Metagenomics One Day Crash Course Documentation

Release 0

Adina Howe

April 08, 2016

Contents

Contents:

These are the tutorials for a one day course I will teach at Cambridge University in April, 2016. They are a merge of my perspective of what I wish someone would have showed me in less than 6 hours when I was a total newbie to metagenomic sequencing analysis.

CHAPTER 1

Three main tutorial sections:

- *Unix Shell*
- *Getting data*
- *Running a program*

Resources:

Practical Computing for Biologist

Bioinformatics Programming with Python

Effective Computation in Physics

1.1 Introduction to the shell

Authored by Tracy Teal with Software Carpentry, with minor modifications by Ashley Shade, Joshua Herr, and Paul Wilburn EDAMAME-2015 wiki

***

1.1.1 Overarching Goal

- This tutorial will contribute towards developing of **computing literacy**

1.1.2 Learning Objectives

- Understand what the shell is, how to access it from your computer, and how to use it.
- Navigate around a Unix file system to view and manipulate files

1.1.3 Using The Shell

1.1.4 Objectives

- What is the shell?
- How do you access it?
- How do you use it?
- Getting around the Unix file system
- looking at files
- manipulating files
- automating tasks
- What is the Shell good for?
- Where are resources where I can learn more? (because the shell is awesome)

1.1.5 What is the shell?

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.

There are many reasons to learn about the shell.

- For most bioinformatics tools, you have to use the shell. There is no graphical interface. If you want to work in metagenomics or genomics you're going to need to use the shell.
- The shell gives you *power*. The command line gives you the power to do your work more efficiently and more quickly. When you need to do things tens to hundreds of times, knowing how to use the shell is transformative.
- To use remote computers or cloud computing, you need to use the shell.
- We're going to use it in this class, for all of the reasons above.

Fig. 1.1: Automation

Unix is user-friendly. It's just very selective about who its friends are.

Today we're going to go through how to access Unix/Linux and some of the basic shell commands.

1.1.6 Information on the shell

Here are some great shell cheat sheets: - Unix-Linux Command Cheat Sheet - Software Carpentry Cheat Sheet - Explain shell - a web site where you can see what the different components of a shell command are doing. - commandlinefu

1.1.7 How to access the shell

The shell is already available on Mac and Linux. For Windows, you'll have to download a separate program.

1.1.8 Mac

On Mac the shell is available through Terminal *Applications* -> *Utilities* -> *Terminal* Go ahead and drag the Terminal application to your Dock for easy access.

1.1.9 Windows

For Windows, we're going to be using MobaXTerm. Download and install MobaXterm Open up the program.

1.1.10 Linux

Well, you should be set if you're already using Linux

1.1.11 Preliminaries

We will spend most of our time using the shell to manipulate example data files.

Open your browser. In the address bar, enter

```
https://s3.amazonaws.com/edamame/EDAMAME_16S.tar.gz
```

Create a folder named `tutorial_shell` and place the EDAMAME_16S.tar.gz file there. We'll come back to it using the shell in just a minute.

1.1.12 Moving around the file system

Let's practice moving around a bit.

We're going to work in that `tutorial_shell` you just created.

Let's navigate there using the regular way by clicking on the different folders.

First we did something like go to the folder of our username. Then we clicked on an n number of directories then eventually 'tutorial_shell'

Let's draw out how that went.

Now let's draw some of the other files and folders we could have clicked on.

This is called a hierarchical file system structure, like an upside down tree with root (/) at the base that looks like this.

Fig. 1.2: Unix

That (/) at the base is often also called the 'top' level.

When you are working at your computer or log in to a remote computer, you are on one of the branches of that tree, your home directory (/home/username)

Now let's go do that same navigation at the command line.

Open The Shell

Congrats! You are in the home directory. Just to be sure, let's type:

```
cd
```

This command will always place you home.

This directory should have some other folders, perhaps files and/or programs. Let's check. Type:

```
ls
```

`ls` stands for 'list' and it lists the contents of a directory.

Oftentimes, a directory will have a mix of objects. If we want to know which is which, we can type:

```
ls -F
```

Anything with a "/" after it is a directory. Things with a "*" after them are programs. It there's nothing there it's a file.

