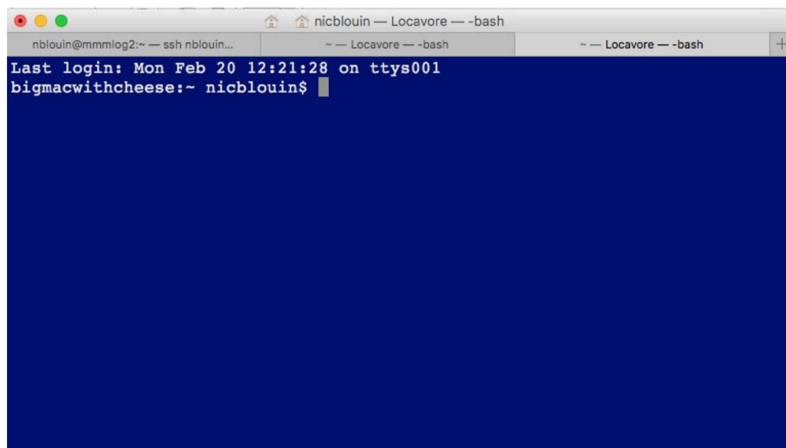
Some of the examples and tasks in this Manual have been heavily influenced by:
Unix and Perl for Biologists: http://korflab.ucdavis.edu/Unix_and_Perl/
Basic UNIX for Biologists:
https://www.bioinformatics.purdue.edu/discoverypark/cyber/bioinformatics/services/workshops.php

The Terminal

This is the common name for the application that gives you text-based access to the operating system of the computer. Basically, it allows you to type input into the computer so that you can receive output from the programs you call. On Unix based machines there is always a 'terminal' program.



1) Some brief tips before we move further:
    a. You will need to frequently resize this window as we do the hands-on activities. This is done as with any other window.
    b. You can change the size of the text by holding 'command' + shift and hitting '+' or 'command' plus the '-' key.
    c. You can have multiple terminal windows open at the same time.
    d. You can also have multiple tabs open in each terminal window.
    e. Everything you type in the terminal is case sensitive. 'grep –F' is not the same as 'grep -f'. This will be a very important thing to remember when using the terminal and creating files or folders.

Connecting to the server

Often with bioinformatics analysis, you will be performing tasks on a remote network server, one much more powerful than your workstation. Let's use the terminal program you just learned about
to connect with this server.
We will use Secure SHell (SSH) protocol to talk to the remote server. In the following example, replace *username* with the user name provided to you.

Two-Factor Authentication
To improve computer security, University of Wyoming has started requiring two factor authentication to grant access to network servers. Two factor implies password plus a second type of authentication. You all have downloaded the DUO app for your phones.

2) Type **ssh** *username***@mtmoran.uwyo.edu**
This will bring up the password prompt.
username1@mtmoran.uwyo.edu's password:

3) As you type the password, you will not see the cursor move. This is normal. During the two-factor authentication, we will use the password in combination with a numeric 'push' in DUO
Enter the two as follows, don't forget the comma:
PASSWORD,Duo push If the authentication succeeds, the server will log you in and present a command prompt of style: user@server:_$.
If you see this, you have successfully logged on to the server.

4) You will see that your terminal line will look like this example when you are working on an HPC.

```
[nblouin@mmmlog2 ~]$
```

5) The 'nblouin@mmmlog2 ~$' text is the Linux command prompt. It contains the username (nblouin), the name of the machine that the user is working on (mmmlog2) and the name of the current directory (~). Note the command prompt may look different on different machines. In this example the $ represents the end of the prompt.

6) What is your username?

7) What is the name of your machine?

8) What is the current directory?

This application is going to be your friend when working with large data sets. Get used to making new windows (command+ shift + n), new tabs (command + shift + t), and resizing this window.

Unix Tutorial

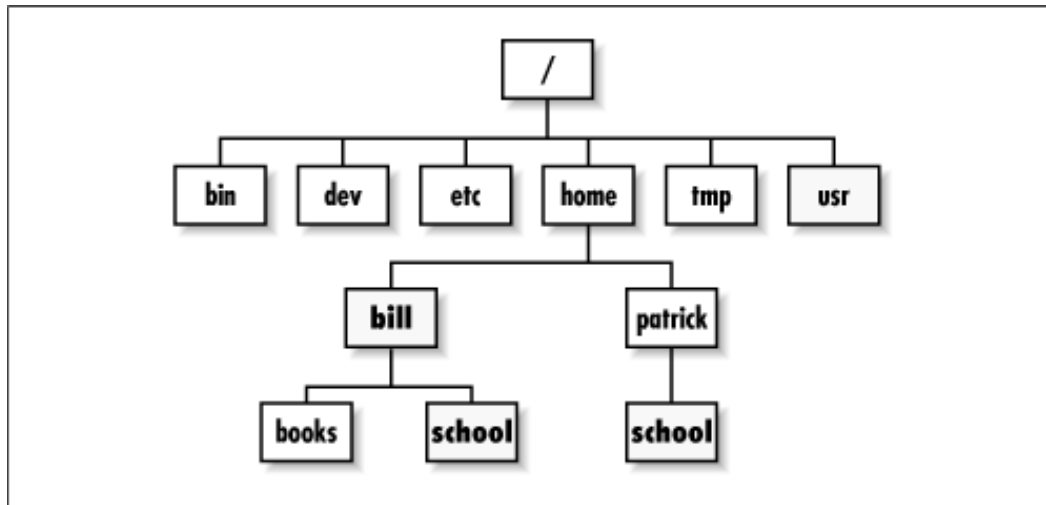
Where am I?

As you will learn everything in Unix is relative to where you are in the file system. Therefore, knowing where you are before launching a command is valuable information. Luckily, there are built in commands for this type of information. Understanding the location of files will be a key part of success.

