Recap From Last Time:

- **Substitution matrices**: Where our alignment match and mis-match scores typically come from.
- **Comparing methods**: The trade-off between *sensitivity*, *selectivity* and *performance*.
- **Sequence motifs and patterns**: Finding functional cues from conservation patterns.
- **Sequence profiles and position specific scoring matrices (PSSMs)**: Building and searching with profiles, their advantages and limitations.
- **PSI-BLAST algorithm**: Application of iterative PSSM searching to improve BLAST sensitivity.
- **Hidden Markov models (HMMs)**: More versatile probabilistic model for detection of remote similarities.
<table>
<thead>
<tr>
<th>Time</th>
<th>Topics</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00-9:15 AM</td>
<td>Setup and Motivation</td>
</tr>
<tr>
<td>9:15-10:00 AM</td>
<td>Beginning Unix</td>
</tr>
<tr>
<td>10:00-10:40 AM</td>
<td>Hands on session</td>
</tr>
<tr>
<td>10:40-11:00 AM</td>
<td>Working with Unix</td>
</tr>
<tr>
<td>11:00-11:20 PM</td>
<td>How to Get Working</td>
</tr>
<tr>
<td>11:20-11:45 PM</td>
<td>Hands on session</td>
</tr>
</tbody>
</table>

https://bioboot.github.io/bggn213_f17/setup/
Setup Checklist

https://bioboot.github.io/bggn213_f17/setup/

- **Mac**: Terminal  or  **PC**: MoblXterm

- **PC**: MoblXterm **CygUtils** plugin (core UNIX tools)

- Downloaded our class specific **jetstream keyfile**
  (See email: This is required for connecting to jetstream virtual machines with your terminal.)

- Example data downloaded:  **bggn213_01_unix.zip**
Let's get started...

Mac Terminal

PC MobaXterm
SideNote: Terminal vs Shell

- **Shell**: A command-line interface that allows a user to interact with the operating system by typing commands.
- **Terminal [emulator]**: A graphical interface to the shell (i.e. the window you get when you launch MobaXterm).

## Setting Up
- **Mac users**: open a Terminal
- **Windows users**: install MobaXterm and then open a terminal Shell prompt
Motivation

Why do we use Unix?
<table>
<thead>
<tr>
<th>Modularity</th>
<th>Core programs are modular and work well with others</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programmability</td>
<td>Best software development environment</td>
</tr>
<tr>
<td>Infrastructure</td>
<td>Access to existing tools and cutting-edge methods</td>
</tr>
<tr>
<td>Reliability</td>
<td>Unparalleled uptime and stability</td>
</tr>
<tr>
<td>Unix Philosophy</td>
<td>Encourages open standards</td>
</tr>
<tr>
<td>Modularity</td>
<td>Core programs are modular and work well with others</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------</td>
</tr>
<tr>
<td>Programmability</td>
<td>Best software development environment</td>
</tr>
<tr>
<td>Infrastructure</td>
<td>Access to existing tools and cutting-edge methods</td>
</tr>
<tr>
<td>Reliability</td>
<td>Unparalleled uptime and stability</td>
</tr>
<tr>
<td>Unix Philosophy</td>
<td>Encourages open standards</td>
</tr>
</tbody>
</table>
Modularity

The Unix shell was designed to allow users to easily build complex workflows by interfacing smaller modular programs together.

wget → awk → grep → sort → uniq → plot

An alternative approach is to write a single complex program that takes raw data as input, and after hours of data processing, outputs publication figures and a final table of results.

All-in-one custom ‘Monster’ program
Which would you prefer and why?

Modular vs Custom
The ‘monster approach’ is customized to a particular project but results in massive, fragile and difficult to modify (therefore inflexible, untransferable, and error prone) code.

With modular workflows, it’s easier to:

• **Spot errors** and figure out where they’re occurring by inspecting intermediate results.

• **Experiment** with alternative methods by swapping out components.

• **Tackle novel problems** by remixing existing modular tools.
Unix ‘Philosophy’

“Write programs that do one thing and do it well. Write programs to work together and that encourage open standards. Write programs to handle text streams, because that is a universal interface.”

