Download and open lesson material:

We will spend most of our time learning about the basics of the shell by manipulating some experimental data. To get the data for this test, please download the file `bash_shell.tar.gz` (it was emailed to you). Please save this file in your home directory. We will use this later in the lesson.

What is the shell and how do I access it?

The *shell* is a program that presents a command line interface which allows you to control your computer using commands entered with a keyboard instead of controlling graphical user interfaces (GUIs) with a mouse/keyboard combination.

A *terminal* is a program you run that gives you access to the shell. There are many different terminal programs that vary across operating systems.

There are many reasons to learn about the shell. In my opinion, the most important reasons are that:

1. It is very common to encounter the shell and command-line-interfaces in scientific computing, so you will probably have to learn it eventually

2. The shell is a really powerful way of interacting with your computer. GUIs and the shell are complementary - by knowing both you will greatly expand the range of tasks you can accomplish with your computer. You will also be able to perform many tasks more efficiently.

3. My reasons: access remote servers, repeatability, documentation

The shell is just a program and there are many different shell programs that have been developed. The most common shell (and the one we will use) is called the Bourne-Again SHell (bash). Even if bash is not the default shell, it is usually installed on most systems and can be started by typing `bash` in the terminal. Many commands, especially a lot of the basic ones, work across the various shells but many things are different. I recommend sticking with bash and learning it well. (Here is a link for more information)

To open a terminal, just single click on the "Terminal" or "Git Bash" icon on the Desktop.

Let's get started

Bash command cheat sheet

COMMAND	ACTION
echo	prints text to screen
pwd	prints path of current working directory
ls	lists all files in a directory
cd	change directory
rm	removes file
mv	moves files from one directory to another (or renames them)
cp	copies files

One very basic command is `echo`. This command just prints text to the terminal. Try the command:

```
 echo Hello, World
```

Then press enter. You should see the text "Hello, World" printed back to you. The echo command is useful for printing from a shell script, for displaying variables, and for generating known values to pass to other programs.

Moving around the file system

Let's learn how to move around the file system using command line programs. This is really easy to do using a GUI (just click on things). Once you learn the basic commands, you'll see that it is really easy to do in the shell too.

First, we have to know where we are. The program `pwd` (print working directory) tells you where you are sitting in the directory tree. The command `ls` will list the files in files in the current directory. Directories are often called "folders" because of how they are represented in GUIs. Directories are just listings of files. They can contain other files or directories.

Whenever you start up a terminal, you will start in a special directory called the *home* directory. Every user has their own home directory where they have full access to do whatever they want. For example, my user ID is `John`, the `pwd`command tells us that we are in the `/Users/John` directory. This is the home directory for the `John` user.

File Types

When you enter the `ls -F` command lists the contents of the current directory. The -F flag tells the computer to list the files in a way that shows their file type. There are (probably) several items in your home directory, notice that many have a slash at the end. This tells us that all of these items are directories as opposed to files. If a file has an asterisk at the end, it is *executable*.

Lets create an empty file using the `touch` command. Enter the command:

```
touch testfile
```

Then list the contents of the directory again. You should see that a new entry, called `testfile`, exists. It does not have a slash at the end, showing that it is not a directory. The `touch` command just creates an empty file.

Some terminals can color the directory entries in this very convenient way. In those terminals, use `ls --color` instead of `ls`. Now your directories, files, and executables will have different colors.

You can also use the command `ls -l` to see whether items in a directory are files or directories. `ls -l` gives a lot more information too, such as the size of the file and information about the owner. If the entry is a directory, then the first letter will be a "d". The fifth column shows you the size of the entries in bytes. Notice that `testfile` has a size of zero.

Now, let's get rid of `testfile`. To remove a file, just enter the command:

```
rm -i testfile
```

When prompted, type: y

The `rm` command can be used to remove files. The `-i` adds the "are you sure?" message. If you enter `ls` again, you will see that `testfile` is gone.

Changing Directories

First, we need to extract the data for this lesson using the following command:

```
tar -xzvf bash_shell.tar.gz
```

Now, let's move to a different directory. The command `cd` (change directory) is used to move around. Let's move into the `bash_shell` directory that contains the shell lesson material. Enter the following command:

```
cd bash_shell
```

Now use the `ls -F` command to see what is inside this directory. You will see that there is an entry which ends in a star. This means that this is an executable.