1) To find out where you are in the file system type **`pwd`** in the terminal window.
2) This will return your current working directory (**p**rint **w**orking **d**irectory)
3) As you can see above the working directory is: **`/home/`***`username`*
4) The present directory is represented as `.` (dot) and the parent directory is represented as `..` (dot dot)

File Structure

Unix files are arranged in a hierarchical structure, or a directory tree. From the root directory (/) there are many subdirectories. Each subdirectory can contain files or other subdirectories, etc, etc. Whenever you're using the terminal you will always be 'in' a directory. The default behavior of opening a terminal window, or logging into a remote computer, will place you in your 'home' directory. This is true when we logged into MtMoran. The home directory contains files and directories that only you can modify, we will get to permissions later.



To see what files, or directories, you have in your home directory we will use the **`ls`** command.

1) Type **`ls`** and hit enter
2) You should see the list of files and directories in your current folder.
3) After the **`ls`** command finishes it produces a new command prompt that is ready for your next command.
4) The **`ls`** command can be used to list the contents of any directory not necessarily just the one you are currently in.
5) Type **`ls Unix`**

Changing Directories

To move between directories (folders) we use the **`cd`** (change directory) command. We are currently in our home directory. Lets move to
/home/*username***/Unix**

The **`cd`** command uses the format:

```
$ cd DIRECTORY
```

1) Type **cd /home/***username***/Unix**
2) Type **ls**
3) Type **pwd**
4) You can see that using the **cd** command moved us to a different directory.
5) By typing **ls** we can see that there is different stuff in this directory.
6) Finally, using **pwd** shows us what directory we are not located in.
7) We could have done the previous example in separate steps
8) Type **cd /home**
9) Type **cd** *username*
10) Type **cd Unix**
11) Note that we needed to type **/home** but not **/***username* When using a **/Directory** you are specifying a directory that is below the root directory. Without the leading **/** the system will look below the current directory.
12) Type **cd home**
13) Type **cd /home**
14) What happened with the first command?

15) You will frequently need to move up a level to a parent directory. Remember that two dots **..** are used to represent the parent directory. Every directory has a parent except the root level.
16) Type **cd ..**
17) Type **pwd**
18) You can move multiple levels at the same time
19) Type **cd /home/***username***/Unix**
20) Type **cd ../..**
21) Type **pwd**

22) When using **cd** everything is relative to your current location. However you can always use the *absolute* location to change directories. Lets move into the **Code** directory and look at two ways to switch to the **Data** directory.
23) Type **cd /home/***username***/Unix/Code**
24) Option 1: Type **cd ../Data**
25) Type **pwd**
26) Type **cd /home/***username***/Unix/Code**

27) Option 2: Type **cd /home/***username***/Unix/Data**
28) Type **pwd**

29) As you can see both options get us to the same place but Option 1 will only work from within a directory below **Data**. Option 2 will work from any location on the machine.

Task:

Remember the command prompt shows you the current directory you're in, and when you're in your home directory it shows a tilde (**~**). This is Unix's short-hand method of specifying a home directory.

Type the following commands and record the results. Follow each command with **pwd** and note how your location changes.

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

What were the locations after each command?

UNIX TIP: Tab complete is your friend. This is the best tip to learn early on as it will save you keystrokes and time. Type enough letters to uniquely identify the name of a file, directory, or program and press tab, Unix will autocomplete the word. E.g. type **tou** and press tab, Unix will autocomplete the word **touch** (we will learn about this soon). Tab completion occurred because there is no other command that starts with **tou.** If pressing tab doesn't work you have not typed enough unique characters, pressing tab twice will show you a list of everything that starts with the characters you've typed thus far.

Change to your home directory, then use **cd** to move to **/home/***username***/Unix/Code/** directory. Use tab complete to do this.

How many keystrokes did it take?

How many does it take to do the whole thing?

In my case it takes 13 keystrokes versus the 27 it would take to type the whole string.

```
$ cd /home/nblouin/Unix/Code/
```

Making Directories

Creating directories in Unix is done with the **mkdir** (make directory) command.

$ mkdir *DirectoryName*

Unlike making directories on your desktop using spaces is not advised in the Unix file system. This is why you see the use of _ in place of spaces. You can *escape* a space in Unix but it creates unnecessary typing and can create issues executing certain programs. Generally, using spaces in file and directory names is something to avoid.

1) Type **cd /home/***username***/Unix/**
2) Type **mkdir Work**
3) Type **ls**
4) Type **mkdir Temp1**
5) Type **cd Temp1**
6) Type **mkdir Temp2**
7) Type **cd Temp2**
8) Type **pwd**

9) In the previous example we created two temp directories but it took two steps. We could have done this in one step with by adding an *option/flag* to the **mkdir** command.
10) Type **cd ..**
11) Type **mkdir -p Temp1.1/Temp2.1**

Task

Practice creating at least 4 directories below ("within") the **Work/** directory.

Move between them using **cd** by using the *absolute* and *relative* paths. Remember tab completion is your friend!

UNIX TIP: You may remember earlier we used a flag with the **mkdir** command that created two directories at once. How did we find out about the **-p** flag? Google? Well, you could do that, but Unix has built in manuals for each command that give you all of the details. Simply type **man COMMAND**, *e.g.* $ **man cd** $ **man ls** or $ **man mkdir**. Even $ **man man**

To navigate in a **`man`** page use the arrow keys to move down one line at a time, space bar will page down. To exit a **`man`** type **`q`**.

Making and Editing Files

In this section you will learn the basics of making files and putting things into those files. There are a variety of ways we can accomplish this as Unix has built in multiple editors for these tasks. We will review a few here.

$ touch FILENAME
This will create a new, empty file.

