Marine Genomics Week 2

The Carpentries, edited by Vanessa Guerra and Serena Caplins

4/6/2021

Week 2

Working With Files

Before we start, make sure to clone or update the github folder MarineGenomics in the user directory

```
$ cd /home/margeno/
$ git clone https://github.com/BayLab/MarineGenomics
```

- Questions:
  - How can I view and search file contents?
  - How can I create, copy and delete files and directories?
  - How can I control who has permission to modify a file?
  - How can I repeat recently used commands?
- Objectives:
  - View, search within, copy, move, and rename files. Create new directories.
  - Use wildcards (`*`) to perform operations on multiple files.
  - Make a file read only.
  - Use the `history` command to view and repeat recently used commands.
- Keypoints:
  - You can view file contents using `less`, `cat`, `head` or `tail`.
  - The commands `cp`, `mv`, and `mkdir` are useful for manipulating existing files and creating new directories.
  - You can view file permissions using `ls -l` and change permissions using `chmod`.
  - The `history` command and the up arrow on your keyboard can be used to repeat recently used commands.

Our data set: FASTQ files

Now that we know how to navigate around our directory structure, let's start working with our sequencing files. We did a sequencing experiment and have two results files, which are stored in our `untrimmed_fastq` directory.

Wildcards

Navigate to your `untrimmed_fastq` directory:

```
$ cd ./shell_data/untrimmed_fastq
```

We are interested in looking at the FASTQ files in this directory. We can list all files with the .fastq extension using the command:

```
$ ls *.fastq
```

```
SRR097977.fastq  SRR098026.fastq
```

The `*` character is a special type of character called a wildcard, which can be used to represent any number of any type of character. Thus, `*.fastq` matches every file that ends with `.fastq`.

This command:

```
$ ls *977.fastq
```

```
SRR097977.fastq
```

lists only the file that ends with `977.fastq`.

This command:

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

```
/usr/bin/gettext.sh  /usr/bin/rescan-scsi-bus.sh
```

Lists every file in `/usr/bin` that ends in the characters `.sh`. Note that the output displays **full** paths to files, since each result starts with `/`.

Exercise

Do each of the following tasks from your current directory using a single `ls` command for each:

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

Bonus: List all of the files in `/usr/bin` that contain the letter 'a' or the letter 'c'.

Hint: The bonus question requires a Unix wildcard that we haven't talked about yet. Try searching the internet for information about Unix wildcards to find what you need to solve the bonus problem.

Solution

1. `ls /usr/bin/c*`
2. `ls /usr/bin/*a*`
3. `ls /usr/bin/*o`
   Bonus: `ls /usr/bin/*[ac]*`

Exercise

`echo` is a built-in shell command that writes its arguments, like a line of text to standard output. The `echo` command can also be used with pattern matching characters, such as wildcard characters. Here we will use the `echo` command to see how the wildcard character is interpreted by the shell.

```
$ echo *.fastq
```

```
SRR097977.fastq SRR098026.fastq
```

The `*` is expanded to include any file that ends with `.fastq`. We can see that the output of `echo *.fastq` is the same as that of `ls *.fastq`.

What would the output look like if the wildcard could *not* be matched? Compare the outputs of `echo *.missing` and `ls *.missing`.

Solution

```
$ echo *.missing
```

```
*.missing
```

```
$ ls *.missing
```

```
ls: cannot access '*.missing': No such file or directory
```

Command History

If you want to repeat a command that you've run recently, you can access previous commands using the up arrow on your keyboard to go back to the most recent command. Likewise, the down arrow takes you forward in the command history.

A few more useful shortcuts:

- Ctrl+C will cancel the command you are writing, and give you a fresh prompt.
- Ctrl+R will do a reverse-search through your command history. This is very useful.
- Ctrl+L or the `clear` command will clear your screen.

You can also review your recent commands with the `history` command, by entering:

```
$ history
```

to see a numbered list of recent commands. You can reuse one of these commands directly by referring to the number of that command.

For example, if your history looked like this:

```
259  ls *
260  ls /usr/bin/*.sh
261  ls *R1*fastq
```

then you could repeat command #260 by entering:

```
$ !260
```

Type `!` (exclamation point) and then the number of the command from your history. You will be glad you learned this when you need to re-run very complicated commands. For more information on advanced usage of `history`, read section 9.3 of Bash manual.

Exercise

Find the line number in your history for the command that listed all the .sh files in `/usr/bin`. Rerun that command.

Solution

First type `history`. Then use `!` followed by the line number to rerun that command.

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?

One way to examine a file is to print out all of the contents using the program `cat`.

Enter the following command from within the `untrimmed_fastq` directory:

```
$ cat SRR098026.fastq
```

This will print out all of the contents of the `SRR098026.fastq` to the screen.

Exercise

1. Print out the contents of the `./shell_data/untrimmed_fastq/SRR097977.fastq` file. What is the last line of the file?
2. From your home directory, and without changing directories, use one short command to print the contents of all of the files in the `./shell_data/untrimmed_fastq` directory.

Solution

1. The last line of the file is `C:CCC::CCCCCCCC<8?6A:C28C<608'&&&,'$`.
2. `cat ./shell_data/untrimmed_fastq/*`

`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. `less` opens the file as read only, and lets you navigate through it. The navigation commands are identical to the `man` program.

Enter the following command:

```
$ less SRR097977.fastq
```

Some navigation commands in `less`:

key	action
Space	to go forward
b	to go backward
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. Use the "/" key to begin a search. Enter the word you would like to search for and press `enter`. The screen will jump to the next location where that word is found.

Shortcut: If you hit "/" then "enter", `less` will repeat the previous search. `less` searches from the current location and works its way forward. Scroll up a couple lines on your terminal to verify you are at the beginning of the file. Note, if you are at the end of the file and search for the sequence "CAA", `less` will not find it. You either need to go to the beginning of the file (by typing `g`) and search again using `/` or you can use `?` to search backwards in the same way you used `/` previously.

For instance, let's search forward for the sequence `TTTTT` in our file. You can see that we go right to that sequence, what it looks like, and where it is in the file. If you continue to type `/` and hit return, you will move forward to the next instance of this sequence motif. If you instead type `?` and hit return, you will search backwards and move up the file to previous examples of this motif.

Exercise

What are the next three nucleotides (characters) after the first instance of the sequence quoted above?

Solution

`CAC`

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 let you look at the beginning and end of a file, respectively.

```
$ head SRR098026.fastq
```

```
@SRR098026.1 HWUSI-EAS1599_1:2:1:0:968 length=35
NNNNNNNNNNNNNNNNCNNNNNNNNNNNNNNNNNN
+SRR098026.1 HWUSI-EAS1599_1:2:1:0:968 length=35
!!!!!!!!!!!!!!!!#!!!!!!!!!!!!!!!!!!
@SRR098026.2 HWUSI-EAS1599_1:2:1:0:312 length=35
NNNNNNNNNNNNNNNNANNNNNNNNNNNNNNNNNN
+SRR098026.2 HWUSI-EAS1599_1:2:1:0:312 length=35
!!!!!!!!!!!!!!!!#!!!!!!!!!!!!!!!!!!
@SRR098026.3 HWUSI-EAS1599_1:2:1:0:570 length=35
NNNNNNNNNNNNNNNNANNNNNNNNNNNNNNNNNN
```

```
$ tail SRR098026.fastq
```

```
+SRR098026.247 HWUSI-EAS1599_1:2:1:2:1311 length=35
#!##!##################!!!!!!!######
@SRR098026.248 HWUSI-EAS1599_1:2:1:2:118 length=35
GNTGNGGTCATCATACGCGCCCNNNNNNNGGCATG
+SRR098026.248 HWUSI-EAS1599_1:2:1:2:118 length=35
B!;?!A=5922:##########!!!!!!!######
@SRR098026.249 HWUSI-EAS1599_1:2:1:2:1057 length=35
CNCTNTATGCGTACGGCAGTGANNNNNNNGGAGAT
+SRR098026.249 HWUSI-EAS1599_1:2:1:2:1057 length=35
A!@B!BBB@ABAB#########!!!!!!!######
```

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

```
$ head -n 1 SRR098026.fastq
```

```
@SRR098026.1 HWUSI-EAS1599_1:2:1:0:968 length=35
```

```
$ tail -n 1 SRR098026.fastq
```

```
A!@B!BBB@ABAB#########!!!!!!!######
```

Details on the FASTQ format

Although it looks complicated (and it is), it's easy to understand the fastq format with a little decoding. Some rules about the format include...