This directory contains all of the material for this boot camp. Now move to the directory containing the data for the shell tutorial:

```
cd data
```

If you enter the `cd` command by itself, you will return to the home directory. Try this, and then navigate back to the `data` directory.

Arguments

Most programs take additional arguments that control their exact behavior. For example, `-F` and `-l` are arguments to `ls`. The `ls` program, like many programs, take a lot of arguments. But how do we know what the options are to particular commands?

Most commonly used shell programs have a manual. You can access the manual using the `man` program. Try entering:

```
man ls
```

This will open the manual page for `ls`. Use the space key to go forward and b to go backwards. When you are done reading, just hit `q` to exit.

Unfortunately Git Bash for Windows does not have the `man` command. Instead, try using the `--help` flag after the command you want to learn about.

```
ls --help
```

And you also find the manual pages at many different sites online, e.g.http://linuxmanpages.com/.

Programs that are run from the shell can get extremely complicated. To see an example, open up the manual page for the `find` program, which we will use later this session. No one can possibly learn all of these arguments, of course. So you will probably find yourself referring back to the manual page frequently.

Examining the contents of other directories

By default, the `ls` commands lists the contents of the working directory (i.e. the directory you are in). You can always find the directory you are in using the `pwd` command. However, you can also give `ls` the names of other directories to view. Navigate to the home directory if you are not already there. Then enter the command:

```
ls
```

This shows you the contents of your current directory (which should be your home directory). Now type:

```
ls bash_shell
```

This will list the contents of the `bash_shell` directory without you having to navigate there. Now enter:

```
ls bash_shell/data
```

This prints the contents of `bash_shell/data`. The `cd` command works in a similar way. Try entering:

```
cd bash_shell/data
```

and you will jump directly to `data` without having to go through the intermediate directory.

Full vs. Relative Paths

The `cd` command takes an argument which is the directory name. Directories can be specified using either a *relative* path or a full *path*. The directories on the computer are arranged into a hierarchy. The absolute path tells you where a directory is in that hierarchy. Navigate to the home directory. Now, enter the `pwd` command and you should see something like:

```
/Users/John
```

which is the full name of your home directory. This tells you that you are in a directory called `John`, which sits inside a directory called `Users` which sits inside the very top directory in the hierarchy. The very top of the hierarchy is a directory called `/` which is usually referred to as the *root directory*. So, to summarize: `John` is a directory in `Users` which is a directory in `/`.

Now enter the following command but replace `/Users/erdavenport/' with your home directory:

```
cd /Users/John/bash_shell/data
```

This jumps to `data`. Now go back to the home directory. We saw earlier that the command:

```
cd bash_shell/data
```

had the same effect - it took us to the `data` directory. But, instead of specifying the absolute path (`/Users/John/bash_shell/data`), we specified a *relative path*. In other words, we specified the path relative to our current directory. A absolute path always starts with a `/`. A relative path does not. You can usually use either a absolute path or a relative path depending on what is most convenient. If we are in the home directory, it is more convenient to just enter the relative path since it involves less typing.

Now, list the contents of the /bin directory. Do you see anything familiar in there?

Saving time with shortcuts, wild cards, and tab completion

Shortcuts

There are some shortcuts which you should know about. Dealing with the home directory is very common. So, in the shell the tilde character, ~, is a shortcut for your home directory. Navigate to the `data` directory, then enter the command:

```
ls ~
```

This prints the contents of your home directory, without you having to type the absolute path. The shortcut `..` always refers to the directory above your current directory. Thus:

```
ls ..
```

prints the contents of the `/Users/John/bash_shell/`. You can chain these together, so:

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

prints the contents of `/Users/John` which is your home directory. Finally, the special directory `.` always refers to your current directory. So, `ls`, `ls .`, and `ls ./././.` all do the same thing, they print the contents of the current directory. This may seem like a useless shortcut right now, but we'll see when it is needed in a little while.

To summarize, the commands `ls ~`, `ls ~/.`, `ls ../../`, and `ls /Users/John` all do exactly the same thing. These shortcuts are not necessary, they are provided for your convenience.

Tab Completion

Navigate to the `bash_shell` directory. Typing out directory names can waste a lot of time. When you start typing out the name of a directory, then hit the tab key, the shell will try to fill in the rest of the directory name. For example, enter:

```
cd d<tab>
```

The shell will fill in the rest of the directory name for `data`. Now enter:

```
ls 0<tab><tab>
```

When you hit the first tab, nothing happens. The reason is that there are multiple directories in the `data` directory which start with 0. Thus, the shell does not know which one to fill in. When you hit tab again, the shell will list the possible choices.

