Unix Basics

History

The first version of Unix was developed by Bell Labs (part of AT&T) in 1969, making it more than forty years old and one of the few cases of a computer technology that has survived more than a decade. Its roots go back to when computers were large and rare, time on them very expensive and shared between many users – Unix was designed from the beginning[1] to have multiple users working simultaneously. While this might seem strange and unnecessary in a world where everyone has their own laptop, computing is again moving back to remote central services with many users: the compute power required for mapping next-generation sequencing data or *de novo* assembly is beyond what is available or desirable to have sitting on your lap. In many ways, the "Cloud" (or what ever has replaced it by the time you read this) requires ways of working that are more in common with traditional Unix machines than the personal computing emphasised by Windows and Apple Macintosh.

USA federal monopoly law prevented AT&T from commercialising Unix but interest in using it increased outside of Bell Labs and eventually they decided to give it away freely, including the source code, which allowed other institutions to modify it. Perhaps the most important of these institutions was the University of Berkeley[2] which distributed a set of tools to make Unix more useful and made changes that significantly increased performance. The involvement of several universities in its development meant Unix was ideally placed when the internet was created and many of the fundamental technologies were developed and tested using Unix machines. Again these improvements were given away freely, some of the code being repurposed to provided networking for early versions of Windows and even today several utilities in Windows Vista incorporate Berkeley code[3].

As well as being a key part in the development of the early internet, a Unix machine was also the first web server, a NeXT cube[4]. NeXT was an early attempt to make a Unix machine for desktop use, extremely advanced for its time but also very expensive so they never really caught on outside of the finance industry. Apple eventually bought NeXT, its operating system becoming OsX, and this heritage can still be seen in its programming interfaces. Apple is now the largest manufacturer of Unix machines; every Apple computer, the iPhone and most recent iPods have a Unix base underneath their facade.

By the early 90s Unix became increasingly commercially important which inevitably lead to legal trouble: with so many people giving away improvements freely and having them integrated into the system, who actually owned it? The legal trouble cast uncertainty over the freely available Unix versions, creating an opening for another free operating system.

The vacuum was filled by Linux, a freely available computer operating system[5] similar to Unix and

1 This is lie. In truth, Unix actually grew out of a desire to play a game called Space Travel http://en.wikipedia.org/wiki/Space_Travel_(video_game) and the features that made it an operating system were incidental. Initially it only supported one user and the name Unix, originally UNICS, is an unfortunate pun on MULTICS, a multi-user system available at the time.

2 A significant proportion of Mac OsX has its roots in the Berkeley Standard Distribution (BSD).

3 For example: **`strings ftp.exe | grep Cal`**
`@(#) Copyright (c) 1983 The Regents of the University of California.`

4 See http://en.wikipedia.org/wiki/NeXT

5 More correctly, Linux is just the kernel, the central program from which all others are run. Many more tools in addition to this are required to make an operating system, tools provided by the GNU project. http://www.gnu.org/.

started by Linus Torvalds in 1991 as a hobby. Importantly, Linux was written from scratch and did not contain any of the original Unix code and so was free of legal doubt. Coinciding with the penetration of the internet onto university campus and the availability of cheap but sufficiently powerful personal computers, Linux rapidly matured with over one hundred developers collaborating over the internet within two years. The real advances driving Linux were social rather than technological, disparate volunteers donating time on the understanding that, in return for giving their work away freely, anything based on their work is also given away freely and so they in turn benefit from improvements. The idea that underpins this sharing and ensures that nobody can profit from anyone else's work without sharing is "copyleft", described in a simple legal document called the GNU General Public Licence http://www.gnu.org/copyleft/[6] which turns the notion of copyright on its head.

Today, Linux has become the dominant free Unix-like operating system with millions of users and support from many large companies.

Getting and installing Ubuntu

This tutorial concentrates on the Ubuntu distribution (packaging) of Linux, which is one of the most widely used, but all the examples are fairly generic and should work with most Linux, Unix and Macintosh computers. There are many different guides on the web about how to install Ubuntu but we recommend installing it as a virtual machine on your current computer, see separate documentation for instructions.

The Ubuntu Linux distribution is generally easy to use and it is updated (for free) every six months. At the time of writing, the current version of Ubuntu is 11.10, named after its release date in October 2011, and also known as "Oneiric Ocelot"; the next version, 12.04 or "Precise Pangolin" will be released in April 2012 and will be designated a Long Term Support (LTS) edition, meaning that it will be receive fixes and maintenance upgrades for five years before being retired, and is the best option if you don't want to be regularly upgrading your system.

Acclimatisation

A significant effort has been undertaken to make Ubuntu easy to use, so even novice computer users should have little trouble using it. There is a considerable number of tutorials available for users new to Ubuntu; the official material is available at https://help.ubuntu.com/11.10/ but a quick search on the web will locate much more. In addition, there is a lot of documentation installed on the machine itself: you can access this by moving the mouse towards *Ubuntu Desktop* at the top left of the screen and clicking on the help menu that appears. In general, the name of the program you are currently using is displayed at the top-left of the screen and moving the mouse to top of the screen will reveal the programs menus in a similar fashion to how they are displayed on the Mac (although, confusingly, some programs display their menus within their own window rather like a Windows computer).

An alternative way to get help is to click on the circular symbol (a stylised picture of three people holding hands) at the top left of the screen and type help in the search box that appears. For want of a better name, we will refer to the people-holding-hands button as the Ubuntu button although the help text that appears describes it as "Dash home".

Ubuntu comes free with many tools, including web browsers, file managers, word processors, etc. Generally there is a free equivalent for most software you might use and you can browse those available by clicking on the *Ubuntu Software Centre*, whose icon at the left of the screen looks like a

6 It should be noted that the GNU project, and the philosophy behind it, predate Linux by almost a decade.

shopping bag full of goodies. The *Ubuntu Software Centre* is just a starting point and there are many other sources available, both of prepackage software specifically for Ubuntu and source code that will require compiling. Search the web for "Ubuntu software repositories" for more information on obtaining additional software.

While there are explicit key combinations for copy and pasting text, just like on Windows or Mac, shift-control-c and shift-control-v in Ubuntu, this convention is not respected by all programs. Unix has traditionally been more mouse centred with the left mouse button used to highlight text and the middle button used to copy it. You may find yourself accidentally doing this occasionally if you aren't used to using the middle mouse button.

Starting applications from icons, opening folders, etc... only requires a single click, rather than the double click required on Windows, making the action of pressing buttons and selecting things from menus more consistent with each other. Accidentally double clicking will generally result in an action being done twice, not normally a bad thing but it does means that impatient users can quickly find their desktop covered in windows.

Perhaps the most important difference you are likely to encounter on a daily basis is that the names of files and directories are case sensitive: README.txt, readme.txt and readme.TXT all refer to different files. This is different from both Windows and Mac OsX[7], where upper and lower-case characters are preserved in the name but the file can be referred to using any case.

Fetching the examples

There are many examples in this tutorial to be tried, enclosed in boxes like the one below which explains the format of the examples. The files required for the examples can be downloaded from http://tinyurl.com/32a2gbk/unix.tgz although the example below shows how to automatically download and unpack the file ready for use.

```
#     Ordinary text, starting with a # and indented on the first line, is a comment
#     on the example.
Bold text is something to type in at the command-line. A single ↩
line wrapped on to multiple lines is indicated by the '↩' symbol
This is now a separate command.
Italic text is a reply from the computer to what was typed in
#     Now we will download and install the examples
#     Firstly, ensure that we are in the home directory
cd
#     Where an example requires you to be in a specific directory, it will start
#     with the command and reply to tell you where you should be. If you are not
#     in the correct directory, move to it before doing the example (see later for
#     how to change directory. Here 'ebi' is just the name of the user (you),
#     yours may vary.
pwd
/home/ebi/
#     Download and unpack the examples. You don't need to understand what this is
#     doing yet, although you will by the time you have worked through this
#     document.
wget -O  -  'http://tinyurl.com/32a2gbk/unix.tgz' | tar -zx
#     If you are trying the examples out on a Mac, the command wget is not
```

7 Despite its Unix heritage. This behaviour is deliberate to maintain compatibility with earlier versions of the Mac operating system.

```
#     available and the above will not work. Instead, use the similar curl command
curl -L 'http://tinyurl.com/32a2gbk/unix.tgz' | tar -zx
#     A directory `examples' should have been created.
ls examples/unix
Compression LineEndings MultipleFiles     SCP         haiku
Escaping    MoveCopy    Pipes             Scripting
#     How to delete the examples, if required
#     Firstly, ensure that we are in the home directory (where the examples where
#     installed).
cd
rm -rf examples
```

The command line

While Ubuntu has all the graphical tools you might expect in a modern operating system, so new users rarely need to deal with its Unix foundations, we will be working with the command-line. An obvious question is why the command-line is still the main way of interacting with Unix or, more relevantly, why we are making you use it? Part of the answer to the first question is that the origins of Unix predate the development of graphical interfaces and this is what all the tools and programs have evolved from. The reason the command-line remains popular is that it is an extremely efficient way to interact with the computer: once you want to do something complex enough that there isn't a handy button for it, graphical interfaces force you to go through many menus and manually perform a task that could have been automated. Alternatively, you must resort to some form of programming (Mac OsX Automator, Microsoft Office macros, etc) which is the moral equivalent of using the command-line.

