Second Unix Lesson

Review

Last time we:

1. Opened the terminal on our local machine
2. Learned about `BASH` and the difference between `GUI` and `CLI`
3. Talked about file / folder structures (e.g. how paths are indicated hierarchically with each level being separated by a `/`)
4. Discussed the difference between
5. Tested out some commands for navigating around our file structures via the command line and figuring out who we are:
   - `ls`: list files in current location
   - `pwd`: print working directory (where am I?)
   - `cd`: change directory (moving from place to place)
   - `whoami`: print username
   - `unzip`: unzip a file compressed with zip function

Goals for today:

Today we will:

1. Get an introduction to HPCs in general and the HPC we will be using for this class (`poseidon.whoi.edu`)
2. Discuss remote machine access (`ssh`) and log on to the HPC
3. Review the commands we learned at the end of last class
4. Learn to work with files and directories

5. Learn about wildcards and regex
6. Learn how to deal with outputs & string commands things together

Logging on to the HPC

For this class (and all subsequent classes) we will be using WHOI's HPC: `poseidon.whoi.edu`. This is a remote cluster of computers (it isn't really *that* remote.. it is in the basement of Clark). Nearly all command line systems (BASH etc.) have Secure Shell (`ssh`) natively installed. `ssh` is a cryptographic network protocol that allows you to provides a secure channel over an unsecured network. `ssh` can be used to log on to any number of platforms such as: remote computers (like a lab computer), computer clusters or high performance computers (HPCs), computers running in the cloud (e.g. AWS), etc.

Today we will be using `ssh` to logon to `poseidon.whoi.edu`. To do this you will need to know what your WHOI username is and be within WHOI's firewall either by connecting to the local network (e.g. `Arctic`) or by logging into WHOI's vpn.

Once you are logged on to WHOI's network you should be ready to `ssh`. In your terminal prompt type the following:

```
ssh username@poseidon.whoi.edu
```

If this is your first time logging onto the network you will see a prompt like the following:

```
    Host key not found from the list of known hosts.
    Are you sure you want to continue connecting (yes/no)?
```

This is a safety protocol ensuring that you do indeed want to add this address to a list of known hosts. Type `yes`.

You should now be prompted for your password (this will be your WHOI email password):

```
Host 'poseidon.whoi.edu' added to the list of known hosts.
```

```
        [usernames]'s password:
```

Once you type the correct password you should see something like the following:

```
  ___                _     _                ___ _          _
 | _ \___ ___ ___(_)__| |___ _ _    / __| |_  _ __| |_ ___ _ _
 |  _/ _ (_-</ -_) / _` / _ \ ' \  | (__| | || (_-<  _/ -_) '_|
 |_| \___/__/\___|_\__,_\___/_||_|  \___|_|\_,_/__/\__\___|_|

Welcome to the Poseidon Cluster at WHOI!

Please remember to copy your files to scratch and move/delete them after each jo
b.
Please do not run anything on the login nodes and submit jobs to SLURM. All runn
ing
jobs/processes on the login nodes will be terminated without notice.
```

Congratulations! You have logged on to the HPC! This terminal now represents the environment of the remote computer that you just logged on to.

> Look at your `prompt`. Has it changed? What information do you see now? What do the different parts mean? Hint: try using the command `whoami` and the command `hostname`.

Navigating to our classroom directory

For this class we have set up a special workspace on `poseidon`. This is where you will do all your homework and projects.

> Navigate to `/vortexfs1/omics/env-bio`. What folders do you see? Hint: if you are typing out the path provided rather than copying it, try hitting the `tab` key in the middle of the word or phrase, this should automatically complete the phrase for you *or* if there is more than one option repeatedly hitting `tab` will bring up a listing of all options.

You should see a folder within `/vortexfs1/omics/env-bio/` called `users/`. This directory holds many subdirectories that we will be using in the class.

> Run `ls users`. What do you see?

Now, navigate into your user directory. This will be your computational space for the rest of the class. Only you and the instructors have access to your directory.

> Try navigating into someone else's directory. What happens? Why? Try `ls -l` to figure it out.

This is a nice example of how file permissions can be used to structure access to a file system. The letters you see next to the file indicate the permissions (read, write, execute) that are given to different groups from left to right (owner, group, all). You can **actively** use file permissions to protect original data (we will come back to this later). We will also learn how to change those permissions later.

