Unix Primer for Biologists

Keith Bradnam, Ian Korf & Richard Smith-Unna

Version 4 — November 2014

Contents

Introduction

Why Unix?

The Unix operating system has been around since 1969. Back then there was no such thing as a graphical user interface. You typed everything. It may seem archaic to use a keyboard to issue commands today, but it's much easier to automate keyboard tasks than mouse tasks. There are several variants of Unix (including Linux), though the differences do not matter much. Though you may not have noticed it, Apple has been using Unix as the underlying operating system on all of their computers since 2001.

Increasingly, the raw output of biological research exists as *in silico* data, usually in the form of large text files. Unix is particularly suited to working with such files and has several powerful (and flexible) commands that can process your data for you. The real strength of learning Unix is that most of these commands can be combined in an almost unlimited fashion. So if you can learn just five Unix commands, you will be able to do a lot more than just five things.

Typeset Conventions

All of the Unix code in these guides is written in constant-width font with line numbering. Here is an example with 3 lines:

```
1. for var in 0 1 2 3 4 5 6 7 8 9 do
3.    echo $var
4. done
```

Text you are meant to type into a terminal is indented in constant-width font without line numbering. Here is an example:

```
ls -lrh
```

Sometimes a paragraph will include a reference to a Unix command, or a file that you should be working with, Any such text will be in a constant-width, boxed font. E.g.

Type the `pwd` command again.

From time to time this documentation will contain web links to pages that will help you find out more about certain Unix commands and Perl functions. Usually, the *first* mention of a command or function will be a hyperlink to Wikipedia. Important or critical points will be styled like so:

> ***This is an important point!***

Part 1: Unix - Learning the essentials

Introduction to Unix

These exercises will (hopefully) teach you to become comfortable when working in the environment of the Unix terminal. Unix contains many hundred of commands but you will probably use just 10 or so to achieve most of what you want to do.

You are probably used to working with programs like the Apple Finder or the Windows File Explorer to navigate around the hard drive of your computer. Some people are so used to using the mouse to move files, drag files to trash etc. that it can seem strange switching from this behavior to typing commands instead. Be patient, and try as much as possible to stay within world of the Unix terminal. Please make sure you complete and understand each task before moving on to the next one.

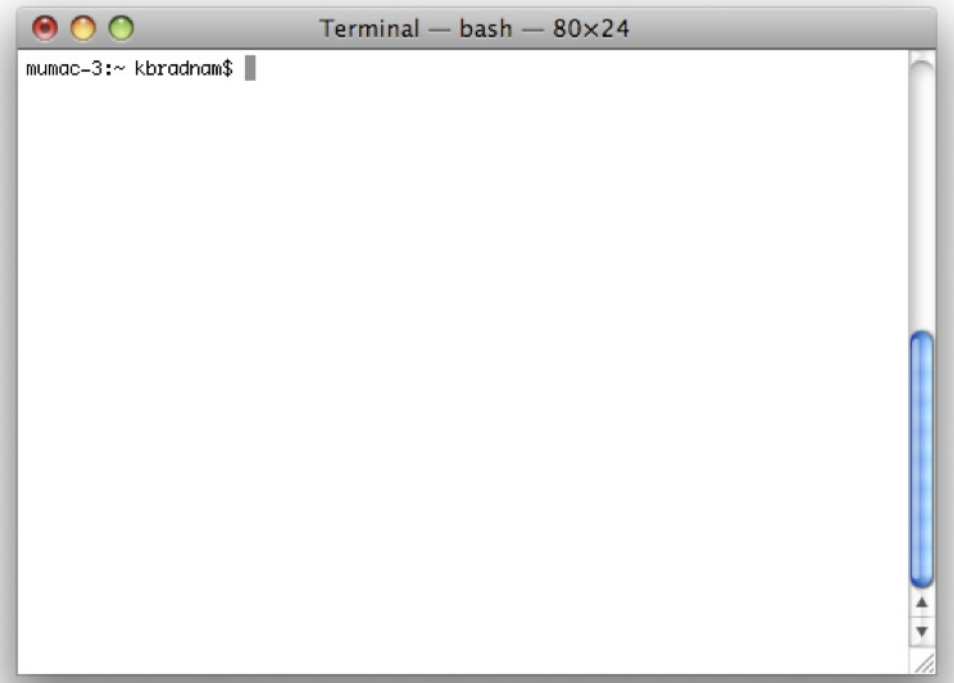
U1. The Terminal

A 'terminal' is the common name for the program that does two main things. It allows you to type input to the computer (i.e. run programs, move/view files etc.) and it allows you to see output from those programs. All Unix machines will have a terminal program, usually unsurprisingly named 'Terminal'.

Task U1.1

Open the Terminal app.

You should now see something that looks like the following (any text that appears inside your terminal window will look different):



Before we go any further, you should note that you can:

- make the text larger/smaller (hold down `ctrl` and either `+` or `-`)
- resize the window (this will often be necessary)
- have multiple terminal windows on screen
- have multiple tabs open within each window

There will be many situations where it will be useful to have multiple terminals open and it will be a matter of preference as to whether you want to have multiple windows, or one window with multiple tabs (there are keyboard shortcuts for switching between windows, or moving between tabs).

Connecting to the server

Now, we will connect to the HPC server at ICIPE.

Click inside the Terminal window and then enter the following: `ssh user01@hpc01.icipe.org` and press enter, replacing '01' with your desk number.

You should see something like:

```
$ ssh user01@hpc01.icipe.org
user01@hpc01.icipe.org's password:
```

Enter the password (password01, where '01' is your desk number) and press enter again. Now you are logged into the server.

U2. Your first Unix command

Unix keeps files arranged in a hierarchical structure. From the 'top-level' of the computer, there will be a number of directories, each of which can contain files and subdirectories, and each of those in turn can of course contain more files and directories and so on, ad infinitum. It's important to note that you will always be 'in' a directory when using the terminal. The default behavior is that when you open a new terminal you start in your

own ‘home’ directory (containing files and directories that only you can modify).

To see what files are in our home directory, we need to use the ls command. This command “lists’ the contents of a directory. So why don’t they call the command ‘list’ instead? Well, this is a good thing because typing long commands over and over again is tiring and time-consuming. There are many (frequently used) Unix commands that are just two or three letters. If we run the `ls` command we should see something like:

```
[user01@hpc01 ~]$ ls
Application Shortcuts   Documents   Library
Desktop                 Downloads
[user01@hpc01 ~]$
```

There are four things that you should note here:

1. You will probably see different output to what is shown here, it depends on your computer. Don’t worry about that for now.
2. The `[user01@hpc01 ~]$` text that you see is the Unix command prompt. It contains a user name (kbradnam), the name of the machine that this user is working on (“olson27-1’ and the name of the current directory (“~’ more on that later). Note that the command prompt might not look the same on different Unix systems. In this case, the $ sign marks the end of the prompt.
3. The output of the `ls` command lists five things. In this case, they are all directories, but they could also be files. We’ll learn how to tell them apart later on.
4. After the `ls` command finishes it produces a new command prompt, ready for you to type your next command.