Unix is built around many little tools designed to work together. Each program does one task well and returns its output in a form easily understood by other programs and these properties allow simple programs to be combined together to produce complex results, rather like building something out of Lego bricks. The forward to the 1978 report in the Bell System Technical Journal[8] describes the Unix philosophy as:

"(i) Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features.

(ii) Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.

(iii)Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.

(iv)Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them."

The rest of this tutorial will be based using the command-line through a “terminal”[9]. The terminal program can be found by clicking on the Ubuntu button and typing terminal in the search box, as

8 See: M.D. McIlroy, E.N.Pinson, and B.A. Tague "Unix Time-Sharing System Forward", The Bell System Technical Jounal, July -Aug 1978 vol 57, number 6 part 2, pg. 1902

9 Terminology that dates back to the early days of Unix when there would be many “terminals”, basically a fancy screen and keyboard, connected to a central computer.

shown in Illustration 1. Once open, the text size can be changed using the *View/Zoom* menu options or the font changed entirely using the *Edit/Profile Preferences* menu option.

While we are using Linux during the workshop, you may not have access to a machine later or may not wish to use Linux exclusively on your computer. While you could install Linux as 'dual-boot' on your computer, or run it in a virtual machine[10], the knowledge of the command-line is fairly transferable between platforms: having Unix foundations, Mac OsX also has a command-line hidden away: /Applications/Utilities/Terminal and, with a small number of eccentricities, everything that works on the Linux command-line should work for OsX. Windows has its own incompatible version of a command-line but Cygwin http://www.cygwin.com/ can be installed and provides an entire Unix-like environment within Windows.

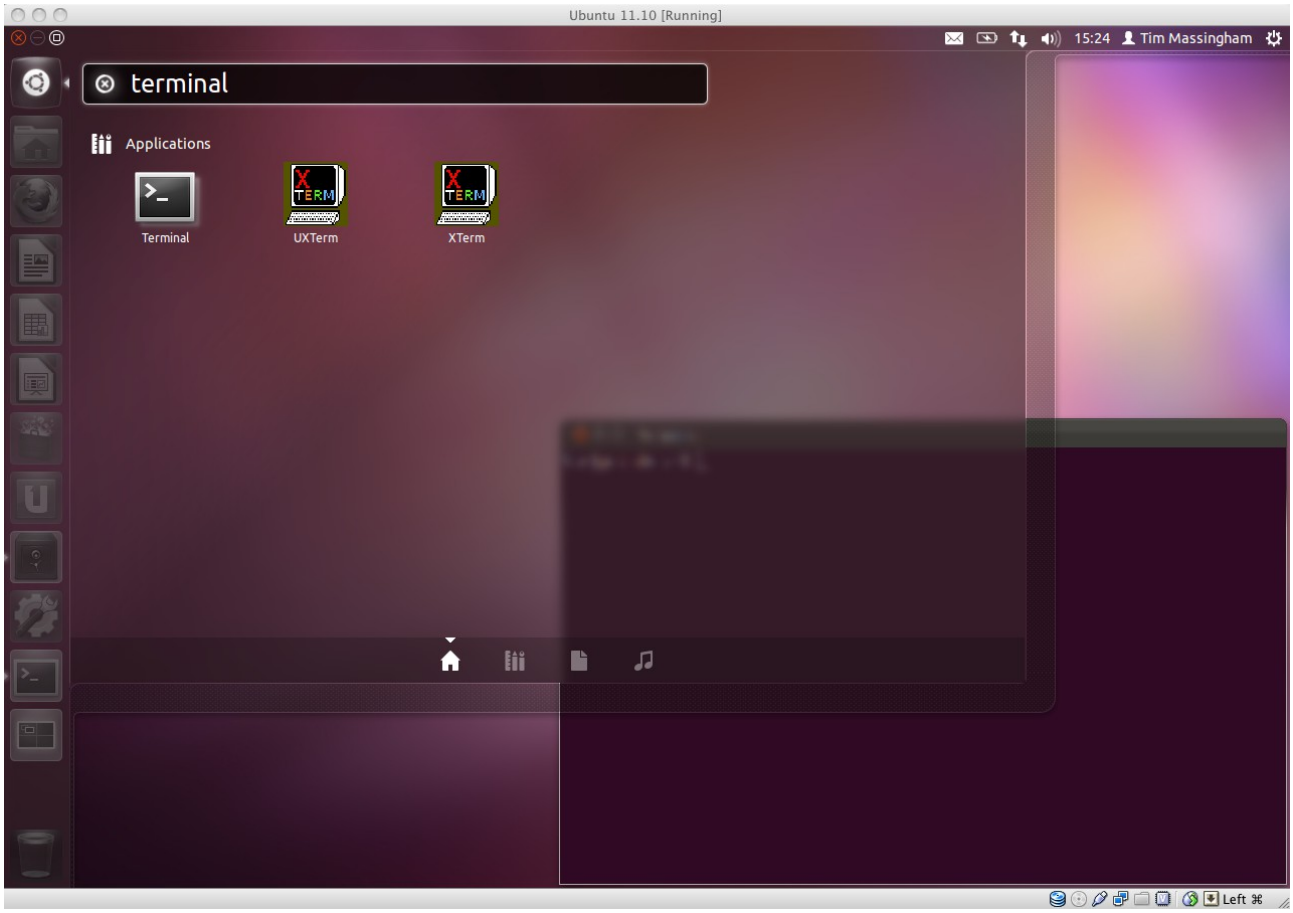

Illustration 1: Opening a terminal in Ubuntu. A partially obscured terminal is shown at the bottom right of the desktop.

At the beginning of the command-line is the *command prompt*, showing that the computer is ready to accept commands. The prompt is text of the form **`user@computer:directory$`**, Illustration 1 having a user called *tim* in the directory `~` on a computer called *coffee-grinder*. Having all this information is handy when you are working with multiple remote computers at the same time. The prompt is configurable and may vary between computers; you may notice later that other prompts are slightly different. Some basic commands are shown in Table 1; try typing them at the command-line[11] – press return after the command to tell the computer to run the command.

10 A Virtual Machine (VM) is a program on your computer that acts like another computer and can run other operating systems. Several VM's are available, VirtualBox http://www.virtualbox.org/ is free and regularly updated.

11 You'll notice that the output of **`pwd`** does not agree with the command prompt, instead printing `/home/ebi`. This is because `~` is a synonym for `/home/ebi`, see Table 2 for more details.

whoami	Your username
hostname	Name of machine being used
pwd	Current directory (Print Working Directory)
uname	Operating system (**uname -a** for the full details).

Table 1: Some basic commands to answer the important questions of life: "who am I?", "where am I?", and "what operating system am I running?"

Files and directories

All files in Unix are arranged in a tree-like structure: directories are represented as branches leading from a single trunk (the "root") and may, in turn, have other branches leading from them (directories inside directories) and individual files are the leaves of the tree. The tree structure is similar to that of every other common operating system and most file browsers can display the filesystem in a tree-like fashion, for example: part of the filesystem for an Ubuntu Linux computer is displayed in Illustration 2. Where Unix differs from other operating systems is that the filesystem is used much more for organising different types of files: the essential system programs are all in ***/bin*** and their shared code (libraries) are in ***/lib***; similarly user programs are in ***/usr/bin***, with libraries in ***/usr/lib*** and manual pages in ***/usr/share/man***.