Getting ready!

Now, if for some reason you have moved away, navigate back to your class user directory. Use `ls` to check the contents of the folder. You should see a file called `master.zip`. This contains the play file structure we are going to work with today.

Let's unzip it. Unlike last class where you had the option of unzipping `master.zip` through your computer's GUI file navigator (`Finder` or the like), on the HPC you DO NOT have access to any sort of GUI and you must do everything via the `CLI`. To unzip the file use the command `unzip master.zip`. This should print a lot of expanding and printing statements. Once your prompt returns

> What happens if you type `unzip` on its own?

Now, navigate into the newly created folder `unix-folders-master/`.

> How can you tell if a listing is a folder?

Working with files and directories

Run `ls -F`. How many directories are there in this level?

> **Exercise:** Let's practice working on navigating around the file structure. There is a folder called `cavity/` hidden somewhere within this directory. Go find it. Once you have found it– put up a sticky note.

Creating directories

Now, let's navigate back to `unix-folders-master/`. In this directory we are going to make a new directory called `observations`. To do this, run the command `mkdir observations`.

> Run `ls -F` again. What has changed since the first time? Try running `ls -F observations`. What do you see?

Since we just made `observations` there is nothing inside of it– let's change that!

Creating a file

Move into the directory `observations` and let's create a file. There are (unsurprisingly) many ways to do this. For now, let's use a text editor. During this class we will be using `nano` as it one of the easiest to learn. There are many other more powerful command line text editors (e.g. `vim`, `emacs`, etc.). If you are comfortable with one of those feel free to use it– otherwise stick with nano.

To create a new text file simply type `nano [name of file]`, for example:

```
nano 2019-observation.txt
```

This will create a new file called `2019-observation.txt` and you should be automatically be entered into the nano environment. Add some text to the file. And then press `ctrl+O`. This will write the text to the output file. Once you are satisfied you can press `ctrl+X` to exit. Note that the bottom of the nano window lists other `ctrl` commands that you might want to use with the program.

> We just made a file with nano. Now investigate the command `touch`. Try typing `touch 2018-observation.txt`. What happened? How is it different from nano? Try looking at its size.
>
> > **Question for thought:** If you make a text file does it need to end in `.txt`?

Looking at files (without editing them)

Often we want to get a quick look at the contents of a file without potentially accidentally modifying it. For that, the command `less` is quite useful. Try typing `less 2019-observation.txt`. This should bring you into an interface where you can read and scroll but cannot edit the document. Type `Q` to exit. Take a look at both files in this directory.

Another convenient way to examine the content of a file is to print it to the screen or print it to `stdout` or Standard Output. The program `cat` or concatenate will print the entire contents of a file to the screen. Try it out by typing cat `2019-observations.txt`.

> How is this different than less?

Moving and renaming

One of the most common things you will likely find yourself doing is managing and reorganizing files in your projects. Let's go to the folder called `unix-folders-master/writing/drafts/`. Here we have many different versions of a paper that we are struggling to write.

First, let's make a new directory called `old-drafts`. We are going to use the command `mv` (move) to move files on the command line. `mv` takes two arguments: the first is the target file that you want to move, the second is destination or the location that you want to move it to. Let's move `paper-v1.txt` into `old-drafts/`.

The command `mv` can also be used to rename files. Let's rename `paper-final2.txt` to `paper-DONE.txt`. To do this, your destination is the new name of the file.

```
mv paper-final2.txt paper-DONE.txt
```

> Run `ls`. What has happened?

`mv` can also be used with directories. For example, you can change the name of the directory `old-drafts` to `extremely-old-drafts`.

A word of caution: You can *very* easily overwrite files using `mv`. Let's check this out. What happens if you run:

```
mv paper-v4.txt paper-v3.txt
```

Do you have the same number of files?

> Write a single command to move `paper-v2.txt` to the directory `unix-folders-master/observations/` *and* rename it `initial-observations`. Now, navigate to `unix-folders-master/observations/` and try copying `paper-v3.txt` to the

> location where you are without moving.