$ nano FILENAME
This is a built in text editor that will allow us to put information into a file.

1) Create two files in **Unix/**
   a. $ **touch earth.txt**
   b. $ **touch heaven.txt**
2) Type **ls**
3) We have now created two empty files called '**earth.txt**' and '**heaven.txt**'
4) Type **cd Work**
5) Type **touch basic_info.txt**
6) Type **nano basic_info.txt**
7) We are now using an internal text editor that we can use to alter the contents of this file.
8) Add your name, email address, and favorite food to this file on separate lines.

9) Press control + x to exit and then y to save the file
10) We can also create and edit files with **nano**
    a. $ **nano onestep.txt**
11) Add a line of text and save the file.

Task

In the **Unix/Work/** directory create a new directory called **Stuff**. Add a file to **Stuff** called **things**. Using **nano** type some text into **things** and save the file. Now edit **things** again in **nano** but save the file as **things.txt**. Finally, type **ls** what files do you see?

Record all your commands.

Moving Directories and Files

To move a file or directory the **mv** (move) command is used. This is the first command we have used that requires two arguments. You need to specify the source and the destination for the moving.

$ **mv SOURCE DESTINATION**

1) Lets move **heaven.txt** and **earth.txt**
2) $ **cd /home/***username***/Unix**
3) $ **mv heaven.txt Work/**
4) $ **mv earth.txt Work/**
5) $ **ls**
6) $ **ls Work/**

7) We could have moved these files all at once using wildcards. An asterisk (*) means match anything.
8) $ **mv *.txt Work/** #This will move any file that ends with .txt
9) $ **mv *t Work/** # This moves any file or directory that ends with a t
10) $ **mv *ea* Work/** #This works because only heaven and earth contain 'ea'
11) $ **mv** can also be used to rename files
12) $ **touch rags**
13) $ **ls**
14) $ **mv rags Work/riches**
15) $ **ls Work/**

16) Here we move and renamed the file **rags** to **Work/riches**
17) We can rename it without moving the file
18) $ **mv Work/riches Work/rags**
19) $ **ls Work/**

20) The **mv** command is used to rename files or directories, as there is no 'rename' command in Unix.

Task

Move to your home directory and create a new directory called **`Here`**.

<u>*Without changing directories*</u> move **`Here`** to your **`Work`** directory on **`/home/`***`username`* **`/Unix/Work`**. After moving **`Here`** to **`Work`** move **`Here`** back to home but rename it **`AndBackAgain`**. Record your commands.

> **UNIX TIP:** You can retrieve previous commands by accessing your history. On the command prompt pressing the up arrow will cycle through your previous commands. Typing history will list the last few hundred commands you entered. We will learn how to search through this type of information later. But using the up arrow to modify previous commands or commands with typo's is another way to save time and keystrokes.

Copying Directories

To copy a file or directory **cp** (copy) command is used. Just like **mv** you will need a source and a destination to copy something.

$ cp SOURCE DESTINATION

1) Copying files is similar to moving them
2) $ **cd /home/***username***/Unix/Work**
3) $ **mkdir Copy**
4) $ **cd Copy**
5) $ **touch file1**
6) $ **cp file1 file2**
7) $ **ls**

8) Remember we do not have to be in a directory to make, move, or copy files.
9) $ **touch ~/file3**
10) $ **ls**
11) $ **cp ~/file3 .** # here we represent the current directory with a **.** (dot)
12) $ **ls**

13) The **cp** command can also move directories using a flag
14) $ **mkdir Example**
15) $ **mv file* Example/**
16) $ **ls**

17) $ **cp -r Example/ Example2**
18) $ **ls Example Example2/**

19) What happens without the **-R** flag?
20) $ **cp Example2/ Example3**

21) The error occurs because the **-r** flag means copy recursively. Since Example2 is not empty **cp** (without **-r**) does not descend into Example2 and copy those files it simply tries to move a directory without moving the things in the directory.

Task

Lets dig into the **`man`** for **`cp`**.

Are there other ways to tell **`cp`** to copy recursively?

How could we use **`cp`** to preserve the newest version of a file?

Besides $ man **`cp`** how else could we get help on this, or other, Unix commands?

How can we be sure **`cp`** will not overwrite files?

Viewing the contents of a directory

To view the contents of directories we use the **ls** (list segments) command.

$ **ls DIRECTORY**

If no directory is provided **ls** will list the contents of the current directory.

1) We have been using **ls** frequently to check directory contents. However, there are many options for using **ls**. As the previous example noted we can use **ls** on multiple directories at the same time.
2) $ **ls –l /home/***username***/Unix**
3) $ **ls –p /home/***username***/Unix**
4) $ **ls –othr /home/***username***/Unix** #Nic's favorite!!

5) As you can see these flags/options change the way **ls** displays the contents of the directory, giving us more or less information.
6) Notice the changes that the **–p** or the combination of **–o**, **-t**, **-h**, and **–r** flags makes to the output.

Task

Try the following commands on any directory of your choosing.

$ **ls -l**

$ **ls -R**

$ **ls -l -t -r**

$ **ls -lhS**

Look through the man and determine what each of these flags does.

How can you display hidden files with **ls**?

Can you sort files by extension? How?

When sorting by extension what is another good flag to use?

The most dangerous Unix command! Proceed with extreme caution.

If you run **`ls`** on your Unix/Work/ directory I'm sure it is full of lots of empty files and directories by this point. Wouldn't it be nice if there were a way to clean that up? Of course there is a way, however it can be dangerous.

Please read this section carefully. Misuse of the `rm` command can delete your entire computer. Seriously.

To delete directors and files from the system we have two options the **`rm`** (<u>r</u>e<u>m</u>ove) and **`rm –r`** commands.