There are two different ways of specifying the location of a file or directory in the tree: the absolute path and the relative path from where we currently are (the current working directory, see **pwd**, previously) in the filesystem. An absolute path is one that starts at the root and does not depend on the location of the current working directory. Starting with a / to signify the root, the path

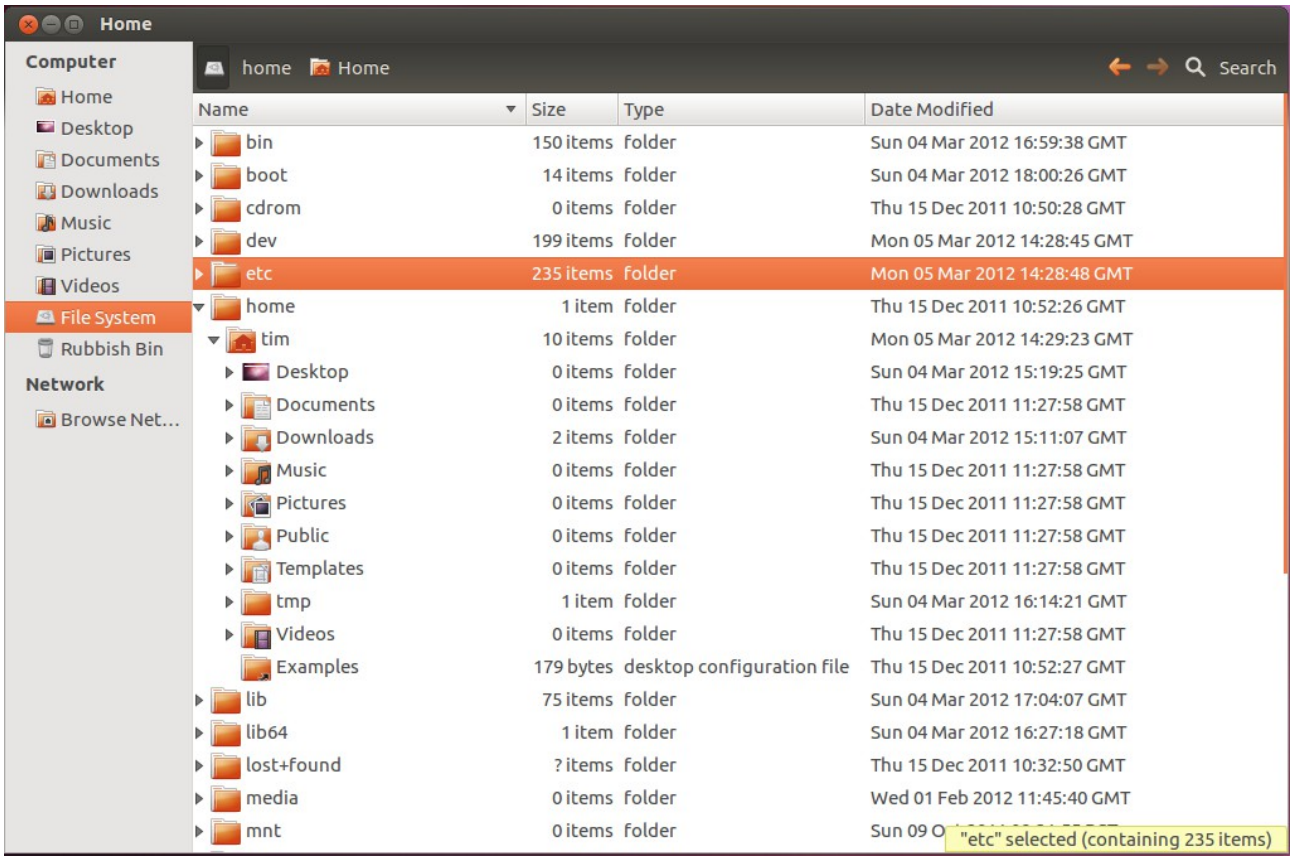

Illustration 2: Tree-like structure of the Ubuntu filesystem. Starting at the root '/', directories are displayed and the home and home/tim directories have been opened to show its contents, relationships indicated by indention. /home/tim contains several more directories which could also be opened.

describes all the directories (branches) we must traverse to get to the file, each directory name separated by a **/**. For example, ***/home/user/Music/TheKinks/SunnyAfternoon.mp3*** refers to the file SunnyAfternoon.mp3 inside the directory TheKinks, which is inside the directory Music, ... , which is inside on the directory home, which is connected to the root. If you are familiar with Microsoft Windows, you might notice that the path separator is different: a forward-slash / rather than the backward-slash \ on Windows; the paths of web pages are also separated by forward-slashes, revealing their Unix origins as a path to a file on a remote machine.

For convenience, a few directories have special symbols that are synonyms for them and the most common of these are listed in Table 2.

Symbol	Description	Notes
/	Root directory	Go to top of tree
.	Current directory	
..	The parent directory	Go up one in tree
~	Home directory	Synonym for $HOME
~user	Home directory for ***user***	

Table 2: Special directory names. Most of the these are only have a special meaning when at the beginning of a path, otherwise they are just a symbol. For example, ***dir/~/*** *is the directory* **~** *inside the directory* ***dir*** *in the current directory, whereas* ***~/dir/*** *is the directory* **dir** *inside the home directory. In both cases the '/' symbols are separators rather than the root directory.*

The current location, the working directory, can be displayed at the command-line using the **pwd** command. Rather than referring to a file by its absolute path, we can refer it by using a path relative to where we are: a file in the current directory can be referred to by its name, a file in a directory inside our working directory can be referred to by ***directory/filename*** (and so on for files inside of directories inside of directories inside of our working directory, etc...). Note that these paths are very similar to how we describe absolute paths except that they do not start with **/**; absolute paths are relative paths relative to the root (alternatively we could read the initial **/** as "goto root" and consider them to be relative paths). As shown in Table 2, the directory above the current directory can be referred to as **..** so, if the working directory is ***/home/user***, then the root directory can be referred to as **../..** (go up one directory, then go up another directory). The symbol **..** can be freely mixed into paths: the directory ***examples*** below the current directory could have path ***examples/../examples/../examples*** (needless to say, simply using just ***examples*** is recommended).

Commands

Commands are just programs elsewhere on the computer and entering their name on the command-line runs them. Commands have a predicable format:

command -flags target

The command is the name of the program to run, the (optional) flags modify its behaviour and the target is what the command is to operate on, often the name of a file. Many commands require neither flags nor target but Unix tools are generally extremely configurable and even simple commands like **date**[12] have many optional flags to change the format of their output.

12 Some utilities also have parodies, see **ddate** or **sl** for example.

As mentioned in Files and directories, there are special directories to contain executable programs and programs within them can be run by typing their name at the command-line. In general you will not have permission to place files in these directories and experienced Unix users create their own, normally ***~/bin/***, to place programs they use frequently[13]. If a program is not in a special directory, you cannot run it just by typing its name: the computer doesn't know where to find it even if the program is in the current directory. Programs which are not in special directories can still be run, but you have to include the path to where it can be found and this can be as simple as ***./program*** (program is in current directory) to a more complex absolute path to somewhere where shared programs are kept (see footnote 13 for a hint of how to alleviate this tedium for commonly used programs) but you can always use the command-line's autocompletion features, see "tab-completion" below, to reduce the amount of typing needed.

One thing you'll quickly discover is that the mouse does not move the cursor in the terminal. The terminal interface predates the popularity of mice by decades and alternative methods of efficiently moving around and editing have been developed, keyboard short-cuts being defined for most common operations. A few of these are listed in Table 3 but probably the most useful is the tab key to complete command names and paths in the filesystem, referred to a 'tab-completion'. Pressing tab once will complete a path up to the first ambiguity encountered and pressing again gives a list of possible completions (you can type the next letter or so of the one you want and press tab again to attempt further auto-completion).

`Control-a`	Move to beginning of line
`Control-e`	Move to end of line
`Alt-f`	Move forward one word
`Alt-b`	Move backwards one word
`Control-l`	Clear screen, leaving current line
`Tab`	Try to automatically complete path

Table 3: Common key bindings for moving around command-line.

```
#     Example of tab completion
#     For the rest of this tutorial, we will assume the user is called ebi
whoami
ebi
pwd
/home/ebi
ls
Desktop    Downloads         Pictures  Templates   Videos
Documents  examples.desktop  Music     Public      Ubuntu One
#     Change to a different directory - don't press enter yet
cd D
#     Pressing tab once has no effect since there are three possible options.
#     Pressing tab again lists the three options, note that cd D remains on the
#     command-line for further editing.
```

13 Creating the directory does not make it special. There is a variable **$PATH** which is a list of directories in which the computer looks for programs and the command **export PATH=~/bin:$PATH** appends the new directory to this list. This command is often added to the file ***~/.bashrc*** which is a list of commands to be run automatically every time a new terminal is opened.

```
cd D<tab><tab>
Desktop/   Documents/ Downloads/
#     Press e to disambiguate options, and tab again to complete.
cd De<tab>
#     Gives
cd Desktop/
```

A record is kept of the commands you have entered, the **history** command can be used to list them so you can refer back to what you did earlier. The history can also be searched: Control-r starts a search and the computer will match against your history as you type; typing enter accepts the current line, typing Control-r again goes to the next match and Control-g cancels the search. History can also be referred to by entry number, listed using the **history** command: entering **!***n* on the command-line will repeat history entry *n*, entering **!!** will repeat the last command.

