortcuts’ in Windows. These, for example, allow you to place an icon on your Desktop which links to a file that lives somewhere else on your computer.

In BASH you create a shortcut using the `ln -s` command (`ln` means ‘LiNk’ - the `-s` flag makes it a ‘symbolic’ link — don’t worry about what this means, just know that it’s pretty much the only sort you’ll ever need!). It is used in just the same way as the copy command (`cp`), but simply creates another name for the same file. Typically you would do this to create the same file in two different directories.

45: Type the following:

```
cd ~/cli
ln -s pdbc/pdb9aat.ent .
```

Here we have gone to the main `cli` directory and made a symbolic link from the file `pdbc9aat.ent` in the `pdbc` sub-directory to a file of the same name in the current directory (remember that `.` represents the current directory and `..` the directory above the current directory).

Note that the second parameter can be either a filename or a directory name. What we did here was the same as typing:

```
ln -s pdbc/pdb9aat.ent pdbc9aat.ent
```

because if you specify a directory name without a filename then the same filename as the input file will be assumed. We could also have typed:

```
ln -s pdbc/pdb9aat.ent ./pdbc9aat.ent
```

which would have had exactly the same effect.

15 Printing on the screen and environment variables

The `echo` command is used simply to type something on the screen. For example

46: Type the following:

```
echo "Hello world"
```

and you will see the text appear on the screen.

Like a normal programming language, BASH has a concept of variables where you can store information. In a shell like BASH these are known as ‘environment variables’ because they can be used to pass information about the environment in which a program is running into a program. The environment variable `$HOME` contains the location of your home directory, so you can use `echo` to find what this is:

47: Type the following:

```
echo $HOME
```

Note that the rule with environment variables is that you do not put a `$` in front of them when you are setting them, but you do put a `$` in front of them when you want the value they contain.

16 Aliases

Aliases allow us to create abbreviations for commonly used commands. For example we often find ourselves wanting to type `ls -ltrh` (i.e. a directory listing in 'long' format sorted by date in reverse order with file sizes in 'human' format). To reduce the number of characters we have to type, we often set up an alias of `ll`:

48: Type the following:

```
alias ll='ls -ltrh'
ll
```

However, we would have to do this every time we open a command line terminal which is not very convenient. BASH provides us with a mechanism to allow us to store a set of commands such as aliases that are executed every time we open a new BASH shell. Depending on the exact setup of your system, BASH reads a file called either `.bashrc` or `.profile` in your home directory⁹. Go to your home directory and see if either file exists:

49: Type the following:

```
cd
ls -a
```

If `.bashrc` exists then, being very careful about the single and double inverted commas and spaces:

50: Type the following:

```
cd
echo "alias ll='ls -ltrh' " >>.bashrc
```

Here we have used the redirect-and-append symbol (`>>`) to add this alias command onto the end of the `.bashrc` file.

If `.profile` exists then we simply specify this file instead of `.bashrc`:

51: Type the following:

```
cd
echo "alias ll='ls -ltrh' " >>.profile
```

If neither exists and since we don't know which is required by your system, we will put the command into `.bashrc` and then create `.profile` as a symbolic link to `.bashrc`

52: Type the following:

```
cd
echo "alias ll='ls -ltrh' " >>.bashrc
ln -s .bashrc .profile
```

Note that there is no harm in using `>>` even if the file does not exist already.

We might also want to add some other aliases. For example, if you are prone to deleting files by mistake, you might want to make `rm` an alias for `rm -i` which prompts to check you really want to delete a file:

⁹Remember that filenames starting with a `'.'` are hidden files and only seen when we do `ls -a`

53: Type the following:

```
cd
echo "alias rm='rm -i' " >>.bashrc
```

(Replace `.bashrc` with `.profile` if appropriate for your system.)

NOTE! The commands in your `.bashrc` or `.profile` file will not be run until you open a new terminal¹⁰.

17 Doing more complex things with pipes

Suppose we want a list of the amino acids used in a particular protein instead of counting the number of atoms, or the number of amino acids, as we did earlier. If we look at a PDB file we find that the amino acid name appears as the fourth field in the `ATOM` records¹¹. Do as we did before and extract the `ATOM` records for $C\alpha$ atoms to confirm this:

54: Type the following:

```
grep ^ATOM pdb9aat.ent | grep CA | less
```

If we want a simple list of the amino acids used in a particular protein, we want to look at this fourth field.

