Basic Unix

2.1 What is Unix?

Unix is an *operating system*, by which we broadly mean the suite of software that make the computer work. Specifically, the operating system provides *services* for the program they run. Unix (and its cousin Linux) is used by the terminals and multi-user servers in most physics/astronomy departments. For people with Windows on their laptop, we will use Cygwin to provide Unix functionality.

The Unix operating system is made up of three parts: the kernel; the shell; and the programs.

2.1.1 The Kernel

The *kernel* of Unix is the hub of the operating system: it allocates time and memory to programs and handles the file store and communications in response to system calls (requests from programs for the kernel to do things on behalf of the program, e.g. print characters on the screen, write data to disk, etc).

2.1.2 The Shell

The *shell* acts as an interface between the user and the kernel. When a user logs in, the login program checks the username and password, and then starts another program called the *shell*. The shell is a command line interpreter (CLI). It interprets the commands the user types in and arranges for them to be carried out. The commands themselves are usually programs: when they terminate, the shell gives the user another prompt to indicate it is ready to accept more commands.

There are many different shells, one of the earliest being the Bourne shell (**`/bin/sh`**) which is pretty much available on all Unix distributions. We will be showing you Bash (Bourne Again SHell[1]) (**`/bin/bash`**). Bash is an sh-compatible shell that incorporates useful features from other more modern shells.

2.1.3 Files and Processes

Everything in Unix is either a file or a process.

A process is an executing program identified by a unique PID (process identifier).

A file is a collection of data. Files can be created by users using text editors, running programs, or copying existing files from elsewhere on the system, or across the network etc. Notice that the idea of a file does not talk specifically about being on disk, it could be in memory, on disk, being transmitted across the network or being generated on the fly by another program or the operating system.

2.2 Working with Files and Directories

Just like in the Windows environment, files are stored in a hierarchical *directory* structure. The location of a particular file (or *subdirectory*) is described using a *path*. Under Windows, the hierarchy starts from a drive letter (for example `C:\`) but in Unix the hierarchy starts from a single slash (`/`) called the *root* directory. An *absolute path* is one that starts from the top of the hierarchy (i.e. begins with a slash). Alternatively, a *relative path* starts from the current directory.

An absolute path points to the same place regardless of what the current directory is. A relative path is relative to your current location.

Don't directories break the *everything is a file or process* rule? Directories are in fact special files which link

[1] Welcome to bad Unix naming humour!

filenames and other information about a file, with the file data itself. It is possible (although not advisable) to open a directory just like any other file on Unix systems.

2.2.1 `pwd`

Each process (a running program) has a *working directory* (also called the *current working directory*) from which all relative paths are interpreted.

The **`pwd`** command prints the current path and thus enables you to work out where you are in the filesystem. For example, if you run **`pwd`** when you first log in, you will find you are in your *home directory*:

```
% pwd
/Users/tara
%
```

Of course your home directory path will be different, other examples are:

```
/cygdrive/c/Users and Documents/tara
/home-astrop/tara
/usr/extern/tara
```

2.2.2 `ls`

When you first login, your current working directory is your home directory. Your home directory has the same name as your user name, for example, **`tara`**, and it is where your personal files and subdirectories are saved.

To find out what is in your home directory, type:

```
% ls
README
%
```

The **`ls`** command lists the contents of your current working directory.

If the home directory is empty no files will be shown (and you will just get a prompt back on the next line). Usually there are some standard files inserted by the System Adminisrator when your account was created.

`ls` does not, in fact, cause *all* the files in your home directory to be listed, but only those ones whose name does not begin with a dot (`.`) Files beginning with a dot (`.`) are known as *hidden files* and usually contain important program configuration information. They are hidden because you should not change them unless you are very familiar with Unix!

To list all files in your home directory including those whose names begin with a dot, type:

```
% ls -a
.
..
.login
README
%
```

`ls` is an example of a command which can take options: **`-a`** is an example of an option. The options change the behaviour of the command. There are online manual pages that tell you which options a particular command can take, and how each option modifies the behaviour of the command. (See later in these lab notes)

2.2.3 `mkdir`

We will now make a subdirectory in your home directory to hold the files you will be creating and using in this lab. To make a subdirectory called **`unixstuff`** in your current working directory type:

```
% mkdir unixstuff
%
```

To see the directory you have just created, type:

```
% ls
README unixstuff
```

```
%
```

2.2.4 `cd`

The *change directory* command **`cd <directory>`**, changes the current working directory to **`<directory>`**. The current working directory may be thought of as the directory you are "in", i.e. your current position in the file-system tree.