There are many, often terse, commands for manipulating files and a few of the more useful of these are shown in Table 4. Many of the commands for Unix have short names, often only two or three letters, so errors typing can easily have unintended and severe consequences – be careful what you write because Unix rarely gives you a second chance to correct mistakes. Some Unix machines have the **sl** command to encourage accurate typing.

cd Change Directory – change the working directory

Examples:

- `cd path   # Change working directory to path`
- `cd        # Change to home directory`
- `cd -      # Change to previous directory`

cp CoPy file – copy a file from place to another

Examples:

- `cp file1 file2           # copy file1 to file2`
- `cp file1 directory/      # copy file1 into directory. The copy of the file has path directory/file1`
- `cp file1 file2 directory/ # copy file1 and file2 to directory. When copying multiple files, the destination must be a path to a directory.`

ls LiSt contents of directory

Examples:

- `ls     # List files`
- `ls -a  # Also show hidden files (those whose name begins with a period).`
- `ls -l  # Show more information about each file (permissions, owner, group, time and date of last modification).`

mkdir MaKe DIRectory – create a new directory

Examples:

- `mkdir path                     # Make directory described by path`
- `mkdir -p directory1/directory2 # Make the directory described and all directories leading to it (its Parents) if necessary.`

mv MoVe file – move (rename) a file. Usage is exactly like cp except that the file is moved rather than copied.

Examples:

- `mv file1 file2           # Rename file1 to file2`
- `mv file1 directory/      # Move file to directory`
- `mv file1 file2 directory/ # Move files to directory`

rm ReMove file – remove (delete) a file. Generally deletion on Unix machines is permanent and instantaneous; there is no *trash* directory to save you from your mistakes.

- `rm file          # Remove file`
- `rm directory     # Fails, can't remove directories`
- `rm -r directory # Recursively descend into directory and delete everything, including other directories inside of it (hence the recursively). This will remove the directory`
- `rm -f file       # Force (ignore warnings) removal of file; ignoring warnings includes read-only files.`

rmdir ReMove DIRectory – remove a directory. Only empty directories can be removed (this includes hidden files).

Example:

- `rmdir directory`

Table 4: A few commands for manipulating files and brief explanations.

```
#     Moving and copying
cd /home/ebi/examples/unix/MoveCopy
pwd
/home/ebi/examples/unix/MoveCopy
ls
alignmnet.fasta read-only test test2
mv alignmnet.fasta alignment.fasta
ls
alignment.fasta read-only test test2
cp alignment.fasta alignment_copy.fasta
ls
alignment.fasta alignment_copy.fasta read-only test test2
#     Several files can be moved or copied if the destination is a directory
mkdir MyAlignments
mv alignment.fasta alignment_copy.fasta MyAlignments
ls MyAlignments
alignment.fasta alignment_copy.fasta
```

```
#     Removing files and directories
pwd
/home/ebi/examples/unix/MoveCopy
#     Some files are read-only (answer ‘‘y’’)
rm read-only
rm: remove write-protected regular empty file `read-only'? y
#     Some files are not yours
rm /dev/null
rm: remove write-protected regular file `/dev/null'? y
rm: cannot remove `/dev/null': Permission denied
#     rm will not delete directories
rm test
rm: cannot remove `test': Is a directory
#     rmdir will only delete empty directories
rmdir test
rmdir: `test': Directory not empty
ls test
afile
rm test/afile
rmdir test
ls test2
afile subdirectory
#     rm -r (-r = ‘‘recursive’’) deletes everything, including subdirectories
#     Danger - a mistake using this option can result in a lot of work being
#     accidentally deleted
rm -r test2
```

There are many circumstances when it is preferable for symbols not to have a special meaning, the most common example being when the file name contains a space[14]. The character in question can be “escaped” by prefixing it with a '\' to remove its special meaning so, for example: / is the root

14 A space is a special character in the sense that it is interpreted as a break between command-line options.

directory but \/ is a file called '/'.

```
#     Escaping examples
pwd
/home/ebi/examples/unix/Escaping
ls
my sequences.fasta  sequence_directory
#     Incorrect version. Space in name treated as command-line argument separator.
#     ‘‘my’’ and ‘‘sequences.fasta’’ treated as separate files.
mv my sequences.fasta sequence_directory
mv: cannot stat `my': No such file or directory
#     The wrong file has been moved
ls sequence_directory
sequences.fasta
#     Correct version, the space is escaped to remove its special meaning.
mv my\ sequences.fasta sequence_directory
ls sequence_directory
my sequences.fasta  sequences.fasta
#     If you use ‘‘tab completion’’ to complete names, spaces and other characters
#     are automatically escaped for you.
```

Files beginning with a **.** character are hidden by default and will not appear in the output of **ls** or equivalent. General hidden files are those important for the computer or programs, containing configuration information not intended for the user.

```
pwd
/home/ebi
#     Show ordinary files
ls
doc-samples  examples
#     Show all files. Note the special directories . and .. are visible.
ls -a
.            .bash_history .bashrc .directory .mpd.conf    examples
..           .bash_logout  .config .kde4      .ssh
.Xauthority .bash_profile .dbus   .local     doc-samples
#     Show only hidden files. For more details, see Dealing with multiple files.
ls -d .*
.  .Xauthority   .bash_logout  .bashrc .dbus      .kde4  .mpd.conf
.. .bash_history .bash_profile .config .directory .local .ssh
```

Reading and writing permission

All files and directories have a set of permissions associated with them, describing who is allowed to read or write a file. There are three basic permissions: read **r**, write **w** and execute **x.** The meanings are fairly obvious other than execute, which has two meanings depending on context: for normal files, execute is just a marker to show that the file contains executable code (i.e. is a program) but execute permission is also needed to open a directory and see the files is contains. There are three categories of user: owner **u**, group **g**, and other **o** and the permissions for each file are described as a string of nine characters, three for each user category. The triplet for each category is either a letter '**rwx**' if users in that category have the corresponding permission or '**-**' if they don't. The permission string **rwxr-x---** means that the owner has permission to read, write or execute, users in the same group have read and execute permission and other users have no permissions.

```
#     Which groups do I belong to?
pwd
/home/ebi
id
uid=521(ebi) gid=100(users)  groups=6(disk),7(lp),11(floppy), ↩
17(console),27(video),80(cdrw),100(users),521(ebi),1002(boinc) ↩
,65533(nogroup),65534(nobody)
#     User ebi has ID 521, in group users by default (Group ID 100). ebi is also a
#     member of several other groups, giving access to features of the computer
#     that would otherwise be denied.
#     List some files and permissions. The initial 'd' means that both doc-samples
#     and examples are directories. The owner of these files is ebi and they are
#     part of the users group.
mkdir test
ls -l
drwxr-xr-x 14 ebi users 352 Mar 22 14:02 doc-samples
drwxr-xr-x  3 ebi users  72 Mar 22 17:42 examples
drwxr-xr-x  2 ebi users  48 Mar 26 14:09 tests
#     ebi has read, write and execute permission for both directories; users in the
#     group users have read and execute permission, as do any other users.
#     Remove execute permission for all users (a means u, g and o)
chmod a-x test
#     Now we can't open the directory.
cd test
-bash: cd: test: Permission denied
#     Give ourselves permission to enter directory.
chmod u+x test
#     This now succeeds but nobody else will be able to enter
cd test
#     Make all files in and below a directory read-only. The -R flag means
#     recursively descend into all directories inside test.
chmod -R a-w test
```

As the owner of a file you can change its permissions to be anything and some programs do this for you automatically, giving the impression that the permissions have been ignored. Running **rm -f** is possibly the only time you might run into this behaviour: by default **rm** will prompt to remove write-protected files but the **-f** (force) flag turns tells it not to bother asking and just remove the file.

Dealing with multiple files

Often, especially when running scripts or organising files, it is desirable to deal with multiple files at once. Rather than typing each file name out explicitly, we can give the computer a pattern instead of a filename: all filenames are checked against the pattern and it is automatically replaced by a list of matching files before running the command. Patterns are just filenames containing symbols that have a special meaning, for example: ***** means match anything, so **a*b** is a pattern that matches any filename beginning with **a** and ending with **b** including the file **ab**. Table 5 contains a list of special symbols useful for constructing patterns.