— Doug McIlory
Unix family tree [1969-2010]

Source: https://commons.wikimedia.org/wiki/File:Unix_history-simple.svg
<table>
<thead>
<tr>
<th>Basics</th>
<th>File Control</th>
<th>Viewing &amp; Editing Files</th>
<th>Misc. useful</th>
<th>Power commands</th>
<th>Process related</th>
</tr>
</thead>
<tbody>
<tr>
<td>ls</td>
<td>mv</td>
<td>less</td>
<td>chmod</td>
<td>grep</td>
<td>top</td>
</tr>
<tr>
<td>cd</td>
<td>cp</td>
<td>head</td>
<td>echo</td>
<td>find</td>
<td>ps</td>
</tr>
<tr>
<td>pwd</td>
<td>mkdir</td>
<td>tail</td>
<td>wc</td>
<td>sed</td>
<td>kill</td>
</tr>
<tr>
<td>man</td>
<td>rm</td>
<td>nano</td>
<td>curl</td>
<td>sudo</td>
<td>Ctrl-c</td>
</tr>
<tr>
<td>ssh</td>
<td>I (pipe)</td>
<td>touch</td>
<td>source</td>
<td>git</td>
<td>Ctrl-z</td>
</tr>
<tr>
<td>scp</td>
<td>&gt; (write to file)</td>
<td>cat</td>
<td>R</td>
<td>bg</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; (read from file)</td>
<td>tmux</td>
<td>python</td>
<td>fg</td>
<td></td>
</tr>
<tr>
<td>Basics</td>
<td>File Control</td>
<td>Viewing &amp; Editing Files</td>
<td>Misc. useful</td>
<td>Power commands</td>
<td>Process related</td>
</tr>
<tr>
<td>--------</td>
<td>--------------</td>
<td>-------------------------</td>
<td>--------------</td>
<td>----------------</td>
<td>----------------</td>
</tr>
<tr>
<td>ls</td>
<td>mv</td>
<td>less</td>
<td>chmod</td>
<td>grep</td>
<td>top</td>
</tr>
<tr>
<td>cd</td>
<td>cp</td>
<td>head</td>
<td>echo</td>
<td>find</td>
<td>ps</td>
</tr>
<tr>
<td>pwd</td>
<td>mkdir</td>
<td>tail</td>
<td>wc</td>
<td>sed</td>
<td>kill</td>
</tr>
<tr>
<td>man</td>
<td>rm</td>
<td>nano</td>
<td>curl</td>
<td>sudo</td>
<td>Crl-c</td>
</tr>
<tr>
<td>ssh</td>
<td>l (pipe)</td>
<td>touch</td>
<td>source</td>
<td>git</td>
<td>Crl-z</td>
</tr>
<tr>
<td>scp</td>
<td>&gt;</td>
<td>write to file</td>
<td>cat</td>
<td>R</td>
<td>bg</td>
</tr>
<tr>
<td></td>
<td>&lt;</td>
<td>read from file</td>
<td>tmux</td>
<td>python</td>
<td>fg</td>
</tr>
</tbody>
</table>
Let's get started...

Mac Terminal

PC MobaXterm
Beginning Unix
Getting started with basic Unix commands

Download the example data and mv to your Desktop
bggn213_01_unix.zip
Information in the file system is stored in files, which are stored in directories (folders). Directories can also store other directories, which forms a directory tree.

The forward slash character ‘/’ is used to represent the root directory of the whole file system, and is also used to separate directory names. E.g. `/home/jono/work/bggn213_notes.txt`
## Basics: Using the filesystem

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ls</code></td>
<td>List files and directories</td>
</tr>
<tr>
<td><code>cd</code></td>
<td>Change directory (i.e. move to a different ‘folder’)</td>
</tr>
<tr>
<td><code>pwd</code></td>
<td>Print working directory (which folder are you in)</td>
</tr>
<tr>
<td><code>mkdir</code></td>
<td>Make a new directories</td>
</tr>
<tr>
<td><code>cp</code></td>
<td>Copy a file or directory to somewhere else</td>
</tr>
<tr>
<td><code>mv</code></td>
<td>Move a file or directory (basically rename)</td>
</tr>
<tr>
<td><code>rm</code></td>
<td>Remove a file or directory</td>
</tr>
</tbody>
</table>
**Side Note: File Paths**

