Bioinformatics cheat sheet

By Allison E. Mann, Krithi Sankaranarayanan, Christina Warinner, Andrew Ozga (Updated 9/2/20)

Before getting started!

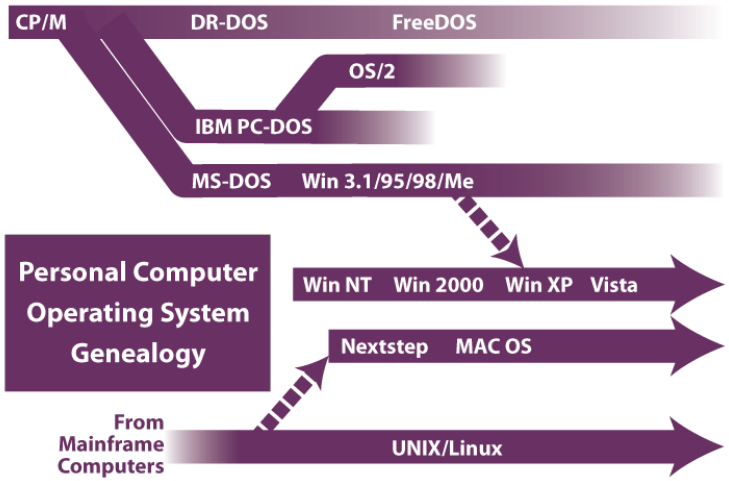
- These commands may not work exactly as written with your operating system (OS) but with some tweaking/googling you should be able to figure out how to use them on your system (see more resources below).
- Remember! Using the full file path is nearly always important!
- If you don't know how to use a command or you want more options try `man command`, `command -h`, or `command --help`
- If you run into a problem, Google is your best friend! Nine times out of ten someone somewhere has had the same problem, posted it to a forum, and had it solved by the community. Helpful online forums include SEQanswers, StackOverflow, and ResearchGate, among many others!

Table of contents

Your computer

Your OS and why it matters

Your operating system (OS) is essential system software that manages all of your computer's hardware and software (the base of which is known as the kernel). If you have a Mac computer your OS is Mac OS, Windows runs Windows OS, and Linux distributions all run different flavors of the Linux OS. While both Linux and Mac OS are part of the wider UNIX family of operating systems, Windows is based on MS-DOS (Microsoft Disk Operating System). It's helpful to thing about operating systems as a genealogy with two major branches – DOS and UNIX (See figure below, from http://ed-informatics.org/healthcare-it/).



Because of their shared family history, Linux and Mac OS have very similar command line tools and most anything you can do in a Linux system you can also do on a Mac and vice versa. Windows, on the other hand, is fundamentally different. The flexibility and accessibility of UNIX-based systems make them the preferred OS for computational biology. As many bioinformatics programs are written with UNIX systems in mind, Windows users should download a Linux version to run in a virtual environment (instructions below). Even if you have a

Mac, it is important to be familiar with Linux. This is especially true if you are interested in cluster computing as nearly all cluster computers run Linux environments.

Getting started with Linux

Congratulations! You're already set up to run nearly everything in this cheat sheet!

Getting started with Mac OS X

Mac OS X has a UNIX operating system that operates similarly to Linux. With a few slight tweaks, you should be able to run everything in this cheat sheet with little problem. Your command line programming is in the terminal. To open the terminal, click the Mac search icon (magnifying glass) at the upper right corner of the screen and type in Terminal. Select the Terminal program with the icon that looks like a black box

Getting started with a Windows PC

You have a Microsoft Windows operating system that runs a MS-DOS like command line. Although Windows is good for running many commercial Graphical User Interface (GUI) programs, it is not good for running programs in the command line. To run the command line scripts and programs described in this manual, you must first download and install a Virtual Box for Linux: https://www.virtualbox.org/wiki/Linux_Downloads. The most commonly used Linux distribution is Ubuntu but don't be afraid to try other choices and find one that works for you!

System requirements

Be aware that successfully running jobs on your own computer depends greatly on the size of the job and your computer's specifications (specs). Because of this the table below might not apply to your particular goals. In general, however, the following features are the minimum desired specs for processing next generation sequencing data on your personal computer. Other jobs will need to be sent to a cluster to complete. It's a good idea to have an external hard drive to serve as backup. As a general rule I always have important data files and scripts saved in triplicate -- whether that is on external/internal hard drives or on online repositories (e.g., GitHub).

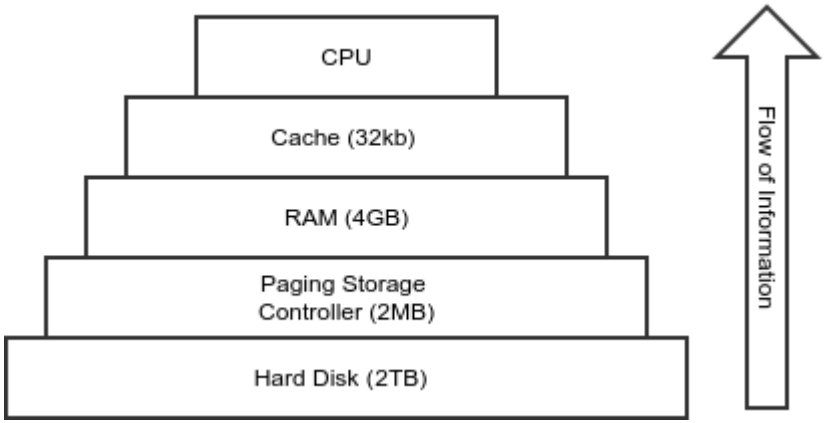
Processor/CPU	4 to 8 cores
Hard drive/Solid state drive	At least 1 TB of memory
RAM	At least 16 GB

Understanding your system's memory & parallel processing

CPU	Stands for Central Processing Unit. Consists of registers that contain the instructions that are sent to your hardware (e.g., fetch, store, operate data)
Cache	Small but very fast bit of system memory that holds data while in transition between your RAM and CPU (also sometimes called static RAM or SRAM)
RAM	Stands for Random Access Memory. Fast because it can be accessed directly in random order and no mechanical devices are required to read it.
Hard Disk	This is what most people think about when talking about a computer's memory (but as you can see is only one type of system memory). The hard disk provides permanent storage, is much slower, but larger than other memory types.
Virtual memory	Acts like RAM but resides on the disk (occasionally important to set for jobs submitted to cluster computers)

You've probably noticed that if you submit a large job or try to open a large data file on your personal computer you run into problems (or you might even crash your computer!). If not a flaw in the program or file itself, this is very likely due to a fundamental architectural component of modern computers known as "memory hierarchy." Essentially, your computer has multiple forms of memory that differ in terms of access speed and size which your computer uses to access and store

data. To perform a task, information needs to be transmitted to your CPU along this memory hierarchy.



Most CPUs can only perform a couple tasks at a time (depending on how many cores the CPU has) so fi the data it needs to continue working is stuck somewhere in the memory hierarchy where it cannot access it the task will exit with an error. Think of it as a sort of information bottleneck. Because of this, most computationally expensive tasks require the program to be parallelized (that is, the job is split into smaller problems and solved simultaneously on several processors or threads). Luckily, many programs are set up to have parallel options and CPUs usually have more than one core or threads, which allow local parallelization. Look for error messages that mention things like “cache dump” or “out of memory” – these point to problems with your memory bottleneck. Even when certain programs can be run in parallel, some jobs require a lot more memory than what your basic computer has so using a cluster computer will be necessary. The key thing to keep in mind here is that increasing the memory allocated to a task doesn’t necessarily “speed up” the process; it provides more space for the process to continue! See the section on parallelization for more information.

Unix/Linux bash basics

Basic terms: The shell

Shell	Program that takes keyboard commands and passes them to the operating system to carry out. While there are different flavors of the shell, most commands are run on the bash shell (Bourne Again Shell) is the standard shell available on Linux and Mac OS.
Terminal	Window created by a GUI through which you can access the shell. The terminal contains your command line and is accessed differently for Linux, Mac OS, and Windows machines. Terminal/command line/shell are often used interchangeably even though they do refer to different things.
Command line	Line where you type your commands. The line itself begins with a shell prompt, which is usually your computer username followed by $ or >.
Directory	Folder that contains files. Directories have a hierarchical structures. All directories have a pathname that show how the directory is connected to your home directory. For example: /home/mann/my_folder/
Executable file	A file that can be invoked as a command. Can contain shell scripts or programming code commands
Command	An executable program. Can be a compiled binary (C++), a scripting language program (e.g., Perl or Python), or a shell function.
Argument	Arguments are a type of parameter that provides information to a command (often other file names)

Structure of commands and pipelines

Structure of a shell command

```
command option(s) argument(s)
```

Example:
You have a fasta file called contigs.fa and you want to know the format of the sequence labels. You can search the file for the pattern “^>” (a regular expression) using the command “grep” and print to your standard output (in this case, your

terminal window). This will print the lines containing your sequence labels but not the sequences themselves.

```
grep "^>" contigs.fa
```

Results:
```
>ASV595
>ASV596
>ASV597
>ASV598
>ASV599
>ASV600
```

Structure of a shell pipeline
```
command option argument | command option argument
```

Example:
From the same fasta file, you want to know how many sequences there are in the file. Since every sequence name begins with > you can just count these and that will tell you how many sequences you have. Use the script above to isolate only the lines containing > at the beginning of the line (^) and then use a pipe (|) the result of the first command to a secondary command called wc (word count) to count the number of lines (-l).

```
grep "^>" contigs.fa | wc -l
```

Result:
```
6
```

Input and output redirection

|	Pipe command result into standard input of next command
;	Give serial commands on a line (without redirect)
<	Input redirect
>	Output redirect
>>	Appends output to file rather than overwriting it
cat	Read input file(s) and copies them to standard output

The default standard output for most commands is to print to screen, but often this is not the desired result. For example, in the previous example, imagine printing all fasta sequence labels for a 5 GB fasta file! If you do this by mistake, press CTRL+C to kill the command. Instead, you may want to pipe the output into the input of the next command using a pipe (|) as shown in the previous example with wc.