Symbol	Description
`*`	Match anything, including match "nothing"
`\*`	Match a '*' character
`?`	Match any character exactly once (excludes matching "nothing")
`[abc]`	Match exactly one of 'a', 'b' or 'c'
`[^abc] or [!abc]`	Match any character but 'a', 'b' or 'c'
`[c-y]`	Match any character between 'c' and 'y'. Note: [-a] is defined to mean a match with '-' or 'a'.
`{pattern1,pattern2}`	Combines the result of pattern1 and pattern2 together. Note if a file is matched by pattern1 and pattern2, it is returned twice.

Table 5: Special symbols for filenames. As with the `\*` *example in the table, any of these symbols can be prevented from having a special meaning by "escaping" them with a '\'.*

```
#     Organise files by type. When moving or copying multiple files, the final
#     argument must be a directory not a file.
pwd
/home/ebi/examples/unix/MultipleFiles
mkdir Fasta Tree Sequences
cp *.fasta Fasta
cp *.tree Tree
#     Copy both Fasta and Fastq format files. Any other files with the suffix .fast?
#     or .f? would also be matched, .fz for example which is occasionally used for
#     compressed fasta files.
cp *.fast? *.f? Sequences
      A more restrictive form, only matching fasta and fastq format
cp *.fast[aq] *.f[aq] Sequences
```

As mentioned above, pattern matching occurs before a command is run and the pattern is replaced by none, one or more matches. The command never sees the pattern, just the results of the match and this can have unintended consequences.

```
#     Why we used the -d flag for ls in the previous section
pwd
/home/ebi
cd
#     Now in home directory. The pattern is expanded to all files and directories
#     matching, including the . and .. directories. When ls is run, it gets the
#     directory names as arguments and so lists their contents.
ls .*
.Xauthority    .bash_logout   .bashrc     .mpd.conf
.bash_history  .bash_profile  .directory

.:
doc-samples  examples

..:
cilia  ebi      giorgos jon      natassa    pet      spyros  tkill
costas fix      ioanna  katerina nodas      pierre   tereza  voula
elena  gioannis jacques maria    panagiotis pvavilis thanos
```

```
… etc
#     The -d flag for ls stops it from listing the contents of directories and
#     instead just prints their names.
ls -d .*
.  .Xauthority   .bash_logout  .bashrc .dbus      .kde4  .mpd.conf
.. .bash_history .bash_profile .config .directory .local .ssh
```

Unintended consequences can be dangerous. Take special care when using patterns with commands. The following example is a "joke" played on inexperienced Unix users.

```
                    This example can be dangerous - BEWARE
pwd
/tmp
      Create a file called -rf *
touch -- -rf\ \*
#     Typing rm -rf * is a really bad idea. * matches everything, r means
#     recursively descend down all directories matched, and f forces deletion even
#     if the file is important or write-protected.
#     Safely delete file. There are other ways.
rm -- '-rf *'
#     The quote marks '-rf *' stop the filename being interpreted as a pattern. The
#     -- prevents rm (and, more generally, most commands) from interpreting
#     anything after it as a flag, so -rf is just a name not the recursive and force
#     flags.
```

Running multiple programs

From early on in its development, Unix was designed to run multiple programs simultaneously on remote machines and support for this is integrated into the command-line. An important distinction is that between foreground jobs and background jobs: a foreground job temporarily replaces the command-line and you cannot enter new commands until it has finished, whereas a background job runs independently and allows you to continue with other tasks. Only foreground jobs receive input from the keyboard, so interactive programs like PAUP* should be run as foreground (although you could set up a compute intensive analysis, background it and continue with other tasks while it is running. Later, when the calculations have finished, the program can be made foreground again so interaction can continue). Background jobs still send their output to stdout, your terminal unless you have redirected it somewhere else, which can be confusing if you are running multiple background jobs – their output will be interleaved without any indication of which line came from which job.

Control-c	Cancel (kill) foreground job
Control-z	Pause foreground job
jobs	List current jobs (started in this command-line)
kill %*n*	Kill job number *n*
killall *name*	Kill all processes called *name*
ps	Show all running processes (distinct from jobs) regardless of how they were started
fg %*n*	Bring job number *n* to foreground
bg %*n*	Run job number *n* in background
program &	Start *program* in background
nohup *program*	Run *program* in background so it will not stop if you log out. **stdout** and **stderr** are redirected to the file nohup.out. Advanced users might like to look at **screen** instead.

Table 6: A few commands and key combinations for job control.

As hinted in Table 6, there is a difference between a job and a process. A process is a single program running on the machine, each of which is uniquely numbered (a pid, Process ID). You can list all the processes you are running, including the command-line itself[15], using **ps** (or **ps -a** if you want to see what all the other users of the machine are doing). The command-line itself is just another process running on the computer, albeit one specially designed for starting, stopping and manipulating other processes. Processes are the fundamental method of keeping track of what is running on the computer. Jobs, on the other hand, are things entered on the command-line and many include several programs logically connected together by pipes (see In, out and pipes for details) to achieve a task. The command-line splits the jobs into several processes and runs them, possibly simultaneously.

```
#     Time a cup of tea. Computer sleeps for 300 seconds and then prints Tea
#     brewed.
sleep 300; echo 'Tea brewed!'
Tea brewed!
#     Backgrounding does not have the effect you might think. The following
#     backgrounds the echo but not the sleep; backgrounding the sleep would have
#     the effect of immediately running both the programs. Similarly, interrupting
#     the sleep with Control-z will immediately allow echo to run.
sleep 300; echo 'Tea brewed!' &
#     Correct method of backgrounding: group programs using brackets and background
#     entire group. The space between the bracket and the command is important.
{ sleep 300; echo 'Tea brewed!'; }&
```

In, out and pipes

Where possible, Unix commands are written as filters: they read from input, manipulate the data and write the output. This might sound trivial, tautologous even, but it enables simple commands to be combined to produce complex results. Every command reads from **stdin** and writes to **stdout**, by default **stdout** is connected to the current command-line, so results are displayed on the screen, but it can be redirected: **> filename** redirects **stdout** to the file specified for later perusal. Rather than

15 Generally called **bash**, the Bourne Again SHell, a pun on the original Unix shell written by Stephen Bourne.

redirecting to a file, a pipe can be used to connect **stdout** of one command to the **stdin** of another – by chaining many simple commands together, complex transformations of the input can be achieved.

Following is an advanced example, showing how a complex output can be achieved using a series of smaller steps. You don't know sufficient yet to understand everything in this example but try to work through it and see what each step is doing. The man pages for each command (see Getting help) might be useful.

```
#     An advanced example using pipes. At the command-line, a pipe is represented
#     by the character |.
pwd
/home/ebi/examples/unix/Pipes
#     The compressed file transcripts.fasta.gz contains the fasta
#     sequences of all transcripts of Homo sapiens chromosome 22
#     (from Ensembl release 57). Want to count how many transcripts
#     there are for each gene.
#     First cat reads the file and writes it to stdout, pipe into the
#     decompression program gunzip
cat transcripts.fasta.gz | gunzip |
#     Another pipe into grep, a tool that extracts lines matching a
#     certain pattern (here, those starting with a >). The pattern to
#     match is described using a Regular Expression - a very important
#     concept than underlies many Unix tools but too advanced for this
#     tutorial.
grep '^>' |
#     The sequence names in the fasta file are of the form
#     >GENE_ID|TRANSCRIPT_ID
#     cut splits a line to bits. Use | as the delimiter and we want
#     the first field. The single quotes prevent the '|' being
#     interpreted as a pipe by the command-line.
cut -d '|' -f 1 |
#     Sort and count the unique entries (uniq requires sorted input)
#     Output to file transcript_counts.txt
sort | uniq -c > transcript_counts.txt
#     Or on one line, redirecting stdin in a similar way to how we redirected
#     stdout before. The < transcripts.fasta.gz connects the stdin of gunzip to the
#     file transcripts.fasta.gz rather than using cat to read it and output it into
#     a pipe.
cat transcripts.fasta.gz | gunzip | grep '^>' |  cut -d '|' -f 1 | ↩
sort | uniq -c > transcript_counts.txt
```

Compression

The aim of compression is to make files smaller, useful for both saving disk space and making it quicker to send files over the internet[16] Simply, compression programs look for frequently repeated patterns in the file and remove this redundancy in a manner that can be undone later. Text files tend to compress very well, 100MB worth of Wikipedia being compressed into less than 16MB[17], and, in

16 Some types of connection over the internet have the ability to transparently compress files before sending and uncompress at the other end. Some web servers implement this but the only important example for us is **scp** / **sftp** which can be given the **-C** option to request compression. E.g. **scp -C sequences.txt auser@anothermachine.org:/home/ebi/**

17 See The Hutter prize http://prize.hutter1.net/

particular, biological sequences tend to be very compressible since the size of the alphabet of nucleotides or amino acids is small compared to the total computer alphabet of all lower-case and upper-case characters, numbers, symbols, etc.