To change to the directory you have just made, type:

```
% cd unixstuff
%
```

Type **`ls`** to see the contents (which should be empty) and **`pwd`** to see the absolute path.

```
% ls
% pwd
/Users/tara/unixstuff
%
```

2.2.5 The Special directories `.` and `..`

Using **`ls -a`** will always display the two directories `.` and `..` They represent the *current directory* and its *parent directory* respectively. Clearly, these special directories are relative paths because they depend (by definition) on the current working directory.

These can be used with **`cd`** to navigate the directories:

```
% pwd
/Users/tara/unixstuff
% cd .
% pwd
/Users/tara/unixstuff
% cd ..
% pwd
/Users/tara
%
```

As you can see, **`cd ..`** takes you up to the parent directory.

When we type **`cd .`** we do not move as we are asking to move to the directory we are already in. It might not seem very useful yet, but there are lots of places later on where we will need to refer to the current directory.

2.2.6 More about directories and pathnames

```
% pwd
/Users/tara
% mkdir foo/bar
mkdir: cannot create directory 'foo/bar': No such file or directory
% mkdir unixstuff/backups
% ls unixstuff
backups
% mkdir -p foo/bar
% ls foo
bar
%
```

We cannot create nested directories (directories within other directories) without first creating the necessary parent directories. The **`-p`** option can be used to force the building of parent directories first. Now we have two directories within our home directory (**`unixstuff`** and **`foo`**). The **`unixstuff`** directory has a **`backups`** subdirectory, and **`foo`** has a **`bar`** subdirectory.

```
% ls backups
ls: backups: no such file or directory
%
```

We can't see inside the **backups** directory directly from our home directory, because it is only accessible from inside the **unixstuff** directory. However, we explicitly put parent directories using slashes (a relative path) like:

```
% cd unixstuff/backups
% pwd
/Users/tara/unixstuff/backups
% ls ..
backups
% ls ../..
README     foo        unixstuff
% ls ../../foo
bar
%
```

Notice we can also use the `..` directory with the slashes to get access to the grandparent directory, and then other directories from there.

~ (the *tilde* or *twiddle* character[2]) is another special directory that represents your home directory. To get to another user's home directory (if they have made it accessible to others), you would use **~<login>**.

```
% ls ~
unixstuff
% cd ~/unixstuff
% pwd
/Users/tara/unixstuff
%
```

Notice that we can combine ~ with other directories using the slash.

2.3 Copy, Moving and Deleting Files and Directories

2.3.1 cp

This is used to copy a file or directory to a new location:

```
% cd ~
% touch afile
% ls
README     afile      unixstuff
% cp afile unixstuff
% ls unixstuff
afile    backups
%
```

touch is used to create an empty file (containing no characters), but you could create a file like this just as easily with a text editor (e.g. pico, vi, emacs, nedit).

Here we have copied the file **afile** into the **unixstuff** directory. So now a copy of **afile** exists in your home directory and also in the **unixstuff** directory.

The file can also be copied to different filenames. Here we make a copy of **afile** called **anotherfile**:

```
% cp afile anotherfile
% ls
README       afile        anotherfile  foo          unixstuff
%
```

As well as specifying a named directory, you can also copy files to the current directory using dot:

```
% cd unixstuff
% cp ../anotherfile .
% ls
afile        anotherfile  backups
%
```

[2]Tilde can be a bit hard to find on some keyboards – typically it is on the top left above the tab key.

If we want to copy directories and all of their subdirectories *recursively*, the **-r** flag should be used to copy recursively:

```
% cd
% cp -r unixstuff moreunixstuff
% ls moreunixstuff
afile       anotherfile  backups
%
```

Notice that the **cd** command without a directory argument returns you to your home directory (just like **cd ~**). We have then created a complete copy of **unixstuff** and its subdirectory **backups**.

2.3.2 mv

This is used to move (or rename) files and directories. Note, unlike MSDOS and the Windows command shell (**cmd.exe**) there is no separate rename command:

```
% ls
README         anotherfile    moreunixstuff
afile          foo            unixstuff
% mv afile foo
% mv anotherfile yetanotherfile
% mv moreunixstuff lessunixstuff
% ls
README          foo             lessunixstuff   unixstuff       yetanotherfile
% ls foo
afile  bar
%
```

Here we have renamed the file **anotherfile** to **yetanotherfile** and **moreunixstuff** to **lessunixstuff**. We have also moved **afile** from our current directory into the **foo** directory.

While we are here, notice that the **ls** command tries to format the filenames in the most readable manner. This will change depending on the size of the shell window, and the length and number of filenames in the directory being listed.