Tab completion can also fill in the names of programs. For example, enter `e<tab><tab>`. You will see the name of every program that starts with an `e`. One of those is `echo`. If you enter `ec<tab>` you will see that tab completion works.

Command History

You can easily access previous commands. Hit the up arrow.
Hit it again. You can step backwards through your command history. The down arrow takes your forwards in the command history.

^-C will cancel the command you are writing, and give you a fresh prompt.

^-R will do a reverse-search through your command history. This is very useful.

Which program?

Commands like `ls`, `rm`, `echo`, and `cd` are just ordinary programs on the computer. A program is just a file that you can *execute*. The program `which`tells you the location of a particular program. For example:

```
which ls
```

Will return "/bin/ls". Thus, we can see that `ls` is a program that sits inside of the `/bin` directory. Now enter:

```
which find
```

You will see that `find` is a program that sits inside of the `/usr/bin` or `/bin`directory.

So ... when we enter a program name, like `ls`, and hit enter, how does the shell know where to look for that program? How does it know to run `/bin/ls` when we enter `ls`. The answer is that when we enter a program name and hit enter, there are a few standard places that the shell automatically looks. If it can't find the program in any of those places, it will print an error saying "command not found". Enter the command:

```
echo $PATH
```

This will print out the value of the `PATH` environment variable. More on environment variables later. Notice that a list of directories, separated by colon characters, is listed. These are the places the shell looks for programs to run. If your program is not in this list, then an error is printed. The shell ONLY checks in the places listed in the `PATH` environment variable.

Navigate to the `bash_shell` directory and list the contents. You will notice that there is a program (executable file) called `hello` in this directory. Now, try to run the program by entering:

```
hello
```

You should get an error saying that hello cannot be found. That is because this directory is not in the `PATH`. You can run the `hello` program by entering:

```
./hello
```

Remember that `.` is a shortcut for the current working directory. This tells the shell to run the `hello` program which is located right here. So, you can run any program by entering the path to that program. You can run `hello` equally well by specifying:

```
/Users/John/bash_shell/hello
```

When there are no `/` characters, the shell assumes you want to look in one of the default places for the program.

Examining Files

We now know how to switch directories, run programs, and look at the contents of directories, but how do we look at the contents of files?

The easiest way to examine a file is to just print out all of the contents using the program `cat`. Enter the following command:

```
cat data/NOTES_1
```

This prints out the contents of the `appaloosa.txt` file. If you enter:

```
cat data/NOTES_1 data/NOTES_1
```

It will print out the contents of `appaloosa.txt` twice. `cat` just takes a list of file names and writes them out one after another (this is where the name comes from, `cat` is short for concatenate).

Short Exercise

Using `cat`, print out the contents of 1404.txt in the data directory.

`cat` is a terrific program, but when the file is really big, it can be annoying to use. The program, `less`, is useful for this case. Enter the following command:

```
less data/1404.txt
```

`less` opens the file, and lets you navigate through it. The commands are identical to the `man` program. Use "space" to go forward and hit the "b" key to go backwards. The "g" key goes to the beginning of the file and "G" goes to the end. Finally, hit "q" to quit.

`less` also gives you a way of searching through files. Just hit the "/" key to begin a search. Enter the name of the word you would like to search for and hit enter. It will jump to the next location where that word is found. Try searching the `1404.txt` file for the string "AGA". If you hit "/" then "enter", `less` will just repeat the previous search. `less` searches from the current location and works its way forward. If you are at the end of the file and search for the word "cat", `less` will not find it. You need to go to the beginning of the file and search.

Remember, the `man` program uses the same commands, so you can search documentation using "/" as well!

Short Exercise

Use the commands we've learned so far to figure out how to search in reverse while using `less`.

Our data set: Sequencing Data

One of the file types you are likely to encounter as a biologist is a fasta file. A fasta file contains DNA sequence information in the following format. The first line starts with a `>` and is followed by some sort of identifying information. The next line is your DNA sequence. You can have as many lines of sequence in one file as you like. To learn some other tools/tricks in the shell, we will be using these files located in `~/bash_shell/data`.