The `ls` command is used to list the contents of *any* directory, not necessarily the one that you are currently in. Plug in your USB drive, and type the following:

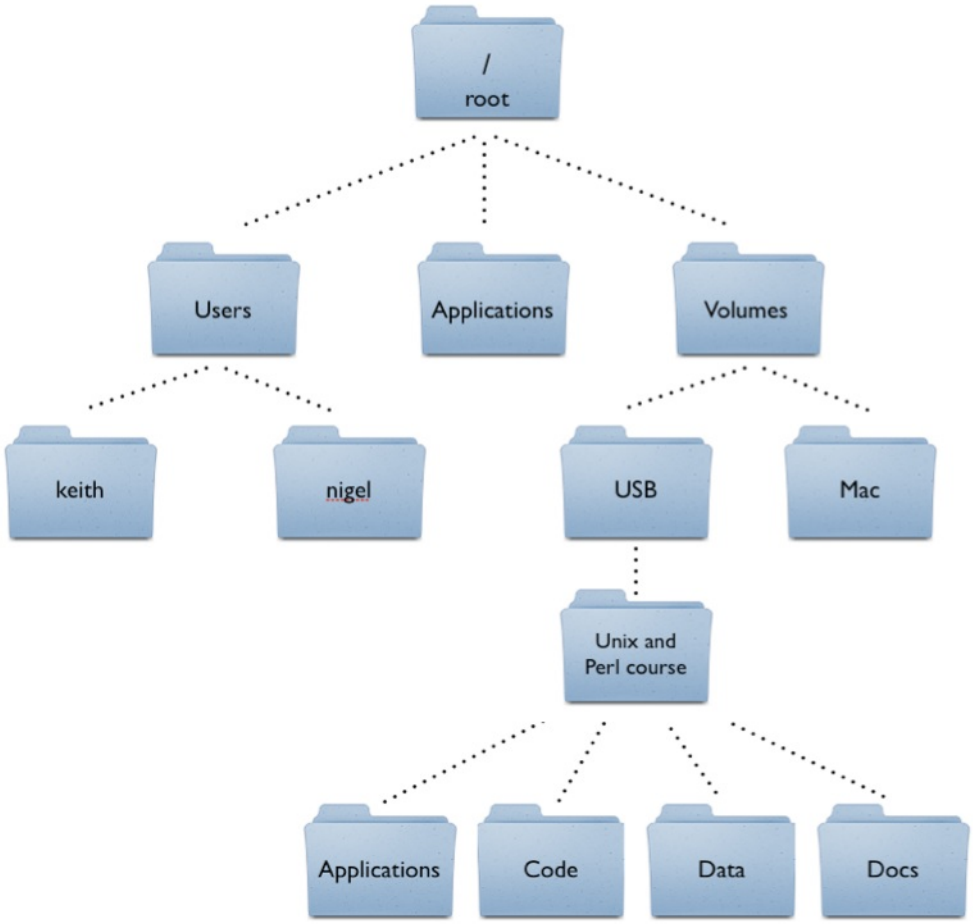
```
[user01@hpc01 ~]$ ls ~/Unix.Course/Data
Applications    Code        Data        Documentation
```

On a Mac, plugged in drives appear as subdirectories in the special ‘Volumes’ directory. The name of the USB flash drive is ‘USB’. The above output shows a set of four directories that are all “inside” the ‘Unix_and_Perl_course’ directory). Note how the underscore character ‘_’ is used to space out words in the directory name.

U3: The Unix tree

Looking at directories from within a Unix terminal can often seem confusing. But bear in mind that these directories are exactly the same type of folders that you can see if you use a graphical file-management program. A tree analogy is often used when describing computer filesystems. From the root level (/) there can be one or more top level directories. In the example below, we show just three. When you log in to a computer you are working with your files in your user directory, and this will nearly always be inside a ‘home’ directory. On many computers there will be multiple users.

It will help to think of this tree when we come to copying and moving files. E.g. in the tree depicted below if we had a file in the ‘Code’ directory and wanted to copy it to the ‘keith’ directory, we would have to go *up* four levels to the root level, and then *down* two levels.



U4: Finding out where you are

There may be many hundreds of directories on any Unix machine, so how do you know which one you are in? The command pwd will Print the Working Directory and that's pretty much all this command does:

```
[user01@hpc01 ~]$ pwd
/home/user01
```

When you log in to a Unix computer, you are typically placed into your *home* directory. In this example, after we log in, we are placed in a directory called 'user01' which itself is a subdirectory of another directory called 'users'. Conversely, 'home' is the parent directory of 'user01'. The first forward slash that appears in a list of directory names always refers to the top level directory of the file system (known as the root directory). The remaining forward slash (between 'users' and 'clmuser') delimits the various parts of the directory hierarchy. If you ever get 'lost' in Unix, remember the `pwd` command.

As you learn Unix you will frequently type commands that don't seem to work. Most of the time this will be because you are in the wrong directory, so it's a really good habit to get used to running the `pwd` command a lot.

U5: Getting from 'A' to 'B'

We are in the home directory on the computer but we want to to work on the USB drive. To change directories in Unix, we use the cd command:

```
[user01@hpc01 ~]$ cd ~/Unix.Course/Data
```

```
[user01@hpc01 Data]$ ls
Applications    Code        Data        Documentation
[user01@hpc01 Data]$ pwd
~/Unix.Course/Data
```

The first command reads as “change directory to the Unix.Course directory that is inside a directory called ‘~’, which itself is a shortcut to the ‘user01’ directory, which is inside the ‘home’ directory”. Did you notice that the command prompt changed after you ran the `cd` command? The ‘~’ sign should have changed to ‘Unix.Course’. This is a useful feature of the command prompt. By default it reminds you where you are as you move through different directories on the computer.

> ***NB. For the sake of clarity, we will now simplify the command prompt to ‘$’ in all of the following examples***

U6: Root is the root of all evil

In the previous example, we could have achieved the same result in three separate steps:

```
$ cd /home
$ cd user01
$ cd Unix.Course
```

Note that the second and third commands do not include a forward slash. When you specify a directory that starts with a forward slash, you are referring to a directory that should exist one level below the root level of the computer. What happens if you try the following two commands? The first command should produce an error message.

```
$ cd home
$ cd /home
```

The error is because without including a leading slash, Unix is trying to change to a ‘home’ directory below your current level in the file hierarchy (~/Unix.Course/Data), and there is no directory called home at this location.

U7: Up, up, and away

Frequently, you will find that you want to go ‘upwards’ one level in the directory hierarchy. Two dots `..` are used in Unix to refer to the *parent* directory of wherever you are. Every directory has a parent except the root level of the computer:

```
$ cd ~/Unix.Course/Data
$ pwd
~/Unix.Course/Data
$ cd ..
$ pwd
/home/user01
```

What if you wanted to navigate up *two* levels in the file system in one go? It’s very simple, just use two sets of the `..` operator, separated by a forward slash:

```
$ cd ~/Unix.Course/Data
$ pwd
~/Unix.Course/Data
$ cd ../..
$ pwd
/Volumes
```

U8: This is all relative

Using `cd ..` allows us to change directory *relative* to where we are now. You can also always change to a directory based on its *absolute* location. E.g. if you are working in the `~/Unix.Course/Data/Code` directory and you then want to change to the `~/Unix.Course/Data/Data` directory, then you could do either of the following:

```
$ cd ../Data
```

or…

```
$ cd ~/Unix.Course/Data
```

They both achieve the same thing, but the 2nd example requires that you know about the full *path* from the root level of the computer to your directory of interest (the 'path' is an important concept in Unix). Sometimes it is quicker to change directories using the relative path, and other times it will be quicker to use the absolute path.

U9: Time to go home

Remember that the command prompt shows you the name of the directory that you are currently in, and that when you are in your home directory it shows you a tilde character (~) instead? This is because Unix uses the tilde character as a short-hand way of specifying a home directory.

Task U9.1

See what happens when you try the following commands (use the `pwd` command after each one to confirm the results):

```
$ cd /
$ cd ~
$ cd /
$ cd
```

Hopefully, you should find that `cd` and `cd ~` do the same thing, i.e. they take you back to your home directory (from wherever you were). Also notice how you can specify the single forward slash to refer to the root directory of the computer. When working with Unix you will frequently want to jump straight back to your home directory, and typing `cd` is a very quick way to get there.