Alternatively, you may want to save the result to a new text file. To do this use the greater than symbol (>):

```
grep "^>" contigs.fa > contigs.list
```

You may also want to redirect the contents of a file to a command that normally takes keyboard strokes as its input. To do this, use the less than symbol (<):

command < file

Note that some commands have options for selecting non-standard input and output files as arguments. For these commands the input option is usually (but not always – check the manual!) –i and the output option is usually -o:

```
fastq_to_fasta -i contigs.fq -o output.fa
```

However, for the –i and -o options to work, they must be valid options for that command. If you're unsure what the command's options are, check the command's man or help file:

`command -h` OR `--help` OR `man command`

The cat (concatenate) command is flexible and can be used to append files, similar to >>. If there were two files containing sample1 contigs, contigs1.fa and sample 2 contigs, contigs2.fa, and you wanted to combine them into one file, use cat:

```
cat contigs1.fa contigs2.fa > all_contigs.fa
```

Standard output and standard error

Every time you run a command, bash generates command output information and command error information. By default, the output prints to screen and error

messages only appear if there are errors. However, the error data also contains additional useful file information that you may want to know, even if there are no errors.

Programs produce output in several streams with numbered file descriptors. The first three streams with file descriptors are: the standard input (0), the output (1), and the errors (2)

By default, bash interprets the input redirect < as taking the file descriptor 0. By default, bash interprets the output redirect > as taking the file descriptor 1. As as result, the following two commands are equivalent.

```
grep "^>" contigs.fa > contigs.list

grep "^>" contigs.fa 1> contigs.list
```

However, if you want to save the error stream instead of the output stream, you just need to modify the command as follows:

```
grep "^>" contigs.fa 2> contigs.err
```

Alternatively, if you want to save both streams, you can specify this as well:

```
grep "^>" contigs.fa 1> contigs.list 2> contigs.err
```

If for some reason you don't want the output or error data streams, you can redirect one or both of them to the bit bucket known as /dev/null. The bit bucket accepts data and does nothing with it. The bit bucket can be useful if a particular command generates a lot of error messages that you want to ignore:

```
grep "^>" contigs.fa 1> contigs.list 2> /dev/null
```

Basic terms: Scripting

Script	A file that contains shell commands. Scripts can be written using basic UNIX/Linux commands or can be written in another scripting language like Perl or Python
Job	A script that is currently running. Normally jobs run in the command line and this freezes your terminal until the job completes. Jobs can be made to run in the background by adding an ampersand (&) after the command or pressing CTL+Z and then bg
Alias	A short name that redirects to a user-created command, pipeline, or script
Function	A mini-script located within another script
Parameter	Information that modifies the execution of the command. Options and arguments are parameters
Variable	An object that can be assigned a string value. Variables in bash are preceded with a $

Aliases can be very useful when you need to perform a complex command over and over again. For example, say you have to count the number of sequences in each fasta file within your working directory. If there are a lot of files, this could be very tedious and take a long time to do. You can simplify this process by writing a short script to automate the process and run the jobs for you. To make it even easier still, you can give your script an alias so that it requires even less typing.

For example, the following script uses the command grep in combination with the option -c, a type of parameter, and a command substitution function $(command) to automate sequence counting within fasta files located within the working directory:

```
grep -c "^>" $(ls)
```

To run this same job in the background

```
grep -c "^>" $(ls) &
```

You can alias this script so you don't have to retype it each time you want to use it:

```
alias fastac='grep -c "^>" $(ls)'
```

If you then navigate to a directory containing the files contigs1.fa and contigs2.fa and type fastac the result is:

```
contigs1: 186342348
contigs2: 87155451
```

Help, history, and panic buttons

`q`	Escapes man files, help files, and text viewing
`CTRL+C`	Stops current command in the terminal
`CTRL+D`	Ends keyboard input (for multi-line command entry, for example using cat); helpful if command hangs
`CTRL+L`	Clears screen and places current line at the top of the terminal (useful for if the terminal gets cluttered)
Up arrow	Recalls the previous command
`history`	Prints numbered list of the last 500 commands
`!88`	Prints the 88th command (can replace with any number)
`exit`	Log out of the terminal shell
man command	View command manual
help command	View help file for command (shell built in commands only)
`command -h`	View help file for command
`command --help`	View help file for command

Sometimes things don't go as planned. If your screen starts scrolling and printing millions of lines of text, don't panic. Press CTRL and C simultaneously and the command will stop.

If you are reading a help file and can't seem to get out of the help file window, don't panic. Press q and you will quit the help file.

If you type a command and press enter and nothing happens, you may have forgotten to complete the command (this happens a lot with the cat command). Try pressing CTRL and D simultaneously to tell bash that you're finished.

If you make a mistake and print a million lines of text and now your terminal is hopelessly cluttered, press CTRL and L simultaneously to clean it up. If it is installed you can also type clear and you will have a cleaned up terminal.

If you want a record of the commands you entered into the shell, type history and a numbered list of your past 500 commands will appear on the screen.

Unprintable characters

Shell programming has a long history, and as a result it carries around programming baggage from when computers (and teletype machines) were very different. It is important to discuss some of these arcane holdovers from the 1960s-1980s so you can understand and troubleshoot frustrating bash quirks.

Many of the characters in use when the shell was developed are no longer in common use. Bash uses the ASCII (pronounced "as-key"; American Standard Code for Information Exchange) character system, which contains 128 characters, of which the first 33 are unprintable and 95 are printable (although one is a blank space). Unprintable characters can be typed by entering the appropriate control code (CTRL + code). By convention, the CTRL key is signified by a caret (^) in text:

Unprintable: null (^@), start of header (^a), start of text (^b) end of text (^c), end of transmission (^d), enquiry (^e), acknowledgement (^f), bell (^g), backspace (^h), horizontal tab (^i), line feed (^j), vertical tab (^k), form feed (^l), carriage return (^m), shift out (^n), shift in (^o), data link escape (^p), device control 1 (^q), device control 2 (^r), device control 3 (^s), device control 4 (^t), negative acknowledgement (^u), synchronous idle (^v), end of transmission block (^w), cancel (^x), end of medium (^y), substitute (^z), escape (^[), file separator (^\), group separator (^]), record separator (^^), unit separator (^_), delete (^?).

Printable: !"#$%&'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ
[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

The 33 unprintable characters are control codes that were used by teletype machines to transmit commands. The ambiguity of control codes in today's world causes incompatibility problems when transferring plain text files across platforms and operating systems. The most notorious examples are the linefeed and carriage return control codes. Typewriters and teletype machines required two commands to advance to a new line: 1) a command to advance the paper (linefeed) and 2) a command to return the carriage to the beginning of the line (carriage return). With the advent of computers, this two-step process was unnecessary. To mark a newline, Unix used only the linefeed, Mac used only the carriage return, and Windows/DOS used both. Because the characters are unprintable, plain text generated on each operating system looks the same, but these "hidden" characters cause trouble if you try to copy text from a Mac or Windows machine into the bash shell. With OS X, Mac switched over to linefeed, but nevertheless problems still arise.

If you use a text editor to write scripts, you must be sure to use one that is compatible with bash. On Macs, the plain text editor TextEdit is NOT compatible with bash. Download and use TextWranger or SublimeText instead. For PCs, the plain text editor Notepad is NOT compatible with bash. NEVER use Word to generate scripts. Within the UNIX/Linux environment, nano, vim, emacs, and gedit are all good text editors made for bash scripting. If you are unsure if your script has hidden characters, test it with cat –a.

```
cat -a broken script
broken script^m$
```

The ^m is a hidden carriage return in the script

Directory and file paths

The locations of all directories and files in your computer are known to bash via their path. The path records how the directory or file in question is connected to the root directory.

For example, I have a fasta file on my computer that contains 16S rRNA sequences generated from kissing bug gut samples called rep_set.fa

The full file path for this file is: `/Users/mann/github/triatomes`

Note that the root directory is not written in this path. Instead, it is presumed by the first /. The first subdirectory is Users, the next is mann, etc. Each subdirectory is separated from the others by a forward slash /.

If I want to access or analyze this file using bash commands, I need to tell bash where the file is. I can do this one of two ways: 1) I can navigate to the directory containing the file and then run the commands through this directory, or 2) I can provide the full file path to the file and run the command from another directory.

When interacting with bash, you are always in a directory. When you first call a terminal you always begin in your home directory. When you navigate to another directory, that directory becomes your working directory.

If you want to know what your working directory is, type pwd (print working directory). If you want to return to your home directory, type cd. To check that you have moved to the correct directory, type pwd to see the file path, or type ls to list all of the files and subdirectories in your new working directory.

Note:
When naming files and directories NEVER use spaces. Bash will interpret the space as indicating two separate names and may cause the program to not work properly. So if you have a directory named My Folder you should change it to something like My_Folder or MyFolder.

Command search path

Every command is actually a file containing executable code. How does bash find these files? When you type a command, bash searches its executable PATH for the command. The PATH is a list of directories, some of which are set by default to the PATH. Most programs you download and install will store their executables in your /bin directory. If not, you will need to move the files there or create a symbolic link so that bash can find it.

To create a symbolic link to executable downloaded to a location other than /usr/local/bin, navigate to the location of the executable and type:

```
sudo cp -L program /usr/local/bin
```

This uses the copy command (cp) in combination with sudo (super user; gives you administrative access to folders otherwise locked by your OS) with -L option (creates link) to link your executable program files to the directory /usr/local/bin. Alternatively, you can add additional directories to the PATH with the following command:

```
PATH=$PATH:/Users/mann/my_program_directory
```

Where /Users/mann/my_program_directory is any valid directory path to where your command is located. To check what directories are already in your path type:

```
echo $PATH
```

You should see a list of all the directories that are currently set for execution by bash. For example, my current setup is:

```
echo $PATH
/usr/local/opt/icu4c/sbin:/usr/local/opt/icu4c/bin:/Users/man
n/.rvm/gems/ruby-2.4.4/bin:/Users/mann/.rvm/gems/ruby-
2.4.4@global/bin:/Users/mann/.rvm/rubies/ruby-
2.4.4/bin:/Users/mann/miniconda3/bin:/Users/mann/miniconda3/c
ondabin:/opt/local/bin:/opt/local/sbin:/Users/mann/miniconda3
/envs/qiime1/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
:/Library/TeX/texbin:/opt/X11/bin:/Users/mann/.rvm/bin
```

Bash searches the PATH directories in order until it finds the executable you called from the terminal, and then it runs it. NEVER put "." in your PATH as this makes your computer vulnerable to hackers through a process called privilege escalation.

If you want to use a command that you used earlier in this session but now you can't remember what it was, use the hash command:

```
hash
```

Results:

```
hits command
   5 /usr/bin/grep
   6   /bin/ls
```

It will return a two-column table of each command you've used and how many times you've used it. Alternatively use the history command to review the past 500 commands you've entered.