$ **`rm FILE`**

This command will remove files, PERMANTELY. There is no trashcan, recycle bin, or archive. Files or directories removed by **`rm`** are gone, forever. With **`rm`** it is possible to delete everything in your home directory, everything. This is why it is such a potentially dangerous command.

$ **`rmdir DIRECTORY`**

This command will remove any directory along with ALL of its contents.

One more time just to be clear. It is possible to delete EVERY file you have ever created with the **`rm`** command. Thankfully there is a way to make **`rm`** a bit safer, and on DT2, this is the default setting. Using the **`–i`** flag **`rm`** will ask for confirmation before deleting anything.

1) $ **`cd /home/`***`username`***`/Unix/Temp1/`**
2) $ **`ls`**

3) If you remember the **`Temp2`** directory is empty therefore we can use **`rm -r`** to delete it.
4) $ **`rm -r Temp2/`**
5) $ **`ls`**
6) We can now move up a level and remove **`Temp1`**
7) $ **`cd ..`**
8) $ **`rm -r Temp1/`**
9) $ **`ls`**

10) Now try $ **`rm Temp1.1`**

11) See the error message letting us know that we are trying to remove a directory not a file.
12) See? Linus is warning us. So back to the recursive flag:
13) $ **rm -r Temp1.1**

14) Note the default behavior of **rm** is to simple delete with out confirmation of what you typed (except for directories of course). This is why it is so dangerous.
15) You can have rm ask for conformation before deleting anything using the **-i** flag (no one does this in practice though).
16) $ **mkdir -p Temp1/Temp2/Temp3/Temp4**
17) $ **cp -r Temp1 Temp1.1**
18) $ **ls**
19) $ **rm Temp1**
20) $ **rm -i Temp1.1**
21) $ **ls**

Task

Use **`rm`** or **`rmdir`** or any combination to remove the **`Work/`** directory and all of its contents. Take some time to review **`man rm`**.

Display the contents of a file

There are various commands available to display/print the contents of a file. The default of all these commands is to display the contents of the file on the terminal. These commands are **`less`, `cat`, `head`,** and **`tail`**.

$ **`less FILENAME`**

Displays file contents on the screen with line scrolling (to scroll you can use 'arrow' keys, 'PgUp/PgDn' keys, 'space bar' or 'Enter' key). Press 'q' to exit.

$ **`cat FILENAME`**

Simplest form of displaying contents. It **cat**alogs the entire contents of the file on the screen. In case of large files, entire file will scroll on the screen without pausing.

$ **`head FILENAME`**

Displays only the 10 starting lines of a file by default. Any number of lines can be displayed with the –n flag followed by the number of lines.

$ **`tail FILENAME`**

As the name implies the opposite of head this displays the last 10 lines. Again –n option can be used to change this.

1) Lets work though these commands.
2) $ **`cd /home/username/Unix`**
3) $ **`less Data/Arabidopsis/At_proteins.fasta`**
4) Try this: type "=**".** This is quite a big file. You can see at the bottom **`less`** displays we are looking at lines 1-32 of 269,463 and we are 0% through the file.
5) We can use '**`h`**' to get help commands for **`less`**.
6) Page forward using 'space', move a line at a time with 'j' (forward) or 'k' (backward) or N lines.
7) Hit 'q' to exit the help
8) Navigate around using the various commands
9) Try hitting 'j' 'enter' '100' 'enter'
10) Press 'q' when ready to exit **`less`**.
11) Navigate the file using the more command, press 'q' to exit.
12) $ **`cat`** is the simplest form of viewing and file. **`cat`** prints all of the file to the screen from start to finish.

13) $ **`cat Data/Arabidopsis/At_genes.gff.short`**
14) Did you get all of that?
15) $ **`cat`** is most useful with combined with other commands using **|** (pipes). We will cover this later.
16) The last two commands **`head`** and **`tail`** are fantastic when you need to look at a file and make sure things are in order.
17) $ **`head Data/GenBank/E.coli.genbank`**
18) $ **`head Data/GenBank/Y.pestis.genbank`**

19) We can change how many lines we see using the **`-n`** flag
20) $ **`head -n 1 Data/GenBank/E.coli.genbank Data/GenBank/Y.pestis.genbank`**

21) $ **`tail Data/GenBank/E.coli.genbank Data/GenBank/Y.pestis.genbank`**

22) This shows us the end of a file. This can be important when transferring files or data and needing to make sure everything transferred completely.

File permissions

All files in any operating system have a set of permissions associated with the file that define what can be done with the file and by whom. What = read, write (modify), and/or execute a file. Whom = user, group, or public.

These permissions are denoted with the following syntax:

Permissions		**Relations**	
Read	**r**	**User**	**u**
Write	**w**	**Group**	**g**
Execute	**x**	**Others**	**o**
		all users	**a**

Permissions Owner Group Size Last modified Name

```
drwxr-xr-x 7 nblouin 128K 02-21 14:19 Data
-rw-rw-r-- 1 nblouin    0 02-24 14:27 heaven.txt
-rw-rw-r-- 1 nblouin    0 02-24 14:27 earth.txt
-rw-rw-r-- 1 nblouin   21 02-24 14:30 basic_info.txt
-rw-rw-r-- 1 nblouin    6 02-24 14:32 onestep.txt
drwxrwxr-x 2 nblouin  512 02-24 14:44 AndBackAgain
drwxrwxr-x 5 nblouin  512 02-24 14:55 Work
drwxrwxr-x 3 nblouin  512 02-24 15:18 Temp1
-rwxrw-r-- 1 nblouin   52 02-24 16:20 hello.sh
drwxr-xr-x 2 nblouin  512 02-24 16:23 Code
```