U10: Making the `ls` command more useful

The `..` operator that we saw earlier can also be used with the `ls` command. Can you see how the following command is listing the contents of the root directory? If you want to test this, try running `ls /` and see if the output is any different.

```
$ cd ~/Unix.Course/Data
$ ls ../../..
Applications    Volumes     net
CRC             bin         oldlogins
Developer       cores       private
Library         dev         sbin
Network         etc         tmp
Server          home        usr
System          mach_kernel var
Users           mach_kernel.ctfsys
```

The `ls` command (like most Unix commands) has a set of options that can be added to the command to change the results. Command-line options in Unix are specified by using a dash ('-') after the command name followed by various letters, numbers, or words. If you add the letter "l' to the `ls` command it will give you a "longer' output compared to the default:

```
$ ls -l ~/Unix.Course/Data
total 192
drwxrwxrwx  1 keith  staff  16384 Oct  3 09:03 Applications
drwxrwxrwx  1 keith  staff  16384 Oct  3 11:11 Code
drwxrwxrwx  1 keith  staff  16384 Oct  3 11:12 Data
drwxrwxrwx  1 keith  staff  16384 Oct  3 11:34 Documentation
```

For each file or directory we now see more information (including file ownership and modification times). The 'd' at the start of each line indicates that these are directories

Task U10.1

There are many, many different options for the ls command. Try out the following (against any directory of your choice) to see how the output changes.

```
ls -l
ls -R
ls -l -t -r
ls -lh
```

Note that the last example combine multiple options but only use one dash. This is a very common way of specifying multiple command-line options. You may be wondering what some of these options are doing. It's time to learn about Unix documentation….

U11: Man your battle stations!

If every Unix command has so many options, you might be wondering how you find out what they are and what they do. Well, thankfully every Unix command has an associated 'manual' that you can access by using the `man` command. E.g.

```
$ man ls
$ man cd
$ man man
```

Yes, even the man command has a manual page!

When you are using the man command, press `space` to scroll down a page, `b` to go back a page, or `q` to quit. You can also use the up and down arrows to scroll a line at a time. The man command is actually using another Unix program, a text viewer called `less`, which we'll come to later on.

Some Unix commands have very long manual pages, which might seem very confusing. It is typical though to always list the command line options early on in the documentation, so you shouldn't have to read too much in order to find out what a command-line option is doing.

U12: Making directories

If we want to make a new directory (e.g. to store some work related data), we can use the mkdir command:

```
$ cd ~/Unix.Course/Data
$ mkdir Work
$ ls
Applications    Code        Data        Documentation   Work
$ mkdir Temp1
$ cd Temp1
$ mkdir Temp2
$ cd Temp2
$ pwd
~/Unix.Course/Data/Temp1/Temp2
```

In the last example we created the two temp directories in two separate steps. If we had used the `-p` option of the `mkdir` command we could

have done this in one step. E.g.

```
$ mkdir -p Temp1/Temp2
```

Task U12.1

Practice creating some directories and navigating between them using the `cd` command. Try changing directories using both the *absolute* as well as the *relative* path.

U13: Time to tidy up

We now have a few (empty) directories that we should remove. To do this use the rmdir command, this will only remove empty directories so it is quite safe to use. If you want to know more about this command (or any Unix command), then remember that you can just look at its man page (`man rmdir`).

```
$ cd ~/Unix.Course/Data
$ rmdir Work
```

Task U13.1

Remove the remaining empty Temp directories that you have created

---

U14: The art of typing less to do more

Saving keystrokes may not seem important, but the longer that you spend typing in a terminal window, the happier you will be if you can reduce the time you spend at the keyboard. Especially, as prolonged typing is not good for your body.

So the best Unix tip to learn early on is that you can tab complete the names of files and programs on most Unix systems. Type enough letters that uniquely identify the name of a file, directory or program and press tab… Unix will do the rest. E.g. if you type 'tou' and then press tab, Unix will autocomplete the word to `touch` (which we will learn more about in a minute). In this case, tab completion will occur because there are no other Unix commands that start with 'tou'.

If pressing tab doesn't do anything, then you have not have typed enough unique characters. In this case pressing tab *twice* will show you all possible completions. This trick can save you a LOT of typing… if you don't use tab-completion then you must be a masochist.

Task U14.1

Navigate to your home directory, and then use the `cd` command to change to the `~/Unix.Course/Data/Code/` directory. Use tab completion for each directory name. This should only take 13 key strokes compared to 24 if you type the whole thing yourself.

Another great time-saver is that Unix stores a list of all the commands that you have typed in each login session. You can access this list by using the history command or more simply by using the up and down arrows to access anything from your history. So if you type a long command but make a mistake, press the up arrow and then you can use the left and right arrows to move the cursor in order to make a change.

U15: You *can* touch this

The following sections will deal with Unix commands that help us to work with files, i.e. copy files to/from places, move files, rename files, remove

files, and most importantly, look at files. Remember, we want to be able to do all of these things without leaving the terminal. First, we need to have some files to play with. The Unix command touch will let us create a new, empty file. The touch command does other things too, but for now we just want a couple of files to work with.

```
$ cd ~/Unix.Course/Data
$ touch heaven.txt
$ touch earth.txt
$ ls
Applications Code   Data    Documentation   earth.txt     heaven.txt
```

U16: Moving heaven and earth

Now, let's assume that we want to move these files to a new directory ('Temp'). We will do this using the Unix mv (move) command:

```
$ mkdir Temp
$ mv heaven.txt Temp/
$ mv earth.txt Temp/
$ ls
Applications   Code     Data    Documentation   Temp
$ ls Temp/
earth.txt heaven.txt
```

For the `mv` command, we always have to specify a source file (or directory) that we want to move, and then specify a target location. If we had wanted to we could have moved both files in one go by typing any of the following commands:

```
$ mv *.txt Temp/
$ mv *t Temp/
$ mv *ea* Temp/
```

The asterisk `*` acts as a wild-card character, essentially meaning "match anything'. The second example works because there are no other files or directories in the directory that end with the letters 't' (if there was, then they would be copied too). Likewise, the third example works because only those two files contain the letters 'ea' in their names. Using wild-card characters can save you a lot of typing.

Task U16.1

Use `touch` to create three files called 'fat', 'fit', and "feet' inside the Temp directory. I.e.

```
$ cd Temp
$ touch fat fit feet
```

Then type either `ls f?t` or `ls f*t` and see what happens. The ? character is also a wild-card but with a slightly different meaning. Try typing `ls f??t` as well.

U17: Renaming files

In the earlier example, the destination for the `mv` command was a directory name (Temp). So we moved a file from its source location to a target location ('source' and 'target' are important concepts for many Unix commands). But note that the target could have also been a (different) file name, rather than a directory. E.g. let's make a new file and move it whilst renaming it at the same time:

```
$ touch rags
$ ls
Applications    Code        Data    Documentation   Temp  rags
$ mv rags Temp/riches
$ ls Temp/
earth.txt      heaven.txt     riches
```

In this example we create a new file ('rags') and move it to a new location and in the process change the name (to 'riches'). So `mv` can rename a file as well as move it. The logical extension of this is using `mv` to rename a file without moving it (you have to use `mv` to do this as Unix does not have a separate 'rename' command):

```
$ mv Temp/riches Temp/rags
$ ls Temp/
earth.txt      heaven.txt     rags
```

U18: Stay on target

It is important to understand that as long as you have specified a 'source' and a 'target' location when you are moving a file, then it doesn't matter what your current directory is. You can move or copy things within the same directory or between different directories regardless of whether you are "in" any of those directories. Moving directories is just like moving files:

```
$ mkdir Temp2
$ ls
Applications    Code        Data    Documentation   Temp  Temp2
$ mv Temp2 Temp/
$ ls Temp/
Temp2       earth.txt       heaven.txt  rags
```

This step moves the Temp2 directory inside the Temp directory.

Task U18.1

Create another Temp directory (Temp3) and then change directory to your home directory (/home/user01). **Without** changing directory, move the Temp3 directory to inside the ~/Unix.Course/Data/Temp directory.