- **An absolute path** specifies a location from the root of the file system. E.g. `/home/jono/work/bggn213_notes.txt`

- **A relative path** specifies a location starting from the current location. E.g. `./bggn213_notes.txt`

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Single dot ‘.’ (for current directory)</td>
</tr>
<tr>
<td>..</td>
<td>Double dot ‘..’ (for parent directory)</td>
</tr>
<tr>
<td>~</td>
<td>Tilda ‘~’ (for your home directory)</td>
</tr>
<tr>
<td>[Tab]</td>
<td>Pressing the tab key can autocomplete names</td>
</tr>
</tbody>
</table>
Finding the Right Hammer

**(man and apropos)**

- You can access the manual (i.e. user documentation) on a command with **man**, e.g:
  
  ```
  > man pwd
  ```

- The man page is only helpful if you know the name of the command you’re looking for. **apropos** will search the man pages for keywords.

  ```
  > apropos "working directory"
  ```
Inspecting text files

- **less** - visualize a text file:
  - use arrow keys
  - page down/page up with "space"/"b" keys
  - search by typing "/"
  - quit by typing "q"

- Also see: **head, tail, cat, more**
Creating text files

Creating files can be done in a few ways:

• With a **text editor** (such as `nano`, `emacs`, or `vi`)
• With the **touch** command (`touch a_file`)
• From the command line with **cat** or **echo** and **redirection** (more on this later)

• **nano** is a simple text editor that is recommended for first-time users. Other text editors have more powerful features but also steep learning curves
Creating and editing text files with **nano**

In the terminal type:

```
> nano yourfilename.txt
```

- There are many other text file editors (e.g. **vim**, **emacs** and **sublime text**, etc.)
Connecting to remote machines (with `ssh`)

- Most high-performance computing (HPC) resources can only be accessed by `ssh` (Secure SHell)

  ```
  > ssh [user@host.address]
  ```

For example:

  ```
  > ssh barry@bio3d.ucsd.edu
  > ssh tb170077@IP_ADDRESS
  ```
Connecting to jetstream (with ssh)

• First we have to login online and fire-up a new “virtual machine”

• Full instructions here: https://bioboot.github.io/bgg213_f17/jetstream/boot/

• We will go through this process in the last section today!
Copying to and from remote machines (**scp**)

- The **scp** (**Secure CoPy**) command can be used to copy files and directories from one computer to another.

  ```
  > scp [file] [user@host]:[destination]
  > scp localfile.txt  bgrant@bigcomputer.net:/remotedir/.
  ```
<table>
<thead>
<tr>
<th>Basics</th>
<th>File Control</th>
<th>Viewing &amp; Editing Files</th>
<th>Misc. useful</th>
<th>Power commands</th>
<th>Process related</th>
</tr>
</thead>
<tbody>
<tr>
<td>ls</td>
<td>mv</td>
<td>less</td>
<td>chmod</td>
<td>grep</td>
<td>top</td>
</tr>
<tr>
<td>cd</td>
<td>cp</td>
<td>head</td>
<td>echo</td>
<td>find</td>
<td>ps</td>
</tr>
<tr>
<td>pwd</td>
<td>mkdir</td>
<td>tail</td>
<td>wc</td>
<td>sed</td>
<td>kill</td>
</tr>
<tr>
<td>man</td>
<td>rm</td>
<td>nano</td>
<td>curl</td>
<td>sudo</td>
<td>Crl-c</td>
</tr>
<tr>
<td>ssh</td>
<td></td>
<td>touch</td>
<td>source</td>
<td>git</td>
<td>Crl-z</td>
</tr>
<tr>
<td>scp</td>
<td>&gt;</td>
<td></td>
<td>cat</td>
<td>R</td>
<td>bg</td>
</tr>
<tr>
<td></td>
<td>&lt;</td>
<td></td>
<td>tmux</td>
<td>python</td>
<td>fg</td>
</tr>
<tr>
<td></td>
<td>(pipe)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(write to file)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(read from file)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Process** refers to a running instance of a program