Unix-like systems provide a mini-programming language called **awk** (the letters stand for the three people who designed it: Aho, Weinberg and Kernigan). This is a very powerful language for doing simple things with files such as extracting fields, counting things, etc. Like the other commands we have seen it will take input from standard input if a file isn't specified, but we also need to give it instructions of what to do on the command line. These instructions have to be enclosed in single inverted commas.

Taking the output of our two piped **grep** commands, we can feed these into an **awk** command to extract the 4th field:

55: Type the following:

```
grep ^ATOM pdb9aat.ent | grep CA | awk '{print $4}'
```

In **awk** you have to put the command you want to be run on each line in curly brackets (`{}`) and the `$4` refers to the 4th field in a line. Consequently, in this case, we are printing the 4th field¹².

¹⁰Strictly it is the shell that needs to start not the terminal, so you could simply type **bash** to start another BASH shell in order to read the file. You could also type **source ~/.bashrc** (or **source ~/.profile** as appropriate) to read the commands in your current shell.

¹¹Actually this won't work for *every* PDB file as the PDB format uses fixed columns rather than space-separated fields. Thus it allows fields to expand so that they are touching each other without a separating space. For example, the third field of the `ATOM` records is the atom name which, strictly speaking is 4 characters wide, with the first character normally a space; the next field is also normally a space, but can sometimes be a letter A or B, so when the atom name is a space followed by 3 characters and the next field contains a letter, those two fields will be touching and also will touch the amino acid name in the next column.

¹²In the previous footnote it was pointed out that using **awk** wouldn't work on every PDB file since **awk** works with space-separated fields and sometimes the fields in a PDB file can merge into one. Another command, **cut**, can work with rigid columns that represent the characters across a line. The amino acid name is in columns 18–20, so we could replace the **awk** command with **cut -c 18–20**

While this gives us a list of all the amino acids in the sequence they are used, what we want is just a list of *which* amino acids are used. To achieve this we want to know which unique entries appear in the output. The first thing we can do is sort the output, using the **sort** command:

56: Type the following:

```
grep ^ATOM pdb9aat.ent | grep CA | awk '{print $4}' | sort
```

If we add the **-u** option to **sort** then it will produce a uniquely sorted list — i.e. if an entry in the list is repeated, it will show it only once:

57: Type the following:

```
grep ^ATOM pdb9aat.ent | grep CA | awk '{print $4}' | sort -u
```

Adding **-u** to **sort** is actually equivalent to sending the output from **sort** into another command **uniq**:

58: Type the following:

```
grep ^ATOM pdb9aat.ent | grep CA | awk '{print $4}' | sort | uniq
```

uniq is normally used when you have a list of items and want to remove repeats that are next to each other in the list, but you want to retain repeats that are separated by other items.

Task:

Send the output of this into **wc** to check whether all 20 amino acids have been used in `pdb9aat.ent`.

18 Programming in BASH

BASH itself is also a simple programming language that lets us automate tasks. Suppose we want to count the number of atoms in all of the PDB files — you already know how to do that for one PDB file at a time, so how would you do it for all of them?

We could count the number of *lines* in all the files very easily:

59: Type the following:

```
cd ~/cli/pdbs
wc -l *.ent
```

We could also count *all* the atoms across all the files very easily:

60: Type the following:

```
cd ~/cli/pdbs
cat *.ent | grep ^ATOM | wc -l
```

However, we want to do `grep ^ATOM xxxxxxxx.ent | wc -l` on each file in turn. We can use a **for** loop within BASH to repeat one or more commands on each of a set of files:

61: Type the following:

```
for file in *.ent
do
  echo -n "$file "
  grep ^ATOM $file | wc -l
done
```

Breaking this down:

1. The **for** loop steps through each file that matches the pattern `*.ent` in turn. For each one, it sets the environment variable `file` (note no `$` sign used at this stage¹³) to the name of the file and then runs the code between **do** and **done**
2. The **do** line simply marks the start of a set of commands to run on the current file
3. The **echo** command prints the name of the file (the content of the `$file` environment variable) followed by a space to the screen. We have seen **echo** before, but here have used a new flag `-n` which tells it *not* to add a newline after printing. In other words the next thing printed will appear on the same line.
4. The **grep** piped into **wc** counts the number of `ATOM` records in the file.
5. The **done** marks the end of the block of commands so that the **for** loop can step onto the next file.