U19: Here, there, and everywhere

The philosophy of "not having to be in a directory to do something in that directory", extends to just about any operation that you might want to do in Unix. Just because we need to do something with file X, it doesn't necessarily mean that we have to change directory to wherever file X is located. Let's assume that we just want to quickly check what is in the Data directory before continuing work with whatever we were previously doing in ~/Unix.Course/Data. Which of the following looks more convenient:

```
$ cd Data
$ ls
Arabidopsis     C_elegans   GenBank     Misc        Unix_test_files
$ cd ..
```

or…

```
$ ls Data/
Arabidopsis     C_elegans   GenBank     Misc        Unix_test_files
```

In the first example, we change directories just to run the ls command, and then we change directories back to where we were again. The second example shows how we could have just stayed where we were.

U20: To slash or not to slash?

Task U20.1

Run the following two commands and compare the output

```
$ ls Documentation

$ ls Documentation/
```

The two examples are not quite identical, but they produce identical output. So does the trailing slash character in the second example matter? Well not really. In both cases we have a directory named "Documentation' and it is optional as to whether you include the trailing slash. When you tab complete any Unix directory name, you will find that a trailing slash character is automatically added for you. This becomes useful when that directory contains subdirectories which you also want to tab complete.

I.e. imagine if you had to type the following (to access a buried directory "ggg') and tab-completion *didn't* add the trailing slash characters. You'd have to type the seven slashes yourself.

```
$ cd aaa/bbb/ccc/ddd/eee/fff/ggg/
```

U21: The most dangerous Unix command you will ever learn!

You've seen how to remove a directory with the `rmdir` command, but `rmdir` won't remove directories if they contain any files. So how can we remove the files we have created (in ~/Unix.Course/Data/Temp)? In order to do this, we will have to use the rm (remove) command.

> ***Please read the next section VERY carefully. Misuse of the rm command can lead to needless death & destruction (...of files)***

Potentially, `rm` is a very dangerous command; if you delete something with `rm`, you will not get it back! It does not go into the trash or recycle can, it is permanently removed. It is possible to delete everything in your home directory (all directories and subdirectories) with `rm`, that is why it is such a dangerous command.

Let me repeat that last part again. It is possible to delete EVERY file you have ever created with the `rm` command. Are you scared yet? You should be. Luckily there is a way of making `rm` a little bit safer. We can use it with the `-i` command-line option which will ask for confirmation before deleting anything:

```
$ pwd
~/Unix.Course/Data/Temp
$ ls
Temp2      Temp3      earth.txt   heaven.txt  rags
$ rm -i earth.txt
remove earth.txt? y
$ rm -i heaven.txt
remove heaven.txt? y
```

We could have simplified this step by using a wild-card (e.g. `rm -i *.txt`).

Task U21.1

Remove the last file in the Temp directory ("rags') and then remove the two empty directories (Temp 2 & Temp3).

U22: Go forth and multiply

Copying files with the cp (copy) command is very similar to moving them. Remember to always specify a source and a target location. Let's create a new file and make a copy of it.

```
$ touch file1
```

```
$ cp file1 file2
$ ls
file1   file2
```

What if we wanted to copy files from a different directory to our current directory? Let's put a file in our home directory (specified by '~' remember) and copy it to the USB drive:

```
$ touch ~/file3
$ ls
file1   file2
$ cp ~/file3 .
$ ls file1 file2 file3
```

This last step introduces another new concept. In Unix, the current directory can be represented by a '.' (dot) character. You will mostly use this only for copying files to the current directory that you are in. But just to make a quick point, compare the following:

```
$ ls
$ ls .
$ ls ./
```

In this case, using the dot is somewhat pointless because `ls` will already list the contents of the current directory by default. Also note again how the trailing slash is optional.

Let's try the opposite situation and copy these files back to the home directory (even though one of them is already there). The default behavior of copy is to overwrite (without warning) files that have the same name, so be careful.

```
$ cp file* ~/
```

Based on what we have already covered, do you think the trailing slash in "~/' is necessary?

U23: Going deeper and deeper

The `cp` command also allows us (with the use of a command-line option) to copy entire directories (also note how the `ls` command in this example is used to specify multiple directories):

```
$ mkdir Storage
$ mv file* Storage/
$ ls
Storage
$ cp -R Storage Storage2
$ ls Storage Storage2
Storage:
file1   file2   file3

Storage2:
file1   file2   file3
```

Task U23.1

The `-R` option means "copy recursively', many other Unix commands also have a similar option. See what happens if you don't include the `-R` option. We've finished with all of these temporary files now. Make sure you remove the Temp directory and its contents (remember to always use `rm -i`).

U24: When things go wrong

At this point in the course, you may have tried typing some of these commands and have found that things did not work as expected. Some

people will then assume that the computer doesn't like them and that it is being deliberately mischievous. The more likely explanation is that you made a typing error. Maybe you have seen one the following error messages:

```
$ ls Codee
ls: Codee: No such file or directory

$ cp Data/Unix_test_files/* Docmentation
usage: cp [-R [-H | -L | -P]] [-fi | -n] [-pvX] source_file target_file
       cp [-R [-H | -L | -P]] [-fi | -n] [-pvX] source_file ... target_directory
```

In both cases, we included a deliberate typo when specifying the name of the directories. With the `ls` command, we get a fairly useful error message. With the `cp` command we get a more cryptic message that reveals the correct usage statement for this command. In general, if a command fails, check your current directory (`pwd`) and check that all the files or directories that you mention actually exist (and are in the right place). Many errors occur because people are not in the directory they thought they were in!

U25: Less is more

So far we have covered listing the contents of directories and moving/copying/deleting either files and/or directories. Now we will quickly cover how you can look at files; in Unix the less command lets you view (but not edit) text files. Let's take a look at a file of *Arabidopsis thaliana* protein sequences:

```
$ less Data/Arabidopsis/At_proteins.fasta
```

When you are using less, you can bring up a page of help commands by pressing `h` , scroll forward a page by pressing `space` , or go forward or backwards one line at a time by pressing `j` or `k` . To exit less, press `q` (for quit). The `less` program also does about a million other useful things (including text searching).

U26: Directory enquiries

When you have a directory containing a mixture of files and directories, it is not often clear which is which. One solution is to use `ls -l` which will put a 'd' at the start of each line of output for items which are directories. A better solution is to use `ls -p` . This command simply adds a trailing slash character to those items which are directories. Compare the following:

```
$ ls
Applications    Data    file1   Code    Documentation   file2

$ ls -p
Applications/   Data/   file1   Code/   Documentation/  file2
```

Hopefully, you'll agree that the second example makes things a little clearer. You can also do things like always capitalizing directory names (like I have done) but ideally we would suggest that you always use `ls -p` . If this sounds a bit of a pain, then it is. Ideally you want to be able to make `ls -p` the default behavior for `ls` . Luckily, there is a way of doing this by using Unix aliases. It's very easy to create an alias:

```
$ alias ls='ls -p'
$ ls
Applications/   Data/   file1   Code/   Documentation/  file2
```

If you have trouble remembering what some of these very short Unix commands do, then aliases allow you to use human-readable alternatives. I.e. you could make a 'copy' alias for the cp command' or even make 'list_files_sorted_by_date' perform the `ls -lt` command. Note that aliases do not replace the original command. It can be dangerous to use the name of an existing command as an alias for a different command. I.e. you could make an `rm` alias that put files to a 'trash' directory by using the `mv` command. This might work for you, but what if you start working on someone else's machine who doesn't have that alias? Or what if someone else starts working on your machine?