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

As you are seeing the list of directories in the `home` folder, pick one and type:

```
cd <name of directory>
```

You have just entered a lower level directory of your choice. Check out its contents by typing:

```
ls
```

To go 'back up a level' we need to use `..`

Type:

```
cd ..
```

Sometimes when we're wandering around in the file system, it's easy to lose track of where we are and get lost.

If you want to know what directory you're currently in, type:

```
pwd
```

This stands for 'print working directory'. The directory you're currently working in.

We are ready. Using `cd <directory>`, `ls` and (optionally) `pwd`, to the `tutorial_shell` directory and list its contents. Remember, if you get lost, going home is easy with `cd` by itself.

Good work. You can now move around in different directories or folders at the command line. Why would you want to do this, rather than just navigating around the normal way?

When you're working with bioinformatics programs, you're working with your data and it's key to be able to have that data in the right place and make sure the program has access to the data. Many of the problems people run into with command line bioinformatics programs result from not having the data in the place the program expects it to be.

1.1.13 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 quit.

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

1.1.14 Examining the contents of other directories

By default, the `ls` command 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 `tutorial_shell` directory if you are not already there.

Type:

```
cd ..
```

Then enter the command:

```
ls tutorial_shell
```

This will list the contents of the `tutorial_shell` directory without you having to navigate there.

The `cd` command works in a similar way. Using `cd ..` twice, navigate two levels higher than `tutorial_shell`. Now navigate back to tutorial shell in one line of code that looks something like:

```
cd <directory_1>/<directory_2>/tutorial_shell
```

and you will jump directly to `tutorial_shell` without having to go through the intermediates.

1.1.15 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 full path tells you where a directory is in that hierarchy. Navigate to the home directory. Now, enter the `pwd` command and you should see:

```
/home/<username>
```

which is the full name of your home directory. This tells you that you are in a directory called `<username>`, which sits inside a directory called `home` 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: `<username>` is a directory in `home` which is a directory in `/`.

Let's try an exercise. Navigate to the `tutorial_shell` directory if you are not already there.

Check where you are with `pwd`. The output should look something like `/home/<username>/.../tutorial_shell` Copy the entire output. Next, go to the home directory with `cd`. Once in home directory, type:

```
cd <pwd output>
```

This jumps back to the `tutorial_shell`.

Now go back to the home directory again with `cd`. Once in home directory, type `cd` plus the output of pwd minus the `/home/<username>`

The reduced command had the same effect - it took us to the `tutorial_shell` directory. But, instead of specifying the *full path*, which starts with the root directory `/`, we specified a *relative path*. In other words, we specified the path relative to our current directory. A full path always starts with a `/`. A relative path does not.

A relative path is like getting directions from someone on the street. They tell you to "go right at the Stop sign, and then turn left on Main Street". That works great if you're standing there together, but not so well if you're trying to

tell someone how to get there from another country. A full path is like GPS coordinates. It tells you exactly where something is no matter where you are right now.

You can usually use either a full 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.

Over time, it will become easier for you to keep a mental note of the structure of the directories that you are using and how to quickly navigate amongst them.

Short Exercise

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

1.1.16 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 `tutorial_shell` directory:

Then enter the command:

```
ls ~
```

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

```
ls ..
```

prints the contents of the directory one level higher than `tutorial_shell`. You can chain these together, so:

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

prints the contents of two levels higher than `tutorial_shell`. 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, while you are in the `shell` directory, the commands `ls ~`, `ls ~/.`, `ls ../../`, and `ls /home/username` all do exactly the same thing. These shortcuts are not necessary, they are provided for your convenience.

Our data set: FASTQ files

We did an experiment and want to look at the bacterial communities a soil chronosequence using 16S sequencing. We get our data back from the sequencing center as FASTQ files, and we stick them all in a folder called MiSeq. This data is actually the data we're going to use for several sections of the course, and it's data generated by the Shade Lab at Michigan State.

We want to be able to look at these files and do some things with them.

First, let's extract the archive we have in `tutorial_shell`. Once in this directory, type:

```
tar -xzvf EDAMAME_16S.tar.gz
```

Done!