Basic terms: Navigation

Path	Navigation information that is used to locate a file or directory
Root directory	Base directory of your computer. All other directories are subdirectories of this directory
Home directory	The directory for your username on a computer. You control permissions for folders and files within your home directory but may have limited permissions for folders and files outside of your home.
Working directory	The folder you are currently "in"

File system navigation

`cd /path/to/directory`	Change directory to the one provided in the path
`cd ..`	Go up one directory in the path
`cd`	Go to your home directory
`pwd`	Print your current directory
`cd ~`	Go to home directory

As described in previous sections, you can use cd (change directory) to navigate your directories, and pwd (print working directory) to view the file path of your current working directory

Check directory contents

`ls`	List all files/folders in your current directory
`ls -a`	List all files including hidden files (preceded with a .)
`ls -l`	List all files and associated metadata
`ls -R`	List all files recursively (in subdirectories as well)
`ls -lhrt`	List all files sorted by time modified, including permissions, file type, and size (this is my go-to)
`man ls`	Open the manual for the ls command. There are many different options

As described in previous sections, you can use ls (list) to list the contents of a directory. You can use options to view additional metadata about the contents of a directory such as when the file was created, how big they are, etc.

Searching for files, directories, and programs

`find foo`	Search for file named "foo" in directory hierarchy
`find –name foo`	Search for file "foo" in any directory
`locate foo`	Locate all instances of file name "foo"
`grep -R foo`	Search all files in directory hierarchy for file containing "foo"
`which program_name`	Shows where executable program is located (i.e., it's path)

Sometimes you need to find a file/directory/program but you don't know where it is. Use the find, locate, grep, and which commands to locate what you need.

Wildcards

Wildcards apply to filenames and can be used with any command that accepts filenames as arguments.

*	Any character(s)
?	Any single character
[abcd]	Any character in abcd
[!abcd]	Any character not in abcd
[[:class:]]	Any character that is a member of the specified class
[[:alnum:]]	Any alphanumeric character
[[:alpha:]]	Any alphabetic character
[[:digit:]]	Any numeral
[[:lower:]]	Any lowercase letter
[[:upper:]]	Any uppercase letter

Wildcards can be combined, as shown in the following examples:

[[:upper:]123]*	Any file beginning with an uppercase letter or numbers 1, 2, or 3
*[[:upper:]123]	Any file ending with an uppercase letter or numbers 1, 2, or 3
[![:upper:]]*	Any file that does not begin with an uppercase letter

For example, if you have the following files in a directory:

```
C214c_16s.seqs.fasta
C214r_16s.seqs.fasta
F1948c_16s.seqs.fasta
F1948r_16s.seqs.fasta
S37c_16s.seqs.fasta
S37r_16s.seqs.fasta
```

But you only wanted to list those that start with "S" and have "c_16s" in their filename you can use ls and wildcard characters:

```
ls S*c_16s*
```

Result:
```
S37c_16s.seqs.fasta
```

Note:
Wildcards should not be used with commands that take regular expressions as arguments, like grep. Many of the characters have different meanings when used in regular expressions and may yield surprising and undesired results if placed within a regular expression. There are flags that can be passed to the grep command that allow use of regular expressions, if these are not called, however, your output will not be what you expect.

Creating, moving, renaming, and removing files and directories

mkdir foo	Make directory called "foo"
cp foo.txt foo_copy.txt	Copy file foo.txt as new file foo_copy.txt
cp -r my_dir my_copy_dir	Copy folder my_dir and all of its contents into new directory my_copy_dir
mv foo.txt new.txt	Rename foo.txt as new.txt
mv -r my_dir new_dir	Rename directory my_dir to new_dir
touch foo.txt	Create new file in working directory called foo.txt
rm foo.txt	PERMANENTLY remove foo.txt
rm -r my_dir	PERMANENTLY remove directory my_dir

CAUTION!! The rm command PERMANENTLY removes the item, it doesn't go into the trash like it would if you deleted from the desktop.

Viewing Files in the Command Line

`less foo.txt`	opens text file in new command window
`more foo.txt`	opens text file in same command window
`head foo.txt`	view first 10 lines of foo.txt
`head -n 25 foo.txt`	view first 25 lines of foo.txt
`tail foo.txt`	view last 10 lines of foo.txt
`tail -n 25 foo.txt`	view last 25 lines of foo.txt

Sometimes you may want to view the contents of a file to 1) make sure that a command worked correctly, or 2) that you have the correct file, or 3) to check how many columns there are in the header. Usually, you don't want to look at the entire file (especially if it contains 200 million lines!). You can view just part of the file using the less, more, head, and tail commands above.

Unpacking and Compressing Files

`tar cvf - file1 file2 file 3 | gzip > foo.tar.gz`	takes files file1, file2, and file3 and creates a tar archive (foo.tar), and then uses gzip to compresses the tar archive (foo.tar.gz)
`tar czf foo.tar.gz file1 file2 file3`	takes files file1, file2, and file3 and creates a tar archive (foo.tar), and then uses gzip to compresses the tar archive (foo.tar.gz)
`tar xfz foo.tar.gz`	unpacks compressed tar file "foo.tar.gz"
`gunzip < foo.tar.gz | tar xvf -`	unpacks compressed tar file "foo.tar.gz"
`gunzip foo.gz`	uncompresses gzip file
`tar -xvf foo.tar`	extract tar file
`unzip foo.zip`	unpacks compressed zip file
`bunzip2 foo.bz2`	uncompresses bz2 file
`bunzip2 foo.tar.bz2 xvf foo.tar`	unpacks compressed bz2 file

Text files are often compressed in order to make them easier to share or store. For genetic data, tar (tape archive) and gzip (GNU zip) are often combined. Tar allows multiple files to be combined into a single archive, and gzip allows this archive to be compressed into a single file. Zip is an unrelated program that performs archiving and compression with a single command. .zip files are typically not as compressed as tar.gz files.

Note: File compression works better for files with repetitive information. Text files with DNA sequence data (e.g., fasta or fastq) typically compress well. Others file formats do not. For example, trying to compress some PDF files may make even bigger files!

Important keyboard shortcuts

`arrow`	recalls previous command
`tab`	autocomplete when typing in terminal
`tab tab`	Show possible options for autocomplete
`ctrl+a`	Move to beginning of line
`ctrl+e`	Move to end of line
`ctrl+k`	Kill forward to end of line
`ctrl+forward/back arrow keys`	Move forward/back by one word
`alt+tab`	Cycle through open applications/windows
`ctrl+alt+l`	Lock screen
`ctrl+a`	Select all text

`ctrl+c`	Copy text
`ctrl+x`	Cut text
`ctrl+v`	Paste text
`ctrl+backspace`	Delete entire word
`ctrl+z`	undo
`ctrl+shift+z`	redo

Note:
When working in the terminal, the tab key is your best friend. Instead of typing out full paths, tab will give you options, autocomplete a command, and other magical things. Don't forget to tab often!

Metacharacters & expansions

Metacharacters are special characters (e.g., *) that let you search for or change text patterns (e.g., any three-letter word beginning with f) rather than literal strings (e.g., the word foo). It is very important to understand that there are two different uses of metacharacters. In the first case, metacharacters are recognized by the bash shell and used in **expansions**. In the second case, metacharacters are recognized by certain programs or commands and used in **regular expressions**. Bash will first interpret any metacharacter it sees as an expansion metacharacter. To indicate to bash that the metachararacter is intended for the command, you must put the metacharacter(s) in quotes. This is very important. Both expansions and regular expressions use the same metacharacter types (asterisks, dots, slashes, etc.), but the metacharacters have different meanings when used in expansions vs. regular expressions. Wow, that's confusing! Expansions and regular expressions are explained in more detail below.

Some characters have special meaning to bash. These metacharacters can be used to perform expansion before the command is performed. Types of expansion include:

Pathname expansion:

```
command partialpath*

ls D*
```

Results:

lists all files and folders in working directory beginning with D

Tilde expansion:

```
command ~

ls ~
```

Results:
lists all files and folders in home directory

Brace expansion:

```
command text{pattern}text

echo G12-{A,B,C}
```

Results:

```
G12-A G12-B G12 C

echo G12-{1..4}.fa
```

Results:

```
G12-1.fa G12-2.fa G12-3.fa G12-4.fa
```

Arithmetic expansion (integers only):

```
command $((expression))

echo $((2+2))
```

Results:
returns answer 4

Parameter expansion:

```
command $variable

head $good
```

Results:
would return the first 50 lines of the file G12-1.fa if I had previously defined the variable $good as G12-1.fa

Regular Expressions

Regular expressions are symbolic notations used to identify patterns in text. The main program used to work with regular expression in bash is grep. The programming language Perl has a richer assortment of notations able to rapidly identify patterns in text, but grep works well enough for searching small files using bash.

grep stands for *global regular expression print* and is a command used to search text files for a specified regular notation and output any line containing a match to standard output. Basically, grep allows you to search through text.

The basic structure of the grep command is as follows:

```
grep options regex input
```

where the first argument is regex, the regular expression or pattern being searched, and the second argument is the input file or directory to be searched.

When searching for a pattern using grep, all characters in the pattern will be interpreted literally except the following metacharacters.

^	Anchor: only accept match at beginning of line
$	Anchor: only accept match at end of line
.	Any character. Example: grep .tar would yield star, ntar, and .tar
{ }	Match element a specific number of times {n} exactly n times. {n,m} at least n times but no more than m times. {n,} n or more times {,m} m or fewer times
x{y}	Exactly y repeats of x
x{y,k}	Between y and k repeats of x
?	Quantifier: make preceding element optional; match element zero or one more times
*	Quantifier: make preceding element optional; match element zero or more times
x*	0 or more repeats of x
+	Quantifier: make preceding element required; match element one or more times
(x|y)	Matches x or y
|	Alternation: signifies “or”. Search for pattern one or pattern two
\d	Any whole number
\D	Any non-number
\w	“word”, meaning letters and digits and _
\W	Any non-word
\s	White space
\S	Non-white space

Note:
Several of the metacharacters above have different meanings to the shell (e.g., tell it to perform an expansion, etc.). Therefore, if the pattern contains a metacharacter, it is essential to put the pattern within single quotes. Alternatively, if you want some of the metacharacters to be interpreted by the shell (e.g., a $variable) then use double quotes.

Quoting and escaping allow you to selectively turn off expansions.