Task:

Modify the code to count the number of amino acids in each file instead.

18.1 Creating multiple files

We can extend this idea to process a set of files in any way required. Using redirection we can send the output of each command to a file. For example:

62: Type the following:

```
for file in *.ent
do
  grep ^ATOM $file | wc -l > $file.atomcount
done
```

This version creates a file with the extension `.atomcount` for each `.ent` file. Here we are simply doing a trivial example of counting atoms — in a real problem, you would probably be running some more complex program, perhaps finding genes in a stretch of DNA and taking several minutes to run. When you do `ls` you will find that there are filenames like `pdb9aat.ent.atomcount`. As mentioned earlier, when using the command line, file extensions are arbitrary and you should choose something that is meaningful to you.

¹³Remember the rule with environment variables is that you do not put a `$` in front of them when you are setting them, but you do put a `$` in front of them when you want the value they contain.

18.2 Renaming a batch of files

However you might want to rename all the files so that they are called, for example, `pdb9aat.atomcount` rather than `pdb9aat.ent.atomcount`. We have already seen the `mv` command for renaming files, and we could do this for each file one at a time, but how can we rename a whole batch of files? Again we can use a `for` loop, but we need to introduce a new command to create the new filename. The `basename` command lets us do that. It is followed by a filename and an extension that you wish to remove.

63: Type the following:

```
basename pdb9aat.ent.atomcount .ent.atomcount
```

This will simply print `pdb9aat`. Somehow we wish to step through each file and do a `mv` of (for example) `pdb9aat.ent.atomcount` to the result of this `basename` command with new extension (`.atomcount`) appended. What we need to do is use the output of a command (`basename`) as a parameter to another command (`mv`). We can do that by enclosing the `basename` command in back-ticks (also known as an opening single inverted comma: ```) — note that both the character before `basename` and before `.atomcount` in this example must be a back-tick.

64: Type the following:

```
for file in *.ent.atomcount
do
  mv $file `basename $file .ent.atomcount`.atomcount
done
```

Do an `ls` to check the output.

Task:

Delete the `.atomcount` files and modify the code in Section 18.1 to generate the files with the `.atomcount` extension directly rather than generating them with the extension `.ent.atomcount` and then renaming them.

19 The `ls` long format and file permissions

When you did `ls -lh`, you obtained output that looked something like:

```
total 1.7M
-rw-r--r-- 1 localuser localuser 69K Jan 21 2016 pdb9ame.ent
-rw-r--r-- 1 localuser localuser 238K Jan 21 2016 pdb9abp.ent
-rw-r--r-- 1 localuser localuser 597K Jan 21 2016 pdb9aat.ent
-rw-r--r-- 1 localuser localuser 315K Jan 21 2016 pdb9atc.ent
-rw-r--r-- 1 localuser localuser 297K Jan 21 2016 pdb9api.ent
-rw-r--r-- 1 localuser localuser 212K Jan 21 2016 pdb9ant.ent
```

The first line shows the total size of the files. After that the lines are divided into 9 columns:

Column 1: These are the file ‘permissions’ — we will return to this in a moment

Column 2: This is the ‘link count’; it’s exact meaning is system and context dependent, but in general it will be 1 for a normal file. For a directory containing other directories it will be one plus the number of contained directories. In general you can ignore it!

Column 3: This is your user name (or a number representing you as a user).

Column 4: This is your group name (or a number representing your group). Each user is assigned to a group allowing several users to share files, but restrict access to other users.

Column 5: This is the file size. Since we used `-h` this is in a ‘human readable’ format; the default if you don’t use `-h` is the number of ‘blocks’ which is the minimum chunk of a disk that can be accessed (usually 512 bytes, but system dependent).