Task U26.1

Create an alias such that typing `rm` will always invoke `rm -i` . Try running the alias command on its own to see what happens. Now open a new

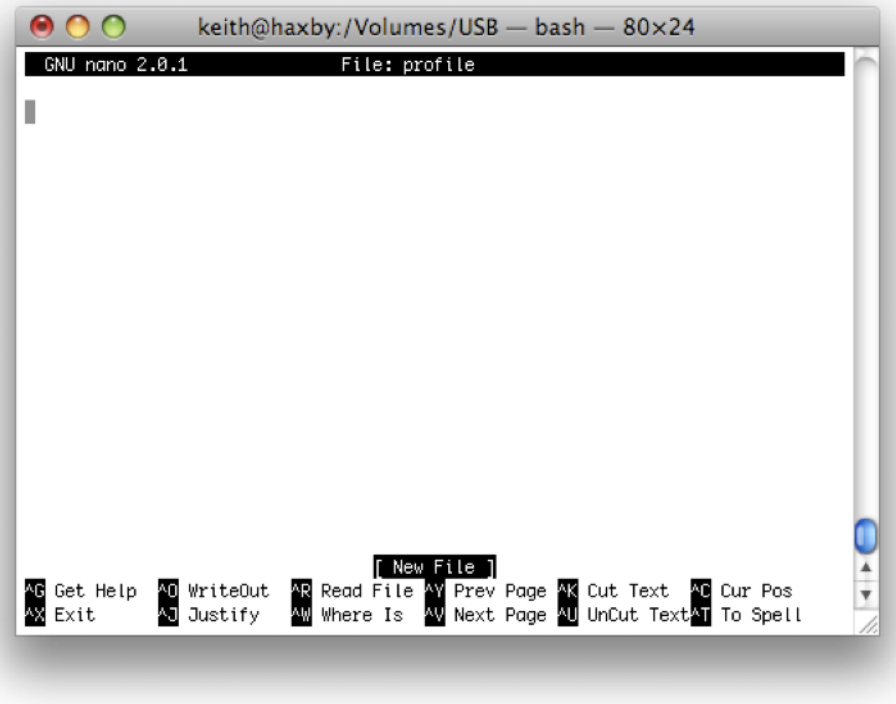
terminal window (or a new tab) and try running your `ls` alias. What happens?

U27: Fire the editor

The problem with aliases is that they only exist in the current terminal session. Once you log out, or use a new terminal window, then you'll have to retype the alias. Fortunately though, there is a way of storing settings like these. To do this, we need to be able to create a configuration file and this requires using a text editor. We could use a program like TextEdit to do this (or even Microsoft Word), but as this is a Unix course, we will use a simple Unix editor called [`][]. Let's create a file called profile:

```
$ cd ~/Unix.Course/Data
$ nano profile
```

You should see something like the following appear in your terminal:



The bottom of the nano window shows you a list of simple commands which are all accessible by typing 'Control' plus a letter. The `^` symbol means 'Control'. E.g. `^X Exit` at the bottom left of the screen means that Control + X exits the program.

Task U27.1

Type the following text in the editor and then save it (Control + O). Nano will ask if you want to "save the modified buffer' and then ask if you want to keep the same name. Then exit nano (Control + X) and use `less` to confirm that the profile file contains the text you added.

```
# some useful command line short-cuts
alias ls='ls -p'
alias rm='rm -i'
```

Now you have successfully created a configuration file (called 'profile') which contains two aliases. The first line that starts with a hash (#) is a comment, these are just notes that you can add to explain what the other lines are doing. But how do you get Unix to recognize the contents of this file? The source command tells Unix to read the contents of a file and treat it as a series of Unix commands (but it will ignore any comments).

Task U27.2

Open a new terminal window or tab (to ensure that any aliases will not work) and then type the following (make sure you first change to the correct directory):

```
$ source profile
```

Now try the `ls` command to see if the output looks different. Next, use `touch` to make a new file and then try deleting it with the `rm` command. Are the aliases working?

Running `\ls`, that is, `ls` with a backslash at the beginning, will bypass the alias. Try it - you should see the original style of ls output.

U28: Hidden treasure

In addition to adding aliases, profile files in Unix are very useful for many other reasons. We have actually already created a profile for you. It's in ~/Unix.Course/Data but you probably won't have seen it yet. That's because it is a hidden file named '.profile' (dot profile). If a filename starts with a dot, Unix will treat it as a hidden file. To see it, you can use `ls -a` which lists all hidden files (there may be several more files that appear).

Task U28.1

Use `less` to look at the profile file that we have created. See if you can understand what all the lines mean (any lines that start with a # are just comments). Use `source` to read this file. See how this changes the behavior of typing `cd` on its own. You can now delete the profile file that you made earlier, from now on we will use the .profile file.

If you have a .profile file in your *home* directory then it will be automatically read every time you open a new terminal.

> ***Remember to type:***
> ***source ~/Unix.Course/Data/.profile***
> ***every time you use a new terminal window***

U29: Sticking to the script

Unix can also be used as a programming language. Depending on what you want to do, a Unix script might solve all your problems and mean that you don't really need to learn another language at all.

So how do you make a Unix script (which are commonly called 'shell scripts')? At the simplest level, we just write one or more Unix commands to a file and then treat that file as if it was any other Unix command or program.

Task U29.1

Copy the following two lines to a file (using `nano`). Name that file hello.sh (shell scripts are typically given a .sh extension) and **make sure that you save this file in ~/Unix.Course/Data/Code**.

```
# my first Unix shell script
echo "Hello World"
```

When you have done that, simply type 'hello.sh' and see what happens. If you have previously run `source .profile` then you should be able to run 'hello.sh' from any directory that you navigate to. If it worked, then it should have printed 'Hello world'. This very simple script uses the Unix command echo which just prints output to the screen. Also note the comment that precedes the `echo` command, it is a good habit to add explanatory comments.

Task U29.2

Try moving the script outside of the Code directory (maybe move it 'up' one level) and then `cd` to that directory. Now try running the script again. You should find that it doesn't work anymore. Now try running `./hello.sh` (that's a dot + slash at the beginning). It should work again.

U30: Keep to the $PATH

The reason why the script worked when it was in the Code directory and then stopped working when you moved it is because we did something to make the Code directory a bit special. Remember this line that is in your .profile file?

```
PATH=$PATH":$HOME/Code"
```

When you try running *any* program in Unix, your computer will look in a set of predetermined places to see if a program by that name lives there. All Unix commands are just files that live in directories somewhere on your computer. Unix uses something called $PATH (which is an environment variable) to store a list of places to look for programs to run. In our .profile file we have just told Unix to also look in your
then we have to run the program by first typing ./ (dot slash). Remember that the dot means the current directory. Think of it as a way of forcing Unix to run a program.

U31: Ask for permission

Programs in Unix need permission to be run. We will normally always have to type the following for any script that we create:

```
$ chmod u+x hello.sh
```

This would use the chmod to add *executable* permissions (+x) to the file called 'hello.sh' (the 'u' means add this permission to just you, the user). Without it, your script won't run. Except that it did. One of the oddities of using the USB drive for this course, is that files copied to a USB drive have all permissions turned on by default. Just remember that you will normally need to run `chmod` on any script that you create. It's probably a good habit to get into now.

The chmod command can also modify read and write permissions for files, and change any of the three sets of permissions (read, write, execute) at the level of "user', "group', and "other'. You probably won't need to know any more about the chmod command other than you need to use it to make scripts executable.

U32: The power of shell scripts

Time to make some Unix shell scripts that might actually be useful.

Task U32.1

Look in the Data/Unix_test_files directory. You should see several files (all are empty) and four directories. Now put the following information into a shell script (using `nano`) and save it as cleanup.sh.

```
#!/bin/bash
mv *.txt Text
mv *.jpg Pictures
mv *.mp3 Music
mv *.fa Sequences
```

Make sure that this script is saved in your `Unix.Course/Code directory`. Now cd to the `Unix.Course/Data/Unix_test_files` directory and run this script. It should place the relevant files in the correct directories. This is a relatively simple use of shell scripting. As you can see the script just contains regular Unix commands that you might type at the command prompt. But if you had to do this type of file sorting every day, and had many different types of file, then it would save you a lot of time.