Line	Description
1	Always begins with '@' and then information about the read
2	The actual DNA sequence
3	Always begins with a '+' and sometimes the same info in line 1
4	Has a string of characters which represent the quality scores; must have same number of characters as line 2

We can view the first complete read in one of the files in our dataset by using `head` to look at the first four lines.

```
$ head -n 4 SRR098026.fastq
```

```
@SRR098026.1 HWUSI-EAS1599_1:2:1:0:968 length=35
NNNNNNNNNNNNNNNNCNNNNNNNNNNNNNNNNNN
+SRR098026.1 HWUSI-EAS1599_1:2:1:0:968 length=35
!!!!!!!!!!!!!!!#!!!!!!!!!!!!!!!!!!
```

All but one of the nucleotides in this read are unknown (`N`). This is a pretty bad read!

Line 4 shows the quality for each nucleotide in the read. Quality is interpreted as the probability of an incorrect base call (e.g. 1 in 10) or, equivalently, the base call accuracy (e.g. 90%). To make it possible to

line up each individual nucleotide with its quality score, the numerical score is converted into a code where each individual character represents the numerical quality score for an individual nucleotide. For example, in the line above, the quality score line is:

```
!!!!!!!!!!!!!!!!#!!!!!!!!!!!!!!!!!!!
```

The `#` character and each of the `!` characters represent the encoded quality for an individual nucleotide. The numerical value assigned to each of these characters depends on the sequencing platform that generated the reads. The sequencing machine used to generate our data uses the standard Sanger quality PHRED score encoding, Illumina version 1.8 onwards. Each character is assigned a quality score between 0 and 42 as shown in the chart below.

```
Quality encoding: !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJK
                  |         |         |         |         |
Quality score:    0........10........20........30........40..
```

Each quality score represents the probability that the corresponding nucleotide call is incorrect. This quality score is logarithmically based, so a quality score of 10 reflects a base call accuracy of 90%, but a quality score of 20 reflects a base call accuracy of 99%. These probability values are the results from the base calling algorithm and dependent on how much signal was captured for the base incorporation.

Looking back at our read:

```
@SRR098026.1 HWUSI-EAS1599_1:2:1:0:968 length=35
NNNNNNNNNNNNNNNNCNNNNNNNNNNNNNNNNNN
+SRR098026.1 HWUSI-EAS1599_1:2:1:0:968 length=35
!!!!!!!!!!!!!!!!#!!!!!!!!!!!!!!!!!!!
```

we can now see that the quality of each of the `N`s is 0 and the quality of the only nucleotide call (`C`) is also very poor (`#` = a quality score of 2). This is indeed a very bad read.

Creating, moving, copying, and removing

Now we can move around in the file structure, look at files, and search files. But what if we want to copy files or move them around or get rid of them? Most of the time, you can do these sorts of file manipulations without the command line, but there will be some cases (like when you're working with a remote computer like we are for this lesson) where it will be impossible. You'll also find that you may be working with hundreds of files and want to do similar manipulations to all of those files. In cases like this, it's much faster to do these operations at the command line.

Copying Files

When working with computational data, it's important to keep a safe copy of that data that can't be accidentally overwritten or deleted. For this lesson, our raw data is our FASTQ files. We don't want to accidentally change the original files, so we'll make a copy of them and change the file permissions so that we can read from, but not write to, the files.

First, let's make a copy of one of our FASTQ files using the `cp` command.

Navigate to the `shell_data/untrimmed_fastq` directory and enter:

```
$ cp SRR098026.fastq SRR098026-copy.fastq
$ ls -F
```

```
SRR097977.fastq  SRR098026-copy.fastq  SRR098026.fastq
```

We now have two copies of the `SRR098026.fastq` file, one of them named `SRR098026-copy.fastq`. We'll move this file to a new directory called `backup` where we'll store our backup data files.

Creating Directories

The `mkdir` command is used to make a directory. Enter `mkdir` followed by a space, then the directory name you want to create:

```
$ mkdir backup
```

Moving / Renaming

We can now move our backup file to this directory. We can move files around using the command `mv`:

```
$ mv SRR098026-copy.fastq backup
$ ls backup
```

```
SRR098026-copy.fastq
```

The `mv` command is also how you rename files. Let's rename this file to make it clear that this is a backup:

```
$ cd backup
$ mv SRR098026-copy.fastq SRR098026-backup.fastq
$ ls
```

```
SRR098026-backup.fastq
```

File Permissions

We've now made a backup copy of our file, but just because we have two copies, it doesn't make us safe. We can still accidentally delete or overwrite both copies. To make sure we can't accidentally mess up this backup file, we're going to change the permissions on the file so that we're only allowed to read (i.e. view) the file, not write to it (i.e. make new changes).

View the current permissions on a file using the `-l` (long) flag for the `ls` command:

```
$ ls -l
```

```
-rw-rw-r-- 1 margeno margeno 43K Apr 6 12:15 SRR098026-backup.fastq
```

The first part of the output for the `-l` flag gives you information about the file's current permissions. There are ten slots in the permissions list. The first character in this list is related to file type, not permissions, so we'll ignore it for now. The next three characters relate to the permissions that the file owner has, the next three relate to the permissions for group members, and the final three characters specify what other users

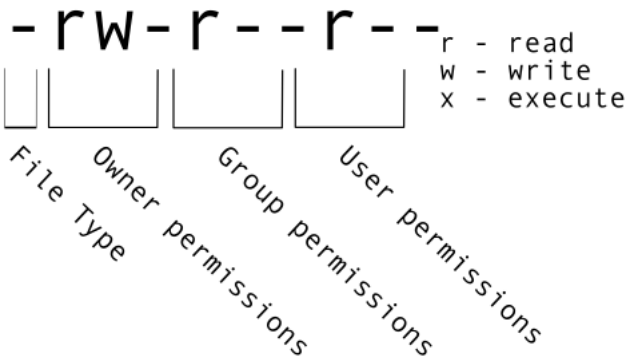

Figure 1: Permissions breakdown

outside of your group can do with the file. We're going to concentrate on the three positions that deal with your permissions (as the file owner).

Here the three positions that relate to the file owner are `rw-`. The `r` means that you have permission to read the file, the `w` indicates that you have permission to write to (i.e. make changes to) the file, and the third position is a `-`, indicating that you don't have permission to carry out the ability encoded by that space (this is the space where `x` or executable ability is stored, we'll talk more about this in a later lesson).

Our goal for now is to change permissions on this file so that you no longer have `w` or write permissions. We can do this using the `chmod` (change mode) command and subtracting (`-`) the write permission `-w`.

```
$ chmod -w SRR098026-backup.fastq
$ ls -l
```

```
-r--r--r-- 1 dcuser dcuser 43332 Nov 15 23:02 SRR098026-backup.fastq
```

Chmod can also change the permission to only the user (u), group (g), and/or other (o). Let's add reading (r), writing (x), and execute (x) permissions to user group.

```
$ chmod u=rwx SRR098026-backup.fastq
$ ls -l
```

Removing

To prove to ourselves that you no longer have the ability to modify this file, try deleting it with the `rm` command:

```
$ rm SRR098026-backup.fastq
```

You'll be asked if you want to override your file permissions:

```
rm: remove write-protected regular file ‘SRR098026-backup.fastq’?
```

You should enter `n` for no. If you enter `n` (for no), the file will not be deleted. If you enter `y`, you will delete the file. This gives us an extra measure of security, as there is one more step between us and deleting our data files.