There are two common tools for compressing files: **gzip** and **bzip2** with their respective tools for uncompressing: **gunzip** and **bunzip2**. **gzip** is the de-facto standard; **bzip2** tends to produce smaller files but takes longer to compress them. On the Windows platform, the Zip[18] compression method is favoured and many Unix platforms provide **zip** and **unzip** tools to deal with these files. Non-Linux Unix platforms, Mac OsX for example, have older tools called **compress** and **uncompress** that are rarely used any more. Support for **compress**'d files on Linux can be patchy, for example: a machine I have access to has a **compress** manual page but no actually tool installed. A final method to be aware of, that is becoming more popular, is 7-zip (**7za**) which can produce smaller files than all the above methods, again at the expense of taking longer to compress. A list of file suffices that can be used to identify what files are compressed using what method is provided in Table 7.

```
pwd
/home/ebi/examples/unix/Compression
ls
sequences.fq
#     Gzip'ing a file
gzip sequences.fq
ls
sequences.fq.gz
#     Unzipping the file
gunzip sequences.fq.gz
ls
sequences.fq
#     Using bzip2
bzip2 sequences.fq
ls
sequences.fq.bz2
#     As part of a pipe, reading from stdin and writing to stdout
cat sequences.fq.bz2 | bunzip2 | gzip > sequences.fq.gz
ls
sequences.fq.bz2  sequences.fq.gz
```

Compression works better if files are combined and then compressed together, rather than compressing them individually, since this allows the compression program to spot repeated patterns between the files. On Unix, the process of packing/unpacking several files into / from a single file has been historically separate from the process of the compression, in keeping with the philosophy of having little tools that do one thing well. The Unix tool for packing and unpacking files is **tar** "Tape Archiver", the odd name because its heritage goes back to 1979 when writing files to magnetic tape was a common method of storage.

```
ls
chimp.fasta human.fasta macaque.fasta orangutan.fasta
#     Pack into single file. The suffix is your responsibility. 'c' means create, and
#     'f' means that the next argument is the filename to write to.
tar -cf sequences.tar *.fasta
#     Note that the original files are untouched
```

18 Popularised and often known as Winzip (http://www.winzip.com/) but originally invented by Phil Katz as **pkzip** and now handled automatically by Windows and Mac OsX.

```
ls
chimp.fasta human.fasta macaque.fasta orangutan.fasta sequences.tar
# Delete all sequences
rm *.fasta
#     'x' means extract
tar -xf sequences.tar
ls
chimp.fasta human.fasta macaque.fasta orangutan.fasta sequences.tar
```

Over time, the features of **tar** have increased to make it more convenient and modern versions are now capable of packing and compressing files.

```
ls
chimp.fasta human.fasta macaque.fasta orangutan.fasta sequences.tar
#     Pack and gzip sequences simultaneously
tar -zcf sequences.tgz *.fasta
#     List the contents without extracting
tar -ztf sequences.tgz
chimp.fasta
human.fasta
macaque.fasta
orangutan.fasta
#     More recent versions of tar can also bzip2 files
tar -jcf sequences.tbz2 *.fasta
tar -jtf sequences.tbz2
chimp.fasta
human.fasta
macaque.fasta
orangutan.fasta
```

Compression	Uncompression[19]	Suffix	Tar'd Suffix
gzip	**gunzip**	.gz	.tgz
bzip2	**bunzip2**	.bz2	.tbz2
compress	**uncompress**	.Z	No convention, .tar.Z
zip	**unzip**	.zip	Not needed
7za	**7za e archive.7z**	.7z	Not needed

Table 7: File suffices for common compression programs. When combined with **tar** *to compress multiple files, often the full suffix* ***.tar.suffix*** *is shortened to that given above.* **zip** *and* **7za** *"7-zip" have a Windows heritage and have built methods to combine multiple files together, so are rarely used in conjunction with* **tar**. *The* **file** *tool can also be used to determine file type, e.g:* **file file.unknown.suffix** . *See* **man file** *for details.*

Working on remote computers

Why use a remote computer? There are many reasons: Firstly, central computing resources tend to

19 The compression programs actual do both compression and decompression. These names are convenience synonyms for the compression program and whatever command-line options it requires to flip it into decompression mode.

be much larger, more reliable[20] and more powerful than your laptop or PC – if you need to do a lot of work, or use a lot of data then you may have no option but to use a bigger computer. If you have a job that will take a long time to run, Bayesian phylogenetic methods being one example, you may not want to commit to leaving your personal computer for long enough (and you really trust your colleagues not to turn it off?) whereas central facilities are permanently on and have batteries to prevent small glitches in the power supply from affecting the computers. Lastly, and most importantly, central computers tend to have much more rigorous and tested policies for backing up data – Do you backup? Is it kept in a separate physical location from the original? When was the last time you checked that the backup actually worked?

Secure SHell is a method of connecting to other computers and giving access to a command-line on them; once we have a command-line we can interact with the remote computer just like we interact with the local one using the command-line. SSH replaces an older method of connecting to remote computers called **telnet**, which sends everything – including your password – as normal undisguised text so anyone can read it. Never use **telnet** unless you know what you are doing and you have no other option; similarly, never use FTP `File Transfer Protocol' for transferring files.

```
#     Connect to a remote computer, here we connect as the user auser to the
#     computer with address anothermachine.org
#     You will need an account on another computer to get the most out of this
#     exercise.
ssh auser@anothermachine.org
Password:
#     You are prompted for a password and then have a command-line
[auser@anothermachine ~]$
#     Type exit to leave.
#     If the username is left off, ssh assumes that the remote username is the same
#     as the local one. Depending on how the local network is set up, machines on
#     it might be referred by their names rather than their full address.
#     This might work and saves a lot of typing
whoami
ebi
ssh anothermachine
```

As well as keeping communications between your computer and a remote computer secure, SSH also allows you to verify that the remote computer is the computer it claims to be – no point keeping traffic secure if you send it to the wrong place – or to prevent someone sitting in the middle of the connection listening to each message then passing it on, pretending to each side to be the other[21]. If verification fails, you will be warned with a message like:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

20 There is world of difference between server-quality hardware and stuff on your desk. Battery backup, Uninterruptible Power Supplies, are one example, servers also tend to have redundant components and memory that can detect and correct errors. At the top end, servers can detect and isolate faulty parts, report the problem and continue running: often the first time companies know that a fault occurred is when an engineer turns up with a replacement part.

21 Known as a Man-in-the-Middle attack http://en.wikipedia.org/wiki/Man-in-the-middle_attack . Both sides think they are communicating with the other but are actually communicating with an intermediary who copies all messages then forwards them on. The method use to verify identity, without possibility of forgery, and even if someone else can copy and manipulate all messages is very interesting and has many other uses like to tell if a message has been send from who it says it has and whether or not it has been tampered with; see http://en.wikipedia.org/wiki/Public-key_cryptography and http://en.wikipedia.org/wiki/Digital_signature for details.

```
@       WARNING: POSSIBLE DNS SPOOFING DETECTED!          @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
The RSA host key for gate.ebi.ac.uk has changed,
and the key for the corresponding IP address 193.62.197.203
is unknown. This could either mean that
DNS SPOOFING is happening or the IP address for the host
and its host key have changed at the same time.
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
76:55:ba:23:87:f8:34:ca:d0:28:80:5d:c6:fb:f9:4f.
Please contact your system administrator.
```

and the computer will refuse to connect. By far, the majority of these warnings are caused by inept computer administration rather than malice – someone has upgraded the other machine incorrectly so it appears to be a different computer; if you are sure it is safe, the warning can be dealt with by deleting the appropriate line for the computer from the ***~/.ssh/known_hosts*** file.