2.3.3 rm

This is used to delete files.

```
% ls
README          foo             lessunixstuff   unixstuff       yetanotherfile
% rm yetanotherfile
% ls
README         foo            lessunixstuff  unixstuff
%
```

Here we have deleted the file **yetanotherfile**. NB: unlike Windows there is no trash can under Unix, once a file is deleted it is gone forever. The only way of getting back a file is to have a backup copy of it somewhere else on the system. Most (all?) departments will backup user accounts every evening, so it is possible to retrieve your previous days work by speaking to your System Administrator.

```
% rm lessunixstuff
rm: lessunixstuff is a directory
% rm -r lessunixstuff
% ls
README     foo        unixstuff
%
```

We cannot use this to delete directories unless we use the **-r** flag. Be careful using this as it will delete everything in that directory, i.e. all of the subdirectories and the files they contain. In fact, be **very careful with `rm -r`**!

2.3.4 rmdir

This is used to delete directories. It will only work when they are empty.

```
% rmdir foo
rmdir: directory "foo": Directory not empty
% rm foo/afile
% rmdir foo/bar
% ls foo
% rmdir foo
% ls
README     unixstuff
%
```

This series of commands deleted the file **afile** and the directory **bar** from within the **foo** directory, so we could then delete the **foo** directory using **rmdir**. You can see why using **rm -r** is tempting, but be careful!

2.4 Displaying Files

For these exercises you will need to download the word list from the BasicUnix wiki page. This file contains a list of standard English words. It is often found in **/usr/dict/words** or **/usr/share/dict/words** on modern Linux systems. Once you have downloaded the file, put it in your **unixstuff** directory.

2.4.1 clear

This command will clear the screen and is sometimes useful when viewing files or listing lots of directories. It doesn't clear the entire history, only the current screenful. Commands prior to that will still be visible if you scroll the terminal window up.

2.4.2 cat

This will print the given file(s) to the standard output (screen).

```
% cd ~/unixstuff
% cat words.txt
10th
1st
...
Zurich
zygote
%
```

You should have seen a lot of words fly by on the screen with this command — all the words contained in this word list. If multiple files are given to **cat** they will be printed one after the other in order.

2.4.3 less

less is a more advanced text viewer. It allows the navigation a file or data read from standard input (normally the keyboard). It is a more advanced version of another text viewer, called **more**[3], which didn't have the capability of scrolling backwards through a file.

Here are some of the supported commands:

arrow keys or **Enter**	up or down one line at a time
PageUp/Down	up or down one page at a time
b or **SpaceBar**	
g	beginning of the file
G	end of the file
/<pattern>	find a given pattern <**pattern**>
n	find the next matching instance of the pattern
q	quit out of less
h	get help on the available commands

less can be used as a direct substitute for **cat**:

```
% less words.txt
10th
```

[3] see what we mean about Unix humour!

```
1st
2nd
...[to end of page]
%
```

But it can also be used in the following way:

```
% cat words.txt | less
10th
1st
2nd
...[to end of page]
%
```

Both these have the same effect. The second idiom can be used to view the output of any program that produces multiple screenfuls of output. Here the vertical bar (|) is a *pipe*, indicating that the output from the first program should not be output to the screen, but rather should be used as input to the second program (instead of the keyboard). This means **cat** is providing input to **less** rather than **less** directly reading it from the file. We will see why this is useful shortly.

2.4.4 **head** and **tail**

Sometimes we want to get access to the first few lines of a file or the last few lines of a file. The **head** and **tail** commands can be used to do that:

```
% head -1 words.txt
10th
% head -2 words.txt
10th
1st
% tail -4 words.txt
Zoroastrian
zounds
zucchini
Zurich
%
```

The number after the **-** specifies how many lines to print. If this option is not specified, the default is 10 lines.

tail also allows you to *ignore* a given number of the first lines:

```
% tail +4 words.txt
3rd
4th
5th
...
%
```

Here, the **+4** means start printing from the 4th line, i.e. ignore the first three lines of the file. **head** and **tail** are often useful for checking the format of a file because they print out enough to see what is going on, but not so much that you lose everything else you had on the screen.

2.4.5 **wc**

wc is used to count characters, words and lines in a file:

```
% wc words.txt
   24001   24001  196537 words.txt
% wc -l words.txt
   24001 words.txt
% wc -w words.txt
  196537 words.txt
%
```

Here we print the listing of lines, words and bytes, then the lines only and then the bytes (or characters) only for **`words.txt`**. Because this file only contains one word per line the number of each is the same for this file.