The scenario: We are going to share our data with collaborators, but the data is a bit of a mess! There are inconsistent file names, there are extraneous "NOTES" files that we'd like to get rid of, and the data is spread across many directories. We are going to use shell commands to get this data into shape. By the end we would like to:

1. Put all of the sequence data into one file called "all_data"

2. Have all of the data files in one folder, and ensure that every file has a ".fasta" extension

3. Get rid of the extraneous "NOTES" files

4. Have a summary file that lists out the number of lines in each file

If we can get through this example in the available time, we will move onto more advanced shell topics...

Short Exercise

We should be able to do one of these tasks already:

1. Remove the silly "NOTES" files from the `data` directory.

Wild cards

Navigate to the `data` directory. This directory contains our sequencing data. If we type `ls`, we will see that there are a bunch of files which are just four digit numbers, a few files called `NOTES`, and a folder called `sequencing_data`. By default, `ls` lists all of the files in a given directory. The `*` character is a shortcut for "everything". Thus, if you enter `ls *`, you will see all of the contents of a given directory. Now try this command:

```
ls *1.txt
```

This lists every file that ends with a `1.txt`. This command:

```
ls /usr/bin/*.sh
```

Lists every file in `/usr/bin` that ends in the characters `.sh`. And this command:

```
ls *9*1.txt
```

lists every file in the current directory which contains the number `9`, and ends with the number `1` and the extension `.txt`. There are three such files: `3901.txt`,`7901.txt`, and `9901.txt`.

So how does this actually work? Well...when the shell (bash) sees a word that contains the `*` character, it automatically looks for files that match the given pattern. In this case, it identified four such files. Then, it replaced the `*9*1.txt`with the list of files, separated by spaces. In other the two commands:

```
ls *9*1.txt
ls 3901.txt 7901.txt 9901.txt
```

are exactly identical. The `ls` command cannot tell the difference between these two things.

Short Exercise

Do each of the following using a single `ls` command without navigating to a different directory.

1. List all of the files in `/bin` that contain the letter `a`
2. List all of the files in `/bin` that contain the letter `a` or the letter `b`
3. List all of the files in `/bin` that contain the letter `a` AND the letter `b`

Redirection

Let's turn to the experimental data from the sequencing experiment. This data is located in the `bash_shell/data` directory. Each file corresponds to the sequencing data for a particular individual in this experiment. We have two samples with similar names. The sequencing for 4490 didn't work well the first time (4490.1) so we tried re-sequencing the sample (4490.2). Let's combine the information for these two samples.

```
cat 4480.*.txt
```

We want to save that information to a file and eliminate the two other files:

```
cat 4480.*.txt > 4480.txt
```

This tells the shell to take the output from the `cat 4480.*.txt` command and dump it into a new file called `4490.txt`. To verify that this worked, examine the `4480.txt` file using `less`. If `4480.txt` had already existed, we would overwritten it. So the `>` character tells the shell to take the output from what ever is on the left and dump it into the file on the right. The `>>` characters do almost the same thing, except that they will append the output to the file if it already exists. If you have a file `4480.txt`, eliminate the two subfiles so we don't have duplicate information:

```
rm 4480.*.txt
```

Short Exercise

Use `>`, to create a single file called `all_data` that sits in the `bash_shell/data/`directory that contains all of the sequencing data in the `data` directory, but not including the `sequencing_data` directory.

Once you have that `all_data` file created, append the sequencing .fasta files from the `sequencing_data` folder to the end of the file.

Creating, moving, copying, and removing

We've created a file called `all_data` using the redirection operators `>` and `>>`. This file is critical - it's our analysis results - so we want to make copies so that

the data is backed up. Lets copy the file using the `cp` command.
The `cp`command backs up the file. Navigate to the `data` directory and enter:

```
cp all_data all_data_backup
```

Now `all_data_backup` has been created as a copy of `all_data`. We can move files around using the command `mv`. Enter this command:

```
mv all_data_backup /tmp/
```

This moves `all_data_backup` into the directory `/tmp`. The directory `/tmp` is a special directory that all users can write to. It is a temporary place for storing files. Data stored in `/tmp` is automatically deleted when the computer shuts down.

The `mv` command is also how you rename files. Since this file is so important, let's rename it:

```
mv all_data all_data_IMPORTANT
```

Now the file name has been changed to `all_data_IMPORTANT`. Let's delete the backup file now:

```
rm /tmp/all_data_backup
```

The `mkdir` command is used to create a directory. Just enter `mkdir` followed by a space, then the directory name.