Did you notice the '#!/bin/bash' line in this script? There are several different types of shell script in Unix, and this line makes it clearer that a) that this is actually a file that can be treated as a program and b) that it will be a bash script (bash is a type of Unix). As a general rule, all types of scriptable programming languages should have a similar line as the first line in the program if you want to run them from the command line.

Task U32.2

Here is another script. Copy this information into a file called change_file_extension.sh and again place that file in the Code directory.

```
#!/bin/bash

for filename in *.$1
do
    mv $filename ${filename%$1}$2
done
```

Now go to the `Data/Unix_test_files/Text` directory. If you have run the exercise from Task U32.1 then your text directory should now contain three files. Run the following command:

```
$ change_file_extension.sh txt text
```

Now run the `ls` command to see what has happened to the files in the directory. You should see that all the files that ended with 'txt' now end with 'text'. Try using this script to change the file extensions of other files.

It's not essential that you understand exactly how this script works at the moment, but you should at least see how a relatively simple Unix shell script can be potentially very useful.

End of part 1.

You can now continue to learn a series of much more powerful Unix commands!

Part 2: Advanced Unix

How to Become a Unix power user

The commands that you have learnt so far are essential for doing any work in Unix but they don't really let you do anything that is very useful. The following sections will introduce a few new commands that will start to show you how powerful Unix is.

U33: Match making

You will often want to search files to find lines that match a certain pattern. The Unix command grep does this (and much more). You might already know that FASTA files (used frequently in bioinformatics) have a simple format: one header line which must start with a '>' character, followed by a DNA or protein sequence on subsequent lines. To find only those header lines in a FASTA file, we can use grep, which just requires you specify a pattern to search for, and one or more files to search:

```
$ cd Data/Arabidopsis/
$ grep ">" intron_IME_data.fasta

>AT1G68260.1_i1_204_CDS
>AT1G68260.1_i2_457_CDS
>AT1G68260.1_i3_1286_CDS
>AT1G68260.1_i4_1464_CDS
```

```
.
.
.
```

This will produce lots of output which will flood past your screen. If you ever want to stop a program running in Unix, you can type Control+C (this sends an interrupt signal which should stop most Unix programs). The grep command has many different command-line options (type `man grep` to see them all), and one common option is to get `grep` to show lines that don't match your input pattern. You can do this with the -v option and in this example we are seeing just the sequence part of the FASTA file.

```
$ grep -v ">" intron_IME_data.fasta

GTATACACATCTCTCTACTTTCATATTTTGCATCTCTAACGAAATCGGATTCCGTCGTTG
TGAAATTGAGTTTTCGGATTCAGTGTTGTCGAGATTCTATATCTGATTCAGTGATCTAAT
GATTCTGATTGAAAATCTTCGCTATTGTACAG
GTTAGTTTTCAATGTTGCTGCTTCTGATTGTTGAAAGTGTTCATACATTTGTGAATTTAG
TTGATAAAATCTGAACTCTGCATGATCAAAGTTACTTCTTTACTTAGTTTGACAGGGACT
TTTTTTGTGAATGTGGTTGAGTAGAATTTAGGGCTTTGGATTAAATGTGACAAGATTTTG
.
.
.
```

U34: Your first ever Unix pipe

By now, you might be getting a bit fed up of waiting for the `grep` command to finish, or you might want a cleaner way of controlling things without having to reach for Ctrl-C. Ideally, you might want to look at the output from any command in a controlled manner, i.e. you might want to use a Unix program like less to view the output.

This is very easy to do in Unix, you can send the output from any command to any other Unix program (as long as the second program accepts input of some sort). We do this by using what is known as a pipe. This is implemented using the '|' character (which is a character which always seems to be on different keys depending on the keyboard that you are using). Think of the pipe as simply connecting two Unix programs. In this next example we send the output from `grep` down a pipe to the less program. Let's imagine that we just want to see lines in the input file which contain the pattern "ATGTGA" (a potential start and stop codon combined):

```
$ grep "ATGTGA" intron_IME_data.fasta | less

TTTTTTGTGAATGTGGTTGAGTAGAATTTAGGGCTTTGGATTAAATGTGACAAGATTTTG
CTGAATGTGACTGGAAGAATGAAATGTGTTAAGATCTTGTTCGTTAAGTTTAGAGTCTTG
GGTGGAATGAATTTATGTATCATGTGATAGCTGTTGCATTACAAGATGTAATTTTGCAAA
GTCTATGTGATGGCCATAGCCCATAGTGACTGATAGCTCCTTACTTTGTTTTTTTTTTCT
TTACTTGCAAAATTCCATGTGATTTTTTATATTACTTTGAAGAATTTTATAATATATTTT
TTGCATCAAGATATGTGACATCTTCAAAAAGATAACTTGTGAGAAGACAATTATAATATG
GTAACTTATTTATTGATTGAATCAGTAACTGTATTGTTATCATGATTTGTGAATATGTGA
AATCTTTGTGGTGGGTCTACGATATGAGCTGTCAATATATTTTTGTTTATACATGTGATC
GTATGTGAGCAAACGATGTCTCGTTTTCTCTCTCTCAATGATCAAGCACCTAACTTAAAT\
.
.
.
```

Notice that you still have control of your output as you are now in the `less` program. If you press the forward slash (/) key in `less`, you can then specify a search pattern. Type ATGTGA after the slash and press enter. The `less` program will highlight the location of these matches on each line. Note that `grep` matches patterns on a per line basis. So if one line ended ATG and the next line started TGA, then `grep` would not find it.

> ***Any time you run a Unix program or command that outputs a lot of text to the screen, you can instead pipe that output into the `less` program.***

U35: Heads and tails

Sometimes we do not want to use `less` to see *all* of the output from a command like grep. We might just want to see a few lines to get a feeling for what the output looks like, or just check that our program (or Unix command) is working properly. There are two useful Unix commands for doing this: `head` and `tail`. These commands show (by default) the first or last 10 lines of a file (though it is easy to specify more or fewer lines of output). So now, let's look for another pattern which might be in all the sequence files in the directory. If we didn't know whether the DNA/protein sequence in a FASTA files was in upper-case or lower-case letters, then we could use the `-i` option of `grep` which 'ignores' case when searching:

```
$ grep -i ACGTC * | head
At_proteins.fasta:TYRSPRCNSAVCSRAGSIACGTCFSPPRPGCSNNTCGAFPDNSITGWATSGEFALDVVSIQSTNGSNPGRFVKIPNLIFS
At_proteins.fasta:FRRYGHYISSDVFRRFKGSNGNFKESLTGYAKGMLSLYEAAHLGTTKDYILQEALSFTSSHLESLAACGTCPPHLSVHIQ
At_proteins.fasta:MAISKALIASLLISLLVLQLVQADVENSQKKNGYAKKIDCGSACVARCRLSRRPRLCHRACGTCCYRCNCVPPGTYGNYD
At_proteins.fasta:MAVFRVLLASLLISLLVLDFVHADMVTSNDAPKIDCNSRCQERCSLSSRPNLCHRACGTCCARCNCVAPGTSGNYDKCPC
chr1.fasta:TGTCTACTGATTTGATGTTTTCCTAAACTGTTGATTCGTTTCAGGTCAACCAATCACGTCAACGAAATTCAGGATCTTA
chr1.fasta:TATGCTGCAAGTACCAGTCAATTTTAGTATGGGAAACTATAAACATGTATAATCAACCAATGAACACGTCAATAACCTA
chr1.fasta:TTGAACAGCTTAGGGTGAAAATTATGATCCGTAGAGACAGCATTTAAAAGTTCCTTACGTCCACGTAAAATAATATATC
chr1.fasta:GGGATCACGAGTCTGTTGAGTTTTCCGACGTCGCTTGGTGTTACCACTTTGTCGAACATGTGTTCTTTCTCCGGAGGTG
chr1.fasta:CTGCAAAGGCCTACCTGTTTGTCCCTGTTACTGACAATACGTCTATGGAACCCATAAAAGGGATCAACTGGGAATTGGT
chr1.fasta:ACGTCGAAGGGGGTAAGATTGCAGCTAATCATTTGATGAAATGGATTGGGATTCACGTGGAGGATGATCCTGATGAAGT
```

The `*` character acts as a wildcard meaning 'search all files in the current directory' and the `head` command restricts the total amount of output to 10 lines. Notice that the output also includes the name of the file containing the matching pattern. In this case, the `grep` command finds the ACGTC pattern in four protein sequences and several lines of the the chromosome 1 DNA sequence (we don't know how many exactly because the head command is only giving us ten lines of output).