2.5 Investigating files

You should do these exercises in your **`unixstuff`** directory:

```
% cd ~/unixstuff
```

2.5.1 `grep`

`grep` prints lines of the input matching a given expression:

```
% grep 'uu' words.txt
continuum
residuum
vacuum
% grep '^x' words.txt
x
x's
xenon
xenophobia
xerography
...
% grep -i '^x' words.txt
x
x's
Xavier
xenon
xenophobia
...
%
```

`grep` is case sensitive by default, so we have to use the **`-i`** flag to set case insensitivity. Other useful flags are **`-v`** (line that do not match) and **`-c`** (produce a count only). If multiple files are given then the results are given by file.

More complex expressions can be used for the pattern, including wildcard characters. These include **`*`** (0 or more of the preceding character), **`+`** (1 or more of the preceding character), **`?`** (0 or 1 of the preceding character), **`.`** (any character), **`[abc]`** (any 1 of the included characters), **`[^abc]`** (any 1 character, except those included). When used (or if a space is used), the expression must be enclosed in single quotes.

These complex patterns are called *regular expressions* and are a powerful way of describing strings. For example

```
% grep '^Z.us' words.txt
Zeus
%
```

This searches for any line with a string starting with 'Z' followed by any character, followed by a 'u' character, and finished with an 's'. Regular expressions are covered in more detail in the Advanced Unix notes.

2.5.2 Redirection

Most processes initiated by Unix commands write to the *standard output* (or stdout) i.e. they write to the terminal window, and many take their input from the *standard input* (or stdin), i.e. they read it from the keyboard. There is also the *standard error* (or stderr), where processes write their error messages, which by default, is also to the terminal window.

Here we see the other use of **`cat`**: taking input from the standard input (keyboard) and printing it to the standard output (the screen). You need to type in the bits in **`??normal text`** and then press **`<Ctrl-D>`** at the end:

```
% cat
foo
foo
bar
bar
```

```
<Ctrl-D>
%
```

Redirection can be used to change the standard input, output and error of a program to a file instead. Running **cat** again, and redirecting standard output to a file looks like this:

```
% cat > list1
apple
pear
<Ctrl-D>
% cat list1
apple
pear
%
```

Here we have redirected the standard output to the file **list1** using the **>** operator. This will create or overwrite the file specified after the redirection operator. We can use the **>>** operator to *append* the output. If the file does not exist it acts the same as the **>** operator.

```
% cat >> list1
banana
orange
<Ctrl-D>
% cat list1
apple
pear
banana
orange
% cat >> list2
foo
bar
<Ctrl-D>
% cat list2
foo
bar
% cat > list2
beef
lamb
<Ctrl-D>
% cat list2
beef
lamb
%
```

You can see in this example above that failing to use **>>** for the second lot of output for **list2** causes the original contents (**foo** and **bar**) to be deleted.

We can also use **cat** to join (or *concatenate* files:

```
% cat list1 list2 > biglist
% cat biglist
apple
pear
banana
orange
beef
lamb
%
```

2.5.3 `sort`

sort can be used to sort the contents of a file:

```
% sort biglist
apple
```

```
banana
beef
lamb
orange
pear
% sort < biglist > sortedlist
% cat sortedlist
apple
banana
beef
lamb
orange
pear
%
```

Sort by default takes input from the standard input and prints the sorted contents to standard output. However, if you specify a filename, here **biglist**, it will sort that file rather than standard input. Alternatively, we can use the **<** operator to redirect standard input to be the file **biglist**. Here we use both operators to take input from a file and send the output to a file. Note that sort can take a series of files as input and sort these.

2.5.4 pipes

Suppose we want to run some input through several programs, such as counting the results of **grep** using **wc**. If we use redirection, it requires the creation of temporary files.

```
% grep 'b' biglist > filteredlist
% cat filteredlist
banana
beef
lamb
% wc filteredlist
       3       3      17 filteredlist
%
```

This requires us to then cleanup the file when complete. Instead we can use the **|** to create a pipe between the two programs.

```
% grep 'b' biglist | wc
       3       3      17
%
```

This pipe makes the output of the first program the input of the second program. These can then be chained together in arbitrarily long sequences.

```
% grep 'b' biglist | wc | wc
       1       3      25
%
```

2.5.5 uniq

uniq merges together duplicate lines *that are adjacent to each other*. It is often used in conjunction with sort.

```
% cat > duplicates
this
test
is
is
is
a
test
test
% uniq duplicates
this
test
```

```
is
a
test
%
```