By default, `rm`, will NOT delete directories. You can tell `rm` to delete a directory using the `-r` option. Enter the following command:

Short Exercise

Do the following:

1. Rename the `all_data_IMPORTANT` file to `all_data`.
2. Create a directory in the `data` directory called `foo`
3. Then, copy the `all_data` file into `foo`
4. Move all of the .fasta files from `sequencing_data` to the `data` folder
5. Remove the `sequencing_data` folder.

Count the words

The `wc` program (word count) counts the number of lines, words, and characters in one or more files. Make sure you are in the `data` directory, then enter the following command:

```
wc *.txt
```

For each of the files indicated, `wc` has printed a line with three numbers. The first is the number of lines in that file. The second is the number of words. Finally, the total number of characters is indicated. The final line contains this information summed over all of the files. Thus, there were 287846 characters in total.

Remember that the `*.txt` and `*.fasta` files were merged into the `all_data` file. So, we should see that `all_data` contains the same number of characters:

```
wc ../foo/all_data
```

Every character in the file takes up one byte of disk space. Thus, the size of the file in bytes should also be 287846. Let's confirm this:

```
ls -l ../foo/all_data
```

Remember that `ls -l` prints out detailed information about a file and that the fifth column is the size of the file in bytes.

The awesome power of the Pipe

Suppose I wanted to only see the total number of character, words, and lines across the files `*.txt` and `*.fasta`. I don't want to see the individual counts, just the total. Of course, I could just do:

```
wc all_data
```

Since this file is a concatenation of the smaller files. Sure, this works, but I had to create the `all_data` file to do this. Thus, I have wasted a precious 7062 bytes of hard disk space. We can do this *without* creating a temporary file, but first I have to show you two more commands: `head` and `tail`. These commands print the first few, or last few, lines of a file, respectively. Try them out on`all_data`:

```
head ../foo/all_data
tail ../foo/all_data
```

The `-n` option to either of these commands can be used to print the first or last `n` lines of a file. To print the first/last line of the file use:

```
head -n 1 all_data
tail -n 1 all_data
```

Let's turn back to the problem of printing only the total number of lines in a set of files without creating any temporary files. To do this, we want to tell the shell to take the output of the `wc *` and send it into the `tail -n 1` command. The `|` character (called pipe) is used for this purpose. Enter the following command:

```
wc * | tail -n 1
```

This will print only the total number of lines, characters, and words across all of these files. What is happening here? Well, `tail`, like many command line programs will read from the *standard input* when it is not given any files to operate on. In this case, it will just sit there waiting for input. That input can come from the user's keyboard *or from another program*. Try this:

```
tail -n 2
```

Notice that your cursor just sits there blinking. Tail is waiting for data to come in. Now type:

```
French
fries
are
good
```

then CONTROL+d. You should see the lines:

```
are
good
```

printed back at you. The CONTROL+d keyboard shortcut inserts an *end-of-file* character. It is sort of the standard way of telling the program "I'm done entering data". The `|` character is replaces the data from the keyboard with data from another command. You can string all sorts of commands together using the pipe.

The philosophy behind these command line programs is that none of them really do anything all that impressive. BUT when you start chaining them together, you can do some really powerful things really efficiently. If you want to be proficient at using the shell, you must learn to become proficient with the pipe and redirection operators: `|`, `>`, `>>`.

A sorting example

Let's create a file with some words to sort for the next example. We want to create a file which contains the following names:

```
Bob
Alice
Diane
Charles
```

To do this, we need a program which allows us to create text files. There are many such programs, the easiest one which is installed on almost all systems is called `nano`.Enter the following command:

```
nano toBeSorted
```

Windows users will need to open Notepad or Notepad++ to create the file. Make sure to save it in the folder bash_shell.

Now enter the four names as shown above. When you are done, press CONTROL+O to write out the file. Press enter to use the file name`toBeSorted`. Then press CONTROL+x to exit `nano`.

When you are back to the command line, enter the command:

```
sort toBeSorted
```

Notice that the names are now printed in alphabetical order.

Short Exercise

Use the `echo` command and the append operator, `>>`, to append your name to the file, then sort it and make a new file called Sorted.