Type (d=directory) u g o

Changing permissions is done via **chmod** (<u>**CH**</u>ange <u>**MOD**</u>e) command

$ **chmod [Options] RELATIONS [+ or -] PERMISSIONS FILE**

1) Lets make a new directory and add some files.
2) From the **Unix/** directory
3) $ **mkdir Allow**
4) $ **cd Allow/**
5) $ **touch read.txt write.txt execute.go all.txt**
6) $ **ls**
7) $ **ls -l**

8) We have created some files but we need to change the permission for these files in order to share these or execute them as programs.
9) Since you created these files you're the owner and have the ability to change their permissions with **`chmod`**.
10) From this you can see the default is for the user to have **`rw`** access and the group and others to have **`r`** access.
11) Lets add execute permissions for everyone on **`execute.go`**
12) $ **`chmod a+x execute.go`**
13) $ **`ls -l`**

14) We have now added the "**`x`**" option to all three levels of permission for this file
15) If we want other members of the group to have write permission for **`write.txt`** we can do that as well.
16) $ **`chmod a+w write.txt`**
17) $ **`ls -l`**

18) Others still cannot modify this file but now members of the group will be able to modify the contents.
19) If we want a file to be completely public we need all of the flags active.
20) $ **`chmod a+rwx all.txt`**
21) $ **`ls -l`**

22) Now the file all.txt can be read, written, or executed by anyone on this system.
23) We can also remove permissions using this same command.
24) $ **`chmod a-rwx all.txt`**
25) $ **`ls -l`**

26) Now we have removed all access to the **`all.txt`** file even the owner's access.
27) Finally we can change the permissions of all the files in a directory with the **`-R`** flag.
28) $ **`cd ..`**
29) $ **`chmod -R a+rwx Allow/`**
30) $ **`ls -l Allow/`**

31) This made all of these files public in one step.

Task

What group does your user account belong to?

Make a directory and some files. Change the permissions using **`chmod`**. Now using **`-R`** flag make the whole folder **`rwx`**. Make a new directory below your test directory with a new file. What are the permissions on that new file?

Your first script

Just like Perl, Python, R, etc. Unix can be used as a programming language. Depending upon the task a shell script might be all you really need to get your task completed.

To make a script we simply write shell commands into a file and then treat that file like any other program or command.

1) From the **`Unix/Code/`** directory
2) $ **`nano hello.sh`**
3) Type the following two lines

```
#This is my first shell script
echo "Hello World"
```

4) Save the file and exit **`nano`**
5) $ **`chmod u+x hello.sh`**
6) $ ./**`hello.sh`**
7) Voilà! It's that simple.
8) Now move to another directory and see if you can still run **`hello.sh`**
9) This works because we've added the **`Code`** directory to the **`PATH`**. Basically Unix knows to look in this directory for commands we type.

Task

From your home directory (execute all commands from your home directory)

1) Move **`hello.sh`** to **`Unix/`**
2) run **`hello.sh`**
3) Now move **`hello.sh`** back to **`Unix/Code`**, it should work again

Useful shell scripts

Look in the **Data/Unix_test_files** directory. You should see several files (all are empty) and four directories.

1) Put the following information into a shell script and save it as **cleanup.sh**.

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

2) Now return to **Data/Unix_test_files**
3) $ **ls -l**
4) $ **cleanup.sh**
5) $ **ls -l**
6) It should place the relevant files in the correct directories. This is a relatively simple use of shell scripting. As you can see the script just contains regular Unix commands that you might type at the command prompt.
7) Did you notice the **#!/bin/bash** line in this script? There are several different types of shell script in Unix, and this line makes it clearer that a) that this is actually a file that can be treated as a program and b) that it will be a bash script (bash is a type of Unix).

Task

Copy this information into a file called **change_file_extension.sh** and again place that file in the **Code** directory.

```
#!/bin/bash

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

Now go to the **Data/Unix_test_files/Text** directory. Run the following command:

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

Now run the **ls** command to see what has happened to the files in the directory?

Try using this script to change the file extensions of other files.

It's not essential that you understand exactly how this script works at the moment (things will become clearer as you learn Python), but you should at least see how a relatively simple Unix shell script could be potentially very useful.

Unix Power Commands

The commands that you have learned so far are essential for doing any work in Unix, but they don't really let you do anything that is very useful. The following section will introduce new commands that will start to show you the power of Unix.

Pipes and redirects

Everything we have done so far has sent the result of the command to the screen. This is feasible when the data being displayed is small enough to fit the screen or if it is the endpoint of your analysis. But for large data outputs, or if you need a new file, printing to the screen isn't very useful. Unix has built in methods to hand output from commands using **>** (greater than) or **<** (lesser than) or **>>** signs.

< redirects the data to the command for processing

> redirects the data from the command's output to a file. The file will be created if it is non-existing and if present it will overwrite the contents with the new output data (you will lose the original file).

>> unlike **>** this redirection lets user append the data to an already existing file or a new file

Another special operator **|** (called pipe) is used to pass the output from a command to another command (as input) before sending it to an output file or display.

Some examples:

```
$ cat FILE1 > FILE2
```

Creates a new file (file2) with same contents as old file (file1)

```
$ cat FILE1 >> FILE2
```

Appends the contents for file1 to file2, equivalent to opening file1, copying all the contents, pasting the copied contents to the end of the file2 and saving it!