Column 6–8: The date on which the file was created. The last of these three fields will be the year if the file wasn’t created recently or the time if it was created recently.

Column 9: The filename

The file permissions are divided into 4 blocks:

Character 1: This is a dash for a normal file, the character ‘d’ for a directory, or the character ‘l’ for a symbolic link.

Characters 2–4: These are permissions for the ‘user’; in other words the owner of the file. They consist of three flags (or ‘bits’) which are the letters ‘rwx’ (in that order) where any of the characters may be replaced by a dash. If the `r` flag is set, then the owner of the file can read the file. If the `w` flag is set, then the owner of the file can write to the file. If the `x` flag is set, then the owner of the file can execute the file — in other words the file is a program or a script that can be run as a program.

Characters 5–7: These are permissions for the group to which the file belongs. The meanings are the same as for characters 2–4.

Characters 8–10: These are permissions for other people (i.e. anyone who isn’t the owner and isn’t in the file’s group). The meanings are the same as for characters 2–4.

The default permissions are system dependent, but usually the owner of a normal file (‘user’) can read and write to it while the group and ‘other’ permissions are generally read only. For a directory, the meaning of the ‘execute bit’ is slightly different — if it is set it allows people to do an `ls` in that directory¹⁴.

¹⁴There are also other special permission bits outside the scope of this tutorial.

20 Changing permissions

The **chmod** ('CHange MODe') command allows you to change the permissions on a file. For example, if you wish to stop other users from reading your PDB files, you would:

65: Type the following:

```
cd ~/cli/pdbs
ls -lh
chmod og-r *.ent
ls -lh
```

You should see that the permissions shown by the second **ls** have changed. In the **chmod** command, the **og** means 'others' and 'group' (**u** represents you, the user), the **-** means 'remove permissions' (**+** means 'add permissions') and the **r** refers to the 'read' permission.

Task:

1. Change the permissions to add back the read permission for others and the group.
2. Change the permissions to remove write permissions for yourself — this allows you to protect files so you can't delete or overwrite them accidentally.

21 Creating a reusable script

One of the permission flags we have just seen is the execute permission (**x**). This allows us to put a set of commands into a file and then type the name of this file to execute the commands it contains. These files are known as 'shell scripts'. Suppose we regularly need to check the number of amino acids in a set of PDB files in the current directory. We already saw how to count the number of atoms in a set of files in Section 18. We could easily modify that to count amino acids instead of atoms:

66: Type the following:

```
cd ~/cli/pdbs
for file in *.ent
do
echo -n "$file "
grep ^ATOM $file | grep CA | wc -l
done
```

We already know an easy way of putting commands into a file, so we will create file called `countaa.sh` — `.sh` is conventionally used as file extension for shell scripts. You need to be careful not to make any mistakes, so you might want to cut-and-paste this (but remember **Ctrl-d** is not typed as letters!):

67: Type the following:

```
cat >countaa.sh
for file in *.ent
do
echo -n "$file "
grep ^ATOM $file | grep CA | wc -l
done
Ctrl-d
```

We now need to set the execute bit on this script:

68: Type the following:

```
chmod a+x countaa.sh
```

The 'a' in **a+x** means that the execute bit should be set for all users.

You can now run the script:

69: Type the following:

```
./countaa.sh
```

Note that you need to tell the shell where to find the script — the './' part of the command tells the shell to look in the current directory. We will see how to get around this so we can just type **countaa.sh** in the next section.

22 Commands, programs and paths

Under Unix/Linux/Mac — and for that matter under Windows as well — commands issued at the command line are all programs that can live in pretty much any directory (folder) on the computer. When you type a command, the shell interprets what you write, finds the program and runs it. A few of the key commands are built into the shell, but most are separate programs.

Obviously you can have hundreds, or thousands, of different directories and it would take the shell ages to search all of them to find a command. Therefore the shell exploits an environment variable called `$PATH` to store a set of directories where it will look for commands. To find out what your path is:

70: Type the following:

```
echo $PATH
```

Typically this is something like:

```
/usr/bin:/bin:/usr/local/bin
```

(the `:` is used to separate each different directory in the path). In other words, if it doesn't find the command you gave is a built-in command, it will first look in `/usr/bin`, then in `/bin`, then in `/usr/local/bin`, etc.