In keeping with its heritage of terminals to remote computers, graphical programs can also be run on remote machines but expect pauses unless you have a low-latency internet connection. The system that enables this is called the X Windows system[22] (or just X, or X11), hence the use of the **-X** flag on following example, and requires software on your local computer that understands the drawing instructions being sent. Linux computers use such software by default for display, Mac OsX comes with software that can be used (and is started automatically by ssh in the following example). On Windows, the Cygwin software provides the required functionality.

```
#     Connect to anothermachine, using the -X flag to enable X11
ssh -X auser@anothermachine.org
#     This gives us a command-line on anothermachine. Start a graphical
#     application; the application that appears is running on anothermachine
#     not your local computer.
gtk-demo &
#     Get another command-line on anothermachine
xterm &
```

Transferring files

As shown in Nice examples, it is possible to transfer files between computers using SSH alone but this is not recommended since more friendly interfaces exist.

```
pwd
/home/ebi/examples/unix/SCP
#     Secure CoPy, scp,  is version of cp
#     Copies transcripts.fasta.gz to anothermachine.org, into the directory
#     ~ebi/examples/ Paths on the remote machine could also be specified absolutely,
#     like.
auser@anothermachine.org:/home/ebi/examples[23].
```

22 The successor to the W Windows System, if you are wondering where the X came from.

23 This is not too dissimilar to how remote files (web pages) are described on the web, which look like ***http://username@computer.address:portnumber/path/to/file***. The portnumber and username

```
scp transcripts.fasta.gz auser@anothermachine.org:examples
#     Secure FTP, sftp, acts like an older method for transferring files and is more
#     useful for transferring multiple files or if you cannot remember the full
#     remote path.
sftp  auser@anothermachine.org
Connecting to anothermachine.org...
Password:
sftp>
#     Have something that looks like a command prompt. Type help for a list of
#     commands that it understands. The cd, ls, mkdir, pwd, rm, put and get
#     commands (and their "local" variants lcd, lls, lmkdir and lpwd) are of
#     especially useful. For multiple files, there are mput and mget commands.
#     Type bye, exit or quit to leave.
bye
```

Of course, there are many graphical file transfer programs available. Without recommending particular programs, Cyber-duck http://cyberduck.ch/ for the Mac OsX and WinSCP http://winscp.net/ for Windows appear to good options but there are many more. Alternatively, under Mac and Unix, it is possible to mount directories on remote computers so there appear to be local; search for ***sshfs*** for details.

When transferring files, silent errors are extremely rare but can happen and so we'd like to be able to verify that the file received is identical to the one sent. Short files could be checked by eye but this can't be automated without transferring the file again (which might also get an error). A common technique to verify correct transfer is to calculate the *md5* (Message Digest algorithm 5) of both files and compare these values. The md5 is short string of characters that identifies a file and two different files are extremely unlikely[24] to share the same string – if a file changes, its md5 will (very probably) change and so we know that that a change occurred. It is extremely difficult to deliberately create two files that have the same sum.

```
#     Calculate the md5 of a file
pwd
/home/ebi/examples/unix/Pipes
md5sum transcripts.fasta.gz
85e2bf58ff544a4f2d8b0e5aca37b726   transcripts.fasta.gz
#     Repeat on remote version of file and check the md5's agree. If the remote
#     machine is Mac, on which the tool is called md5 rather than md5sum
md5 transcripts.fasta.gz
MD5 (transcripts.fasta.gz) =  85e2bf58ff544a4f2d8b0e5aca37b726
#     Was the examples files we downloaded the one intended?
#     If you're md5sum doesn't match the one below, the examples file has changed.
wget -O  -  'http://tinyurl.com/32a2gbk/unix.tgz' | tar -zx
b00fbf51d3c7f2c5113de3be330c25a1  -
```

More rarely, you may come across *SHA* sums, **shasum** on both Unix and Mac computers, which are very similar to md5's but have an even smaller chance that two files share the same string.

---

are almost never used in practise so tend not to be explicitly written. If you examine your spam email closely, you will occasionally see links using a username to hide its true destination:
http://www.bank.com@www.dodgysite.com/

24 The chances of two non-identical random files having the same md5 is about 3.4×10^{-38}. When checking large numbers of files, the Birthday Paradox (http://en.wikipedia.org/wiki/Birthday_problem) occurs and the chance that there are two files in the set with the same md5 decreases rapidly but will still be small enough for realistic uses.

Getting help

General help with Ubuntu has already been covered in "Acclimatisation", alternatively just find someone to ask.

As with everything else, the web is a verdant source of good, bad and down-right weird[25] tutorials. Unix is general very well documented, although the documentation is generally aimed at experienced users. The manual pages follow the same format, starting with a description of what the command does and a summary of all its flags; optional flags are enclosed in square brackets. Next is generally a full description of the command and detailed descriptions of what each flag does. Sometimes there is also a section containing examples of usage, Mac OsX is generally very consistent about this but Linux derivatives can be a mix.

```
#     Look at manual page for man
man man
man(1)                                                          man(1)



NAME
       man - format and display the on-line manual pages

SYNOPSIS
       man  [-acdfFhkKtwW]  [--path]  [-m system] [-p string] [-C config_file]
       [-M pathlist] [-P pager] [-B browser] [-H htmlpager] [-S  section_list]
       [section] name ...


DESCRIPTION
       man formats and displays the on-line manual pages.  If you specify sec-
       tion, man only looks in that section of the manual.  name  is  normally
…
#     You can use the arrow keys to move up and down the man page, alternatively
#     the page-up and page-down can be used to scroll entire pages at once (in some
#     terminals you'll have to hold shift and press page-up/down to get the same
#     effect). Pressing 'q' quits the viewer and returns back to the command-line.
```

Variables and programming

So far, we have only used the command-line to run other programs and to chain them together to achieve more complex results. The command-line is a programming language in its own right and we can write little programs to automate common tasks; often this referred to as *scripting* rather than *programming* although the distinction is not really relevant. Obviously learning to program is not something that can be taught in an hour or two, even experienced programmers take several days to become productive in a new language, so this section can give little more than a taste of what is possible and hopefully show how you could save a lot of time. If you are doing similar things to large

25 Try Why's Poignant Guide to Ruby http://www.ember.co.nz/files/resources/whys-poignant-guide-to-ruby.pdf or Learn You a Haskell for Great Good http://learnyouahaskell.com/ for a taste of just how strange programming tutorials can get. I recommend neither of these languages: python http://python.org/ is a good choice for someone with little programming experience starting out in bioinformatics, especially in conjunction with the biopython http://biopython.org/ libraries.

number of files, many sequences for example, scripting can save you a lot of time and allow you to get on with something else rather than repetitively typing variations on the same thing with inevitable mistakes (think about how you would rename 100 files, or change the format of thousands of gene alignments so they are compatible with your phylogeny program). As with everything, there are many tutorials available on the web and a search for *bash scripting tutorial* or *bash scripting introduction* will yield many examples of varying completeness and comprehensibility.

The first thing to introduce are variables. A variable is just a name for another piece of data, a useful analogy is that of a labelled box: every time we see the label, we replace it with the contents of the box. The ability to manipulate variables, changing the state of the computer, have many uses and are considered fundamental to imperative programming but we'll just introduce two useful cases: shortening common directory paths and performing the same operations on many files. In bash scripting, variables are referred to as **$NAME**. There are some restrictions on the punctuation that can be part of a name and it cannot start with a number.

```
#     Some variables are already defined for us. We have already met $HOME which is
#     a variable containing the path to your home directory
echo $HOME
/home/ebi
#     The other defined variables can also be listed using the env (environment)
#     command. Few of these relevant to us. The output is of the form:
#     NAME=contents
env
…
PYTHONDOCS_2_6=/usr/share/doc/python-docs-2.6.4/html/library
LESSOPEN=|lesspipe.sh %s
AMANDA_GROUP_GID=87
R_HOME=/usr/lib64/R
…
#     The data files for blast are in /bio_data/blastdb
BLASTDATA=/bio_data/blastdb
echo $BLASTDATA
/bio_data/blastdb
#     List contents of directory /bio_data/blastdb
ls $BLASTDATA
GO_db           ego   fish_models_49  ncbi        nt         uniprot
SeaBream_BACend elena jacques         ncbi_taxa   swissprot  vector
data            est   mge             nr          tereza
#     Variables are only accessible from the current command-line. Programs you
#     start from it, other scripts for example, will not be able to see them; they
#     can if declared using export.
export BLASTDATA=/bio_data/blastdb
```