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

By default, `rm` will not delete directories. You can tell `rm` to delete a directory using the `-r` (recursive) option. Let's delete the backup directory we just made.

Enter the following command:

```
$ cd ..
$ rm -r backup
```

This will delete not only the directory, but all files within the directory. If you have write-protected files in the directory, you will be asked whether you want to override your permission settings.

Exercise

Starting in the `shell_data/untrimmed_fastq/` directory, do the following: 1. Make sure that you have deleted your backup directory and all files it contains.
2. Create a backup of each of your FASTQ files using `cp`. (Note: You'll need to do this individually for each of the two FASTQ files. We haven't learned yet how to do this with a wildcard.)
3. Use a wildcard to move all of your backup files to a new backup directory.
4. Change the permissions on all of your backup files to be write-protected.

Solution

1. `rm -r backup`

2. `cp SRR098026.fastq SRR098026-backup.fastq` and `cp SRR097977.fastq SRR097977-backup.fastq`

3. `mkdir backup` and `mv *-backup.fastq backup`
4. `chmod -w backup/*-backup.fastq`
   It's always a good idea to check your work with `ls -l backup`. You should see something like:

```
-r--r--r-- 1 dcuser dcuser 47552 Nov 15 23:06 SRR097977-backup.fastq
-r--r--r-- 1 dcuser dcuser 43332 Nov 15 23:06 SRR098026-backup.fastq
```

Redirections

- Questions: +How can I search within files? +How can I combine existing commands to do new things?
- Objectives: +Employ the `grep` command to search for information within files. +Print the results of a command to a file. +Construct command pipelines with two or more stages. +Use `for` loops to run the same command for several input files.
- Keypoints: +`grep` is a powerful search tool with many options for customization. +`>`, `>>`, and `|` are different ways of redirecting output. +`command > file` redirects a command's output to a file. +`command >> file` redirects a command's output to a file without overwriting the existing contents of the file. +`command_1 | command_2` redirects the output of the first command as input to the second command. +`for` loops are used for iteration. +`basename` gets rid of repetitive parts of names.

Searching files

We discussed in a previous episode 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 files for lines matching a specific set of characters (sometimes called a string) or a particular pattern (which can be specified using something called regular expressions). We're not going to work with regular expressions in this lesson, and are instead going to specify the strings we are searching for. Let's give it a try!

Nucleotide abbreviations

The four nucleotides that appear in DNA are abbreviated `A`, `C`, `T` and `G`. Unknown nucleotides are represented with the letter `N`. An `N` appearing in a sequencing file represents a position where the sequencing machine was not able to confidently determine the nucleotide in that position. You can think of an `N` as being aNy nucleotide at that position in the DNA sequence.

We'll search for strings inside of our fastq files. Let's first make sure we are in the correct directory:

```
$ cd ./shell_data/untrimmed_fastq
```

Suppose we want to see how many reads in our file have really bad segments containing 10 consecutive unknown nucleotides (Ns).

Determining quality

In this lesson, we're going to be manually searching for strings of `N`s within our sequence results to illustrate some principles of file searching. It can be really useful to do this type of searching to get a feel for the quality of your sequencing results, however, in your research you will most likely use a bioinformatics tool that has a built-in program for filtering out low-quality reads. You'll learn how to use one such tool in a later lesson.

Let's search for the string NNNNNNNNNN in the SRR098026 file:

```
$ grep NNNNNNNNNN SRR098026.fastq
```

This command returns a lot of output to the terminal. Every single line in the SRR098026 file that contains at least 10 consecutive Ns is printed to the terminal, regardless of how long or short the file is. We may be interested not only in the actual sequence which contains this string, but in the name (or identifier) of that sequence. We discussed in a previous lesson that the identifier line immediately precedes the nucleotide sequence for each read in a FASTQ file. We may also want to inspect the quality scores associated with each of these reads. To get all of this information, we will return the line immediately before each match and the two lines immediately after each match.

We can use the `-B` argument for grep to return a specific number of lines before each match. The `-A` argument returns a specific number of lines after each matching line. Here we want the line *before* and the two lines *after* each matching line, so we add `-B1 -A2` to our grep command:

```
$ grep -B1 -A2 NNNNNNNNNN SRR098026.fastq
```

One of the sets of lines returned by this command is:

```
@SRR098026.177 HWUSI-EAS1599_1:2:1:1:2025 length=35
CNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
+SRR098026.177 HWUSI-EAS1599_1:2:1:1:2025 length=35
#!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Exercise

1. Search for the sequence `GNATNACCACTTCC` in the `SRR098026.fastq` file. Have your search return all matching lines and the name (or identifier) for each sequence that contains a match.
2. Search for the sequence `AAGTT` in both FASTQ files. Have your search return all matching lines and the name (or identifier) for each sequence that contains a match.

Solution

1. `grep -B1 GNATNACCACTTCC SRR098026.fastq`

   ```
   @SRR098026.245 HWUSI-EAS1599_1:2:1:2:801 length=35
   GNATNACCACTTCCAGTGCTGANNNNNNNGGGATG
   ```

2. `grep -B1 AAGTT *.fastq`

   ```
   SRR097977.fastq-@SRR097977.11 209DTAAXX_Lenski2_1_7:8:3:247:351 length=36
   SRR097977.fastq:GATTGCTTTAATGAAAAAGTCATATAAGTTGCCATG
   --
   SRR097977.fastq-@SRR097977.67 209DTAAXX_Lenski2_1_7:8:3:544:566 length=36
   SRR097977.fastq:TTGTCCACGCTTTTCTATGTAAAGTTTATTTGCTTT
   --
   SRR097977.fastq-@SRR097977.68 209DTAAXX_Lenski2_1_7:8:3:724:110 length=36
   SRR097977.fastq:TGAAGCCTGCTTTTTTATACTAAGTTTGCATTATAA
   --
   SRR097977.fastq-@SRR097977.80 209DTAAXX_Lenski2_1_7:8:3:258:281 length=36
   SRR097977.fastq:GTGGCGCTGCTGCATAAGTTGGGTTATCAGGTCGTT
   --
   SRR097977.fastq-@SRR097977.92 209DTAAXX_Lenski2_1_7:8:3:353:318 length=36
   SRR097977.fastq:GGCAAAATGGTCCTCCAGCCAGGCCAGAAGCAAGTT
   --
   SRR097977.fastq-@SRR097977.139 209DTAAXX_Lenski2_1_7:8:3:703:655 length=36
   SRR097977.fastq:TTTATTTGTAAAGTTTTGTTGAAATAAGGGTTGTAA
   --
   SRR097977.fastq-@SRR097977.238 209DTAAXX_Lenski2_1_7:8:3:592:919 length=36
   SRR097977.fastq:TTCTTACCATCCTGAAGTTTTTTCATCTTCCCTGAT
   --
   SRR098026.fastq-@SRR098026.158 HWUSI-EAS1599_1:2:1:1:1505 length=35
   SRR098026.fastq:GNNNNNNNNCAAAGTTGATCNNNNNNNNNTGTGCG
   ```

Redirecting output

`grep` allowed us to identify sequences in our FASTQ files that match a particular pattern. All of these sequences were printed to our terminal screen, but in order to work with these sequences and perform other operations on them, we will need to capture that output in some way.

We can do this with something called "redirection". The idea is that we are taking what would ordinarily be printed to the terminal screen and redirecting it to another location. In our case, we want to print this information to a file so that we can look at it later and use other commands to analyze this data.

The command for redirecting output to a file is `>`.

Let's try out this command and copy all the records (including all four lines of each record) in our FASTQ files that contain 'NNNNNNNNNN' to another file called `bad_reads.txt`.