```
$ cat FILE1 | less
```

Here, **cat** command displays the contents of the file1, but instead of sending it to standard output (screen) it sends it through the pipe to the next command **less** so that contents of the file are now displayed on the screen with line scrolling.

1) From the **Unix/Data/** directory
2) $ **cat seq.fasta**
3) $ **head seq.fasta > new.txt**
4) $ **cat new.txt**
5) $ **tail seq.fasta > new.txt**
6) $ **cat new.txt**
7) Now lets try that with the append option.
8) $ **head -n 1 seq.fasta > new.txt**
9) $ **tail -n 1 seq.fasta >> new.txt**

Task

The **`Data/`** directory contains a few fasta files with the extension **`.fa`**. Combine all of these files into a single file called **`sequences.fasta`** using redirects.

Grep

The **grep** (**g**lobally search a **r**egular **e**xpression and **p**rint) is one of the most useful commands in Unix and it is commonly used to filter a file/input, line by line, against a pattern.

grep [OPTIONS] PATTERN FILENAME

Like any other command there are various options available **man grep** for this command. Most useful options include:

```
-v inverts the match or finds lines NOT containing the pattern.
--color colors the matched text for easy visualization
-i ignore case for the pattern matching.
-l lists the file names containing the pattern
-n prints the line number containing the pattern
-c counts the number of matches for a pattern
```

Some typical scenarios to use **grep**:

- Counting number of sequences in a multi-fasta sequence file
- Get the header lines of fasta sequence file
- Find a matching motif in a sequence file
- Find restriction sites in sequence(s)
- Get all the Gene IDs from a multi-fasta sequence files and many more.

You might already know that fasta files header must start with a '**>**' character, followed by a DNA or protein sequence on subsequent lines. To find only those header lines in a fasta file, we can use **grep**.

1) Move to **Unix/Data/Arabidopsis**
2) $ **grep ">" intron_IME_data.fasta**
3) Did you get that?
4) Remember the default for a program is to output to the screen.
5) We can fix this with a redirect or a pipe.
6) $ **grep ">" intron_IME_data.fasta | less**
7) This takes the output from **grep** and sends it as input to less
8) What if we want to know how many sequences are in a file?
9) $ **grep –c ">" intron_IME_data.fasta**
10) We can also get lines that don't match our string.
11) $ **grep –v ">" intron_IME_data.fasta | less**
12) Given a the fasta file structure we can use **grep** to separate this information
13) $ **grep ">" intron_IME_data.fasta > intron_headers.txt**

14) $ **grep -v ">" intron_IME_data.fasta > intron_sequences.txt**
15) We can even get some biological information from **grep**
16) $ **grep --color "GAATTC" chr1.fasta**
17) GAATTC is the EcoR1 cut site. The **--color** option highlights the matches in this sequence.

Task

Let's use **grep** to get some information that we might be interested in knowing from these files. Using **grep** and these files lets determine:

1) How many EcoR1 cut sites are there in chr1.fasta?
2) Can you find a transcription factor in At_proteins.fasta? How many?
3) What does the following command do?
   $ **grep –c –w "ATP" At_proteins.fasta**
   $ **grep –l "AAA" ../*.fa**

4) Let's find all of the transcription factor annotations!

   $ **grep -i "transcription factor" At_proteins.fasta | less**

5) What if we don't want to see genes on chromosome 1?
6) What if we only want to genes on chromosome 5?

Regular expressions

`grep` + regular expressions = power! Before we get into this let's start with a task.

Task

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

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

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

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

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

When working with the sequences (protein or DNA) we are often interested to see if a particular feature is present or not. This could be various things like a start codon, restriction site, or even a motif. In Unix all strings of text that follow some pattern can be searched using some formula called regular expressions. *e.g.* As you learned above regular expressions consist of normal and metacharacters. Commonly used characters include

Expression	Function
`.`	matches any single character
`$`	matches the end of a line
`^`	matches the beginning of a line
`*`	matches one or more character
`\`	quoting character, treat the next character followed by this as an ordinary character.
`[]`	matches one or more characters between the brackets
`[range]`	match any character in the range
`[^range]`	match any character except those in the range
`\{N\}`	match N occurrences of the character preceding (sometimes simply +N) where N is a number.
`\{N1,N2\}`	match at least N1 occurrences of the character preceding but not more than N1
`?`	match 1 occurrence of the character preceding
`|`	match 2 conditions together, \(this\\|that)\ *matches both this or that in the text*

Some common patterns for Nucleotide/Protein searches.

Patterns	Matches
`^ATG`	Find a pattern starting with ATG
`TAG$`	Find a pattern ending with TAG
`^A[TGC]G`	Find patterns matching either ATG, AGG or ACG
`TA[GA]$`	Find patterns matching either TAG or TAA
`^A[TGC]G*TGTGAACT*TA[GA]$`	Find gene containing a specific motif
`[YXN][MPR]_[0-9]\{4,9\}`	Find patterns matching NCBI RefSeq (*eg* XM_012345)
`\(NP|XP\)_[0-9]\{4,9\}`	Find patterns matching NCBI RefSeq proteins

Let's use **grep** to find a zinc finger motif. For simplicity let's assume zinc finger motif to be CXXCXXXXXXXXXXXXHXXXH. Either you can use dots to represent any amino acids or use complex regular expressions to come up with a more representative pattern.