Wild cards

Navigate to the `tutorial_shell/EDAMAME_16S/Fastq` directory. This directory some of our FASTQ files we'll need for analyses. If we type `ls`, we will see that there are a bunch of files with long file names. Some of the end with .fastq

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 *fastq
```

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

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

Lists every file in `/usr/bin` that ends in the characters `.sh`.

We have paired end sequencing, so for every sample we have two files. If we want to just see the list of the files for the forward direction sequencing we can use:

```
ls *F*fastq
```

lists every file in the current directory whose name contains the letter `F`, and ends with `fastq`.

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

```
ls *F*fastq
ls C01D01F_sub.fastq    C01D02F_sub.fastq   C01D03F_sub.fastq
```

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 start with the letter 'c
2. List all of the files in `/bin` that contain the letter 'a'
3. List all of the files in `/bin` that end with the letter 'o'

BONUS: List all of the files in '/bin' that contain the letter 'a' or 'c'

Tab Completion

Navigate to the home 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 <someletter><tab>
```

The shell will fill in the rest of the directory name.

Now go to `tutorial_shell/EDAMAME_16S/Fastq`

```
ls C<tab><tab>
```

When you hit the first tab, the name is partially filled in. The reason is that there are multiple directories in the home directory which start with `C01D0`. 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.

1.1.17 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.

You can also review your recent commands with the `history` command. Just enter:

```
history
```

to see a numbered list of recent commands, including this just issues `history` command. You can reuse one of these commands directly by referring to the number of that command.

If your history looked like this:

```
259  ls *
260  ls /usr/bin/*.sh
261  ls *F*fastq
```

then you could repeat command #260 by simply entering:

```
!260
```

(that's an exclamation mark by the way).

Short Exercise

1. Find the line number in your history for the last exercise (listing files in /bin) and reissue that command.

1.1.18 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 C01D01R_sub.fastq
```

This prints out the contents of the `C01D01R_sub.fastq` file.

Short Exercises

1. Print out the contents of the `tutorial_shell/EDAMAME_16S/MappingFiles/Centralia_Full_Map.txt` file. What does this file contain?
2. Without changing directories, (you should still be in `edamame-data`), use one short command to print the contents of all of the files in the `tutorial_shell/EDAMAME_16S/Fastq` directory.

Make sure we're in the right place for the next set of the lessons. We want to be in the `tutorial_shell/EDAMAME_16S/Fastq` directory. Check if you're there with `pwd` and if not navigate there.

`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 C01D01R_sub.fastq
```

`less` opens the file, and lets you navigate through it. The commands are identical to the `man` program. To quit `less` and go back to the shell, press `q`.

Some commands in ''less''

key	action
"space"	to go forward
"b"	to go backwarsd
"g"	to go to the beginning
"G"	to go to the end
"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. 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 that does not exist from that point forward, `less` will not find it. You need to go to the beginning of the file and search.

For instance, let's search for the sequence `HWI-M03127:41:ACE13:1:1114:22908:11882` in our file. You can see that we go right to that sequence and can see what it looks like.

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

There's another way that we can look at files, and in this case, just look at part of them. This can be particularly useful if we just want to see the beginning or end of the file, or see how it's formatted.

The commands are `head` and `tail` and they just let you look at the beginning and end of a file respectively.

```
head C01D01R_sub.fastq
tail C01D01R_sub.fastq
```

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 C01D01R_sub.fastq
tail -n 1 C01D01R_sub.fastq
```

1.1.19 Searching files

We showed a little how to search within a file using `less`. We can also search within files without even opening them, using `grep`. Grep is a command-line utility for searching plain-text data sets for lines matching a string or regular expression. Let's give it a try!

Let's search for that sequence ACE13:1:2109:11596: in the C01D01R_sub.fastq file.

```
grep ACE13:1:2109:11596 C01D01R_sub.fastq
```

We get back the whole line that had '1101:14341' in it. What if we wanted all four lines, the whole part of that FASTQ sequence, back instead.

```
grep -A 3 ACE13:1:2109:11596 C01D01R_sub.fastq
```

The `-A` flag stands for "after match" so it's returning the line that matches plus the three after it. The `-B` flag returns that number of lines before the match.

** Exercise **

Search for the sequence `CCTGTTTGCTCCCCACGCTCTCGCACCTCAGTGTCA` in the `C01D01R_sub.fastq` file and in the output have the sequence name and the sequence. e.g.