‘	All special characters within single quotes lose their meaning to the shell. No expansions are performed on characters inside single quotes.
“	All special characters within double quotes lose their meaning to the shell, except $ (dollar sign), \ (back slash), and ‘ (back tick). Only expansions relating to arithmetic expansion, parameter expansion, and command substitution are performed.
\	Used to quote (escape) a single character. Often used in combination with double quotes to suppress one (but not all) expansions. Also used to escape spaces in file names.
\t	tab
\\	backslash
\’	single quote
\”	double quote
\n	newline
\r	carriage return

Say, for example, you want to find and print to screen all full sentences in the following file that begin with a capital letter:

```
Source.
License by making exceptions from one or more of its
conditions.
License would be to refrain entirely from conveying the
Program.
all NECESSARY SERVICING, REPAIR OR CORRECTION.
SUCH DAMAGES.
also add information on how to contact you by electronic and
paper mail.
```

You can use a combination of regular expression and escape characters to do so.

```
grep "^[A-Z].*\." file.txt
```

```
Source.
License by making exceptions from one or more of its
conditions.
License would be to refrain entirely from conveying the
Program.
SUCH DAMAGES.
```

So let's break down what this grep command does. It tells the shell to look for any capital letter `[A-Z]` at the beginning `^` of each line and pull any other information afterwards `.*`
Because we want to pull sentences that end in a period, and regular expressions consider periods as special characters with special meaning, we must escape it with `\.`
The `\` character tells grep to treat the following character literally.

Basic GREP and Other File Search Functions

`grep ‘pattern’ foo.txt`	search for lines with pattern in foo.txt and return it
`grep -i ‘pattern’ foo.txt`	search for lines with pattern (ignoring case) in foo.txt and return it
`grep ‘^pattern’ foo.txt`	search for lines with pattern only at beginning of line in foo.txt and return it

`grep '$pattern' foo.txt`	search for lines pattern at end of line in foo.txt and return it
`grep '^$' foo.txt`	search for blank lines
`grep '^pattern$' foo.txt`	search for lines that contain the pattern and nothing else
`grep '^..j.r$' foo.txt`	search for lines that contain only a five letter pattern in which the third letter is j and the last letter is r
`grep –v 'pattern' foo.txt`	search for lines WITHOUT pattern in foo.txt and return it
`grep –v '^@' foo.txt`	search for lines WITHOUT null values (empty lines) and return it
`grep 'pattern' -A 1 foo.txt`	search for pattern in foo.txt and return it and one line after it. Useful for fasta files!
`grep 'pattern' -B 1 foo.txt`	search for pattern in foo.txt and return it and one line before it
`grep –E 'pattern1|pattern2|pattern3' foo.txt`	search for any of the three patterns in foo.txt (interprets patterns as regular expressions, -E)
`grep 'pattern' foo.txt > foo2.txt`	search for pattern in foo.txt and print results to foo2.txt
`grep -c 'pattern' foo.txt`	search for lines with pattern in foo.txt and counts number of times pattern appears
`wc -l foo.txt`	output number of lines in foo.txt
`grep 'pattern' foo.txt | wc -l`	output number of lines in which pattern is found in foo.txt

Other Common Commands for Text Processing

`cat`	Concatenate files and print on the standard output
`sort`	Sort lines of text files
`uniq`	Report or omit repeated lines
`rev`	Reverse lines of a file or files
`tr`	Translate/modify characters in a file
`comm`	Compare two sorted files line by line
`diff`	Compare files line by line and find differences
`sed`	Stream editor for filtering and transforming text
`awk`	Pattern scanning and processing language
`echo`	Display a line of text

sed stands for stream editor and it is a powerful command for editing file names or sequence names within a file on the fly (not altering the original file). Sed is especially useful in loops and in pipelines in which the output extension of one program needs to be changed in order for the files to be used by the next program in the pipeline. For example, suppose you have folder full of fasta files with the extension .fa but you need to change the extension to .fasta in order to run the imaginary program foo. You can do this easily with sed:

`ls | sed -e 's/.fa/.fasta/' |` foo

The syntax used above for sed is as follows: `-e` = regular expression, `'s` = substitute, `/.fa/` = remove everything between the / characters, `//` = replace with everything between the / characters (in this example replace with nothing), ' =

end substitution. However, note that sed does not actually change the original file name.

Loops

Loops are useful for performing repetitive tasks, such as systematically renaming all files in a folder or executing a command for multiple files. Say for example you have a directory of fasta files that you want to blast using the same parameters but don't want to type out the blast command for all of them -- this is a perfect time to use a loop!

Basic bash loop structure:

```
input | while read line; do command; done
```

Your input should be a list of the variables/filenames/identifiers that you use to execute the loop. In our example it will be the IDs of the fasta files. You then pipe this command to the looping command while read line; do. Note that "line" in this statement can be any variable name you choose, line makes most sense, however, as each time you iterate through the loop the function will read a line in from your input to include as the new variable until it reaches the end of the list. You can then insert your command with $line as your variable. Finally, the loop is closed with the ; done statement.

So for this example let's use a loop to blast all of the 16S sequence fasta files in the following directory:

```
ls –lhrt
total 76M
-rw-rw-r-- 1 allison allison 5.4M Nov 22 23:11
C214c_16s.seqs.fasta
-rw-rw-r-- 1 allison allison 2.7M Nov 22 23:11
C214r_16s.seqs.fasta-rw-rw-r-- 1 allison allison 7.1M Nov 22
23:11 F1948c_16s.seqs.fasta
-rw-rw-r-- 1 allison allison 2.6M Nov 22 23:11
F1948r_16s.seqs.fasta
-rw-rw-r-- 1 allison allison 4.8M Nov 22 23:11
S37c_16s.seqs.fasta
-rw-rw-r-- 1 allison allison 4.1M Nov 22 23:11
S37r_16s.seqs.fasta
-rw-rw-r-- 1 allison allison 4.4M Nov 22 23:11
S454c_16s.seqs.fasta
-rw-rw-r-- 1 allison allison 895K Nov 22 23:11
S454r_16s.seqs.fasta
-rw-rw-r-- 1 allison allison 3.7M Dec  5 09:19
C214c_SHOT_rename.fasta
-rw-rw-r-- 1 allison allison 1.8M Dec  5 09:20
C214r_SHOT_rename.fasta
-rw-rw-r-- 1 allison allison 5.0M Dec  5 09:21
F1948c_SHOT_rename.fasta
-rw-rw-r-- 1 allison allison 1.8M Dec  5 09:26
F1948r_SHOT_rename.fasta
```

We first need a list of the sequence IDs we want to use. Use a combination of ls and sed to print your IDs to

```
ls *16s.seqs.fasta | sed 's/_16s.seqs.fasta//'
C214c
C214r
F1948c
F1948r
S37c
S37r
S454c
S454r
```

Now we're ready to initiate our loop! Each time the your command is called the $line variable is replaced by the next line in your input (so in the first iteration C214c will be called, the second C214r, the third F1948c, etc...). Notice that we

used the $line variable again to name our output files according to their sample IDs. Another important thing to remember about using variables in loops is that you must escape the variable if it is followed by text ($line_16s.seqs.fasta) but this is not required if it is followed by a "." ($line.blast.out).

```
ls *16s.seqs.fasta | sed 's/_16s.seqs.fasta//' | while read
line; do blastn -query $line\_16s.seqs.fasta -out
$line.blast.out -db greengenes.fasta -outfmt 6 -evalue 1e-10;
done
```

Advanced: customizing your shell

When you work on several computers at once it is helpful to modify your shell prompt to differentiate between them. For example, I currently have accounts on a cluster and two other computers other than my own laptop. To differentiate between them I've modified a specific shell variable called PS1 in my .bashrc file to control how the prompt looks across computers.

Your bash profile is a hidden file in your user directory (~) that is loaded every time you load your shell environment and contains configuration information and preferences for your terminal. For example, when you add a new directory to your path you are essentially adding a new line of text to your bash profile telling it that along with the default path you want to add this new path. On Linux, the bash profile will be located here: ~/.bashrc while on Macs it is here ~/.bash_profile.

To change the color, computer name, and information displayed by your terminal prompt you need to redefine the PS1 variable in your bash profile. There are many things you can change but here's what I have as an example:

```
PROMPT_HOSTNAME='lobsang'
PROMPT_COLOR='1;34m'
PS1='\[\e]1;${PROMPT_HOSTNAME}\a\e]2;${PROMPT_HOSTNAME}:${PWD
}\a\
\e[${PROMPT_COLOR}\]\
[\u@${PROMPT_HOSTNAME} \w]\n \#\$ \
\[\e[m\]'
```

Which gives me the following prompt:

[mann@lobsang ~/github/fad_mouse]
5$

In my custom PS1 variable I set what I want my computer's name to be (PROMPT_NAME='lobsang'), that I want the prompt itself to be bolded and blue (PROMPT_COLOR='1;34m' where '1;34m' is a color code), that I want it to explicitly tell me what working directory I am currently in (PWD), how many commands have I entered into the command prompt since beginning my current session (\#, in this case I've done 5) and that I want the place where I type commands to be just below the initial prompt (\n) (which is why it is two lines). Any commands that I enter into the terminal come directly after the $. There are endless possibilities to customize your prompt!

Another way to customize your shell is to create aliases for programs or commands. For example, typing out a SSH command every time you want to log onto a server can be a bit tedious. To get around this I've set up different aliases to speed up this process. In my bash profile the following lines:

```
alias zoo='ssh username@zoology.ubc.ca'
alias ent='ssh username@entamoeba.zoology.ubc.ca'
alias talon='ssh username@talon3.hpc.unt.edu'
```

mean that instead of typing out the full SSH command, path and all I can just type in "zoo" or "ent" or "talon" and my shell knows that what I really mean are those SSH commands.

You can even rename built in programs by creating an alias in your bash profile. As an example, I work on both Linux and Mac machines and while they have a lot of similarities, there are some annoying differences. The command "sed" is found in both Linux and Mac OS but they behave differently. To run the Linux style sed on my Mac I downloaded a different program "gsed" but I also don't want to have to remember that it's called a different thing between OS. Aliases to the rescue!

```
alias sed=gsed
```

Working on servers

Connecting to remote hosts (i.e., other computers)

Often bioinformatic pipelines require more computational power than what you typically find on a personal laptop or desktop. In these cases you'll need to connect to and run jobs on a remote computer cluster. A computer cluster is a set of connected computers that work together where each computer is known as a node that greatly improve the performance and speed of a single computer alone. You can "tunnel" or log into a remote cluster from your own computer via the terminal by using the SSH protocol (Secure Shell tunneling). Similarly SCP (Secure Copy Protocol) can move folders and files between your personal computer (the local host) and a cluster (remote host).