You can modify the path using

```
export PATH="$PATH:/directory/I/want/to/add"
```

(where `/directory/I/want/to/add` is replaced by an actual directory and generally needs to be a full path including the full directory specification obtained by typing

pwd in that directory). Note that there must not be any spaces around the equals sign. You can put this command in your `$HOME/.bashrc` (or `$HOME/.profile`) file so that it works whenever you open a shell.

Often people will create a `bin` (for 'binary') directory within their home directory, add that to their path and put all their programs in there, ensuring that `$HOME/bin` is in their path. Consequently, we could put our `count.sh` shell script developed in Section 21 into our `$HOME/bin` directory and then simply type **countaa.sh** to run the script whatever directory we happen to be in.

23 Command Summary

awk *instructions filename* A special language for processing files including extracting fields from each line. If the filename is not specified, then input is taken from standard input.

basename *filename extension* Strip the extension from a filename.

cat *filename* Types a file to the screen. Can also be used to concatenate files.

cd *dir* 'Change Directory'. Typing **cd** on its own will take you back to your home directory. If you have created a sub-directory called, for example, `alignments`, you would move into that sub-directory by typing **cd alignments**. (See **mkdir** on how to create a sub-directory.) To move up a level in the hierarchy, type `cd ..`

chmod *mode filename* Change the permissions on a file

cp *file1 file2* Copy a file. *file1* must be the name of a file; *file2* may be another filename or a directory name.

curl **-O** *url* Grab a file from the internet.

cut **-c** *col1-col2 filename* Extract a column from a file. If the filename is not specified, then input is taken from standard input.

diff *file1 file2* View the differences between two files.

echo *text* Print text to the screen. The **-n** flag stops it starting a new line after printing.

grep *pattern filename* Search for a pattern in a file. If the filename is not specified, then input is taken from standard input.

gzip *filename* Used to compress a file.

gunzip *filename* Used to uncompress a file.

head *filename* Display the start of a file. Use a minus sign and a number to specify the number of lines. If the filename is not specified, then input is taken from standard input.

less filename Types a file to the screen, but pauses at the end of each screen full of text. Press the space-bar to view the next page or the **b** key to move back a page. Press the **q** key to quit without viewing the whole file. If the filename is not specified, then input is taken from standard input.

ls directory List the files in the specified directory. If no directory is specified, then list the files in the current directory. Options include **-l** to obtain a 'long' format listing, **-h** to have file sizes in 'human readable' format, **-t** to sort by time, **-r** to reverse the sort, **-a** to list all files (including hidden files)

mkdir dir Create a new sub-directory. For example, to create a sub-directory called *alignments*, use the command **mkdir alignments**. (See **cd** on how to navigate between sub-directories.)

more filename Page through a file. An older version of **less**, but only allows you to move forward through the file. If the filename is not specified, then input is taken from standard input.

mv file1 file2 Moves or renames a file or directory. *file1* is the name of the file or directory to be moved or renamed; *file2* may be a new filename or a directory to which you wish to move the first file or directory.

pwd 'Print Working Directory'. Shows you which directory/sub-directory you are currently in.

rm filename Removes (deletes) a file. Use with the **-r** flag to delete a directory and all its contents recursively.

rmdir directory Remove an empty directory.

sort filename Sorts a file. Various options include **-n** to do a numeric rather than an alphabetical sort and **-k** to specify a column on which to sort. If the filename is not specified, it reads from standard input.

ssh machine Allows you to log into another machine. The connection is secure so that passwords, etc., are not passed around the network in plain text.

tail filename Display the end of a file. Use a minus sign and a number to specify the number of lines. If the filename is not specified, then input is taken from standard input.

tar Used to create or unpack an archive containing several files. Typical usage: **tar zcvf filename.tgz directory** to create an archive called *filename.tgz* of the directory, *directory* and its contents; **tar zxvf filename.tgz** to unpack an archive called *filename.tgz*

uniq filename Remove repeated lines from a file. If the filename is not specified, then input is taken from standard input.

wc filename Count the lines, words and characters in a file. **-l** only count lines; **-w** only count words; **-c** only count characters. If the filename is not specified, then input is taken from standard input.

wget url Grab a file from the internet.