```
@HWI-M03127:41:ACE13:1:1114:14857:17361 2:N:0:GGAGACAAGGGA
CCTGTTTGCTCCCCACGCTCTCGCACCTCAGTGTCAGTATCTGCCCAGGTCGCCGCCTT
```

Search for that sequence in all the FASTQ files. ****

1.1.20 Redirection

We're excited we have all these sequences that we care about that we just got from the FASTQ files. That is a really important motif that is going to help us answer our important question. But all those sequences just went whizzing by with grep. How can we capture them?

We can do that with something called "redirection". The idea is that we're redirecting the output to the terminal (all the stuff that went whizzing by) to something else. In this case, we want to print it to a file, so that we can look at it later.

The redirection command for putting something in a file is >

Let's try it out and put all the sequences that contain 'CCTGTTTGCTCCCCACGCTCTCGCACCTCAGTGTCA' from all the files in to another file called 'good-data.txt'

```
grep -B 2 CCTGTTTGCTCCCCACGCTCTCGCACCTCAGTGTCA * > good-data.txt
```

The above code makes use of the `*` wilcard to search *ALL* of the files in your current directory for the sequence. The `>` here says to write the results from the `grep` command we just ran to a new file called good-data.txt The prompt should sit there a little bit, and then it should look like nothing happened. But type `ls`. You should have a new file called good-data.txt. Take a look at it and see if it has what you think it should.

There's one more useful redirection command that we're going to show, and that's called the pipe command, and it is `|`. It's probably not a key on your keyboard you use very much. What `|` does is take the output that scrolling by on the terminal and then can run it through another command. When it was all whizzing by before, we wished we could just slow it down and look at it, like we can with `less`. Well it turns out that we can! We pipe the `grep` command through `less`.

```
grep CCTGTTTGCTCCCCACGCTCTCGCACCTCAGTGTCA * | less
```

Now we can use the arrows to scroll up and down and use `q` to get out.

We can also do something tricky and use the command `wc`. `wc` stands for `word count`. It counts the number of lines or characters. So, we can use it to count the number of lines we're getting back from our `grep` command. And that will magically tell us how many sequences we're finding. We're

```
grep CCTGTTTGCTCCCCACGCTCTCGCACCTCAGTGTCA * | wc
```

That tells us the number of lines, words and characters in the file. If we just want the number of lines, we can use the `-l` flag for `lines`.

```
grep CCTGTTTGCTCCCCACGCTCTCGCACCTCAGTGTCA * | wc -l
```

Redirecting is not super intuitive, but it's really powerful for stringing together these different commands, so you can do whatever you need to do.

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: `|`, `>`, `>>`.

1.1.21 Creating, moving, copying, and removing

Now we can move around in the file structure, look at files, search files, redirect. But what if we want to do normal things like copy files or move them around or get rid of them. Sure we could do most of these things without the command line, but what fun would that be?! Besides it's often faster to do it at the command line, or you'll be on a remote server like Amazon where you won't have another option.

The `Centralia_Full_Map.txt` is one that tells us what environmental data goes with which samples. This is a really important file, so we want to make a copy so we don't lose it.

Lets copy the file using the `cp` command. The `cp` command backs up the file. Navigate to the `Fastq/MappingFiles` directory and enter:

```
cp Centralia_Full_Map.txt Centralia_Full_Map_backup.txt
```

Now `Centralia_Full_Map_backup.txt` has been created as a copy of `Centralia_Full_Map.txt`.

Let's make a `backup` directory where we can put this file.

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

```
mkdir backup
```

We can now move our backed up file in to this directory. We can move files around using the command `mv`. Enter this command:

```
mv Centralia_Full_Map_backup.txt backup/
```

This moves `Centralia_Full_Map_backup.txt` into the directory `backup/`. Check the full path of backup with `pwd`

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

```
mv Centralia_Full_Map.txt Centralia_Full_Map_IMPORTANT.txt
```

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