`ssh user@address`	Remote connect to host
`scp foo.txt user@address:/path/`	Upload file foo.txt from your current working directory to somewhere on the remote host (must specify path!)
`scp user@address:/path/foo.txt /home/`	Download foo.txt from remote host and save in folder /home/
`ftp ftp.ncbi.ncbi.nih.gov`	Remote connect to NCBI using FTP (File Transfer Protocol)

Note:
By default SSH and SCP will use port 22 at the remote host to form a connection between computers. While SSH and SCP are already secure protocols, occasionally system administrators will change the port number to increase security. In this case you would add -p (SSH) or -P (SCP) to your command along with the new port number.

Job queue systems

Because many people may have access to a cluster, they often come equipped with software that controls when and how long a job runs on the cluster called job schedulers. The job queue refers to a data structure that contains jobs that are in a list to run. Users submit a small script that contains the "job" they want executed to the queue to be processed after which the job scheduler maintains the queue in the background. Along with the program that the User wants to run, typically a job script also lists the amount of memory that needs to be allocated, how long the job should run, and other information. There are many different flavors of job scheduling software, but they all follow the same general structure. For example, if your cluster uses PBS (Portable Batch System), your job script may look something like this:

```
#!/bin/bash
#PBS -l nodes=1:ppn=2
#PBS -l walltime=00:00:59
cd /home/mann/test/
align_seqs.py -i ASV18.fa -t LTPs132_salmonella_ref.align.fa
```

In the above example, the first line identifies which shell you are using (in this case bash), the second line tells the job scheduler the number of nodes and processors that you would like it to allocate to your job, the third line is how much wall-clock time is requested, and finally the fourth and fifth lines are the code that you want executed (in this case the first line changes into the /test directory and the second runs the program align_seqs.py). Every cluster is different but there are plenty of examples of scripts online!

Parallelization

Parallel computing is a type of computing where a large task can be split into smaller tasks which are solved simultaneously, greatly speeding up the time that it takes to solve the problem. Parallel computing can be performed on multi-core and multi-processor computers or can be split over multiple computers in the case of cluster computing. One of the easiest ways to run a program in parallel on Unix and Linux computers is by using the shell tool GNU parallel (https://www.gnu.org/software/parallel/). Say you have multiple fastq files that you need to perform the same task on. You could run these through a for loop (see section on loops) but as loops run serially this could take a long time if you have many many files. Instead you can run multiple files at once using GNU parallel. For example, if you wanted to run the following command in parallel:

This:
```
ls *fastq.gz | while read line; do gzip -d $line; done
```
Becomes this:
```
ls *fastq.gz | parallel 'gzip -d {}'
```

Easy! In the above example your zipped up fastq files will be unzipped by GNU parallel. By default, parallel will take up as many cores as your computer has at its

disposal. You can set a preferred number of cores with the option -j (e.g., -j 10 will use 10 cores).

File permissions

Typically, the user who created or uploaded a file or directory to a server is the "owner" which by default limits access from other users. To share files or folders across multiple users you may need to change the permissions for that file or folder. To do this you can use the chmod and chown commands.

You can see the current permissions and other information about a file using the list command in long format:

```
ls -l
```

Beside your file you'll see a string of characters that looks something like this:

```
-rwxrwxrwx
```

Each character in this string tells you what the item is as well as the permissions set for the owner of the item, the group assigned to that item, and others. If the first character in your string is – that means the item is a file, if it is d the item is a directory. The next three characters indicate the permissions set for the owner of the item, the following the permissions of the group, and the final the permissions of other users on the server. If all positions have a r, w, and x that means that that particular person or group has read (r), write (w), and execution (x) permissions. A dash indicates that that particular permission is not allowed. So for example the following string is a file that the owner and group have all permissions while others only have read permissions but cannot write or execute the file.

```
-rwxrwxr--
```

You can assign new permission status to all three groups using chmod and a three digit code:

0: No permission
1: Execute permission only
2: Write permission only
3: Write and execute permissions
4: Read permission only
5: Read and execute permissions
6: Read and write permissions
7: Read, write, and execute permissions

So if I wanted to give the owner all permissions (7), the group read and write permissions (6) and read and execute permissions to others (5) we would type:

```
chmod 765 my_file.txt
```

To completely change ownership or group ownership of a file or folder you can use chown where a colon separates the owner:group status of an item on your server. For example, to change the owner of a file (that, importantly, you already have ownership of) to yan and the group to viviana:

```
chown yan:viviana my_file.txt
```

Advanced: other useful commands

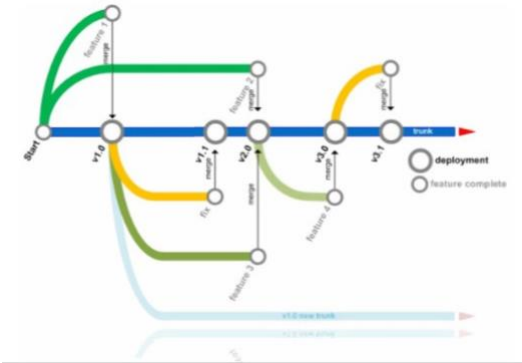
`who`	See list of all users currently logged into the server
`whoami`	Prints your user name
`mail`	Send mail from the command line. Example with file attachment: `mail -A my_file.txt -s "subject header" user@mail.com < /dev/null`
`wall`	Send a message to all users on the server. Type in wall then your message and then CTRL+D to send the message
`df`	Check disk space on your machine
`du`	Check how much disk space your current working directory is using
`kill`	Kill a process currently running on your machine. You need to give the kill command your job's PID number which can be found by typing `ps`
`tree`	Prints directories and folders in a tree like fashion from your current working directory
`passwd`	Change your user account password
`uname -a`	List information about your current system
`top`	Consistently updated log of what processes are running, how much memory they are using, who submitted the job, etc. For a better experience install htop as an alternative!

`wget`	Download files from the web. For example: `wget http://github.com/aemann01/my_file.py`

Note:
If you send emails to yourself or others be aware that they are often automatically filtered into the spam folder so if it didn't seem to work check there!

Version control

Version control is a valuable tool in bioinformatics and refers to software tools that record changes in a file or a set of file overtime. Version control software allows you to manage and track changes to files as well as provides a way for teams of researchers to collaborate on projects. Scripts and other documents are kept on a "master branch" in the software from which branches can be split and merged. A visual example of version control workflow can be found below:



Git and GitHub

There are many different version control options but one of the most popular is Git. Git is an open source code management system while GitHub refers to the online hosting of Git "repositories" (essentially a project folder). Many open source bioinformatics software is hosted on GitHub so it is important to know how to access GitHub repositories. Additionally, you can host your own projects (for free) on GitHub. To do this you need to 1) add a new repo to your GitHub account, 2) initialize a directory on your local computer as a Git repository, 3) add this to your GitHub repository as the main branch, 4) add those files within the directory that you want to track, 5) commit any changes that you've made and finally, 6) push your changes to the main branch of your repository. On the command line this looks like:

```
cd /path/to/my/git/repository
echo "# MyRepo" >> README.md
git init
git add README.md
git commit -m "My first commit message"
git remote add origin https://github.com/<your account>/MyRepo.git
git push -u origin master
```

You can check whether or not your files are added to the push queue or if they have been changed with:

```
git status
```

Once your GitHub repository is set up you can start tracking your files with the add-commit-push trio of commands.

```
git add my_file.txt
git commit -m "comment on the changes I made"
git push
```

Conda environments

A fundamental challenge in bioinformatics is analytic reproducibility. In a perfect world, someone should be able to take code and data that you've generated and run them on their own computer to produce the same figures and results. In practice this is tricky because the other person might have different versions of the software or software dependencies you used installed (a situation often called "dependency hell"). You can limit these problems by generating "containers" or "environments" that package up all of your code and its dependencies independent from other packages installed on your computer. Popular container programs include Docker and Snakemake. For the purposes of this cheat sheet we'll look at

one of the simpler ways to manage package distributions for scientific reproducibility with conda environments. The first step is to install conda using miniconda (https://docs.conda.io/en/latest/miniconda.html) or anaconda (https://www.anaconda.com/). Once installed you can create a conda environment with specific package versions in YAML format ("YAML Ain't Markup Language"). Example environment.yml file:

```
name: rpo_gene
channels:
  - defaults
  - bioconda
  - conda-forge
dependencies:
  - bioconda::blast=2.10.
  - conda-forge::parallel=20200522
  - bioconda::bedtools=2.29.2
  - bioconda::seqtk=1.3
```

In the first line we need to give a name to the environment (in this case: rpo_gene), next we set the conda channels to look for packages in (defaults, bioconda, conda-forge), and finally list the packages and version of those packages we want to install within that environment (notice that the syntax includes which channel we want to look for the package in). To then create this environment, you would run:

```
conda env create environment.yml
```

Then to activate the environment:

```
conda activate rpo_gene
```

And finally, to exit the environment:

```
conda deactivate
```

Jupyter notebooks

Another useful tool for data analysis reproducibility are jupyter notebooks (https://jupyter.org/). Jupyter notebook is a web-based application that allows researchers to share documents containing code, text, analyses, and visualizations. While jupyter notebooks were originally developed to share code written in the python language, other languages can be incorporated (e.g., R) to create visually appealing and reproducible analysis files. To start a jupyter notebook server:

```
jupyter notebook
```

This command will open up a webpage that shows your current working directory contents. To create a new notebook click on the "New" dropdown menu and choose the appropriate programming language – this will create a new file in your working directory called Untitled.ipynb. From here you can add code and notes to your notebook which will compile any analyses you do within the ipynb file. For an example jupyter notebook using R: https://hub.gke.mybinder.org/user/binder-examples-r-xfdla3li/notebooks/index.ipynb

Genomics basics

Structure of common files in genomics

Data files in the biological sciences are typically stored as text files or binary files that follow specific formatting rules. Much of the work of someone who does bioinformatics is converting data from one file format to another so that it can be read or used by a program. Knowing how files should be formatted can be key in getting quick information about your data as well as troubleshooting when things go wrong.

FASTA

FASTA-formatted files can contain many sequence entries. Each entry consists of two lines. The first line contains the sequence name and any associated comments or metadata. Different parts of the first line may be separated common separators, such as commas, colons, semicolons, or pipes. The second line contains the sequence information consisting of IUPAC (nucleotide) and IUB (amino acid) codes. Dashes signify gaps. X signifies a masked position. Case is ignored; numbers are ignored; special characters such as spaces, tabs, and asterisks are ignored. FASTA files are usually given an extension. Common extensions for

nucleic acid data are: fasta, .fas, .fa, .fna, .ffn, and .seq. Non-coding RNA data is given the extension .frn. The extension for protein data is .faa.