A variable can be the name of a file and we can write things at the command-line using the variable instead of the name explicitly – change the variable and we run exactly the same commands on a different file. One way to take advantage of this this would be to set the variable to one of several files and use the *history* to repeat a set of commands. Of course, if the commands write their output to a file then that would have to be renamed each time otherwise the output for each file would be written over that for the previous. Shell scripting provides an alternative: the computer can be told to set the variable to each of many file names in turn and the value of the variable can be edited automatically to provide the name of a unique output file.

```
FASTAFILE=sequences.fasta
```

```
#     The special form ${VAR%%suffix} is the variable with suffix removed from it.
#     There is an equivalent but less useful version for prefixes ${VAR##prefix} and
#     many others.
PREFIX=${FASTAFILE%%.fasta}
echo $PREFIX
sequences
#     This behaviour can be used to loop through and manipulate many files.
pwd
/home/ebi/examples/unix/Scripting
#     This example requires the muscle alignment program to be installed. The
#     following will only work on Ubuntu and a few other versions of Linux.
#     Like a lot of software, there is 'package' of muscle ready to be installed
#     on your computer. The program that downloads and installs packages is called
#     apt-get. The program sudo ('super-user do') takes its command-line options
#     (the name of another program) and runs it with permission to make changes to
#     your computer.
#     You will need to be connected to the internet for this to work.
sudo apt-get install muscle
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed
  muscle
#     etc … You may need to type 'y' to confirm that you want to install the
#     package
ls
ENSG00000000971.fasta ENSG00000107593.fasta ENSG00000161326.fasta
ENSG00000002745.fasta ENSG00000107643.fasta ENSG00000161544.fasta
ENSG00000003096.fasta ENSG00000107859.fasta ENSG00000161940.fasta
ENSG00000003987.fasta ENSG00000107863.fasta ENSG00000162009.fasta
ENSG00000003989.fasta ENSG00000107890.fasta ENSG00000162402.fasta
#     + lots more. How many? Use wc (Word Count), -l means count lines instead of
#     words (see man wc for more details).
ls *.fasta | wc -l
1015
#     Use a for loop to go through all 1015 fasta files in a directory and align
#     them.
mkdir output
#     Remember that *.fasta expands to the name of every file ending in .fasta
#     The for loop sets the variable I to the name of each fasta file in turn and
#     runs all the code between the do and done statements.
for I in *.fasta
do
      # The variable I holds each file in turn. Get prefix
      PREFIX=${I%%.fasta}
      # New varible holding name of output file, ends in .align
      OUTPUT=$PREFIX.align
      #  cat the file whose name is held in variable $I into muscle to align it and
      # send output (the alignment) to the file whose name is in variable $OUTPUT.
      cat $I | muscle -maxiters 2 > output/$OUTPUT 2>/dev/null
      # Update us on progress
```

```
      echo Alignment of $I written to output/$OUTPUT
      # Go back to beginning and repeat using the next file.
done
#     Takes a few minutes but we have aligned more than a thousand sets of
#     paralogues.
```

A common Unix idiom is to place frequently used sets of functions into a file, called a script, for reuse and so preventing errors retyping them in. Writing a file also means that complex scripts with many steps can be tested before committing to running them over many files, something that could potentially take days if we are dealing with large numbers of genes. Scripts can be written and modified in any common editor but must be saved in text format; *nano* is a good basic editor that is fairly intuitive to use but there are many others more specifically designed with programmers in mind. Alternatively you could use **gedit**, a program more like Notepad on Windows (click the Ubuntu button and search for gedit; entering **gedit &** at the command-line will also work).

```
#     Create a simple script, one that just prints a variable. The single quotes
#     here are very important - anything enclosed in single quotes is not processed
#     by the command-line and treated exactly as it was typed it. In particular,
#     variable names are not replaced by what they refer to.
pwd
/home/ebi/examples/unix
echo 'echo $VAR' > script.sh
#     Make the script executable
chmod +x script.sh
VAR="Hello"
#     A 'dot' followed by a name runs a script at the current command-line, as if
#     you had typed it in yourself.
. script.sh
Hello
#     We've made the script executable, so we can run it. This starts a new
#     command-line to run the program.
./script.sh

#     Nothing happened. This is because the script was run as a separate program
#     and we did not export the variable we created.
export VAR="Hello"
./script.sh
Hello
#     Scripts can also have command-line arguments of their own which can be
#     accessed using the special variables $@ (all arguments), $1 (the first
#     argument), $2 (the second argument), etc
cat > script.sh
echo $@
echo $1
echo $2
#     control-d to end input
./script Hello Goodbye
Hello Goodbye
Hello
Goodbye
```

Line endings – compatibility problems

Even after the standard alphabet for computers was established (ASCII – American Standard Code for Information Interchange) there was no agreement about how to how to indicated the end of a line. ASCII provides two possibilities: line-feed '**\n**' and carriage-return '**\r**', based on how old type-writers and tele-type terminals used to work: a carriage-return moves the carriage, the position to print the next character at, back to be beginning of the line and line-feed moves the paper one line down but doesn't change where the carriage is. On Unix a '**\n**' character is taken to mean "line-feed and carriage return" and this is used to separate lines of text. On Windows, lines are separated by the pair of characters '**\r\n**' (in that order) and old versions of Apple operating systems (prior to OsX) use '**\r**' to separate lines. The situation on Mac OsX is more complex since it must deal with both its Mac and Unix heritage; officially '**\n**' now separates lines in files but programs have to be able to deal with both conventions.

```
#     Unix provides a simple tool, tr "translate", to change the line endings in a
#     text file to a Unix compatible ones. tr either translates one character into
#     another or deletes a specific character.
#     Old Mac to Unix: The first argument is the character to convert and the second
#     argument is the character to convert it into.
pwd
/home/ebi/examples/unix/LineEndings
cat chimp_mac.fasta | tr '\r' '\n' > chimp1.fasta
#     Windows to Unix: The -d flag means delete the characters in the second
#     argument.
cat chimp_win.fasta | tr -d '\r' > chimp2.fasta
#     Look at md5 for files. Note that chimp1.fasta and chimp2.fasta are identical,
#     whereas the other two files are different.
md5sum *.fasta
6650a16886a6c8f8cbf878edbc29804a  chimp1.fasta
6650a16886a6c8f8cbf878edbc29804a  chimp2.fasta
6e611e58b3e5dc7a4689fab0d8cf938d  chimp_mac.fasta
f04ffe97cb945c23c46d751d7281b9e6  chimp_win.fasta
```

To further complicate things, some methods of transferring files between machines try to automatically convert the line endings for you. This is generally a mistake. Specifically an old file transfer method called FTP "File Transfer Protocol" has two modes: text and binary, text mode will attempt to translate line endings. Unix platforms default to binary and are safe, the only case where you need to be careful is transferring files from Windows using the command-line FTP application. **If you transfer a binary file over FTP in text mode, the received file will be corrupted irretrievably.** If in doubt, see Transferring files for how to verify that your file has transferred correctly.

If you've managed to read through to here, you're probably thinking: a) that's complicated, and b) why haven't I noticed this? The answer is that it used to cause problems in the past but programmers are aware of the issues nowadays and programs tend to do the right thing. Some programming languages like Perl even deal with these problems transparently so even programmers don't need to be aware of them any more.

Nice examples

Here follows some examples of using the command line. You should be able to work out what most

of them are doing, although you may have to refer to the relevant **man** page to determine what a specific option does.

Most of the following examples use `regular expressions' to extract specific bits of text from a large file. In a nut shell, regular expressions are a way of describing patterns to be matched and programs are available to search through a file and print the matches (**grep**) or search, edit and replace the matches (**sed**). Understanding and writing regular expressions are beyond the scope of this tutorial but can quickly save you a lot of time and effort. A large part of the popularity of the language PERL (Practical Extraction and Report Language[26]) in bioinformatics is due to the ease that regular expressions can be used within it.

```
#     Reverse complement sequences, assuming one sequence per line
rev sequences.txt | tr 'ACGTacgt' 'TGCAtgca' > rc_sequences.txt
#     Transfer files without using a special program
tar -zc *.fasta | ssh auser@anothermachine.org "cat | tar -zx"
#     How many sequences do we have in a file?
#     Uses grep to match lines beginning with >
#     The initial caret ^ means match the beginning of a line.
grep '^>'  sequences.fasta | wc -l
#     Extract sequence names from Fasta format file
grep '^>' sequences.fasta | cut -c 2-
#     Extract names from a phylogenetic tree
#     Uses grep to match either an open-bracket or a comma and then matches all
#     characters until a colon, close-bracket or comma is found. The square bracket
#     mean match any character inside, unless the first character inside is a caret,
#     in which case any character except those inside are matched. The \+ means
#     that multiple characters should be matched.
#     The tr is to remove extra characters that may be matched (if you think about
#     the tree format, only open-bracket and comma actually possible).
grep -o '[(,][^:),]\+' tree.txt | tr -d '():,'
#     Extract branch lengths from a phylogenetic tree
#     Uses grep to match text that starts with a colon (branch lengths always
#     follow a colon) and matches every character until a close-bracket or a comma
#     is found. The cut is to remove the initial colon from each match.
grep -o ':[^),]\+' tree.txt | cut -c 2-
```

26 Otherwise known as the Pathologically Eclectic Rubbish Lister