```
rm backup/Centralia_Full_Map_backup.txt
```

The `rm` file removes the file. Be careful with this command. It doesn't just nicely put the files in the Trash. They're really gone.

Short Exercise

Do the following:

1. Rename the `Centralia_Full_Map_IMPORTANT.txt` file to `Centralia_Full_Map.txt`.
2. Create a directory in the `Fastq` directory called `new`
3. Then, copy the `Centralia_Full_Map.txt` file into `new`

By default, `rm`, will NOT delete directories. You can tell `rm` to delete a directory using the `-r` option. Let's delete that `new` directory we just made. Enter the following command:

```
rm -r new
```

1.1.22 Writing files

We've been able to do a lot of work with files that already exist, but what if we want to write our own files. Obviously, we're not going to type in a FASTA file, but you'll see as we go through other tutorials, there are a lot of reasons we'll want to write a file, or edit an existing file.

To write in files, we're going to use the program `nano`. We're going to create a file that contains the favorite grep command so you can remember it for later. We'll name this file 'awesome.sh'.

```
nano awesome.sh
```

Type in your command, so it looks like

```
grep -B 1 CCTGTTTGCTCCCCACGCTCTCGCACCTCAGTGTCA * > good_data.txt
```

Now we want to save the file and exit. At the bottom of nano, you see the "^X Exit". That means that we use Ctrl-X to exit. Type `Ctrl-X`. It will ask if you want to save it. Type `y` for yes. Then it asks if you want that file name. Hit 'Enter'.

Now you've written a file. You can take a look at it with less or cat, or open it up again and edit it.

Exercise

Open `awesome.sh` and add `echo AWESOME\!` after the grep command and save the file.

We're going to come back and use this file in just a bit.

1.1.23 Running programs

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

```
hello.sh
```

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

```
./hello.sh
```

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

1.1.24 Writing scripts

We know how to write files and run scripts, so I bet you can guess where this is headed. We're going to run our own script!

Go in to the 'MiSeq' directory where we created 'awesome.sh' before. Remember we wrote our favorite grep command in there. Since we like it so much, we might want to run it again, or even all the time. Instead of writing it out every time, we can just run it as a script.

It's a command, so we should just be able to run it. Give it try.

```
./awesome.sh
```

Alas, we get `-bash: ./awesome.sh: Permission denied`. This is because we haven't told the computer that it's a program. To do that we have to make it 'executable'. We do this by changing its mode. The command for that is `chmod` - change mode. We're going to change the mode of this file, so that it's executable and the computer knows it's OK to run it as a program.

```
chmod +x awesome.sh
```

Now let's try running it again:

```
./awesome.sh
```

Now you should have seen some output, and of course, it's AWESOME! Congratulations, you just created your first shell script! You're set to rule the world.

1.1.25 For Future Reference

1.1.26 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).

1.2 Fetching Data with the NCBI API

1.2.1 Learning objectives

This is a tutorials for working with the data that is available in NCBI. The learning objectives for this tutorial are as follows:

1. To be able to download specific gene sequences or genomes from NCBI (even with a big list of gene sequences).
2. To be able to create use these genes as a database to annotate a sequencing dataset.
3. To estimate the number of genes and their corresponding annotations in multiple sequencing datasets.

You will need to know some things prior to this tutorial:

1. Familiarity with the structure of NCBI website and their nucleotide and genome databases.
2. Ability to navigate in the unix shell.
3. Ability to execute programs in the shell.
4. Access and login to an Amazon EC2 instance or similar ubuntu-based server

The key challenge that we will work through...or your mission, if you choose to accept it, is to identify nitrogen fixation genes found in sequencing DNA from soils.

You have been delivered three dogma-changing metagenomes (sequencing datasets) originating from three different Iowa crop soils (corn, soybean, and prairie). You are interesting in identifying nitrogen fixation genes that are associated with native bacteria in these soils. Nitrogen fixation is a natural process performed by bacteria that converts nitrogen in the atmosphere into a form that is usable for plants. If we can optimize natural nitrogen fixation, our hope is to reduce nitrogen fertilizer inputs that may contribute to the eutrophication of downstream waters (e.g., dead zones in the Gulf of Mexico).

1.2.2 Getting the data

Get the metagenome datasets and scripts related to this tutorial.

All the tutorial materials are contained on a Github repository. The reason for using Github is that this material can be updated by me and grabbed by you lucky folk seamlessly with just a couple commands. If you are interested in learning more about Git, see these [tutorials]():