U36: Getting fancy with regular expressions

A concept that is supported by many Unix programs and also by most programming languages (including Perl) is that of using regular expressions. These allow you to specify search patterns which are quite complex and really help restrict the huge amount of data that you might be searching for to some very specific lines of output. E.g. you might want to find lines that start with an 'ATG' and finish with 'TGA' but which have at least three AC dinucleotides in the middle:

```
$ grep "^ATG.*ACACAC.*TGA$" chr1.fasta

ATGAACCTTGTACTTCACCGGGTGCCCTCAAAGACGTTCTGCTCGGAAGGTTTGTCTTACACACTTTGATGTCAAATGA
ATGATAGCTCAACCACGAAATGTCATTACCTGAAACCCTTAAACACACTCTACCTCAAACTTACTGGTAAAAACATTGA
ATGCATACCTCAGTTGCATCCCGGCGCAGGGCAAGCATACCCGCTTCAACACACACTGCTTTGAGTTGAGCTCCATTGA
```

You'll learn more about regular expressions when you learn Perl. The `^` character is a special character that tells `grep` to only match a pattern if it occurs at the start of a line. Similarly, the `$` tells `grep` to match patterns that occur at the end of the line.

Task U36.1

The `.` and `*` characters are also special characters that form part of the regular expression. Try to understand how the following patterns all differ. Try using each of these these patterns with `grep` against any one of the sequence files. Can you predict which of the five patterns will generate the most matches?

```
ACGT
AC.GT
AC*GT
AC.*GT
```

> ***The asterisk in a regular expression is similar to, but NOT the same, as the other asterisks that we have seen so far. An asterisk in a regular expression means: 'match zero or more of the preceding character or pattern'.***

Try searching for the following patterns to ensure you understand what `.` and `*` are doing:

```
A...T
AG*T
A*C*G*T*
```

U37: Counting with `grep`

Rather than showing you the lines that match a certain pattern, `grep` can also just give you a count of how many lines match. This is one of the frequently used `grep` options. Running `grep -c` simply counts how many lines match the specified pattern. It doesn't show you the lines themselves, just a number:

```
$ grep -c i2 intron_IME_data.fasta
9785
```

Task U37.1

Count how many times each pattern from Task [U36.1] occurs in all of the sequence files (specifying `*.fasta` will allow you to specify all sequence files).

U38: Regular expressions in `less`

You have seen already how you can use `less` to view files, and also to search for patterns. If you are viewing a file with `less`, you can type a forward-slash `/` character, and this allows you to then specify a pattern and it will then search for (and highlight) all matches to that pattern. Technically it is searching forward from whatever point you are at in the file. You can also type a question-mark `?` and `less` will allow you to search backwards. The real bonus is that the patterns you specify can be regular expressions.

Task U38.1

Try viewing a sequence file with `less` and then searching for a pattern such as `ATCG.*TAG$`. This should make it easier to see exactly where your regular expression pattern matches. After typing a forward-slash (or a question-mark), you can press the up and down arrows to select previous searches.

U39: Let me transl(iter)ate that for you

We have seen that these sequence files contain upper-case characters. What if we wanted to turn them into lower-case characters (because maybe another bioinformatics program will only work if they are lower-case)? The Unix command tr (short for transliterate) does just this, it takes one range of characters that you specify and changes them into another range of characters:

```
$ head -n 2 chr1.fasta

>Chr1 dumped from ADB: Mar/14/08 12:28; last updated: 2007-12-20
CCCTAAACCCTAAACCCTAAACCCTAAACCTCTGAATCCTTAATCCCTAAATCCCTAAATCTTTAAATCCTACATCCAT

$ head -n 2 chr1.fasta | tr 'A-Z' 'a-z'

>chr1 dumped from adb: mar/14/08 12:28; last updated: 2007-12-20
ccctaaaccctaaaccctaaaccctaaacctctgaatccttaatccctaaatccctaaatctttaaatcctacatccat
```

U40: That's what she `sed`

The `tr` command let's you change a range of characters into another range. But what if you wanted to change a particular pattern into

something completely different? Unix has a very powerful command called sed that is capable of performing a variety of text manipulations. Let's assume that you want to change the way the FASTA header looks:

```
$ head -n 1 chr1.fasta >Chr1 dumped from ADB: Mar/14/08 12:28; last updated: 2007-12-20

$ head -n 1 chr1.fasta | sed 's/Chr1/Chromosome 1/' >Chromosome 1 dumped from ADB: Mar/14/08 12:28; last
updated: 2007-12-20
```

The 's' part of the `sed` command puts `sed` in 'substitute' mode, where you specify one pattern (between the first two forward slashes) to be replaced by another pattern (specified between the second set of forward slashes). Note that this doesn't actually change the contents of the file, it just changes the screen output from the previous command in the pipe. We will learn later on how to send the output from a command into a new file.

U41: Word up

For this section we want to work with a different type of file. It is sometimes good to get a feeling for how large a file is before you start running lots of commands against it. The `ls -l` command will tell you how big a file is, but for many purposes it is often more desirable to know how many 'lines' it has. That is because many Unix commands like `grep` and `sed` work on a line by line basis. Fortunately, there is a simple Unix command called wc (word count) that does this:

```
$ cd Data/Arabidopsis/ $ wc At_genes.gff 531497 4783473 39322356 At_genes.gff
```

The three numbers in the output above count the number of lines, words and bytes in the specified file(s). If we had run `wc -l`, the `-l` option would have shown us just the line count.

U42: GFF and the art of redirection

The Arabidopsis directory also contains a GFF file. This is a common file format in bioinformatics and GFF files are used to describe the location of various features on a DNA sequence. Features can be exons, genes, binding sites etc, and the sequence can be a single gene or (more commonly) an entire chromosome.

This GFF file describes of all of the gene-related features from chromosome I of *A. thaliana*. We want to play around with some of this data, but don't need all of the file…just 10,000 lines will do (rather than the ~500,000 lines in the original). We will create a new (smaller) file that contains a subset of the original:

```
$ head -n 10000 At_genes.gff > At_genes_subset.gff
$ ls -l
total 195360
-rwxrwxrwx  1 keith  staff  39322356 Jul  9 15:02 At_genes.gff
-rwxrwxrwx  1 keith  staff    705370 Jul 10 13:33 At_genes_subset.gff
-rwxrwxrwx  1 keith  staf f 17836225 Oct  9  2008 At_proteins.fasta
-rwxrwxrwx  1 keith  staff  30817851 May  7  2008 chr1.fasta
-rwxrwxrwx  1 keith  staff  11330285 Jul 10 11:11 intron_IME_data.fasta
```

This step introduces a new concept. Up till now we have sent the output of any command to the screen (this is the default behavior of Unix commands), or through a pipe to another program. Sometimes you just want to redirect the output into an actual file, and that is what the `>` symbol is doing, it acts as one of three redirection operators in Unix.

As already mentioned, the GFF file that we are working with is a standard file format in bioinformatics. For now, all you really need to know is that every GFF file has 9 fields, each separated with a tab character. There should always be some text at every position (even if it is just a '.' character). The last field often is used to store a lot of text.