Notice that there are still two instances of **test** in the output. That's because the first **test** is separated from the other instances and so they won't be merged. If you want to remove *all* of the duplicates, sort the file first:

```
% sort duplicates | uniq
a
is
test
this
%
```

uniq can also count the number of times each line has occurred:

```
% sort duplicates | uniq -c
   1 a
   3 is
   3 test
   1 this
%
```

uniq also allows you to only print out the *unique* lines or only the *duplicate lines*:

```
% sort duplicates | uniq -u
a
this
% sort duplicates | uniq -d
is
test
%
```

2.6 File Access Rights

2.6.1 **ls**

Using the **-l** flag for **ls**, we can generate a *long listing* of a file or directory:

```
% ls -l
total 40
lrwxrwxrwx   1 tara  astrop        29 Mar  6 11:10 README -> /local/usr/lib/newuser/README
-rw-r--r--   1 tara  astrop        35 Mar  6 10:57 biglist
-rw-r--r--   1 tara  astrop        17 Mar  6 10:58 filteredlist
-rw-r--r--   1 tara  astrop        26 Mar  6 11:09 list1
-rw-r--r--   1 tara  astrop        10 Mar  6 11:10 list2
drwx------   3 tara  astrop      4096 Mar  6 01:50 unixstuff
%
```

Reading left to right, the properties listed are type and permissions, number of links, owner (username), owner (group), file size, date last modified and name. The type is the first character of the listing. These have the value **-** for files, **d** for directories and **l** for symbolic links. The next nine characters are the permissions. They are in three character blocks for *user*, *group* and *world* (or other) permissions. The each block is made up of three flags: **r**, **w**, and **x**, which have slightly different meanings for files and directories. If the permission is not granted a **-** appears.

2.6.2 Access Rights for Files

r indicates read permission (or otherwise), that is, the presence or absence of permission to read and copy the file;

w indicates write permission (or otherwise), that is, the permission (or otherwise) to change a file, including to delete it;

x indicates execution permission (or otherwise), that is, the permission to execute a file, where appropriate.

2.6.3 Access Rights for Directories

r allows users to list files in the directory;

w means that users may delete files from the directory or move files into it;

x means the right to access files in the directory. This implies that you may read files in the directory provided you have read permission on the individual files.

For the above, everyone can read all the files listed, but only tara can write (or delete) them. Everyone can see into the directory **unixstuff** and access files within it (to which they have permission), but, again, only tara can modify it.

2.6.4 chmod

The owner of a file can change the permissions on a file using **chmod**. This allows him to hide or reveal files to other users on the computer. **chmod** takes options as follows:

u user	**r** read	**+** add permission
g group	**w** write (and delete)	**−** remove permission
o other (world)	**x** execute (and access directory)	
a all		

which are used as follows:

```
% ls -l biglist
-rw-r-----   1 tara   astrop          35 Mar  6 10:57 biglist
% chmod g+rwx biglist
% ls -l biglist
-rw-rwx---   1 tara   astrop          35 Mar  6 10:57 biglist
% chmod go-rwx biglist
% ls -l biglist
-rw-------   1 tara   astrop          35 Mar  6 10:57 biglist
% chmod a+r,a-w biglist
% ls -l biglist
-r--r--r--   1 tara   astrop          35 Mar  6 10:57 biglist
% cat > biglist
-bash: biglist: Permission denied
%
```

2.7 Processes and Job Control

2.7.1 ps

Each process executed on the system is given a unique identifier, its PID. **ps** can be used to list your (and others') running processes. To see this we can use the **sleep** command, which will waits (or sleeps) for the given number of seconds. To be able to see this working, we also use the background modifier **&** (the ampersand). Any process executed with this at the end of its command will be run in the *background*. This allows the user to continue to type commands into the shell while the program is running.

When a process is run in the background, the shell will tell you the *jobspec* (in brackets) and the PID of the process so it can later be brought to the foreground using the **fg** command (using the last backgrounded process or taking the jobspec as its argument).

```
% sleep 10 &
[1] 20746
% ps
  PID TTY          TIME CMD
18116 pts/4    00:00:00 bash
```

```
20746 pts/4    00:00:00 sleep
20747 pts/4    00:00:00 ps
%
```

Notice, there are three of your processes running (from this shell window). One of them is the shell itself (**bash**), another is the **ps** itself and the third is the **sleep** command.