Example of FASTA entry:

```
>EAS139:136:FC706VJ:2:2104:15343:197393
ATCACGGATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCAC
AGTTT
```

Example of FASTA nucleotide entry formatted according to the NCBI standard:

```
>gi|862687|gb|U26045.1|MCU26045 Mycoplasma capricolum
capricolum California kid 16S ribosomal RNA (rrnA) gene,
partial sequence
CTGGCGGCATGCCTAATACATGCAAGTCGAACGGGGGTGCTTGCACCTCAGTGGCGAACGG
GTG
```

Example of FASTA protein entry formatted according to the NCBI standard:

```
>gi|31563518|ref|NP_852610.1| microtubule-associated proteins
1A/1B light chain 3A isoform b [Homo sapiens]
MKMRFFSSPCGKAAVDPADRCKEVQQIRDQHPSKIPVIIERYKGEKQLPVLDKTKFLVPDH
VNMSELVKIIRRRLQLNPTQAFFLLVNQHSMVSVSTPIADIYEQEKDEDGFLYMVYASQEF
```

FASTQ

FASTQ-formatted files are similar to FASTA files but contain additional quality information for each nucleotide. Each entry consists of four lines. The first line is proceeded with a @ and contains the sequence name. The second line contains the sequence data. The third line consists of only a + character. It signifies that the fourth line contains the sequence quality data. FASTQ files are usually given an extension. Common extensions are: fastq and .fq.

Example of FASTQ entry obtained from Illumina sequencer (after Casava v.1.8):

```
@EAS139:136:FC706VJ:2:2104:15343:197393 1:Y:18:ATCACG
GGGTGATGGCCGCTGCCGATGGCGTCAAATCCCACC
+
IIIIIIIIIIIIIIIIIIIIIIIIIIIII9IG9IC
```

Illumina uses a systematic naming system for its output. For Illumina data generated after Casava 1.8, it follows this system:

EAS139	Unique instrument name
136	Run ID
FC706VJ	Flowcell ID
2	Flowcell lane
2104	Tile number within flowcell lane
15343	'x'-coordinate of the cluster within the tile
197393	'y'-coordinate of the cluster within the tile
1	the member of a pair: always 1 for single end sequencing; 1 or 2 for paired-end sequencing
Y	Y if read is filtered, N otherwise
18	0 when none of control bits are on, otherwise it is an even number
ATCACG	index sequence

The quality value (Q) for each basecall is an integer mapping of the probability (*p*) that the basecall is incorrect. Q is usually calculated using the Phred quality score formula:

$$Q = -10 \log_{10} p$$

Low Phred scores denote low quality, while high scores denote high quality. For example, a Phred score of 10 corresponds to a basecall accuracy of 90%, a score of 20 corresponds to a basecall accuracy of 99%, a score of 30 corresponds to a basecall accuracy of 99.9%, and so forth. Applied to a 100 bp sequence, using a Phred score cutoff of 30 results in a 90% chance that all bases were correctly called. Using a Phred score cutoff of 35 results in a 95% chance that all bases were correctly called. Applying a Phred score cutoff of 35 to an Illumina dataset of 1x100 bp reads means that, under ideal conditions, about 95% of your reads will have no basecall errors, while about 5% of your reads will contain sequencing error(s).

FASTQ quality information is encoded in printable ASCII characters (characters 34 to 128) corresponding to Phred scores 0 to 93. The quality values in order from lowest quality to highest quality are:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ
[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

The above ASCII characters have been color coded for ease of interpretation. Red characters correspond to Phred scores <16. These base calls are unreliable and should be discarded. Orange characters correspond to medium confidence Phred scores of 16-34. A judgment call must be made on whether to keep these basecalls. Green characters correspond to Phred scores 35-41 and signify high confidence basecalls that should be kept. Blue characters correspond to Phred scores >41 and signify very high quality base calls. Phred scores >41 may be generated for Sanger data, but are almost never seen for current Illumina chemistry.

Example of FASTQ entry retrieved from NCBI Sequence Read Archive (SRA, formerly Short Read Archive):

```
@SRR001666.1 EAS139:136:FC706VJ:2:2104:15343:197393 length=36
GGGTGATGGCCGCTGCCGATGGCGTCAAATCCCACC
+SRR001666.1 EAS139:136:FC706VJ:2:2104:15343:197393 length=36
IIIIIIIIIIIIIIIIIIIIIIIIIIIIII9IG9IC
```

SAM and BAM

Sequence Alignment/Map (SAM) files are sequence alignment files. Bowtie2 outputs SAM files. SAM files are tab-delimited text files consisting of an optional header and an alignment section. Header lines begin with @. Each alignment line has 11 mandatory fields. Eight additional optional fields may also be present. SAM files are unsorted - this means that the alignment data appears in the order that the reads that were searched using Bowtie2; they are not ordered by where they appear in the reference genome. Here is an example of a SAM file with one sequence:

```
@HD VN:1.0 SO:unsorted
@SQ SN:gi|674660337|ref|NC_024711.1| LN:97065
@PG ID:bowtie2 PN:bowtie2 VN:2.1.0
ACB052:117:D2AGKACXX:6:1101:6343:2414 89
gi|674660337|ref|NC_024711.1| 76501 3 100M = 76501 0
CATTTTTAACATCACCGTCTAAATCACCTGATA
DDDDDDEDDDDBB?DDEDEEEEED@DFFFHHHH AS:i:-38 XN:i:0 XM:i:7
XO:i:0 XG:i:0 NM:i:7 MD:Z:14T0T0A7T24A2T1C45 YT:Z:UP
```

The header (shown in blue) may consist of up to five lines, each containing different information. Typically, the SAM files we generate will have three header lines. Header @HD contains information about the SAM file version and setup. Header SQ contains information about the reference sequence. Header PG contains information about the program that generated the SAM file. Refer to the samtools manual for more detailed information about the header: http://samtools.github.io/hts-specs/SAMv1.pdf

The eleven mandatory tab separated fields (columns) are: (1) QNAME: the query/sequence name; (2) FLAG: the bitwise flag; (3) RNAME: the reference sequence name; (4) POS: 1-based leftmost mapping position; (5) MAPQ: the mapping quality; (6) CIGAR: CIGAR string; (7) RNEXT: reference name of the mate/net read; (8) PNEXT: position of the mate/next read; (9) TLEN: observed template length; (10) SEQ: segment sequence; (11) QUAL: ASCII of Phred-scaled base quality+33.

In addition to these mandatory fields, additional optional fields may follow with the format TAG:TYPE:VALUE. The only optional field we commonly use is the last one, which reports whether the paired-end reads mapped concordantly within 500bp to the same genome (YT:Z:CP), or if the paired-end reads did not map (YT:Z:UP). Discordant or unpaired reads indicate genome rearrangement (e.g., indels, inversions, etc.) in the query sequence relative to the reference sequence. For more information on these optional fields, see the samtools manual: http://samtools.github.io/hts-specs/SAMv1.pdf

Note: When using samtools, it is VERY important to provide the flags in the correct order!

BAM

BAM is a SAM file that has been converted into binary format (Binary Sequence Alignment/Map). BAM files can typically be compressed to much smaller sizes than their corresponding SAM files. For this reason, it is often more efficient to transfer sequence alignment files as BAM files. BAM files, however, are not human readable. This is what the first few lines of the BAM file corresponding to the above SAM file looks like:

```
?BC?sr?e?```p?p?
                ?2?3?
                     ??*?+?/*IM??

??Jб137133066?)JM??s?702174?3????4703?p??t?J?//?L5?
?b?%?xMdЬfdS?
???3?F
I@?????E??e?q?-`!Fd?yVbdE ?YvnwK?Cp,$hU???=gō???<?e@u?1?4 s0

?0k?`1???PÙ?>q??}?ِ??????ᾳ??[???ŏ~?s????g??gx???p6???7?qz??t?
?37s?z8??O????t4??蹿
?2w<?g?????'|W:r????J//?0?y&s??u?^!>p~??B??~EQ?_??Z?DB??X\
```

GenBank

The GenBank database is a collection of all publically available nucleotide sequences along with their corresponding protein translations maintained by NCBI (the National Center for Biotechnology Information). Submissions to GenBank can be downloaded via NCBI's FTP server or through the web using the Batch Entrez site (https://www.ncbi.nlm.nih.gov/sites/batchentrez) using the unique accession numbers assigned to each sequence.

GenBank formatted files have a number of defined fields that provide information about what, where, and how the sequence was obtained. Below is an example GenBank entry. The LOCUS filed includes the locus name (BI784134), length of the sequence (150 bp), molecule type (mRNA), GenBank division (EST), and the modification date (26-SEP-2001). The DEFINITION field is a brief description of the sequence and can include the source organism, gene or protein name, etc.

The ACCESSION is a unique identifier that is assigned to that particular GenBank entry. Because the accession number never changes after being uploaded to the database a separate VERSION field includes the accession number followed by a period and the version of that sequence. If any changes are made to the example below, for example, the version number would be updated to BI784134.2. The KEYWORDS field includes a word or phrase that describes the sequence. If no keywords are included this field only has a period. SOURCE indicates what organism the sequence derived from but doesn't necessarily need to be the scientific name while ORGANISM is always the formal scientific name. The REFERENCE field includes the sequence coordinates related to any publications related to this entry followed by the AUTHORS, TITLE, and JOURNAL, even if the project is unpublished.