Let's navigate back to `bash_shell/data`. Enter the following command:

```
wc data/* | sort -k 1 -n
```

We are already familiar with what the first of these two commands does: it creates a list containing the number of characters, words, and lines in each file in the `data` directory. This list is then piped into the `sort` command, so that it can be sorted. Notice there are two options given to sort:

1. `-k 1`: Sort based on the first column
2. `-n`: Sort in numerical order as opposed to alphabetical order

Notice that the files are sorted by the number of lines.

Short Exercise

Use the `man` command to find out how to sort the output from `wc` in reverse order.

Short Exercise

Combine the `wc`, `sort`, `head` and `tail` commands so that only the `wc` information for the largest file is listed

Hint: To print the smallest file, use:

```
wc data/* | sort -k 3 -n | head -n 1
```

Shell script:

One powerful way to use shell is in the form of a shell script. Making scripts that you can rerun can make your work go faster and make it reproducible.

Let's write a shell script that will take as input a directory, and then output a list of files ordered by the number of lines in each file in that directory. Navigate to the `data` directory, then:

To start, type:

```
nano ordered_lines
```

Then enter the following text:

```
#!/bin/bash
wc * | sort -k 1 -n > $1
```

Now, `cd` into the `data` directory and enter the command `../ordered_lines data_lines.txt`. Notice that it says permission denied. This happens because we haven't told the shell that this is an executable file. If you do `ls -l ../ordered_lines`, it will show you the permissions on the left of the listing.

Enter the following commands:

```
chmod a+x ../ordered_lines
../ordered_lines data_lines.txt
```

The `chmod` command is used to modify the permissions of a file. This particular command modifies the file `../ordered_lines` by giving all users (notice the `a`) permission to execute (notice the `x`) the file. If you enter:

```
ls -lF ../ordered_lines
```

You will see that the file name has a `*` at the end and the permissions have changed. Congratulations, you just created your first shell script!

Searching files

You can search the contents of a file using the command `grep`.
The `grep`program is very powerful and useful especially when combined with other commands by using the pipe. Navigate to the `bert` directory. Every data file in this directory has a line which says "Range". The range represents the smallest frequency range that can be discriminated. Lets list all of the ranges from the tests that bert conducted:

```
grep ">" 4480.txt
```

Short Exercise

Create an executable script called `count_sequences` in the `data` directory, that is similar to the `ordered_lines` script, but prints the total number of sequences in each file along with the name of that file.

Finding files

The `find` program can be used to find files based on arbitrary criteria. Navigate to the `data` directory and enter the following command:

```
find . -print
```

This prints the name of every file or directory, recursively, starting from the current directory. Let's exclude all of the directories:

```
find . -type f -print
```

This tells `find` to locate only files. Now try these commands:

```
find . -type f -name "*1*"
find . -type f -name "*1*" -or -name "*2*" -print
find . -type f -name "*1*" -and -name "*2*" -print
```

The `find` command can acquire a list of files and perform some operation on each file. Try this command out:

```
find . -type f -exec grep Volume {} \;
```

This command finds every file starting from `.`. Then it searches each file for a line which contains the word "Volume". The `{}` refers to the name of each file. The trailing `\;` is used to terminate the command. This command is slow, because it is calling a new instance of `grep` for each item the `find` returns.

A faster way to do this is to use the `xargs` command:

```
find . -type f -print | xargs grep Volume
```

`find` generates a list of all the files we are interested in, then we pipe them to `xargs`. `xargs` takes the items given to it and passes them as arguments to `grep.xargs` generally only creates a single instance of `grep` (or whatever program it is running).

Short Exercise

Navigate to the `data` directory. Use one `find` command to perform each of the operations listed below (except number 2, which does not require a `find`command):

1. Find any file whose name is "NOTES" within `data` and delete it

2. Create a new directory called `cleaneddata`

3. Move all of the files within `data` to the `cleaneddata` directory

4. Rename all of the files to ensure that they end in `.txt` (note: it is ok for the file name to end in `.fasta.fasta` or `.txt.fasta`

Hint: If you make a mistake and need to start over just do the following:

1. Navigate to the `bash_shell` directory

2. Delete the `data` directory

3. Enter the command: `git checkout -- data` You should see that the data directory has reappeared in its original state

BONUS

Redo exercise 4, except rename only the files which do not already end in `.txt`. You will have to use the `man` command to figure out how to search for files which do not match a certain name.

Bonus:

backtick, xargs: Example find all files with certain text

alias -> rm -i

variables -> use a path example

.bashrc

du

ln

ssh and scp

regular expressions

permissions

chaining commands together