Copying

The command `cp` (copy) works very similarly to `mv`. As with `mv` it requires two commands (the target file you want to copy and the destination/name of the new file). Return to `unix-folders-master/writing/drafts/`. Let's try to create the files that we had previously by copying our final drafts.

```
cp paper-final.txt paper-v1.txt
cp paper-final.txt paper-v2.txt
cp paper-final.txt paper-v3.txt
```

> `cp` can also be used with directories. Try copying `extremely-old-drafts` to a new directory called `very-old-drafts`. What happened? Use `man` to figure out if there is a flag that can help you.

Removing things

On the command line… removal is permanent.
Removing with the command `rm` just as `cp` and `mv` can take a target file or a list of more than file. So, for example `rm paper-v1.txt` removes `paper-v1.txt`. If you run `ls` again you will note that it is gone. And it is *truly gone*. Short of having backed things up– there is *no way* to get a `rm`-ed file back. A flag that I like to add (or have aliased) is `rm -i` for interactive. Let's try `rm -i` to remove `paper-v2.txt`.

The danger zone: You can also use `rm` to remove directories and this is where you *really* need to be careful. Using the flag `rm -r` will recursively remove all files within a directory. I strongly suggest (especially if you are new the the command line) using the `-i` flag. Note: `rmdir` is another option for removing directories – the default requires that the directory be empty.

Wildcards and Regular Expressions

Wildcards

Wildcards, such as `*` are used to identify multiple files or folders that match a requested pattern. `*` matches zero or more characters and is often used to grab groups of files. Navigate to the folder `unix-folders-master/data/` and type `ls`. If we wanted to see what files ended in `.out`, we can use the wildcard `*`. For example, `ls *.out` will return any file name that ends in the pattern `.out`.

> Use `*` to list all files that end in `.sh`.

A note on usage: When the shell sees a wildcard, it expands the wildcard to create a list of matching filenames before running the command. If a wildcard expression does not match any file, Bash will pass the expression as an argument to the command as it is. For example typing `ls *.pdf` here will result in an error message that there is no file called `*.pdf`. Ultimately, it is the shell, not the other programs, that deals with expanding wildcards, and this is another example of orthogonal design.

Navigate to `unix-folders-master/measurements/`. This folder contains a bunch of different measurement files. Use `less` to take a look at one of them.

> Use `*` to print all the measurements that *end in* the number 5.
> Use `*` to print all the measurements that *contain* the number 5.
> Use `*` to print all the measurements whose number *starts with* the number 5.

Regular expressions

Regular expressions (abbreviated regex) are a concept used in many different programming languages for sophisticated pattern matching. When used well, they can be a very powerful tool to help you find and transform your data and files.

A regular expression is a sequence of characters used to define a search to match strings (think find and replace in Word). A ‘string’ is a contiguous sequence of symbols or values. For example, a word, a date, a set of numbers (e.g., a phone number), or an alphanumeric value (e.g., an identifier). The wildcard `*` we just learned about is taken from regular expressions, but there are many more features to the complete regular expressions syntax:

- Match on types of characters (e.g. upper case letters, digits, spaces, etc.).
- Match patterns that repeat any number of times
- Capture the parts of the original string that match your pattern

Regular expressions look at both the actual charter and the literal characters and `metacharacter`. A metacharacter is any American Standard Code for Information Interchange (ASCII) character that has a special meaning. Using these together, you can construct a regex for finding strings or files that match a pattern rather than a specific string.

Common regex metacharacters

Square brackets define a list or range of characters to be found:

- `[ABC]` matches A or B or C.
- `[A-Z]` matches any upper case letter.
- `[A-Za-z]` matches any upper or lower case letter.
- `[A-Za-z0-9]` matches any upper or lower case letter or any digit.
  And then some special commands are :
- `.` matches any character
- `\d` matches digits
- `\w` matches any word character (i.e. not spaces)
- `\s` matches white space (space, tab new line etc.)
- `\` is a general ”escape character`
  And then there are positional commands:
- `^` means that the position must be at the start of the line
- `$` means it must appear at the end of the line

> So, what is `^[Aa]naly.e` going to match?

Now, let's use it like we were using the wildcard `*`.

> **Exercises**
> Write a command that uses regex to list all files that contain a 5 or a 6 somewhere within their name.
> Write a command that lists all files that lists all files that end in 3, 7 or 8.
> Write a command that returns all files that contain a 2 followed by a 1 or an 8.