FEATURES include information about the gene and gene products. Finally, ORIGIN is the sequence itself followed by // that signals the end of the file.

```
LOCUS       BI784134 150 bp mRNA linear EST 26-SEP-2001
DEFINITION  kh31c04.y1 Ascaris suum male head pAMP1
ACCESSION   BI784134
VERSION     BI784134.1
KEYWORDS    EST
SOURCE      Ascaris suum (pig roundworm)
ORGANISM  Ascaris suum
REFERENCE   1  (bases 1 to 150)
AUTHORS   McCarter,J et al.
TITLE     The Washington Univ. Nematode EST Project
JOURNAL   Unpublished
COMMENT     Contact: McCarter JP
FEATURES    Location/Qualifiers
         source  1..150
                 /organism="Ascaris suum"
                 /mol_type="mRNA"
                 /db_xref="taxon:6253"
                 /sex="male"
                 /dev_stage="Adult"
ORIGIN  1 atcgcatggt ctcgaaccgg cgacgtgtct atcaagtgtc
        61 gtagtttatg tgcctaccat ggttgtaacg ggtaacggag
       121 gagggagcct gagaaacggc taccacatcc
//
```

BLAST

BLAST (Basic Local Alignment Search Tool) was developed in the mid 1980s and to this day remains the gold standard for comparing query sequences to reference sequences. BLAST can be performed through NCBI's web portal (https://blast.ncbi.nlm.nih.gov/Blast.cgi) or through the command line. While BLAST is a powerful bioinformatics tool, the algorithm is very slow and therefore should not be used for large sequence datasets. BLAST results can be stored in a variety of different formats set on the command line with the --outfmt option. The most useful format (for most downstream applications) is --outfmt 6 which gives the results in a tab separated text file. The default fields in this format option are as follows:

1	qseqid	Query sequence ID
2	sseqid	Subject (reference) sequence ID
3	pident	Percentage of identical matches
4	length	Alignment length
5	mismatch	Number of mismatches
6	gapopen	Number of gap openings
7	qstart	Start of the alignment in the query sequence
8	qend	End of the alignment in the query sequence
9	sstart	Start of the alignment in the subject sequence
10	send	End of the alignment in the subject sequence
11	evalue	Expected value score
12	bitscore	Bit score

Note:
The BLAST e-value is the number of expected hits of a similar quality score that could be found by chance. So for example, an e-value of 10 means that up to 10 hits can be expected to be found just by chance. The lower the e-value the better.

By default, BLAST uses an e-value of 10 so you should set a lower cutoff threshold when running the program. On the other hand higher the bit score, the better the sequence similarity. The bit score is a $\log_2$ scaled value that reflects the required size of a theoretical sequence database in which the match between the query and reference could be found by chance.

You can also set your own output format in the BLAST command line. For example, the following output format flag will record the data in a tab separated text file where the first column is the query ID, the second is the subject ID, and the third is the subject common name(s) separated by ";".

```
--outfmt '6 qseqid sseqid scomnames'
```

GFF

General Feature Format (GFF) is used to describe genes and other features of DNA, RNA, and protein sequences. GFF files are tab delimited with 9 fields per line of text:

1	sequence	Name of the sequence where the feature is located
2	source	Keyword identifying the source of the feature (e.g., a program or an organization)
3	feature	Feature type name (e.g., "gene" or "exon")
4	start	Genomic start of the feature with a 1-base offset
5	end	Genomic end of the feature with a 1-base offset
6	score	Numeric value that indicates the confidence of the source in the annotated feature. Null value = "."
7	strand	Single character that indicates the strand of the feature (positive or negative)
8	phase	Phase of CDS feature; can be either 0, 1, 2 (for CDS features) or "." (for everything else)
9	attributes	All the other information pertaining to this feature

VCF

Variant Call Format (VCF) files are text files that store information on gene sequence variations. Unlike GFF which store all of the genetic data, VCF files only record the variations and reference genome making them more portable. An example VCF file can be found below:

```
##fileformat=VCFv4.3 ##fileDate=20090805
##source=myImputationProgramV3.1
##reference=file:///seq/references/1000GenomesPilot-NCBI36.fasta
##contig=<ID=20,length=62435964,assembly=B36,md5=f126cdf8a6e0c7f379d
618ff66beb2da,species="Homo sapiens",taxonomy=x> ##phasing=partial
##INFO=<ID=NS,Number=1,Type=Integer,Description="Number of Samples
With Data"> ##INFO=<ID=DP,Number=1,Type=Integer,Description="Total
Depth"> ##INFO=<ID=AF,Number=A,Type=Float,Description="Allele
Frequency">
##INFO=<ID=AA,Number=1,Type=String,Description="Ancestral Allele">
##INFO=<ID=DB,Number=0,Type=Flag,Description="dbSNP membership,
build 129"> ##INFO=<ID=H2,Number=0,Type=Flag,Description="HapMap2
membership"> ##FILTER=<ID=q10,Description="Quality below 10">
##FILTER=<ID=s50,Description="Less than 50% of samples have data">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
##FORMAT=<ID=GQ,Number=1,Type=Integer,Description="Genotype
Quality"> ##FORMAT=<ID=DP,Number=1,Type=Integer,Description="Read
Depth"> ##FORMAT=<ID=HQ,Number=2,Type=Integer,Description="Haplotype
Quality"> #CHROM POS      ID          REF    ALT    QUAL  FILTER
INFO                              FORMAT       NA00001
NA00002          NA00003 20     14370    rs6054257  G     A      29
PASS    NS=3;DP=14;AF=0.5;DB;H2           GT:GQ:DP:HQ
0|0:48:1:51,51  1|0:48:8:51,51   1/1:43:5:.,. 20     17330    .
T     A      3     q10     NS=3;DP=11;AF=0.017
GT:GQ:DP:HQ  0|0:49:3:58,50  0|1:3:5:65,3     0/0:41:3 20
1110696  rs6040355  A     G,T    67    PASS
NS=2;DP=10;AF=0.333,0.667;AA=T;DB GT:GQ:DP:HQ  1|2:21:6:23,27
2|1:2:0:18,2     2/2:35:4 20     1230237  .          T     .      47
PASS    NS=3;DP=13;AA=T                   GT:GQ:DP:HQ
0|0:54:7:56,60  0|0:48:4:51,51   0/0:61:2 20     1234567  microsat1
GTC   G,GTCT 50    PASS     NS=3;DP=9;AA=G
GT:GQ:DP     0/1:35:4        0/2:17:2         1/1:40:3
```

VCF files begin with a header denoted by ## and includes metadata describing the file as well as keywords that describe the fields used in the body of the file (e.g., INFO, FILTER, FORMAT). The body of VCF files is tab separated and includes 8 mandatory columns:

1	CHROM	The name of the sequence on which the variation is called
2	POS	The 1-based position of the variation on the given sequence

3	`ID`	The identifier of the variation (e.g., using a database like dbSNP). If unknown: "."
4	`REF`	The reference base at the given position on the given reference sequence
5	`ALT`	A list of alternative alleles at this position
6	`QUAL`	The quality score associated with the inference of the given alleles
7	`FILTER`	A flag indicating which of a given set of filters the variation has passed
8	`INFO`	An extensible list of key-value pairs describing the variation
9	`FORMAT`	An (optional) list of fields for describing the samples
+	`SAMPLEs`	For each (optional) sample described in the file, values are given for the fields listed in FORMAT

A brief introduction to programming

Free online programming courses

- Code Academy: http://www.codeacademy.com
- edX: https://www.edx.org/learn/computer-programming
- Harvard University: https://online-learning.harvard.edu/subject/programming
- Khan Academy: https://www.khanacademy.org/computing/computer-programming
- And many more!

Python

Python is a flexible and easy to read programming language often used in the biological sciences. It was designed in the late 1980s by Guido van Rossum (Python's official Benevolent Dictator for Life) and was named after the British skit group, Monty Python. Unlike other common programming languages (e.g., Perl), Python relies heavily on whitespace to structure its code.

Official Python link: https://python.org

Best Python books:
Python Cookbook: Recipes for Mastering Python 3. Brian K. Jones & David M. Beazley
Python for Data Analysis: Data wrangling with Pandas NumPy. Wes McKinney
Automate the Boring Stuff With Python. Al Sweigart.

Online Python learning resources:
Learn Python the Hard Way: https://learnpythonthehardway.org/
BioPython Tutorial and Cookbook:
http://biopython.org/DIST/docs/tutorial/Tutorial.html
Code Academy: https://www.codeacademy.com/learn/learn-python

Installing Python
Windows: https://www.python.org/downloads/windows/
On Mac and Linux, Python should already be installed, check which version in the terminal by typing:

```
python --version
```

Note:
A major shift happened between Python 2.7 and Python 3 in that the old syntax rules in Python 2.7 are not compatible with Python 3 (this is the first time this happened in any Python version). As a result some Python scripts run with 2.7 while other with 3. As people transition over to 3 it's a good idea to learn Python 3 but also to have a copy of Python 2.7 on your computer to run scripts that aren't updated to the new rules. If you try to run a python program using 2.7 and you get a "syntax error" try running it with 3 (or vice versa), 9/10 this fixes the problem!

Basics of Python syntax
Python programs are written as text documents that end with the .py extension. Typically, python scripts begin with #!/usr/bin/python (#! Is called a shebang), which indicates the location of your python install (though this isn't strictly necessary). Unlike other programming languages, (e.g., Perl or R), blocks of code are not specified by brackets or bracers. Instead, indentations and colons indicate blocks of code in python. For example:

```
If (x > y):
      print("x is greater than y")
```

```
else:
      print("x is less than or equal to y")
```

In this script, a print statement is run depending on whether argument x is larger or smaller than y. It's good practice to add comments to your code so that other people can more easily read it (or you can still understand it 3 months after you wrote it!). Comments in both Python and R are denoted by the hash symbol #.

```
 If (x > y):
      print("x is greater than y")
else:
      print("x is less than or equal to y") # this line
won't be read if first condition is true
```

Anything beyond the # on the same line is ignored by the Python interpreter.

Data types in Python

Variables or objects that you define or import into the Python workspace have different data types. Data types are not declared by the user (as is the case with other low- or middle-level languages like C), instead they are inferred from the assigned statement. It's important to note that because of this, sometimes Python will infer the WRONG data type. Be sure to check that your variables are assigned in the proper format to avoid problems down the road.