2.7.2 **bg** and **fg**

A currently running process can be *stopped* without destroying it by using pressing **<Ctrl-Z>**. This sends the process to sleep and prints the jobspec to screen. **bg** can then be used to start the process again in the background. It will then run as if it had been run from the command line using the **&** command. Using **<Ctrl-Z>** is a good trick when you forgot to put an ampersand on the end of your command line. **fg** can be used to run the process in the foreground.

Here is an example of these tricks in action:

```
% sleep 30
<Ctrl-Z>
[1]+  Stopped                 sleep 30
% bg
[1]+ sleep 30 &
% fg 1
sleep 30

%
```

If you actually want to kill a program on the other hand, press **<Ctrl-C>**. Many people confuse the two and run **<Ctrl-Z>** and so rather than killing their programs, they just put them to sleep, so they end up with lots of sleeping processes hanging around the system.

2.7.3 **jobs**

This is used to list the current sleeping and backgrounded jobs:

```
% sleep 30 &
[1] 21119
% sleep 30 &
[2] 21120
% sleep 30
<Ctrl-Z>
[3]+  Stopped                 sleep 30
% jobs
[1]   Running                 sleep 30 &
[2]-  Running                 sleep 30 &
[3]+  Stopped                 sleep 30
%
```

2.7.4 **kill**

This is used to send *signals* to programs to stop, halt or terminate them. It takes a PID or jobspec (preceded by a **%**). **<Ctrl-C>** is the equivalent to killing on a running process.

```
% sleep 30
<Ctrl-C>
% sleep 30 &
[1] 21158
% jobs
[1]+  Running                 sleep 30 &
% kill %1
% jobs
[1]+  Terminated              sleep 30
% sleep 30
[1] 21165
% kill 21165
```

```
% jobs
[1]+  Terminated              sleep 30
%
```

Sometimes a specific signal sent by kill must be specified. The most common of these is **-9**, which will cause a program to exit immediately. The normal kill signal gives the program the opportunity to cleanup after itself. This does not. **-9** is most often used when a program refuses to stop.

It is also possible to kill all of your processes, in all of your shell windows at once using **kill -9 -1**. This kills everything at once, including your shell processes, so it logs you out as well.

2.8 Working with Columns of Data

There are several Unix commands that allow you to process text files that consist of columns of data. Typically these columns are separated by *whitespace*, that is, either a space or tab character[4].

2.8.1 paste

The **paste** command takes two or more files and pastes the contents of these files together column-wise, with each column separated by a *delimiter* character. This delimiter is tab by default:

```
% cat > names
kilo
mega
giga
tera
peta
exa
% cat > powers10
3
6
9
12
15
18
% paste names powers10
kilo    3
mega    6
giga    9
tera    12
peta    15
exa     18
%
```

Note, the **kilo** and **3** etc are separated by tabs. We can change that by giving **paste** the **-d<delim>** delimiter option (or **-d <delim>**), where **<delim>** is one or more characters to use as the delimiter:

```
% paste -d ' >' names names powers10
kilo kilo>3
mega mega>6
giga giga>9
tera tera>12
peta peta>15
exa exa>18
%
```

This example demonstrates:

- you can repeat a filename
- specifying multiple delimiters
- using single quotes to stop the shell interpreting a string

[4] newlines are also counted as whitespace but not in this context because newline characters separate rows.

Multiple delimiters used in the order they are given to separate the columns (if there are more columns than delimiters they are used in cyclical order). The generalisation of this is that if you only specify one delimiter character, then it will be used to separate all columns!

Quoting arguments to avoid shell interpretation is required frequently. In our previous example, we wanted to use space as the first separator and **>** as the second. If we did not quote the argument, then the shell would ignore the whitespace and interpret the **>** as indicating redirection of standard output. If you are not sure whether the shell might be accidentally interpreting an argument, single quotes can be used to protect it from the shell.

paste can also take redirected standard input where the placeholder **-** means read the standard input:

```
% paste -d '_' - powers10 < names
kilo_3
mega_6
giga_9
tera_12
peta_15
exa_18
%
```

Here, **names** has been *redirected* as standard input in place of the keyboard input. **paste** then reads from standard input when the **-** placeholder is used.

The final use of **paste** is to turn a flat file into multiple columns (something we only discovered reading the manpages recently, but is a very useful feature!). This involves repeated use of the **-** placeholder filename and is a very cunning trick:

```
% paste - - - < names
kilo    mega    giga
tera    peta    exa
%
```

2.8.2 cut

Not surprisingly, there is a command to do the reverse of pasting, and that command is called **cut**. This allows you to select specific columns from a given file or standard input.