```
$ grep -B1 -A2 NNNNNNNNNN SRR098026.fastq > bad_reads.txt
```

File extensions

You might be confused about why we're naming our output file with a `.txt` extension. After all, it will be holding FASTQ formatted data that we're extracting from our FASTQ files. Won't it also be a FASTQ file? The answer is, yes - it will be a FASTQ file and it would make sense to name it with a `.fastq` extension. However, using a `.fastq` extension will lead us to problems when we move to using wildcards later in this episode. We'll point out where this becomes important. For now, it's good that you're thinking about file extensions!

The prompt should sit there a little bit, and then it should look like nothing happened. But type `ls`. You should see a new file called `bad_reads.txt`.

We can check the number of lines in our new file using a command called `wc`. `wc` stands for **word count**. This command counts the number of words, lines, and characters in a file. The FASTQ file may change over time, so given the potential for updates, make sure your file matches your instructor's output.

As of Sept. 2020, wc gives the following output:

```
$ wc bad_reads.txt
```

```
  802    1338   24012 bad_reads.txt
```

This will tell us the number of lines, words and characters in the file. If we want only the number of lines, we can use the `-l` flag for `lines`.

```
$ wc -l bad_reads.txt
```

```
802 bad_reads.txt
```

Exercise

How many sequences are there in `SRR098026.fastq`? Remember that every sequence is formed by four lines.

Solution

```
$ wc -l SRR098026.fastq
```

```
996
```

Now you can divide this number by four to get the number of sequences in your fastq file

Exercise

How many sequences in `SRR098026.fastq` contain at least 3 consecutive Ns?

Solution

```
$ grep NNN SRR098026.fastq > bad_reads.txt
$ wc -l bad_reads.txt
```

```
249
```

We might want to search multiple FASTQ files for sequences that match our search pattern. However, we need to be careful, because each time we use the `>` command to redirect output to a file, the new output will replace the output that was already present in the file. This is called "overwriting" and, just like you don't want to overwrite your video recording of your kid's first birthday party, you also want to avoid overwriting your data files.

```
$ grep -B1 -A2 NNNNNNNNNN SRR098026.fastq > bad_reads.txt
$ wc -l bad_reads.txt
```

```
802 bad_reads.txt
```

```
$ grep -B1 -A2 NNNNNNNNNN SRR097977.fastq > bad_reads.txt
$ wc -l bad_reads.txt
```

```
0 bad_reads.txt
```

Here, the output of our second call to `wc` shows that we no longer have any lines in our `bad_reads.txt` file. This is because the second file we searched (`SRR097977.fastq`) does not contain any lines that match our search sequence. So our file was overwritten and is now empty.

We can avoid overwriting our files by using the command `>>`. `>>` is known as the "append redirect" and will append new output to the end of a file, rather than overwriting it.

```
$ grep -B1 -A2 NNNNNNNNNN SRR098026.fastq > bad_reads.txt
$ wc -l bad_reads.txt
```

```
802 bad_reads.txt
```

```
$ grep -B1 -A2 NNNNNNNNNN SRR097977.fastq >> bad_reads.txt
$ wc -l bad_reads.txt
```

```
802 bad_reads.txt
```

The output of our second call to `wc` shows that we have not overwritten our original data.

We can also do this with a single line of code by using a wildcard:

```
$ grep -B1 -A2 NNNNNNNNNN *.fastq > bad_reads.txt
$ wc -l bad_reads.txt
```

```
802 bad_reads.txt
```

File extensions - part 2

This is where we would have trouble if we were naming our output file with a `.fastq` extension. If we already had a file called `bad_reads.fastq` (from our previous `grep` practice) and then ran the command above using a `.fastq` extension instead of a `.txt` extension, `grep` would give us a warning.

```
grep -B1 -A2 NNNNNNNNNN *.fastq > bad_reads.fastq
```

```
grep: input file 'bad_reads.fastq' is also the output
```

`grep` is letting you know that the output file `bad_reads.fastq` is also included in your `grep` call because it matches the `*.fastq` pattern. Be careful with this as it can lead to some unintended results.

Since we might have multiple different criteria we want to search for, creating a new output file each time has the potential to clutter up our workspace. We also thus far haven't been interested in the actual contents of those files, only in the number of reads that we've found. We created the files to store the reads and then counted the lines in the file to see how many reads matched our criteria. There's a way to do this, however, that doesn't require us to create these intermediate files - the pipe command (`|`).

What `|` does is take the output that is scrolling by on the terminal and uses that output as input to another command. When our output was scrolling by, we might have wished we could slow it down and look at it, like we can with `less`. Well it turns out that we can! We can redirect our output from our `grep` call through the `less` command.

```
$ grep -B1 -A2 NNNNNNNNNN SRR098026.fastq | less
```

We can now see the output from our `grep` call within the `less` interface. We can use the up and down arrows to scroll through the output and use `q` to exit `less`.

If we don't want to create a file before counting lines of output from our `grep` search, we could directly pipe the output of the grep search to the command `wc -l`. This can be helpful for investigating your output if you are not sure you would like to save it to a file.

```
$ grep -B1 -A2 NNNNNNNNNN SRR098026.fastq | wc -l
```

Because we asked `grep` for all four lines of each FASTQ record, we need to divide the output by four to get the number of sequences that match our search pattern. Since 802 / 4 = 200.5 and we are expecting an integer number of records, there is something added or missing in `bad_reads.txt`. If we explore `bad_reads.txt` using `less`, we might be able to notice what is causing the uneven number of lines. Luckily, this issue happens by the end of the file so we can also spot it with `tail`.

```
$ grep -B1 -A2 NNNNNNNNNN SRR098026.fastq > bad_reads.txt
$ tail bad_reads.txt
```

```
@SRR098026.133 HWUSI-EAS1599_1:2:1:0:1978 length=35
ANNNNNNNNNTTCAGCGACTNNNNNNNNNNGTNGN
+SRR098026.133 HWUSI-EAS1599_1:2:1:0:1978 length=35
#!!!!!!!!!##########!!!!!!!!!!##!#!
--
--
@SRR098026.177 HWUSI-EAS1599_1:2:1:1:2025 length=35
```

```
CNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
+SRR098026.177 HWUSI-EAS1599_1:2:1:1:2025 length=35
#!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

The fifth and six lines in the output display "–" which is the default action for `grep` to separate groups of lines matching the pattern, and indicate groups of lines which did not match the pattern so are not displayed. To fix this issue, we can redirect the output of grep to a second instance of `grep` as follows.

```
$ grep -B1 -A2 NNNNNNNNNN SRR098026.fastq | grep -v '^--' > bad_reads.fastq
tail bad_reads.fastq
```

```
+SRR098026.132 HWUSI-EAS1599_1:2:1:0:320 length=35
#!!!!!!!!!##########!!!!!!!!!!##!#!
@SRR098026.133 HWUSI-EAS1599_1:2:1:0:1978 length=35
ANNNNNNNNNTTCAGCGACTNNNNNNNNNNGTNGN
+SRR098026.133 HWUSI-EAS1599_1:2:1:0:1978 length=35
#!!!!!!!!!##########!!!!!!!!!!##!#!
@SRR098026.177 HWUSI-EAS1599_1:2:1:1:2025 length=35
CNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
+SRR098026.177 HWUSI-EAS1599_1:2:1:1:2025 length=35
#!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

The `-v` option in the second `grep` search stands for `--invert-match` meaning `grep` will now only display the lines which do not match the searched pattern, in this case `'^--'`. The caret (^) is an **anchoring** character matching the beginning of the line, and the pattern has to be enclose by single quotes so `grep` does not interpret the pattern as an extended option (starting with –).

Custom `grep` control

Use `man grep` to read more about other options to customize the output of `grep` including extended options, anchoring characters, and much more.

Redirecting output is often not intuitive, and can take some time to get used to. Once you're comfortable with redirection, however, you'll be able to combine any number of commands to do all sorts of exciting things with your data!

None of the command line programs we've been learning do anything all that impressive on their own, but when you start chaining them together, you can do some really powerful things very efficiently.

File manipulation and more practices with pipes

To practice a bit more with the tools we've added to our tool kit so far and learn a few extra ones you can follow this extra lesson which uses the SRA metadata file.

Writing for loops

Loops are key to productivity improvements through automation as they allow us to execute commands repeatedly. Similar to wildcards and tab completion, using loops also reduces the amount of typing (and typing mistakes). Loops are helpful when performing operations on groups of sequencing files, such as unzipping or trimming multiple files. We will use loops for these purposes in subsequent analyses, but will cover the basics of them for now.

When the shell sees the keyword `for`, it knows to repeat a command (or group of commands) once for each item in a list. Each time the loop runs (called an iteration), an item in the list is assigned in sequence to the **variable**, and the commands inside the loop are executed, before moving on to the next item in the list. Inside the loop, we call for the variable's value by putting `$` in front of it. The `$` tells the shell interpreter to treat the **variable** as a variable name and substitute its value in its place, rather than treat it as text or an external command. In shell programming, this is usually called "expanding" the variable.

Sometimes, we want to expand a variable without any whitespace to its right. Suppose we have a variable named `foo` that contains the text `abc`, and would like to expand `foo` to create the text `abcEFG`.