24 Text Editors

24.1 Emacs

Emacs is an incredibly powerful text editor. It is not a wordprocessor so does not embed formatting information in the document, just plain simple text. You start emacs by typing

```
emacs filename
```

where *filename* is the file to be edited.

Emacs allows you to move around using the arrow keys and to type to insert characters at cursor location. Anything more complex is handled through ‘control-key’ sequences, ‘extended control-key’ sequences, or ‘meta’ key sequences.

‘Control-key’ sequences are of the form **Ctrl-s** (i.e. press and hold the **Ctrl** key and simultaneously press the letter ‘s’). These are used for the most common commands.

‘Meta’ key sequences require you first to press the **Esc** key and then to press a letter. On some terminals you can use the **Alt** key instead and this then works like the **Ctrl** key (i.e. hold it down while you press the other key. These are generally either ‘bigger’ versions of control-key sequence (e.g. move forward a word rather than a letter) or to reverse a control-key sequence (e.g. move up a screen rather than down a screen). In descriptions below these will be described as **Meta-x** which means press and release the **Esc** key then press and release the ‘x’ key. Alternatively, if your terminal supports it you may press and hold the **Alt** key and simultaneously press the ‘x’ key.

‘Extended control-key’ sequences are always of the form **Ctrl-x** followed by some other control-key sequence. These are used for the more uncommon commands such as saving a file and exiting from emacs.

As a last resort some commands are not bound to keys at all (the whole system is configurable and you can map any key combination to do anything — you can even remap all the normal letter keys to insert different letters!). These are accessed by doing **Meta-x** followed by the command name.

Here is a summary of some of the most useful commands:

Exit emacs: **Ctrl-x Ctrl-c**

Save the contents of the buffer: **Ctrl-x Ctrl-s**

Move down a page: **Ctrl-v**

Move up a page: **Meta-v**

Search for a string: **Ctrl-s** Note that this is an ‘incremental search’ — it will start to find matches as soon as you type any characters. To search again, press **Ctrl-s** again.

Search backwards for a string: **Ctrl-r**

Delete character to the right of the cursor: **Ctrl-d**

Kill (remove) a line: **Ctrl-k** Kills the line from under the cursor to the end of the line on the right, press a second time to remove the end-of-line marker and move the following line up. If you repeatedly press **Ctrl-k** more lines will be killed. All lines are saved in a buffer and can be ‘yanked’ back.

Yank back a set of deleted text: **Ctrl-y**

Delete a block of text: Of course you can use **Ctrl-k** and the delete key, but it is often easier to delete a marked block — especially if you have a lot to delete or you want to yank it back later. First you set a ‘mark’ at one end of the block — do this with **Ctrl-@**. Now move to where you want to end the block. You can use a combination of screen-at-a-time moves, searches, etc. Finally press **Ctrl-w** to ‘wipe’ the marked block. (You can yank it back as before with **Ctrl-y**.)

Interactive Tutorial: New users may find it useful to work through the emacs interactive tutorial. This can be accessed by pressing **Ctrl-h t**.

25 Getting help

Manual pages for all commands should be available on your computer. Just type **man command** (short for ‘manual’) to read the manual page for *command*. If you don’t know what command you are looking for you can use `man -k keyword` to list all manual pages containing keyword. For example, to read the man pages for the text editor **emacs** type:

```
man emacs
```

If they aren’t available on your computer, you can search online for man pages. For example, search for ‘emacs man page’.

26 Other Tutorials

You are also recommended to look at some of these for more information:

- <https://www.youtube.com/watch?v=w97NDZEf-yA>
— A video of my 2016 lecture/tutorial session on computing for Bioinformatics and using the Bash command line
- <http://www.ee.surrey.ac.uk/Teaching/Unix/>
— An excellent set of tutorials from the University of Surrey
- <http://linuxcommand.org/>
— Linux Command Line tutorial
- <https://www.codecademy.com/learn/learn-the-command-line>
— Linux Command Line tutorial