U43: Not just a pipe dream

The 2nd and/or 3rd fields of a GFF file are usually used to describe some sort of biological feature. We might be interested in seeing how many different features are in our file:

```
$ cut -f 3 At_genes_subset.gff | sort | uniq

CDS
chromosome
exon
five_prime_UTR
gene
mRNA
miRNA
ncRNA
protein
pseudogene
pseudogenic_exon
pseudogenic_transcript
snoRNA
tRNA
three_prime_UTR
transposable_element_gene
```

In this example, we combine three separate Unix commands together in one go. Let's break it down (it can be useful to just run each command one at at time to see how each additional command is modifying the preceding output):

1. The cut command first takes the At_genes_subset.gff file and 'cuts' out just the 3rd column (as specified by the `-f` option). Luckily, the default behavior for the `cut` command is to split text files into columns based on tab characters (if the columns were separated by another character such as a comma then we would need to use another command line option to specify the comma).
2. The sort command takes the output of the cut command and sorts it alphanumerically.
3. The uniq command (in its default format) only keeps lines which are unique to the output (otherwise you would see thousands of fields which said 'curated', 'Coding_transcript' etc.)

Now let's imagine that you might want to find which features start earliest in the chromosome sequence. The start coordinate of features is always specified by column 4 of the GFF file, so:

```
$ cut -f 3,4 At_genes_subset.gff | sort -n -k 2 | head

chromosome  1
exon    3631
five_prime_UTR  3631
gene    3631
mRNA    3631
CDS 3760
protein 3760
CDS 3996
exon    3996
CDS 4486
```

Here we first cut out just two columns of interest (3 & 4) from the GFF file. The `-f` option of the `cut` command lets us specify which columns we want to remove. The output is then sorted with the `sort` command. By default, `sort` will sort alphanumerically, rather than numerically, so we use the `-n` option to specify that we want to sort numerically. We have two columns of output at this point and we could sort based on either column. The `-k 2` specifies that we use the second column. Finally, we use the `head` command to get just the 10 rows of output. These should be lines from the GFF file that have the lowest starting coordinate.

U44: The end of the line

When you press the return/enter key on your keyboard you may think that this causes the same effect no matter what computer you are using. The visible effects of hitting this key are indeed the same…if you are in a word processor or text editor, then your cursor will move down one line. However, behind the scenes pressing enter will generate one of two different events (depending on what computer you are using). Technically speaking, pressing enter generates a newline character which is represented internally by either a *line feed or carriage return* character (actually, Windows uses a combination of both to represent a newline). If this is all sounding confusing, well it is, and it is even more complex than we are revealing here.

The relevance of this to Unix is that you will sometimes receive a text file from someone else which looks fine on their computer, but looks unreadable in the Unix text viewer that you are using. In Unix (and in programming languages) the patterns `\n` and `\r` can both be used to denote

newlines. A common fix for this requires substituting `\r` for `\n`.

Use `less` to look at the Data/Misc/excel_data.csv file. This is a simple 4-line file that was exported from a Mac version of Microsoft Excel. You should see that if you use `less`, then this appears as one line with the newlines replaced with ^M characters. You can convert these carriage returns into Unix-friendly line-feed characters by using the tr command like so:

```
$ cd Data/Misc
$ tr '\r' '\n' < excel_data.csv
sequence 1,acacagagag
sequence 2,acacaggggaaa
sequence 3,ttcacagaga
sequence 4,cacaccaaacac
```

This will convert the characters but not save the resulting output, if you wanted to send this output to a new file you will have to use a second redirect operator:

```
$ tr '\r' '\n' < excel_data.csv > excel_data_formatted.csv
```

U45: This one goes to 11

Finally, let's parse the Arabidopsis `intron_IME_data.fasta` file to see if we can extract a subset of sequences that match criteria based on something in the FASTA header line. Every intron sequence in this file has a header line that contains the following pieces of information:

- gene name
- intron position in gene
- distance of intron from transcription start site (TSS)
- type of sequence that intron is located in (either CDS or UTR)

Let's say that we want to extract five sequences from this file that are: a) from first introns, b) in the 5' UTR, and c) closest to the TSS. Therefore we will need to look for FASTA headers that contain the text 'i1' (first intron) and also the text '5UTR'.

We can use `grep` to find header lines that match these terms, but this will not let us extract the associated sequences. The distance to the TSS is the number in the FASTA header which comes after the intron position. So we want to find the five introns which have the lowest values.

Before I show you one way of doing this in Unix, think for a moment how you would go about this if you didn't know any Unix… would it even be something you could do without manually going through a text file and selecting each sequence by eye? Note that this Unix command is so long that — depending on how you are viewing this document — it may appear to wrap across two lines. When you type this, it should all be on a single line:

```
$ tr '\n' '@' < intron_IME_data.fasta | sed 's/>/#>/g' | tr '#' '\n' | grep "i1_.*5UTR" | sort -nk 3 -t "_" |
head -n 5 | tr '@' '\n'

>AT4G39070.1_i1_7_5UTR
GTGTGAAACCAAAACCAAAACAAGTCAATTTGGGGGCATTGAAAGCAAAGGAGAGAGTAG
CTATCAAATCAAGAAAATGAGAGGAAGGAGTTAAAAAAGACAAAGGAAACCTAAGCTGCT
TATCTATAAAGCCAACACATTATTCTTACCCTTTTGCCCACACTTATACCCCATCAACCT
CTACATACACTCACCCACATGAGTGTCTCTACATAAACACTACTATATAGTACTGGTCCA
AAGGTACAAGTTGAGGGAG

>AT5G38430.1_i1_7_5UTR
GCTTTTTGCCTCTTACGGTTCTCACTATATAAAGATGACAAAACCAATAGAAAAACAATT
AAG

>AT1G31820.1_i1_14_5UTR
GTTTGTACTTCTTTACCTCTCGTAAATGTTTAGACTTTCGTATAAGGATCCAAGAATTTA
TCTGATTGTTTTTTTTTCTTTGTTTCTTTGTGTTGATTCAG

>AT3G12670.1_i1_18_5UTR
GTAGAATTCGTAAATTTCTTCTGCTCACTTTATTGTTTCGACTCATACCCGATAATCTCT
TCTATGTTTGGTAGAGATATCTTCTCAAAGTCTTATCTTTCCTTACCGTGTTCTGTGTTT
TTTGATGATTTAG

>AT1G26930.1_i1_19_5UTR
GTATAATATGAGAGATAGACAAATGTAAAGAAAAACACAGAGAGAAAATTAGTTTAATTA
ATCTCTCAAATATATACAAATATTAAAACTTCTTCTTCTTCAATTACAATTCTCATTCTT
```

```
TTTTTCTTGTTCTTATATTGTAGTTGCAAGAAAGTTAAAAGATTTTGACTTTTCTTGTTT
CAG
```

That's a long command, but it does a lot. Try to break down each step and work out what it is doing (you will need to consult the man page for some commands maybe). Notice that I use one of the other redirect operators `<` to read from a file. It took seven Unix commands to do this, but these are all relatively simple Unix commands; it is the combination of them together which makes them so powerful. One might argue that when things get this complex with Unix that it might be easier to do it in Perl!

Summary

Congratulations are due if you have reached this far. If you have learnt (and understood) all of the Unix commands so far then you probably will never need to learn anything more in order to do a lot of productive Unix work. But keep on dipping into the man page for all of these commands to explore them in even further detail.

The following table provides a reminder of most of the commands that we have covered so far. If you include the three, as-yet-unmentioned, commands in the last column, then you will probably be able to achieve >95% of everything that you will ever want to do in Unix (remember, you can use the `man` command to find out more about `top`, `ps`, and `kill`). The power comes from how you can use combinations of these commands.

The absolute basics	**File control**	**Viewing, creating, or editing files**	**Misc. useful commands**	**Power commands**	**Process-related commands**
ls	mv	less	man	uniq	top
cd	cp	head	chmod	sort	ps
pwd	mkdir	tail	source	cut	kill
	rmdi				