```
$ foo=abc
$ echo foo is $foo
foo is abc
$ echo foo is $fooEFG      # doesn't work
foo is
```

The interpreter is trying to expand a variable named `fooEFG`, which (probably) doesn't exist. We can avoid this problem by enclosing the variable name in braces (`{` and `}`, sometimes called "squiggle braces"). `bash` treats the `#` character as a comment character. Any text on a line after a `#` is ignored by bash when evaluating the text as code.

```
$ foo=abc
$ echo foo is $foo
foo is abc
$ echo foo is ${foo}EFG      # now it works!
foo is abcEFG
```

Let's write a for loop to show us the first two lines of the fastq files we downloaded earlier. You will notice the shell prompt changes from `$` to `>` and back again as we were typing in our loop. The second prompt, `>`, is different to remind us that we haven't finished typing a complete command yet. A semicolon, `;`, can be used to separate two commands written on a single line.

```
$ cd ../untrimmed_fastq/
```

```
$ for filename in *.fastq
> do
> head -n 2 ${filename}
> done
```

The for loop begins with the formula `for <variable> in <group to iterate over>`. In this case, the word `filename` is designated as the variable to be used over each iteration. In our case `SRR097977.fastq` and `SRR098026.fastq` will be substituted for `filename` because they fit the pattern of ending with .fastq in the directory we've specified. The next line of the for loop is `do`. The next line is the code that we want to execute. We are telling the loop to print the first two lines of each variable we iterate over. Finally, the word `done` ends the loop.

After executing the loop, you should see the first two lines of both fastq files printed to the terminal. Let's create a loop that will save this information to a file.

```
$ for filename in *.fastq
> do
> head -n 2 ${filename} >> seq_info.txt
> done
```

When writing a loop, you will not be able to return to previous lines once you have pressed Enter. Remember that we can cancel the current command using

- Ctrl+C

If you notice a mistake that is going to prevent your loop for executing correctly.

Note that we are using `>>` to append the text to our `seq_info.txt` file. If we used `>`, the `seq_info.txt` file would be rewritten every time the loop iterates, so it would only have text from the last variable used. Instead, `>>` adds to the end of the file.

Using Basename in for loops

Basename is a function in UNIX that is helpful for removing a uniform part of a name from a list of files. In this case, we will use basename to remove the `.fastq` extension from the files that we've been working with.

```
$ basename SRR097977.fastq .fastq
```

We see that this returns just the SRR accession, and no longer has the .fastq file extension on it.

```
SRR097977
```

If we try the same thing but use `.fasta` as the file extension instead, nothing happens. This is because basename only works when it exactly matches a string in the file.

```
$ basename SRR097977.fastq .fasta
```

```
SRR097977.fastq
```

Basename is really powerful when used in a for loop. It allows to access just the file prefix, which you can use to name things. Let's try this.

Inside our for loop, we create a new name variable. We call the basename function inside the parenthesis, then give our variable name from the for loop, in this case `${filename}`, and finally state that `.fastq` should be removed from the file name. It's important to note that we're not changing the actual files, we're creating a new variable called name. The line > echo $name will print to the terminal the variable name each time the for loop runs. Because we are iterating over two files, we expect to see two lines of output.

```
$ for filename in *.fastq
> do
> name=$(basename ${filename} .fastq)
> echo ${name}
> done
```

Exercise

Print the file prefix of all of the `.txt` files in our current directory.

Solution

Solution

```
$ for filename in *.txt
> do
> name=$(basename ${filename} .txt)
> echo ${name}
> done
```

Solution

One way this is really useful is to move files. Let's rename all of our .txt files using `mv` so that they have the years on them, which will document when we created them.

```
$ for filename in *.txt
> do
> name=$(basename ${filename} .txt)
> mv ${filename}  ${name}_2019.txt
> done
```

Exercise

Remove `_2019` from all of the `.txt` files.

Solution

```
$ for filename in *_2019.txt
> do
> name=$(basename ${filename} _2019.txt)
> mv ${filename} ${name}.txt
> done
```

Writing Scripts and Working with Data

- Questions: +How can we automate a commonly used set of commands?
- Objectives: +Use the `nano` text editor to modify text files. +Write a basic shell script. +Use the `bash` command to execute a shell script. +Use `chmod` to make a script an executable program.
- Keypoints: +Scripts are a collection of commands executed together. +Transferring information to and from virtual and local computers.

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? We're not going to type in a FASTA file, but we'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 add text to files, we're going to use a text editor called Nano. We're going to create a file to take notes about what we've been doing with the data files in `./shell_data/untrimmed_fastq`.

This is good practice when working in bioinformatics. We can create a file called `README.txt` that describes the data files in the directory or documents how the files in that directory were generated. As the name suggests, it's a file that we or others should read to understand the information in that directory.

Let's change our working directory to `./shell_data/untrimmed_fastq` using `cd`, then run `nano` to create a file called `README.txt`:

```
$ cd ./shell_data/untrimmed_fastq
$ nano README.txt
```

You should see something like this:

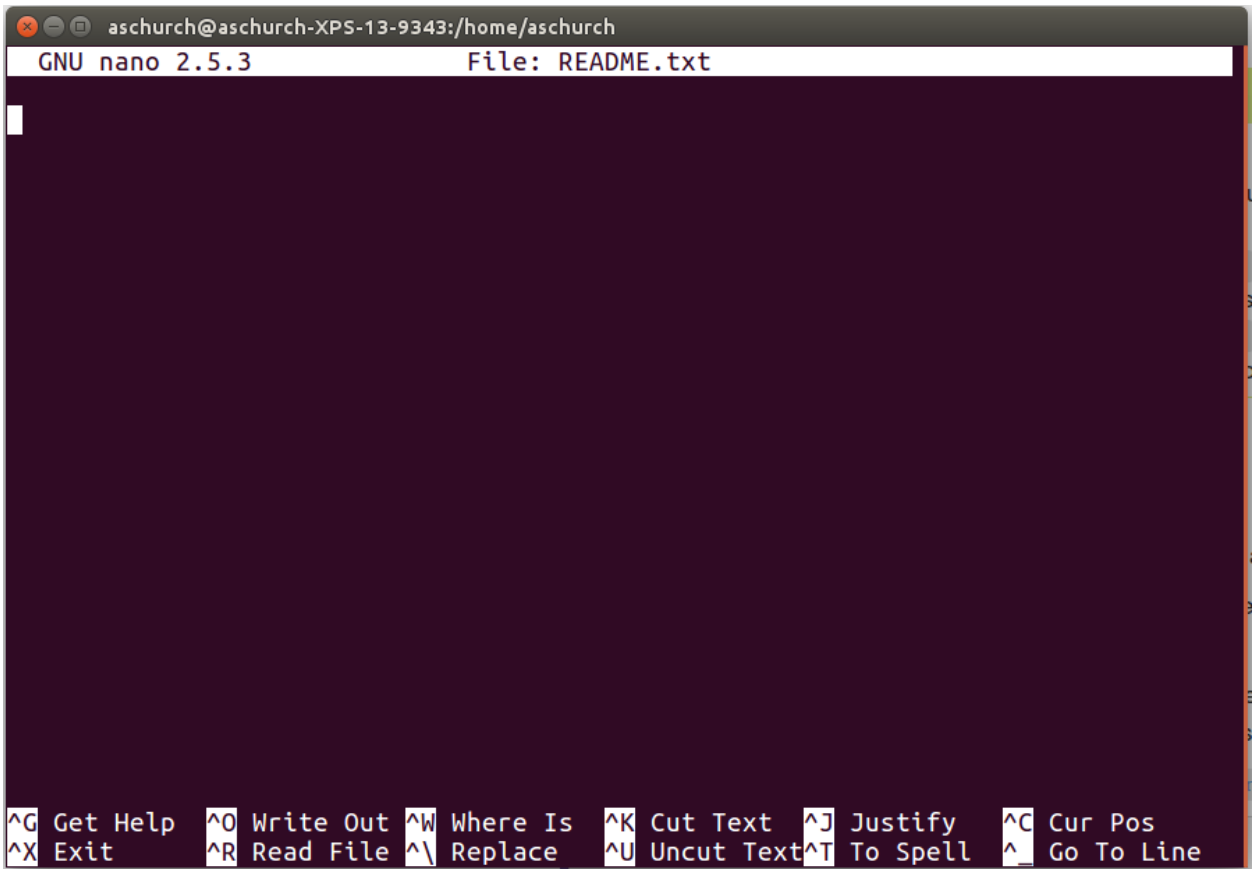

Figure 2: nano201711.png

The text at the bottom of the screen shows the keyboard shortcuts for performing various tasks in `nano`. We will talk more about how to interpret this information soon.

Which Editor?

When we say, "`nano` is a text editor," we really do mean "text": it can only work with plain character data, not tables, images, or any other human-friendly media. We use it in examples because it is one of the least complex text editors. However, because of this trait, it may not be powerful enough or flexible enough for the work you need to do after this workshop. On Unix systems (such as Linux and Mac OS X), many programmers use Emacs or Vim (both of which require more time to learn), or a graphical editor such as Gedit. On Windows, you may wish to use Notepad++. Windows also has a built-in editor called `notepad` that can be run from the command line in the same way as `nano` for the purposes of this lesson.

No matter what editor you use, you will need to know where it searches for and saves files. If you start it from the shell, it will (probably) use your current working directory as its default location. If you use your computer's start menu, it may want to save files in your desktop or documents directory instead. You can change this by navigating to another directory the first time you "Save As..."

Let's type in a few lines of text. Describe what the files in this directory are or what you've been doing with them. Once we're happy with our text, we can press Ctrl-O (press the Ctrl or Control key and, while holding it down, press the O key) to write our data to disk. You'll be asked what file we want to save this to: press Return to accept the suggested default of `README.txt`.

Once our file is saved, we can use Ctrl-X to quit the editor and return to the shell.

Control, Ctrl, or ^ Key

The Control key is also called the "Ctrl" key. There are various ways in which using the Control key may be described. For example, you may see an instruction to press the Ctrl key and, while holding it down, press the X key, described as any of:

- `Control-X`
- `Control+X`
- `Ctrl-X`
- `Ctrl+X`
- `^X`
- `C-x`

In `nano`, along the bottom of the screen you'll see `^G Get Help ^O WriteOut`. This means that you can use Ctrl-G to get help and Ctrl-O to save your file.

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 with `nano`.

Exercise

Open `README.txt` and add the date to the top of the file and save the file.

Solution

Use `nano README.txt` to open the file.
Add today's date and then use Ctrl-X followed by `y` and Enter to save.

Writing scripts

A really powerful thing about the command line is that you can write scripts. Scripts let you save commands to run them and also lets you put multiple commands together. Though writing scripts may require an additional time investment initially, this can save you time as you run them repeatedly. Scripts can also address the challenge of reproducibility: if you need to repeat an analysis, you retain a record of your command history within the script.

One thing we will commonly want to do with sequencing results is pull out bad reads and write them to a file to see if we can figure out what's going on with them. We're going to look for reads with long sequences of N's like we did before, but now we're going to write a script, so we can run it each time we get new sequences, rather than type the code in by hand each time.

We're going to create a new file to put this command in. We'll call it `bad-reads-script.sh`. The `sh` isn't required, but using that extension tells us that it's a shell script.

```
$ nano bad-reads-script.sh
```

Bad reads have a lot of N's, so we're going to look for `NNNNNNNNNN` with `grep`. We want the whole FASTQ record, so we're also going to get the one line above the sequence and the two lines below. We also want to look in all the files that end with `.fastq`, so we're going to use the `*` wildcard.

```
grep -B1 -A2 -h NNNNNNNNNN *.fastq | grep -v '^--' > scripted_bad_reads.txt
```

Custom `grep` control

We introduced the `-v` option in the previous episode, now we are using `-h` to "Suppress the prefixing of file names on output" according to the documentation shown by `man grep`.

Type your `grep` command into the file and save it as before. Be careful that you did not add the `$` at the beginning of the line.

Now comes the neat part. We can run this script. Type:

```
$ bash bad-reads-script.sh
```

It will look like nothing happened, but now if you look at `scripted_bad_reads.txt`, you can see that there are now reads in the file.

Exercise

We want the script to tell us when it's done.
1. Open `bad-reads-script.sh` and add the line `echo "Script finished!"` after the `grep` command and save the file.
2. Run the updated script.

Solution

```
$ bash bad-reads-script.sh
Script finished!
```

Making the script into a program

We had to type `bash` because we needed to tell the computer what program to use to run this script. Instead, we can turn this script into its own program. We need to tell it that it's a program by making it executable. We can do this by changing the file permissions. We talked about permissions in an earlier episode.

First, let's look at the current permissions.

```
$ ls -l bad-reads-script.sh
```

```
-rw-rw-r-- 1 dcuser dcuser 0 Oct 25 21:46 bad-reads-script.sh
```

We see that it says `-rw-r--r--`. This shows that the file can be read by any user and written to by the file owner (you). We want to change these permissions so that the file can be executed as a program. We use the command `chmod` like we did earlier when we removed write permissions. Here we are adding (`+`) executable permissions (`+x`).

```
$ chmod +x bad-reads-script.sh
```

Now let's look at the permissions again.

```
$ ls -l bad-reads-script.sh
```

```
-rwxrwxr-x 1 dcuser dcuser 0 Oct 25 21:46 bad-reads-script.sh
```

Now we see that it says `-rwxr-xr-x`. The `x`'s that are there now tell us we can run it as a program. So, let's try it! We'll need to put `./` at the beginning so the computer knows to look here in this directory for the program.