These are just the basics. If you are wanting to learn more about regex I recommend checking out:

- https://regexone.com/ : a great interactive online learning tool.
- https://regexr.com/ : a useful regex tester
- https://regexcrossword.com/ : for the regex fans who have too much time on their hands.

Outputs & string commands things together

Alright, so we can now navigate around file structures with the command line, make files and directories, copy things, remove things, rename things. Great! These are all things we could arguably do just as easily with your GUI interfaces. The real power of shell comes from our ability to string commands together to do something greater.

Navigate into the directory `data`. Type `ls` to familiarize yourself with what we are doing. Again let's use `ls` to look at the files. There are many data files that end in `.out`. Take a look at one of with `less`.

We are going to use a new command now called `wc` which counts the lines, words, and characters in a file.

Run `wc *out`. This is printing information to standard out `stdout`. This is the default output from many programs– it prints an answer for you on the command line. This answer, however, is not saved anywhere. Within command line we can actively redirect outputs to save it into a file or to pass the output into a new function. Let's check out what that looks

like.

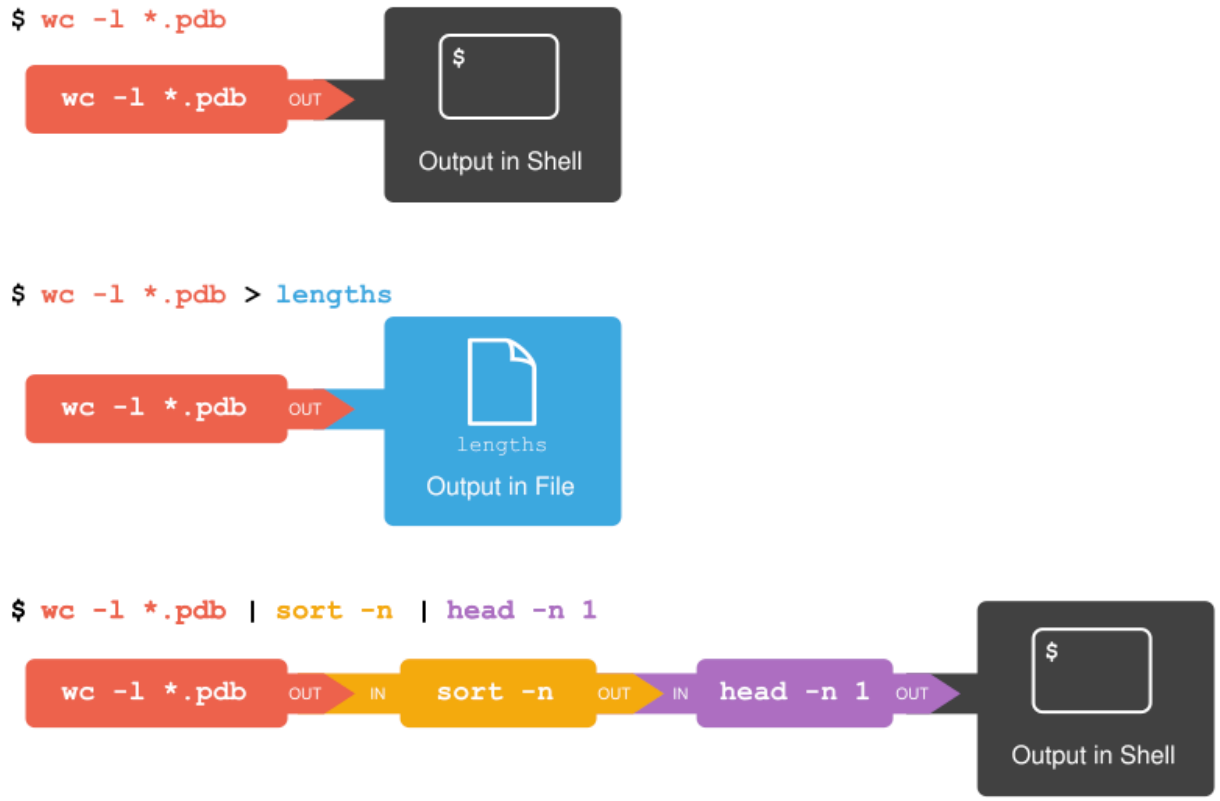


Image from Software Carpentry.