Firstly, have a look at the **columns.txt** file using **less** or **cat**. Notice that the entries are separated in columns:

```
% cat columns.txt
15:48:55.81 -40:32:17.5 J1600M40 2 1393.1 333.9 3
15:49:10.47 -40:14:18.7 J1600M40 2 1380.9 397.2 3
15:49:12.16 -40:02:13.1 J1600M40 2 1381.1 439.6 3
...
%
```

The columns are Right Ascension, Declination, Mosaic Name and various other source information.

So, to get a list of just the declinations we can go:

```
% cut -d ' ' -f 2 columns.txt
-40:32:17.5
-40:14:18.7
-40:02:13.1
...
%
```

The **-d** option allows you to specify the column deliminator. Try using **-d ':'** instead.

Cut allows you to specify the fields in a fairly flexible manner with the **-f** option:

-f 1,3,5	selects columns 1, 3 and 5 (surprising huh!)
-f 4-6	selects columns 4, 5 and 6
-f -3	selects columns 1, 2 and 3
-f 4-	selects columns 4 and any more columns to the right

2.8.3 **awk** and **gawk**

AWK is a programming language for processing text files containing columns of data. It was developed by Aho, Weinberger and Kernighan more than 20 years ago. AWK partly inspired the development of *Perl*, which is a very popular modern scripting language.

awk (or the Gnu version **gawk**) is the Unix command for interpreting AWK programs on text input. AWK allows you to write quite complicated expressions for calculations involving columns of data. We are only going to cover a few simple but powerful ones here.

The first is selecting rows based on properties of particular columns:

```
% paste names powers10 > prefixes
% awk '$2 > 6' prefixes
giga    9
tera    12
peta    15
exa     18
% awk '$1 == "giga"' prefixes
giga    9
```

The first example only prints the row out if column 2 (represented by **$2**) has a numerical value larger than 6. The second example only prints out rows that have the word **giga** in their first column. There are lots of powerful matching techniques which we will discuss in later labs and lectures.

```
% awk '{ print $2, $1; }' prefixes
3 kilo
6 mega
9 giga
12 tera
15 peta
18 exa
% awk '{ print $1, NR*10; }' prefixes
kilo 10
mega 20
giga 30
tera 40
peta 50
exa 60
%
```

It is also possible to produce different output, rather than simply selecting interesting rows. In the first example we swap the two columns around by printing **$2** before **$1**. Notice that the **print** statement needs to be within *braces* (or squiggly brackets) and followed by a semi-colon.

The second example demonstrates that we can also calculate new values on the fly and print these out as well. Here we have calculate the powers of 2 for SI units rather than the powers of 10. We use a special variable **NR** which holds the line number we are up to in the input (the *number of records*). Another useful special variable is **NF** which is the number of fields in the current row.

In both of these examples, the output columns were separated by spaces rather than tabs, because this is the default output field separator (**OFS**) which we can change:

```
% awk 'BEGIN { OFS="-->"; } { print $1, $2; }' prefixes
kilo-->3
mega-->6
giga-->9
tera-->12
peta-->15
exa-->18
```

Here we use the special **BEGIN** rule to set the value of the output field separator *before* AWK starts processing any lines of input. We can also use **BEGIN** to set the input field separator (**FS**) before processing begins.

There is also an equivalent **END** rule which is run after the input has been processed:

```
% awk '{ sum = sum + $2; } END { print sum; }' prefixes
63
```

This program stores the sum of all of the values in column 2 in a variable called **sum**, and then after it has processed all of the rows, prints out the value of **sum**.

AWK has all of the power of a programming language rather than a simple command line tool. You can find out more about what it can do from the manpage **awk(1)**, but it is a rather terse description.

2.8.4 More on **sort**

sort can also work in a column aware manner. By default it sorts on the first column of input in ascending alphabetical order. However, all of these can be changed with command line arguments.

Firstly, you can reverse the sorting order with the **-r** option:

```
% sort -r prefixes
tera    12
peta    15
mega    6
kilo    3
giga    9
exa     18
%
```

Secondly, you can sort on a different column using the **-k** option to indicate which column to sort on first:

```
% sort -k 2 prefixes
tera    12
peta    15
exa     18
kilo    3
mega    6
giga    9
%
```

Hmm, what went wrong here? The rows don't seem to be sorted in any sensible order at all? Well in fact they are sorted, but the sorted order is still *lexicographic*, that is, phone book ordering in alphanumeric ordering. In lexicographic ordering, the length of the word is not important, only the relative order of the first characters in the word. For example, **aardvark** comes before **ant** because 'a' comes before 'n'.