```
$ ./bad-reads-script.sh
```

The script should run the same way as before, but now we've created our very own computer program!

You will learn more about writing scripts in a later lesson.

Moving and Downloading Data

So far, we've worked with data that is pre-loaded on the instance in the cloud. Usually, however, most analyses begin with moving data onto the instance. Below we'll show you some commands to download data onto your instance, or to move data between your computer and the cloud.

Getting data from the cloud

There are two programs that will download data from a remote server to your local (or remote) machine: `wget` and `curl`. They were designed to do slightly different tasks by default, so you'll need to give the programs somewhat different options to get the same behaviour, but they are mostly interchangeable.

- `wget` is short for "world wide web get", and it's basic function is to *download* web pages or data at a web address.
- `cURL` is a pun, it is supposed to be read as "see URL", so its basic function is to *display* webpages or data at a web address.

Which one you need to use mostly depends on your operating system, as most computers will only have one or the other installed by default.

Let's say you want to download some data from Ensembl. We're going to download a very small tab-delimited file that just tells us what data is available on the Ensembl bacteria server. Before we can start our download, we need to know whether we're using `curl` or `wget`.

To see which program you have, type:

```
$ which curl
$ which wget
```

`which` is a BASH program that looks through everything you have installed, and tells you what folder it is installed to. If it can't find the program you asked for, it returns nothing, i.e. gives you no results.

On Mac OSX, you'll likely get the following output:

```
$ which curl
```

```
/usr/bin/curl
```

```
$ which wget
```

```
$
```

This output means that you have `curl` installed, but not `wget`.

Once you know whether you have `curl` or `wget`, use one of the following commands to download the file:

```
$ cd
$ wget ftp://ftp.ensemblgenomes.org/pub/release-37/bacteria/species_EnsemblBacteria.txt
```

or

```
$ cd
$ curl -O ftp://ftp.ensemblgenomes.org/pub/release-37/bacteria/species_EnsemblBacteria.txt
```

Since we wanted to *download* the file rather than just view it, we used `wget` without any modifiers. With `curl` however, we had to use the -O flag, which simultaneously tells `curl` to download the page instead of showing it to us **and** specifies that it should save the file using the same name it had on the server: species_EnsemblBacteria.txt

It's important to note that both `curl` and `wget` download to the computer that the command line belongs to. So, if you are logged into AWS on the command line and execute the `curl` command above in the AWS terminal, the file will be downloaded to your AWS machine, not your local one.

Moving files between your laptop and your instance

What if the data you need is on your local computer, but you need to get it *into* the cloud? There are also several ways to do this, but it's *always* easier to start the transfer locally. **This means if you're typing into a terminal, the terminal should not be logged into your instance, it should be showing your local computer. If you're using a transfer program, it needs to be installed on your local machine, not your instance.**

Transferring Data Between your Local Machine and the Cloud

These directions are platform specific, so please follow the instructions for your system:

Please select the platform you wish to use for the exercises: UNIX Windows

Uploading Data to your Virtual Machine with scp

`scp` stands for 'secure copy protocol', and is a widely used UNIX tool for moving files between computers. The simplest way to use `scp` is to run it in your local terminal, and use it to copy a single file:

```
scp <file I want to move> <where I want to move it>
```

Note that you are always running `scp` locally, but that *doesn't* mean that you can only move files from your local computer. In order to move a file from your local computer to an AWS instance, the command would look like this:

```
$ scp <local file> <AWS instance>
```

To move it back to your local computer, you re-order the `to` and `from` fields:

```
$ scp <AWS instance> <local file>
```

Uploading Data to your Virtual Machine with scp Open the terminal and use the `scp` command to upload a file (e.g. local_file.txt) to the dcuser home directory:

```
$  scp local_file.txt dcuser@ip.address:/home/dcuser/
```

Downloading Data from your Virtual Machine with scp Let's download a text file from our remote machine. You should have a file that contains bad reads called ./shell_data/scripted_bad_reads.txt.

Download the bad reads file in ./shell_data/scripted_bad_reads.txt to your home ~/Download directory using the following command **(make sure you substitute dcuser@ip.address with your remote login credentials)**:

```
$ scp dcuser@ip.address:/home/dcuser/shell_data/untrimmed_fastq/scripted_bad_reads.txt ~/Downloads
```

Remember that in both instances, the command is run from your local machine, we've just flipped the order of the to and from parts of the command.

##Project Organization

- Questions +How can I organize my file system for a new bioinformatics project? +How can I document my work?
- Objectives: +Create a file system for a bioinformatics project. +Explain what types of files should go in your `docs`, `data`, and `results` directories. +Use the `history` command and a text editor like `nano` to document your work on your project.
- Keypoints: +Spend the time to organize your file system when you start a new project. Your future self will thank you! +Always save a write-protected copy of your raw data.

Getting your project started

Project organization is one of the most important parts of a sequencing project, and yet is often overlooked amidst the excitement of getting a first look at new data. Of course, while it's best to get yourself organized before you even begin your analyses, it's never too late to start, either.

You should approach your sequencing project similarly to how you do a biological experiment and this ideally begins with experimental design. We're going to assume that you've already designed a beautiful sequencing experiment to address your biological question, collected appropriate samples, and that you have enough statistical power to answer the questions you're interested in asking. These steps are all incredibly important, but beyond the scope of our course. For all of those steps (collecting specimens, extracting DNA, prepping your samples) you've likely kept a lab notebook that details how and why you did each step. However, the process of documentation doesn't stop at the sequencer!

Genomics projects can quickly accumulate hundreds of files across tens of folders. Every computational analysis you perform over the course of your project is going to create many files, which can especially become a problem when you'll inevitably want to run some of those analyses again. For instance, you might have made significant headway into your project, but then have to remember the PCR conditions you used to create your sequencing library months prior.

Other questions might arise along the way: - What were your best alignment results? - Which folder were they in: Analysis1, AnalysisRedone, or AnalysisRedone2? - Which quality cutoff did you use? - What version of a given program did you implement your analysis in?

Good documentation is key to avoiding this issue, and luckily enough, recording your computational experiments is even easier than recording lab data. Copy/Paste will become your best friend, sensible file names will make your analysis understandable by you and your collaborators, and writing the methods section for your next paper will be easy! Remember that in any given project of yours, it's worthwhile to consider a future version of yourself as an entirely separate collaborator. The better your documenation is, the more this 'collaborator' will feel indebted to you!

With this in mind, let's have a look at the best practices for documenting your genomics project. Your future self will thank you.

In this exercise we will setup a file system for the project we will be working on during this workshop.

We will start by creating a directory that we can use for the rest of the workshop. First navigate to your home directory. Then confirm that you are in the correct directory using the `pwd` command.

```
$ cd
$ pwd
```

You should see the output:

```
/home/margeno
```

Tip

If you aren't in your home directory, the easiest way to get there is to enter the command `cd`, which always returns you to home.

Exercise

Use the `mkdir` command to make the following directories:
- `dc_workshop` - `dc_workshop/docs` - `dc_workshop/data` - `dc_workshop/results`

Solution

```
$ mkdir dc_workshop
$ mkdir dc_workshop/docs
$ mkdir dc_workshop/data
$ mkdir dc_workshop/results
```

Use `ls -R` to verify that you have created these directories. The `-R` option for `ls` stands for recursive. This option causes `ls` to return the contents of each subdirectory within the directory iteratively.

```
$ ls -R dc_workshop
```

You should see the following output:

```
dc_workshop/:
data  docs  results

dc_workshop/data:

dc_workshop/docs:

dc_workshop/results:
```

Organizing your files

Before beginning any analysis, it's important to save a copy of your raw data. The raw data should never be changed. Regardless of how sure you are that you want to carry out a particular data cleaning step, there's always the chance that you'll change your mind later or that there will be an error in carrying out the data cleaning and you'll need to go back a step in the process. Having a raw copy of your data that you never modify guarantees that you will always be able to start over if something goes wrong with your analysis. When starting any analysis, you can make a copy of your raw data file and do your manipulations on that file, rather than the raw version. We learned in a previous episode how to prevent overwriting our raw data files by setting restrictive file permissions.

You can store any results that are generated from your analysis in the `results` folder. This guarantees that you won't confuse results file and data files in six months or two years when you are looking back through your files in preparation for publishing your study.

The `docs` folder is the place to store any written analysis of your results, notes about how your analyses were carried out, and documents related to your eventual publication.

Documenting your activity on the project

When carrying out wet-lab analyses, most scientists work from a written protocol and keep a hard copy of written notes in their lab notebook, including any things they did differently from the written protocol. This detailed record-keeping process is just as important when doing computational analyses. Luckily, it's even easier to record the steps you've carried out computational than it is when working at the bench.

The `history` command is a convenient way to document all the commands you have used while analyzing and manipulating your project files. Let's document the work we have done on our project so far.

View the commands that you have used so far during this session using `history`:

```
$ history
```

The history likely contains many more commands than you have used for the current project. Let's view the last several commands that focus on just what we need for this project.

View the last n lines of your history (where n = approximately the last few lines you think relevant). For our example, we will use the last 7:

```
$ history | tail -n 7
```

Exercise

Using your knowledge of the shell, use the append redirect `>>` to create a file called `dc_workshop_log_XXXX_XX_XX.sh` (Use the four-digit year, two-digit month, and two digit day, e.g. `dc_workshop_log_2017_10_27.sh`)
> > ## Solution > `html > $ history | tail -n 8 >> dc_workshop_log_2017_10_27.sh` > > > Note we used the last 7 lines as an example, the number of lines may vary.

You may have noticed that your history contains the `history` command itself. To remove this redundancy from our log, let's use the `nano` text editor to fix the file:

```
$ nano dc_workshop_log_2017_10_27.sh
```

(Remember to replace the `2017_10_27` with your workshop date.)

From the `nano` screen, you can use your cursor to navigate, type, and delete any redundant lines.

Navigating in Nano

Although `nano` is useful, it can be frustrating to edit documents, as you can't use your mouse to navigate to the part of the document you would like to edit. Here are some useful keyboard shortcuts for moving around within a text document in `nano`. You can find more information by typing Ctrl-G within `nano`.

key	action
Ctrl-Space OR Ctrl-→	to move forward one word
Alt-Space OR Esc-Space OR Ctrl-←	to move back one word
Ctrl-A	to move to the beginning of the current line
Ctrl-E	to move to the end of the current line
Ctrl-W	to search

Add a date line and comment to the line where you have created the directory. Recall that any text on a line after a `#` is ignored by bash when evaluating the text as code. For example:

```
# 2017_10_27
# Created sample directories for the Data Carpentry workshop
```

Next, remove any lines of the history that are not relevant by navigating to those lines and using your delete key. Save your file and close `nano`.

Your file should look something like this:

```
# 2017_10_27
# Created sample directories for the Data Carpentry workshop

mkdir dc_workshop
mkdir dc_workshop/docs
mkdir dc_workshop/data
mkdir dc_workshop/results
```

If you keep this file up to date, you can use it to re-do your work on your project if something happens to your results files. To demonstrate how this works, first delete your `dc_workshop` directory and all of its subdirectories. Look at your directory contents to verify the directory is gone.

```
$ rm -r dc_workshop
$ ls
```

```
shell_data  dc_workshop_log_2017_10_27.sh
```

Then run your workshop log file as a bash script. You should see the `dc_workshop` directory and all of its subdirectories reappear.

```
$ bash dc_workshop_log_2017_10_27.sh
$ ls
```

```
shell_data  dc_workshop dc_workshop_log_2017_10_27.sh
```

It's important that we keep our workshop log file outside of our `dc_workshop` directory if we want to use it to recreate our work. It's also important for us to keep it up to date by regularly updating with the commands that we used to generate our results files.

Congratulations! You've finished your introduction to using the shell for genomics projects. You now know how to navigate your file system, create, copy, move, and remove files and directories, and automate repetitive tasks using scripts and wildcards. With this solid foundation, you're ready to move on to apply all of these new skills to carrying out more sophisticated bioinformatics analysis work. Don't worry if everything doesn't feel perfectly comfortable yet. We're going to have many more opportunities for practice as we move forward on our bioinformatics journey!

References

A Quick Guide to Organizing Computational Biology Projects

##Awk

If we need to count the number of lines in a file, we can use the previously showed command for word counting wc. Let's use a representative table from the "Table S2. Significant QTL loci for mate choice and morphology" from the Bay et al. 2017 publication /shell_data/TableS2_QTL_Bay_2017.txt

```
$ wc -l TableS2_QTL_Bay_2017.txt
```

```
19 TableS2_QTL_Bay_2017.txt
```

As you probably remember, -l is an option that asks for the number of lines only.

However, wc counts the number of newlines in the file, if the last line does not contain a carriage return (i.e. there is no emptyline at the end of the file), the result is going be the actual number of lines minus one.

A workaround is to use Awk. Awk is command line program that takes as input a set of instructions and one or more files. The instructions are executed on each line of the input file(s).

The instructions are enclosed in single quotes or they can be read from a file.

Example:

```
$ awk '{print $0}' TableS2_QTL_Bay_2017.txt
```

```
Trait   n   LOD Chr Position (cM)   Nearest SNP
mate choice 200 4.5 14  22.43   chrXIV:1713227
mate choice     200 4.61    21  8   chrXXI:9373717
discriminant function   200 4.83    12  17  chrXII:7504339
discriminant function   200 4.23    14  8.1 chrXIV:4632223
PC2 200 4.04    4   30.76   chrIV:11367975
PC2 200 6.67    7   47  chrVII:26448674
centroid size   200 6.97    9   47.8    chrIX:19745222
x2* 200 3.93    7   60  chrUn:29400087
y2* 200 9.99    4   32  chrIV:11367975
x3  200 4.45    1   32.3    chrI:15145305
x4  200 5.13    16  30.9    chrXVI:12111717
x5* 200 4.54    15  6   chrXV:505537
y5  200 4.21    4   24.9    chrIV:15721538
x6  200 3.96    16  29.5    chrXVI:13588796
y6* 200 4.14    9   30.2    chrIX:18942598
y15*    200 5.3 2   27  chrII:19324477
x16 200 5.49    7   60  chrUn:29400087
x17     200 4.92    1   32.8    chrI:14261764
Table S2. Significant QTL loci for mate choice and morphology
```

This command has the same output of "cat": it prints each line from the example.fasta file.

The structure of the instruction is the following: - curly braces surround the set of instructions - print is the instruction that sends its arguments to the terminal - $0 is a variable, it means "the content of the current line"

As you can see, the file contains a table.

Awk automatically splits the processed line by looking at spaces: in our case it has knowledge of the different columns in the table.

Each column value for the current line is stored into a variable: $1 for the first column, $2 for the second and so on.

So, if we like to print only the second column from the table, we execute

```
$ awk '{print $2}' TableS2_QTL_Bay_2017.txt
```

```
200
200
200
200
200
200
200
200
200
200
200
200
200
200
```

```
200
200
200
200
S2.
```

We can also print more than one value, or add text to the printed line:

```
$ awk '{print "Significant_QTL_loci",$4,$5}' TableS2_QTL_Bay_2017.txt
```

```
Significant_QTL_loci Chr Position
Significant_QTL_loci 14 22.43
Significant_QTL_loci 21 8
Significant_QTL_loci 12 17
Significant_QTL_loci 14 8.1
Significant_QTL_loci 4 30.76
Significant_QTL_loci 7 47
Significant_QTL_loci 9 47.8
Significant_QTL_loci 7 60
Significant_QTL_loci 4 32
Significant_QTL_loci 1 32.3
Significant_QTL_loci 16 30.9
Significant_QTL_loci 15 6
Significant_QTL_loci 4 24.9
Significant_QTL_loci 16 29.5
Significant_QTL_loci 9 30.2
Significant_QTL_loci 2 27
Significant_QTL_loci 7 60
Significant_QTL_loci 1 32.8
Significant_QTL_loci QTL loci
```

The comma puts a space between the printed values. Strings of text should be enclosed in double quotes. In this case we are printing the text "Significant_QTL_loci", the fourth and the fifth column for each row in the table.

So, $0 is the whole line, $1 the first field, $2 the second and so on. What if we want to print the last column, but we don't know its number? Maybe it is a huge table, or maybe different lines have a different number of columns.

Awk helps us thanks to the variable NF. NF stores the number of fields (our columns) in the row.