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>top</strong></td>
<td>Provides a real-time view of all running processes</td>
</tr>
<tr>
<td><strong>ps</strong></td>
<td>Report a snapshot of the current processes</td>
</tr>
<tr>
<td><strong>kill</strong></td>
<td>Terminate a process (the “force quit” of the unix world)</td>
</tr>
<tr>
<td><strong>Crl-c</strong></td>
<td>Stop a job</td>
</tr>
<tr>
<td><strong>Crl-z</strong></td>
<td>Suspend a job</td>
</tr>
<tr>
<td><strong>bg</strong></td>
<td>Resume a suspended job in the background</td>
</tr>
<tr>
<td><strong>fg</strong></td>
<td>Resume a suspended job in the foreground</td>
</tr>
<tr>
<td><strong>&amp;</strong></td>
<td>Start a job in the background</td>
</tr>
</tbody>
</table>
Hands-on time

Sections 1 to 3 of software carpentry UNIX lesson

https://swcarpentry.github.io/shell-novice/

~20 mins
Working with Unix

How do we actually use Unix?
<table>
<thead>
<tr>
<th>Basics</th>
<th>File Control</th>
<th>Viewing &amp; Editing Files</th>
<th>Misc. useful</th>
<th>Power commands</th>
<th>Process related</th>
</tr>
</thead>
<tbody>
<tr>
<td>ls</td>
<td>mv</td>
<td>less</td>
<td>chmod</td>
<td>grep</td>
<td>top</td>
</tr>
<tr>
<td>cd</td>
<td>cp</td>
<td>head</td>
<td>echo</td>
<td>find</td>
<td>ps</td>
</tr>
<tr>
<td>pwd</td>
<td>mkdir</td>
<td>tail</td>
<td>wc</td>
<td>sed</td>
<td>kill</td>
</tr>
<tr>
<td>man</td>
<td>rm</td>
<td>nano</td>
<td>curl</td>
<td>sudo</td>
<td>Crl-c</td>
</tr>
<tr>
<td>ssh</td>
<td>l (pipe)</td>
<td>touch</td>
<td>source</td>
<td>git</td>
<td>Crl-z</td>
</tr>
<tr>
<td></td>
<td>&gt; (write to file)</td>
<td>cat</td>
<td>R</td>
<td>bg</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; (read from file)</td>
<td>python</td>
<td>fg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Combining Utilities with Redirection (>, <) and Pipes (|)

• The power of the shell lies in the ability to combine simple utilities (i.e. commands) into more complex algorithms very quickly.

• A key element of this is the ability to send the output from one command into a file or to pass it directly to another program.

• This is the job of >, < and |
Two very important concepts that unpin Unix workflows:

• **Standard Output** (**stdout**) - default destination of a program's output. It is generally the terminal screen.

• **Standard Input** (**stdin**) - default source of a program's input. It is generally the command line.
Output redirection and piping

> ls /usr/bin  # stdin is "/usr/bin"; stdout to screen
> `ls /usr/bin`  # stdin is "/usr/bin"; stdout to screen

> `ls /usr/bin > binlist.txt`  # stdout **redirected** to file

> `ls /usr/bin | less`  # stdout **piped** to less (no file created)
Output redirection and piping

> `ls /usr/bin`  # stdin is “/usr/bin”; stdout to screen

> `ls /usr/bin > binlist.txt`  # stdout redirected to file

> `ls /usr/bin | less`  # stdout piped to less (no file created)

> `ls -l /usr/bin`  # extra optional input argument “-l”
Output redirection and piping

```plaintext
> ls /usr/bin  # stdin is "/usr/bin"; stdout to screen

> ls /usr/bin > binlist.txt  # stdout redirected to file

> ls /usr/bin | less  # stdout piped to less (no file created)

> ls /nodirexists/  # stderr to screen

> ls /nodirexists/  # stderr to screen
```
Output redirection and piping

> `ls /usr/bin`  # stdin is "/usr/bin"; stdout to screen

> `ls /usr/bin > binlist.txt`  # stdout redirected to file

> `ls /usr/bin | less`  # stdout piped to less (no file created)