Therefore, when we treat numbers in a lexicographic way, we end up putting all of the numbers starting with the digit '1' first. If we want to treat the column as numeric, we can use the **-n** flag (in first column sorting) or add **n** to the column specifier with the **-k** option.

```
% sort -k 2n prefixes
kilo    3
mega    6
giga    9
tera    12
peta    15
exa     18
% sort -k 2rn prefixes
exa     18
peta    15
tera    12
giga    9
mega    6
kilo    3
%
```

The Gnu version also provides a **-g** option (or **g** for use with the **-k** option) which treats the sorting column as a floating point number with the exponent notation.

2.9 Other Useful Commands

2.9.1 `exit`

This ends the current interactive shell session and so causes you to be logged out. If you still have processes running, it will hang there until the processes have finished.

2.9.2 `gzip` and `gunzip`

`gzip` and **`gunzip`** are the Gnu zipping programs for compressing and decompressing files. Notice that they remove the original and compressed version of the files after they have run:

```
% gzip biglist
% ls biglist*
biglist.gz
% gunzip biglist.gz
% ls biglist*
biglist
%
```

2.9.3 `tar`

`gzip` is designed to compress a single file, so how do you compress and transfer multiple files around the system?

The **`tar`** *archiving* program collates many files and directories into a single file. The **`c`** flag specifies create an archive file, the **`x`** flag specifies extract an archive and the **`t`** flag specifies list the contents of an archive without extracting files. The **`v`** flag specifies verbose output and the **`f`** flag indicates that the following argument is the output file.

The **`z`** flag can be added to use gzip compression and decompression, which is the same as running **`gzip`**/**`gunzip`** after/before running the **`tar`** command.

```
% tar cvf lists.tar list1 list2
list1
list2
% tar xvf lists.tar
list1
list2
% tar cvzf lists.tar.gz list1 list2
list1
list2
% ls -l lists.tar*
-rw-rw-rw-   1 info1903  cs1        10240 Mar  6 14:56 lists.tar
-rw-rw-rw-   1 info1903  cs1          178 Mar  6 14:57 lists.tar.gz
% tar tzvf
```

2.9.4 Tab completion

The tab can be used to auto-complete shell strings on the command line. In its simplest form it allows for completion of filenames. Later versions of Bash also allow completion of arguments and restrict irrelevant file types. Try typing a single character and then pressing **`<tab>`**. If there is only one possible completion, the shell will automatically complete it for you. If there is more than one completion, you will hear a beep instead. Press **`<tab>`** again to see the list of possible completions.

2.9.5 Wildcards

Wildcards can be used on the command line to identify files. **`*`** will match zero or more of any character. **`?`** will find exactly one of any character. Square brackets can be used to create sets of characters to match from. **`[abc]`** will match either the 'a', 'b' or 'c' in a filename.

```
% ls list*
list1
list2
% ls *list
biglist
```

```
filteredlist
% ls *list*
biglist
filteredlist
list1
list2
% ls ?list
biglist
%
```

2.9.6 Shell Variables

Variables are a way of passing information from the shell to programs when you run them. Programs look "in the environment" for particular variables and if they are found will use the values stored in them. Some are set by the system, others by you, yet others by the shell, or any program that loads another program.

Standard Unix variables are split into two categories, environment variables and shell variables. In broad terms, shell variables apply only to the current instance of the shell and are used to set short-term working conditions; environment variables have a farther reaching significance, and those set at login are valid for the duration of the session. By convention, environment variables have UPPER CASE and shell variables have lower case names.

2.9.7 `echo`

The **`echo`** command writes its arguments to standard output and can therefore be used to display the contents of variables. It will repeat to standard output whatever followed it on the command line. Environment and shell variables can be accessed by using the **`$`** symbol before their name.

```
% echo $USER
tara
%
```

Some common environment variables are:

OSTYPE	the operating system type
USER	your login name
HOME	the path name of your home directory
HOST	the name of the computer you are using
ARCH	the architecture of the computers processor
DISPLAY	the name of the computer screen to display X windows
PRINTER	the default printer to send print jobs
PATH	the directories the shell should search to find a command

2.9.8 `printenv`

`printenv` can be used to display all the currently set environment variables.

2.10 Acknowledgements

These notes are based on those used for our course INFO1903 `http://www.it.usyd.edu.au/~info1903`. Thanks to James Gorman for compiling the first version of the notes. They are largely derived from the notes written by M Stonebank from the University of Surrey. `http://www.ee.surrey.ac.uk/Teaching/Unix/` Material was also taken from the University of Utah's Unix command summary at `http://www.math.utah.edu/computing/unix/unix-commands.html`