The most common data types in Python include:

- Boolean: True/False
- Integer: Any full number (e.g., 5, 1000, 48939)
- Float: A number with a fractional value (e.g., 4.5, 3.002, 0.000004)
- String: A letter, sentence, word (e.g., "A", "hello there", "kdjfijis4527hfdj")
  → Note that strings can also contain numbers if they are mixed up with other data types

Statements and expressions

Variables or objects are assigned to some value with the = sign. For example, if I wanted the string "Hello world" to be assigned to a variable I could do so by:

```
myString = "Hello world"
```

This stores "Hello world" in the variable myString but it does nothing else to it. Now if I wanted my string to be printed to the standard output of the terminal you can call the print statement:

```
print(myString)
```

Some object types in Python

- List: a mutable (i.e., changeable) and ordered sequence of elements delineated in Python code by square brackets [ ]
- Dictionary: An object type that consists of keys that are associated with specific values. Think of a word in a dictionary as the key and the definition as the value. Values can be the same across keys but the keys themselves must be unique. Delineated by curly brackets and a colon {key:value}
- Tuple: an immutable (i.e., unchangeable) sequence of objects (very similar to a list)

Importing libraries

There are many useful libraries written in Python that are set up to manage biological data that can be imported into a script so that various modules can be used (some of my favorites include Pandas, NumPy, BioPython, MatplotLib). As many libraries in Python are not included by default you must explicitly load them into your Python environment. For example, if you wanted to use modules from the BioPython library in your script you would add the following line somewhere near the beginning of your script:

```
import Bio
```

Where import is the library importation command and Bio is the name of the BioPython module. More typically, however, you'll want to only load specific tools from the module. For example, if I want to import SeqIO, a function that can be used to load various sequence file types:

```
from Bio import SeqIO
```

Writing a function

Functions are typically short programs that perform a single or a couple tasks that can be linked together to create more complicated scripts. The basic structure of a Python function is:

```
def myFunctionName(input parameters):
	some set of conditions to run
```

Example script Hello World:

```
#!/usr/bin/python3

import sys

def main():
	if len(sys.argv) >= 2:
		name = sys.argv[1]
	else:
		print("Please enter your name:")
	print("Hello", name)
```

In this script we import a module called sys which gathers information from your system and defines a function that prints out "Hello" and the name of the person running it. From the command line this program would be run as such:

```
python3 helloWorld.py <name>
```

Python interactively with iPython

One of the nicest things about Python is it's interactive shell environment iPython. It provides an easy way to debug or test out parts of your script. It also has the extra benefit of tab completion and you can use most Unix command lines within the iPython environment. Instructions for downloading and installing iPython can be found here: http://ipython.org/install.html

Example script Parsing a GenBank File in iPython:

So let's test this out by writing a small script that converts a GenBank formatted file to a fasta formatted file. First open up a terminal and run iPython by typing

```
ipython
```

Now write the script by first importing SeqIO from BioPython

```
from Bio import SeqIO
```

In one line you can now read in your GenBank file and parse through it line by line to extract information from it in a loop:

```
for seqRecord in SeqIO.parse("my_file.gb", "genbank"):
	print(seqRecord)
```

The results of this command shows you all of the information in your GenBank file by assigning each record to the object seqRecord while parsing the file. Note that the name seqRecord is not important and you could have named this object anything you wanted – think of these variable as saying for each **element** in my file: do this where they are a random name for the element. You can pull specific fields from the GenBank file. For example, if you want to only pull the sequence ID (a.k.a. the accession number) you could replace print(seqRecord) in the above script with:

```
print(seqRecord.id)
```

If you wanted just the sequence:

```
print(seqRecord.seq)
```

In my experience the best way to start programming is to think of a task you do often and already have a solution for (e.g., pulling all of the sequence headers from a fasta file) and figure out how to do the same thing in Python. Just like any spoken language, learning computer languages requires practice and consistent use!

R

R is a statistical and graphic package. R is freely available, highly flexible, and users can take advantage of a large community of users. While R can initially be intimidating, once you learn the basic syntax and vocabulary there are very few things you cannot do with it. If you are looking to do something specific with your data, chances are someone has already written a package for it! R can be used for basic statistical tests or can be used to write complex functions

Official R link: http://cran.r-project.org/

Best R books:

Norman Matloff. The Art of R Programming. No Starch Press, 2011.

Paul Teetor. R Cookbook. O'Reilly, 2011.
Alain Zuur, Elena Ieno, & Erik Meesters. A Beginner's Guide to R. 2009

Online R learning resources:
Beginner:
Code School Try R: https://www.codeschool.com/courses/try-r
Software Carpentry R Lesson: http://swcarpentry.github.io/r-novice-inflammation/
RStudio resources: http://www.rstudio.com/resources/training/online-learning/
Intermediate/Advanced:
Advanced R: http://adv-r.had.co.nz/
Statistics Using R with Biological Examples: http://cran.r-project.org/doc/contrib/Seefeld_StatsRBio.pdf
An Introduction to R for Dynamic Models in Biology:
https://people.cam.cornell.edu/~dmb/DynamicModelsLabsInR.pdf

Using R - The Basics:
Remember, R is a statistics and graphics package, not a programming language (though the syntax resembles many programming languages). R uses an object-oriented language meaning the basic unit in R are objects.

- Object: Any input or output that you pass to R (e.g. numbers, letters, strings, etc)
- Vector: Collection of objects that are all the same type (e.g. a collection of numbers)
- Function: Set of instructions to be carried out on a single object or vector of objects
- Parameter: Information passed to a function
- Argument: Similar to a parameter, this is information for the function that tells it how to use or handle the information passed to it

Basic command structure
```
object assignment expression
```

So for example (> is command prompt, you don't type it in):
```
> mean.limb.size <- c(2,3,4,5)
> mean.limb.size
[1] 2 3 4 5
```

In this example we used an assignment operator (<-) to assign a vector of objects (2,3,4,5 -- the "c" stands for concatenate) to an object (mean.limb.size). If we call our vector again R returns a line number [1] and our numerical objects.

```
> mean.limb.size <- 2+3+4+5
> mean.limb.size
14
```

In this example instead of assigning a vector of objects to mean.limb.size we used the addition operator to assign relationships between the objects. If we call mean.limb.size again, R evaluates the expression and returns 14. Now that we have our named object mean.limb.size we might want to do something to it. To this end we can use a function. One useful function that is pre-loaded into the R

environment is summary(). This function takes an object or vector and outputs basic summary statistics about it. So for example if we have the following object:

```
> daily.attendance <- c(20, 25, 58, 108, 11, 1, 44, 46, 70)
```

We can get a summary of our data by supplying our object as an argument in the function:

```
> summary(daily.attendance)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00   20.00   44.00   42.56   58.00  108.00
```

You can see that this function finds the minimum and maximum value in your object as well as the median and mean values and values that represent the first and third quartiles.

Rules for naming objects:

- Must begin with a letter
- Cannot contain the following characters: , - + # & % [ ] { } *
- Be specific! The name of your object should tell you what your object contains!
- Object names are case sensitive. Mean.limb.size =\= mean.limb.size
- If you need to separate words in your name use "." or "_"

The R workspace:
You can run R in the command prompt or terminal or a GUI like R commander (http://www.rcommander.com), RStudio (http://www.rstudio.org), or Tinn-R (http://www.sciviews.org/Tinn-R/). Your 'workspace' refers to your current session and all of the objects/packages/etc that you are using in R. There are various commands that can help you keep track of your workspace.

`ls()`	lists all of the objects in your workspace
`rm()`	remove an object from your workspace by entering its name between the parentheses
`getwd()`	see which directory you are using with this workspace. This is important as all files you want to use/load/etc in R should be in your working directory!
`setwd()`	change your working directory (enter path to new directory in parentheses)
`library()`	tells you which packages are currently loaded into your workspace

Functions and other utilities in R are stored in what are known as packages. Any that are not included in the base R installation need to be installed and loaded to use them in your workspace.

```
install.packages(name, repository)
```

install.packages() is a function that takes two parameters: name of the desired function and the repository that you want to use to download the package. For example, to install the package car from the University of Washington repository:

```
install.packages(car, repos="http://cran.wustl.edu")
```

To get a list of all available packages to download type the install.packages function with no arguments. Once you have a package installed you need to load it into your workspace (you will need to do this every time you start a new workspace!)

```
library(car)
```

You have two options to get help with a particular function in R. If you know the exact name of the function (here we are using the car package) and want to see its help page type:

```
help(car)
```

If you are unsure of the exact function name or want to get information about a general topic/statistical test/etc. (this example would return information about using t-test in R):

```
?t-test
```

You can also request a demo of a function of utility by typing (this will only work if the person who wrote the function or package included a demo):

```
demo(function)
```

To see how awesome R graphic functions can be try:

```
demo(graphics)
```

Other programming languages used in biology

- Perl: https://www.perl.org/
- Java: https://www.java.com/en/
- C family of languages: https://isocpp.org/

Note:
Often, you'll hear references to high-level and low-level programming languages. A high-level programming languages are thought to be more similar to human languages in the sense that they are far removed from the actual language of the computer which at its lowest level is a binary system of just zeros and ones. Python, Java, Perl, and others are high-level languages while C and C++ are increasingly being described as "middle-level" languages. Languages like C and C++ are more abstract than high-level languages but are generally more memory efficient.

Reference databases

There are a plethora of biological databases, this is just a sampling of some of them and should not be considered a comprehensive list.

- EzBiocloud: https://ezbiocloud.net
  - Bacterial and Archaeal 16S rRNA and Genome database
- UNITE: https://unite.ut.ee/
  - Fungal ITS database
- SILVA: http://www.arb-silva.de/
  - Both 18S rRNA and 16S rRNA
- PR2: https://github.com/pr2database/pr2database
  - 18S rRNA focused mostly on protists
- EMBL (European Bioinformatics Institute): https://www.ebi.ac.uk/ena
  - DNA & RNA, Gene Expression, Proteins, Structural, Systems, Chemical Biology, Ontology database
- GenBank (National Center for Biotechnology Information): https://www.ncbi.nlm.nih.gov/genbank/
  - Genetic sequence database, annotated collection of all publically available DNA sequences
- Ensembl: https://useast.ensembl.org/index.html
  - Vertebrate genomes
- RGD: https://rgd.mcw.edu/
  - Rat genome database
- HPA: https://www.proteinatlas.org/
  - Human protein atlas
- miRBase: http://www.mirbase.org/
  - microRNA database
- Rfam: https://rfam.xfam.org/
  - RNA families
- UniProtKB/Swiss-Prot: https://www.uniprot.org/
  - Protein sequences
- SNPedia: https://www.snpedia.com/index.php/SNPedia
  - Human genome SNP database
- NCBI: https://www.ncbi.nlm.nih.gov/
  - Many different biomedical and genomic information

Silly things

Some fun linux commands, most of these you'll need to install yourself but they're good for procrastination!

`sl`	Choo choo
`fourtune`	Get your fortune!
`cowsay`	Get a cow to tell you your future!
`telnet towel.blinkenlights.nl`	In a galaxy far far away…..
`ruby -e 'C=`stty size`.scan(/\d+/)[1].to_i;S=["2743`	Snow day

```
".to_i(16)].pack("U*");a={};puts
"\033[2J";loop{a[rand(C)]=0;a.each
{|x,o|;a[x]+=1;print
"\033[#{o};#{x}H
\033[#{a[x]};#{x}H#{S}
\033[0;0H"};$stdout.flush;sleep
0.1}'
```