> `ls /nodirexists/ > binlist.txt`  # stderr to screen
Output redirection and piping

> `ls /usr/bin`  # stdin is "/usr/bin"; stdout to screen

> `ls /usr/bin > binlist.txt`  # stdout **redirected** to file

> `ls /usr/bin | less`  # stdout **piped** to less (no file created)

> `ls /nodirexists/ 2> binlist.txt`  # stderr to **file**
Output redirection summary

stdin --- program --- stdout

-arg --- args --- stderr

I <|> disk

I <|> network

X11
ls -l
`ls -l > list_of_files`
We have piped (`|`) the stdout of one command into the stdin of another command!
ls -l /usr/bin/ | grep "tree" > list_of_files

**grep**: prints lines containing a string. Also searches for strings in text files.
<table>
<thead>
<tr>
<th>Basics</th>
<th>File Control</th>
<th>Viewing &amp; Editing Files</th>
<th>Misc. useful</th>
<th>Power commands</th>
<th>Process related</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ls</strong></td>
<td>mv</td>
<td>less</td>
<td>chmod</td>
<td>grep</td>
<td>top</td>
</tr>
<tr>
<td>cd</td>
<td>cp</td>
<td>head</td>
<td>echo</td>
<td>find</td>
<td>ps</td>
</tr>
<tr>
<td><strong>pwd</strong></td>
<td>mkdir</td>
<td>tail</td>
<td>wc</td>
<td>sed</td>
<td>kill</td>
</tr>
<tr>
<td><strong>man</strong></td>
<td>rm</td>
<td>nano</td>
<td>curl</td>
<td>sudo</td>
<td>CrI-c</td>
</tr>
<tr>
<td>ssh</td>
<td>l (pipe)</td>
<td>touch</td>
<td>source</td>
<td>git</td>
<td>CrI-z</td>
</tr>
<tr>
<td></td>
<td>&gt; (write to file)</td>
<td>cat</td>
<td>R</td>
<td>bg</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; (read from file)</td>
<td>python</td>
<td>fg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Side-Note: grep ‘power command’

- grep - prints lines containing a string pattern. Also searches for strings in text files, e.g.

  > grep --color "GESGKS" sequences/data/seqdump.fasta

  REVKLLLLLGAGESGKSTIVKQMKIIHEAGYSEECECKQYK

- grep is a ‘power tool’ that is often used with pipes as it accepts regular expressions as input (e.g. “G..GK[ST]”) and has lots of useful options - see the man page for details.
grep example using regular expressions

Suppose a program that you are working with complains that your input sequence file contains non-nucleotide characters. You can eye-ball your file or …

```
> grep -v "^>" seqdump.fasta | grep --color "[^ATGC]"
```

**Exercises:**

1. Use “man grep” to find out what the -v argument option is doing!

2. How could we also show line number for each match along with the output?
   (tip you can grep the output of “man grep” for ‘line number’)

Do it Yourself!
grep example using regular expressions

- Suppose a program that you are working with complains that your input sequence file contains non-nucleotide characters. You can eye-ball your file or …

```
grep -v "^>" seqdump.fasta | grep --color -n "[^[ATGC]]"
```

- First we remove (with 
-v option) lines that start with a “>” character (these are sequence identifiers).

- Next we find characters that are not A, T, C or G. To do this we use ^ symbols second meaning: match anything but the pattern in square brackets. We also print line number (with -n option) and color output (with --color option).
Key Point: Pipes and redirects avoid unnecessary i/o

- Disc i/o is often a bottleneck in data processing!

- Pipes prevent unnecessary disc i/o operations by connecting the stdout of one process to the stdin of another (these are frequently called “streams”)

  ```
  > program1 input.txt 2> program1.stderr | \n  program2 2> program2.stderr > results.txt
  ```

- Pipes and redirects allow us to build solutions from modular parts that work with stdin and stdout streams.
Unix ‘Philosophy’ Revisited

“Write programs that do one thing and do it well. Write programs to work together and that encourage open standards. Write programs to handle text streams, because that is a universal interface.”