```
git clone https://github.com/adina/bodega-howe-ncbi.git
```

This command will make a directory (or folder for those more Finder/Explorer inclined) named "bodega-howe-ncbi" in the location where it was run. Within that directory, there will be two directories containing "data" and "scripts". You can see this by navigating (hint: cd) to the "bodega-howe-ncbi" directory and typing:

```
ls -lah
```

1.2.3 Understand the Data

Navigate to the data directory and identify the number of sequences in each file. Hint: To find specific characters in a file, you can use [grep](http://www.gnu.org/software/grep/manual/html_node/Usage.html). For example, to find all instances of AGTC in the corn.fa file, we could:

```
grep AGTC corn.fa
```

To find sequences, we know that each sequence will start with a special character, ">". This character in the shell, remember, is a bit special. So to find it as a symbol in the text, we're going to put a '^' right before it in quotes:

```
grep ^">" corn.fa
```

Now, to count, you'll remember we can use the command "wc", with a pipe...So you're command will look something like this:

```
grep ^">" corn.fa | wc
```

Or...if you want to do this quicky:

```
for x in *fa; do echo $x; grep ^">" $x | wc; done
```

1.2.4 Understand the gene of interest (based on a literature)

To identify nitrogen fixation genes, you've been tasked to build a database of all previously observed known nitrogen fixation genes (nifH). To build this database, you have been reading literature for about two weeks and come up with a list of about 30 genes. You'll also see this list in a file in the data directory (hint: use cat).

Check out the file containing these gene IDs.

You have a sinking feeling like this isn't really leveraging the big data biology that everyone says sequencing technologies have provided. You've decided to check out NCBI for its contents.

1.2.5 Find more genes of interest (based on NCBI)

Go to the NCBI webpage and identify an estimate of total nifH genes and download a list of their accession numbers.

You'll want to navigate in a web-browser to the http://www.ncbi.nlm.nih.gov/. You'll see in the search query box that you can search a number of databases. Here, we want to look at the nucleotide database and query something along the lines of nifH or nitrogen fixation.

When I did this, there were nearly 180,000 genes that were hit by this query. You will want to look for the "Send To" link at the upper right of the page (put on a magnifying glass!), and download the GI list for this query.

1.2.6 Determine the list of genes to build a reference database

Find that file on your computer and give it a peek.

To make this tutorial not-as-painful to complete in a reasonable amount of time, I've also made a list of 300 nifH genes from NCBI and put them in a file '300-nifh-genes.txt' in the data directory. I would highly suggest you use this gene to build your database going forward in this tutorial.

Take a look at this file. Prove to yourself that it contains 300 genes (Hint: wc)

Note: Some of these hits, I am sure, are likely not nifH. Typically, I would do some clean up of these genes to filter out any annotation that did not contain "nifH". In case you're interested, this script is in the scripts directory and is called "clean-up.py". You are welcome to play with it. Here's the command: python clean-up.py <fasta-file-uncleaned> > <fasta-file-cleaned>

Now, we are going to learn how to download these genes (by learning about the NCBI API below)

1.2.7 Download the associated sequence for genes

Think about how you would download this data if you didn't have this tutorial.

You may have thought about some of the following:

1. Go to the web portal and look up each FASTA
2. Go to the FTP site, find each genome, and download manually
3. Use the NCBI Web Services API to download the data

Among these, I'm going to assume many of you are familiar with the first two. This tutorial then is going to focus on using APIs.

Here's some answers, among which my favorite is "an interface through which you access someone else's code or through which someone else's code accesses yours – in effect the public methods and properties."

The NCBI has a whole toolkit which they call *Entrez Programming Utilities* or *eutils* for short. You can read all about it in the documentation. There are a lot of things you can do to interface with all things NCBI, including publications, etc., but I am going to focus today on downloading sequencing data.

To do this, you're going to be using one tool in *eutils*, called *efetch*. There is a whole chapter devoted to efetch – when I first started doing this kind of work, this documentation always broke my heart. Its easier for me to just show you how to use it.

1.2.8 Understanding NCBI's API

Open a web browser, and check out what NCBI knows about this gene. Check it out here.

Download the gene with eutils commands in your web-browser and take a look at the file.

On your web-browser, paste the following URL to download the nucleotide genome for gene X51500.1:

```
http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=nuccore&id=X51500.1&rettype=fasta&retmode
```

1.2.9 Bringing this to the command line

Try downloading the GenBank file instead by pasting this onto your web-browser:

```
http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=nuccore&id=CP000962&rettype=gb&retmode=te
```

Do you notice the difference in these two commands? Let's breakdown the command here:

1. <http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?> This is command telling your computer program (or your browser) to talk to the NCBI API tool efetch.
2. <db=nuccore> This command tells the NCBI API that you'd like it to look in this particular database for some data. Other databases that the NCBI has available can be found here.
3. <id=X51500.1> This command tells the NCBI API efetch the ID of the gene/genome you want to find.
4. <rettype=gb&retmode=text> These two commands tells the NCBI how the data is returned. You'll note that in the two examples above this command varied slightly. In the first, we asked for only the FASTA sequence, while in the second, we asked for the Genbank file. Here's some elusive documentation on where to find these "return" objects.

Also, a useful command is also <version=1>. There are different versions of sequences and some times that is useful. For reproducibility, I try to specify versions in my queries, see these comments.

Note: Notice the "&" that comes between each of these little commands, it is necessary and important.

Ok, let's think of automating this sort of query. So...we're moving from your lil laptop to your jumbo EC2 instance now.

Download a gene sequence on the command line.

Going back onto your instance, in the shell, you could run the same commands above with the addition of *curl* on your EC2 instance:

```
curl "http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=nuccore&id=X51500.1&rettype=fasta&r
```

You'll see it fly on to your screen. Don't panic - you can save it to a file and make it more useful BUT note the path you are in and where you will save this file (as long as you know...that's fine):

```
curl "http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=nuccore&id=X51500.1&rettype=fasta&r
```

You could now imagine writing a program where you made a list of IDs you want to download and put it in a for loop, *curling* each genome and saving it to a file. The following is a script. Thanks to Jordan Fish who gave me the original version of this script before I even knew how and made it easy to use.

To see the documentation for this script in the scripts directory:

```
python fetch-genomes-fasta.py
```

You'll see that you need to provide a list of IDs and a directory where you want to save the downloaded files.

1.2.10 Scaling up sequencing downloading from a list

Run this script (note that your paths for the script or data may need to be specified) – also see note below:

```
python scripts/fetch-genomes-fasta.py data/300-nifh-genes.txt data/nifh-database-fastas
```

Sit back and think of the glory that is happening on your screen right now...

Note: If you are nervous....you may want to run this on just a few of these IDs to begin with. You can create a smaller list using the *head* command with the -n parameter in the shell. For example, head -n 3 300-nifh-genes.txt > 3genes.txt.

1.2.11 Build your giant database

After all the 300 genes are downloaded, you will want to concatenate them into one file (Hint cat and >>), named "all-nifH.fa".

1.2.12 Under the hood

Look at the script/program content in "fetch-genomes-fasta.py".

The meat of this script uses the following code:

```
url_template = "http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=nucleotide&id=%s&rettype=
```

You'll see that the *id* here is a string character which is obtained from list of IDs contained in a separate file. The rest of the script manages where the files are being placed and what they are named. It also prints some output to the screen so you know its running.

1.2.13 A challenge exercise

You've downloaded all the fasta files from a list. Modify the script to now download genbank files for a list of genomes of interest (not nifH – genomes!, you can pick a phyla and download say 100 genomes IDs). Using shell commands you have learned in the past, can you count the number of 16S rRNA genes that are present in the file? Can you print out a list of all genes contained within the genomes? Note that you'll have to understand what genbank information is contained in a file.

1.3 Running a bioinformatic program

1.3.1 Learning objectives

This is a tutorial for running a BLAST alignment for three metagenomes against a reference gene set. The learning objectives for this tutorial are as follows:

1. To be able to run a specific bioinformatic program, BLAST.
2. To format a reference gene database for BLAST.
3. To estimate the number of genes and their corresponding annotations in multiple sequencing datasets.
4. To run a for loop in the shell.

You will need to know some things prior to this tutorial:

1. Ability to navigate in the unix shell.

1.3.2 Running BLAST

First! We need some data. In the last tutorial, we learned how to get data for a reference genome from an API, specifically NCBI's API. In this tutorial, we will use a subset of this reference database.

Also, we will imagine that we have three metagenomes from three soil types: a corn field, a soybean field, and a prairie field. We are going to identify known nitrogen fixation genes in these metagenomes.

You can grab this data by the following command. Navigate to your home directory and execute the following:

```
git clone https://github.com/germs-lab/blast-tutorial-data.git
```

This will go to a version-controlled server on Github and make a copy of this data onto your local computer in a folder called "blast-tutorial-data". You should be aware of where this folder and data exist. Git is a great program and VERY useful for bioformatics – I would highly suggest you learn more about it and maybe we will have time to discuss it more.

The first thing we need to do is to tell BLAST that our nifH reference genes contained in nifh-ref.fa are (a) a database, and (b) a nucleotide database. That's done by calling 'makeblastdb':

```
makeblastdb -in nifh-ref.fa -dbtype nucl
```

Next, we can run BLAST, or more specifically blastn (nucleotide against nucleotides alignment) by using the following command:

```
blastn -query corn.fa -db nifh-ref.fa
```

If you get an error, and if you are copy and pasting you WILL get an error, think about paths and where the data is contained. Navigate to the data directory and try:

```
blastn -query metags/corn.fa -db nifh-ref.fa
```

You'll see a BLAST output print to the screen really fast. To save it, you can identify the name of an output file:

```
blastn -query metags/corn.fa -db nifh-ref.fa -out metags/corn.fa.x.nifh.blastnout.txt
```

Maybe you don't want to see all the alignments but would rather have a tabular output. You can check out how to do this in the manual and with the command:

```
blastn -query metags/corn.fa -db nifh-ref.fa -out metags/corn.fa.x.nifh.blastnout.tsv -outfmt 6
```

You can save the BLAST output to a tabular output that can be easily parsed.

1.3.3 Automation exercise

You can save a lot of time if you learn how to automate things in the shell. Try the following command when in the data folder:

```
for x in metags/*fa; do blastn -query $x -db nifh-ref.fa -outfmt 6 -out $x.x.nifh.blastnout.tsv; done
```

What does it do?

1.3.4 Getting the best hit

BLAST will find every gene which "hits" a sequence read and a single read will have multiple hits. Often, you will only want to associate one annotation per read. I have written a very simple script to parse only the best hit for each read. Try from in the data directory:

```
python ../scripts/best-hit.py metags/corn.fa.x.nifh.blastnout.tsv > metags/corn.fa.x.nifh.blastnout.t
```

Compare the metags/corn.fa.x.nifh.blastnout.tsv and metags/corn.fa.x.nifh.blastnout.tsv.best.

1.3.5 For loop exercise - best hits

Using a for loop, save files for each blast output that only include the best hits. Make sure you end the file names with the word "best". Also, make sure you have only 3 files like this. You can delete ones from previous exercises if necessary.

1.3.6 More practice executing a program/script

So now we have a file where we have all the best hit reads associated with known nifH genes for each metagenome. However, we want to compare the number of genes per metagenome. I've written a script for this and you can try this from the data directory:

```
python ../scripts/count-up.py metags/*best
```

This creates a file in the data folder called "summary-count.tsv".

Take a look at this file.

1.3.7 Adding annotations

Finally, you might think to yourself that the NCBI accession numbers aren't that useful. But we can grab more details for these IDs, after all we have the sequence file with the associated gene description, nif-ref.fa. Try this script:

```
python ../scripts/import-ann.py nifh-ref.fa summary-count.tsv > summary-count.annotations.tsv
```

Take a look at the summary-count.annotations.tsv

1.3.8 Conclusion

So now you've executed at least 3 programs within this single tutorial. There is a lot more to learn about how to write your own scripts, but this is the first step towards understanding the value of being able to code. And actually, you've been coding along! executing for loops in shell. How much have you learned in one day? Hopefully its an incentive to keep learning!