```
$ grep --color "C..C............H...H" At_proteins.fasta

$ grep --color "C.\{2\}C.\{12\}H.\{3\}H" At_proteins.fasta

$ grep --color "C[A-Z][A-Z]C[A-Z]\{12\}H[A-Z][A-Z][A-Z]H"
At_proteins.fasta
```

These all do the exact same thing. As you can see regular expressions can be very useful for finding patters of all kinds.

UNIX TIP: You can use regular expressions in **`grep, sed, $ awk, less, perl, python`**, certain text editors almost any programing language or tool can utilize the power of regex.

tr

The **tr** (**tr**ansliterate) command is used to translate the input file and produce a modified output. It uses two sets of parameters and replaces the occurrence of the characters in the first set with the elements from the other set.

$ **tr [options] "String1" "String2" < INFILE > OUTFILE**

This makes changing things in files very easy.

1) From the **Data/Arabidopsis/** directory
2) $ **head –n 2 chr1.fasta**
3) $ **head –n 2 chr1.fasta | tr 'A-Z' 'a-z'**
4) Using **tr** we changed the fasta from uppercase to lowercase *as well as the headers*
5) Here are some useful **tr** commands:
6) $ **tr "[:lower:]" "[:upper:]"**
7) $ **tr "ATCG" "AUCG"** # Turns cDNA into RNA
8) $ **tr ' ' '\n'** #Single spaces a document
9) In the previous command **\n** is the syntax for a newline Unix. Sadly, Mac, PC's, and Unix **DO NOT** use the same newline character.
10) From the **Unix/Data/Misc** directory
11) $ **cat -v excel_data.csv**
12) Do you see all of those **^M** characters? This is how a Windows (Dos) represents the newline, Unix does not like this.
13) $ **tr -d '^M' < excel_data.csv**
14) $ **tr –d '^M' < excel_data.csv > excel.fixed**
15) Newer versions of Unix (including this one) have built in commands to deal with this exact issue mac2unix, dos2unix, and unix2dos. These programs will edit the file and save it without any redirects.
16) $ **dos2unix excel_data.csv**
17) Anytime you bring a table or file from your PC or Mac to the command line I would recommend you fix the newline character.

sed

tr lets you change a range of characters but what if you want to change a specific string? Enter **sed**. **sed** is a stream editor that reads one or more text files and makes changes or edits then writes the results to standard output. The simple syntax for **sed** is:

$ **sed 'OPERATION/REGEXP/REPLACEMENT/FLAGS' FILENAME**

Above, **/** is the delimiter but you can use **_** **|** or **:** as well.

OPERATION = the action to be performed, the most common being **s** which is for substitution.

REGEXP and REPLACEMENT = the search term and the substitution for the operation be executed.

FLAGS = additional parameters that control the operation, common FLAGS include:

g replace all the instances of REGEXP with REPLACEMENT (globally)
n (n=any number) replace nth instance of the REGEXP with REPLACEMENT
p If substitution was made, then prints the new pattern space
i ignores case for matching REGEXP
w If substitution was made, write out the result to the given file
d when specified without REPLACEMENT, deletes the found REGEXP

1) From **Unix/Data/Arabidopsis**
2) $ **head -n 1 chr1.fasta**
3) $ **sed 's/Chr1/Chromosome_1/g' chr1.fasta | head -n 1**
4) $ **sed 's:Chr1:Chromosome_1:g' chr1.fasta | head -n 1**
5) As you can see these two commands do the same thing with different delimiters. We Changed "Chr1" to "Chromosome_1" in the file. However this was not done permanently. To do that we would have to write to a new file or use a flag within **sed**.
6) $ **touch greetings.txt**
7) $ **echo "Hello there" >> greetings.txt**
8) $ **head greetings.txt**

9) Now we have our file to manipulate with **sed**. We have three options for altering and saving the file

10) Option 1: Make a new file
11) $ **sed ‘s/Hello/Hi/g’ greetings.txt > greetings_short.txt**
12) $ **head greetings***

13) Option 2: Edit in place but make a backup of the original with the given extension
14) $ **sed –i.bak ‘s/Hello/Hi/g’ greetings.txt**
15) $ **head greetings***

16) Option 3: Edit in place but without a backup. NOTE if you run out of system memory or have an error this will rewrite the original file. You will not get that file back.
17) $ **sed –i ‘s/Hello/Hi/g’ greetings.txt.bak**
18) $ **head greetings***

word count

wc (word count) is a useful command in bioinformatics because it can quickly identify how many lines or words are in a file.

$ **wc FILENAME**

1) From **Data/Arabidopsis**
2) $ **wc At_genes.gff**
3) Here we have the total number of lines, words, and bytes in this file
4) $ **wc –l At_genes.gff**
5) This prints out just the line count for the input file.
6) $ **wc** is best used in with pipes but it can be useful to count things as well
7) $ **ls /home/**_UserName_**/Unix/Data/Sequences | grep ".fa" | wc –l**
8) This command tells you how many .fa files there are in the Sequences directory.

Sort

$ **sort** command can be used to arrange things in a file. Simplest way to use this command is:

```
$ sort FILE1 > SORTED_FILE1
```

`-n`	numerical sort
`-r`	reverse sort
`-k N,N`	sort the Nth field (column), where N is a number. Sorting can also be done on the exact character on a particular field *eg.* `-k 4.3,4.4` sorts based on 3rd and 4th character of the 4th field. Additionally you can supply additional `-k` for resolving ties.
`-t`	specify the delimiters to be used to identify fields (default is TAB) **`-t ':'`** to use ':' as delimiter

Task

The Unix/Data/Sequences directory consists of numerically labeled files. Unix can sort either alphabetically or numerically (not both) and hence they are arranged in `Seq1.fa`, `Seq10.fa`, `Seq11.fa` etc. In order to sort them in an easy to read way, try using

```
$ ls |sort –t 'q' -k 2n
```

This command lists all the files in sequences directory and then passes it to sort command. Sort command then sorts it numerically but only using 3rd and 4th letters of the first field (file name)

Try using **`sort`** on **`Data/Arabidopsis/At_genes.gff`**

```
$ sort -r -k 1,1 At_genes.gff
$  sort -r -k 4,4 At_genes.gff
```

uniq

uniq (**uniq**ue) command removes duplicate lines from a sorted file, retaining only one instance of the running matching lines. Optionally, it can show only lines that appear exactly once, or lines that appear more than once. **uniq requires sorted input** since it compares only consecutive lines.