— Doug McIlory
Pipes provide speed, flexibility and sometimes simplicity...

• In 1986 “Communications of the ACM magazine” asked famous computer scientist Donald Knuth to write a simple program to count and print the $k$ most common words in a file alongside their counts, in descending order.

• Kunth wrote a literate programming solution that was 7 pages long, and also highly customized to this problem (e.g. Kunth implemented a custom data structure for counting English words).

• Doug McIlroy replied with one line:

   > cat input.txt | tr A-Z a-z | sort | uniq -c | sort -rn | sed 10q
Key Point:

You can chain any number of programs together to achieve your goal!

This allows you to build up fairly complex workflows within one command-line.
Hands-on time

Section 4 of software carpentry UNIX lesson

https://swcarpentry.github.io/shell-novice/

~15 mins
Shell scripting

#!/bin/bash
# This is a very simple hello world script.
echo "Hello, world!"

Exercise:
• Create a "Hello world"-like script using command line tools and execute it.

• Copy and alter your script to redirect output to a file using > along with a list of files in your home directory.

• Alter your script to use >> instead of >. What effect does this have on its behavior?
Variables in shell scripts

```
#!/bin/bash
# Another simple hello world script
message='Hello World!'
echo $message
```

- “message” - is a **variable** to which the string 'Hello World!' is assigned
- echo - prints to screen the contents of the variable "$message"
Side-Note: Environment Variables

- stdin
- stdout
- args
- stderr

Environment Variables

I/O

disk

X11

network
$PATH ‘special’ environment variable

• What is the output of this command?

> echo $PATH

• Note the structure: <path1>:<path2>:<path3>

• PATH is an environmental variable which Bash uses to search for commands typed on the command line without a full path.

• Exercise: Use the command env to discover more.
Summary

• Built-in unix shell commands allow for easy data manipulation (e.g. sort, grep, etc.)

• Commands can be easily combined to generate flexible solutions to data manipulation tasks.

• The unix shell allows users to automate repetitive tasks through the use of shell scripts that promote reproducibility and easy troubleshooting

• Introduced the 21 key unix commands that you will use during ~95% of your future unix work…
<table>
<thead>
<tr>
<th>Basics</th>
<th>File Control</th>
<th>Viewing &amp; Editing Files</th>
<th>Misc. useful</th>
<th>Power commands</th>
<th>Process related</th>
</tr>
</thead>
<tbody>
<tr>
<td>ls</td>
<td>mv</td>
<td>less</td>
<td>chmod</td>
<td>grep</td>
<td>top</td>
</tr>
<tr>
<td>cd</td>
<td>cp</td>
<td>head</td>
<td>echo</td>
<td>find</td>
<td>ps</td>
</tr>
<tr>
<td>pwd</td>
<td>mkdir</td>
<td>tail</td>
<td>wc</td>
<td>sed</td>
<td>kill</td>
</tr>
<tr>
<td>man</td>
<td>rm</td>
<td>nano</td>
<td>curl</td>
<td>sudo</td>
<td>Ctrl-c</td>
</tr>
<tr>
<td>ssh</td>
<td><code>l</code> (pipe)</td>
<td>touch</td>
<td>source</td>
<td>git</td>
<td>Ctrl-z</td>
</tr>
<tr>
<td></td>
<td><code>&gt;</code> (write to file)</td>
<td>cat</td>
<td></td>
<td>R</td>
<td>bg</td>
</tr>
<tr>
<td></td>
<td><code>&lt;</code> (read from file)</td>
<td>python</td>
<td></td>
<td></td>
<td>fg</td>
</tr>
</tbody>
</table>
How to Get Working

1) Connecting to jetstream

2) Best practices for organizing your computational biology projects
http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1000424

All files and directories used in your project should live in a single project directory.

- Use sub-directories to divide your project into sub-projects.
- Do not use spaces in file and directory names!

Document your methods and workflows with plain text README files

- Also document the origin of all data in your project directory
- Also document the versions of the software that you ran and the options you used.
- Consider using Markdown for your documentation.

Use version control and backup to multiple destinations!

Be reproducible:
http://ropensci.github.io/reproducibility-guide/sections/introduction/