```
$ uniq [OPTIONS] INFILE OUTFILE
```

Useful options include:

`-c` count; prints lines by the number of occurrences
`-d` only print duplicate lines
`-u` only print unique lines
`-i` ignore differences in case when comparing
`-s N` skip comparing the first N characters (N=number)

Task

From Data/

1) $ **cat uniq.txt**
2) Using the above options do the following:
   a. Count the occurrence of each unique line
   b. Print only duplicated lines.
   c. Print only unique lines.

Comparing two files

diff (**diff**erence) reports differences between two files.

$ diff [OPTIONS] FILE1 FILE2

Useful options include

-b ignore blanks
-w ignore white space (spaces and tabs)
-i ignore case
-r recursively compare all files (when comparing folders)
-s list all similar files (when comparing folders)
-y side by side comparison of files

Differences are reported line by line using a (addition), c (changed), and d (deleted) along with < or > indicating the direction of the change.

1) From **Unix/Data**
2) $ **diff uniq.txt diff.txt > diff_test.txt**
3) $ **cat diff_test.txt**

4) He we see the output from **diff** that covers each change
5) **2d1** = line 2 in file 1 is deleted from file 2 < aa
6) **5,12d3** = lines 5-12 in file 1 are deleted at line 3 in file 2
7) **15a7,8** = at line 15 in file 1 there is an addition of lines 7,8 in file 2.

comm (**comm**on) command compares two sorted files line by line.

$ comm [OPTIONS] FILE1 FILE2
-1 suppress lines unique to FILE1
-2 suppress lines unique to FILE2
-3 suppress lines that appear in both files

Task

Use **comm** with all three options on uniq.txt and diff.txt

Dividing files

cut divides the file into several parts and displays selected columns or fields from each line of a file. Normally **cut** command requires how the fields are separated and what fields need to be displayed.

$ cut -f 1 FILE displays 1st column of a file, assumes TAB as delimiter

$ cut -d ',' -f 2-4 FILE displays columns 2,3 and 4 of a file separated by ","

$ cut -d '|' -f 1,9 FILE displays 1st and 9th columns of a file separated by "|"

split generates output files of a fixed size (bytes or lines). Useful when a huge file needs to be processed.

$ **split -d -l 100 FILENAME SUFFIX**

here –d specifies numeric suffix only (suffix00, sufix01, suffix02 *etc.*) while **-l** specifies number of lines in each file (100 in this case). If you want to split based on bytes, you can use **-b** option (**-b 1k** or **-b 1m** for 1 KB and 1 MB respectively)

You can join these files back using

$ **cat suffix0[0-2] >> joinedfile**

Task

1) Display only first column of the **At_genes.gff** file using **cut**
2) Now can you display that in a way so you can actually see what the output looks like?
3) What if you want column 1, 4, and 5?
4) Let's split **At_genes.gff** into 10,000 line segments. Use "gff_split" as the suffix for the new files.
5) How many files did that make?
6) Can you put them back together?

A Unix "One-liner"

Let's say that we want to extract five sequences from **intron_IME_data.fasta** that are: a) from first introns, b) in the 5' UTR, and c) closest to the TSS. Therefore we will need to look for FASTA headers that contain the text 'i1' (first intron) and also the text '5UTR'. Every intron sequence in this file has a header line that contains the following pieces of information:

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

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

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

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

Task

Break down the above command and figure out what it is doing.

See if you can write your own one-liner. In the **At_genes.gff** file there are multiple types in column 3 (gene, CDS, mRNA, etc) how may of each type are there? This can be done with a one-liner in 3 commands.

Command	Function	Syntax/example usage
Navigation		
ls	list contents	ls [OPTIONS] DIRECTORY
pwd	print working directory	pwd
cd	change directory	cd ~ or cd *#home directory* cd .. *#previous (parent directory)*
File/Directory operations		
mkdir	make directory	mkdir DIRECTORY
cp	copy files/directories	cp SOURCE DESTINATION
man	manual page (help)	man COMMAND
mv	move files/directories	mv SOURCE DESTINATION
touch	create file	touch FILE
nano	edit file	nano FILE
less	view file (with more options)	less FILE
more	view file (with less options)	more FILE
cat	catalog file contents	cat FILE
head	show first few lines of a file	head FILE
tail	show last few lines of a file	tail FILE
rmdir	remove empty directory	rmdir DIRECTORY
rm	remove file(s)	rm FILE
Compression/archiving		
zip	zip compress	zip OUTFILE.zip INFILE.txt zip -r OUTDIR.zip DIRECTORY
unzip	decompress zipped file	unzip ANYTHING.zip
tar	archive and compress files/directories	tar -czvf OUTFILE.tar.gz DIRECTORY tar -xzvf OUTFILE.tar.gz
gzip	gzip files	gzip SOMEFILE
gunzip	decompress gzipped files	gunzip SOMEFILE.gz
File permissions		
chmod	change permissions for files/directories	chmod [OPTIONS] RELATIONS[+ or -]PERMISSIONS FILE
File manipulations		
grep	search a pattern	grep [OPTIONS] "PATTERN" FILENAME
sed	stream edit a file	sed 's/search/replace/g' FILENAME
awk	multi-purpose command	awk 'PATTERN {ACTION}' FILENAME
tr	translate or transliterate a file	tr [OPTIONS] "STRING1" "STRING2" <INFILE
wc	word count	wc FILENAME
sort	sort files	sort FILE1 > SORTED_FILE1
uniq	display unique lines	uniq [OPTIONS] INFILE > OUTFILE
diff	display difference	diff [OPTIONS] FILE1 FILE2
comm	display common lines among files	comm [OPTIONS] FILE1 FILE2
cut	break files vertically based on fields	cut -d "DELIMITER" -f NUMBER FILE
split	break files horizontally	split [OPTIONS] FILENAME