Let's run `wc -l *out` (this only counts the lines for each of the files. We can now use the `>` to pass this output to a file. Let's make a file called `lengths`.

```
wc -l *out > lengths
```

> What happened? Was anything printed to standard out?
>
> > As an aside: `>` writes to a new file. It will overwrite anything present within a file. `>>` by contrast can be used to *append* to an existing file by adding the output to the end of an existing file.

What is more is you can actually pass out put to other programs. Let's introduce briefly a couple very useful programs for looking at and dealing with files:

- `wc` : counts the words in the file
- `sort` : sort the contents of a file
- `uniq` : returns only unique values from a file. Requires the file has been sorted
- `head` : returns the head (top) of the file
- `tail` : returns the tail (bottom) of the file
- `cut` : cuts a column of interest based on whatever the delimiter is
- `paste` : pastes columns together
- `cat` : in addition to printing a file to screen it can be used to concatenate files together
- Any other class favorites?

These are some of my most used text handling functions that I use. All of them default to outputting to `stdout` and so can be easily piped together. [Note: `sed`, `grep`, and `find` are still to come– they deserve their own special attention.]

So, let's try piping the output of `wc -l` into a new function. Let's try sort so that we can figure out what file has the most lines. To pass the output of one file to the next we use the pipe character `|` (located above the `\` on American keyboards). So, we will pass the output of `wc -l` to `sort -n` to get a list of the files sorted based on the numeric value.

```
wc -l *out|sort -n
```

Now we are looking at a sorted list. We can then programmatically identify the shortest command by piping the output of the above to another command `head -n 1` which will return only the first line of the input.

```
wc -l *out|sort -n|head -n 1
```

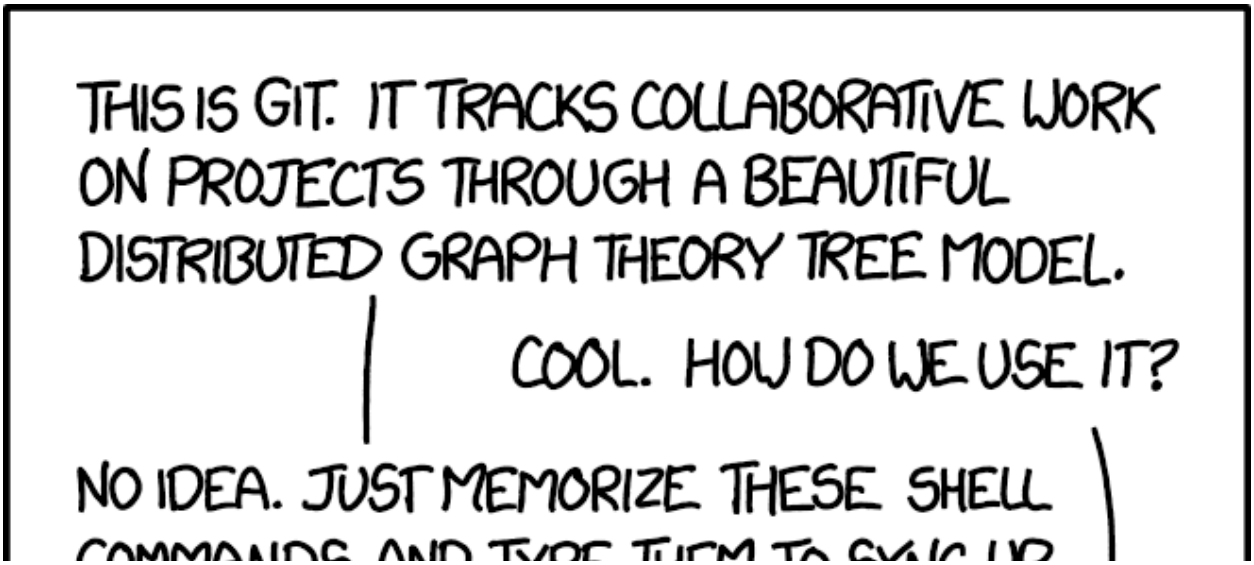
We can then even save this output to a file!

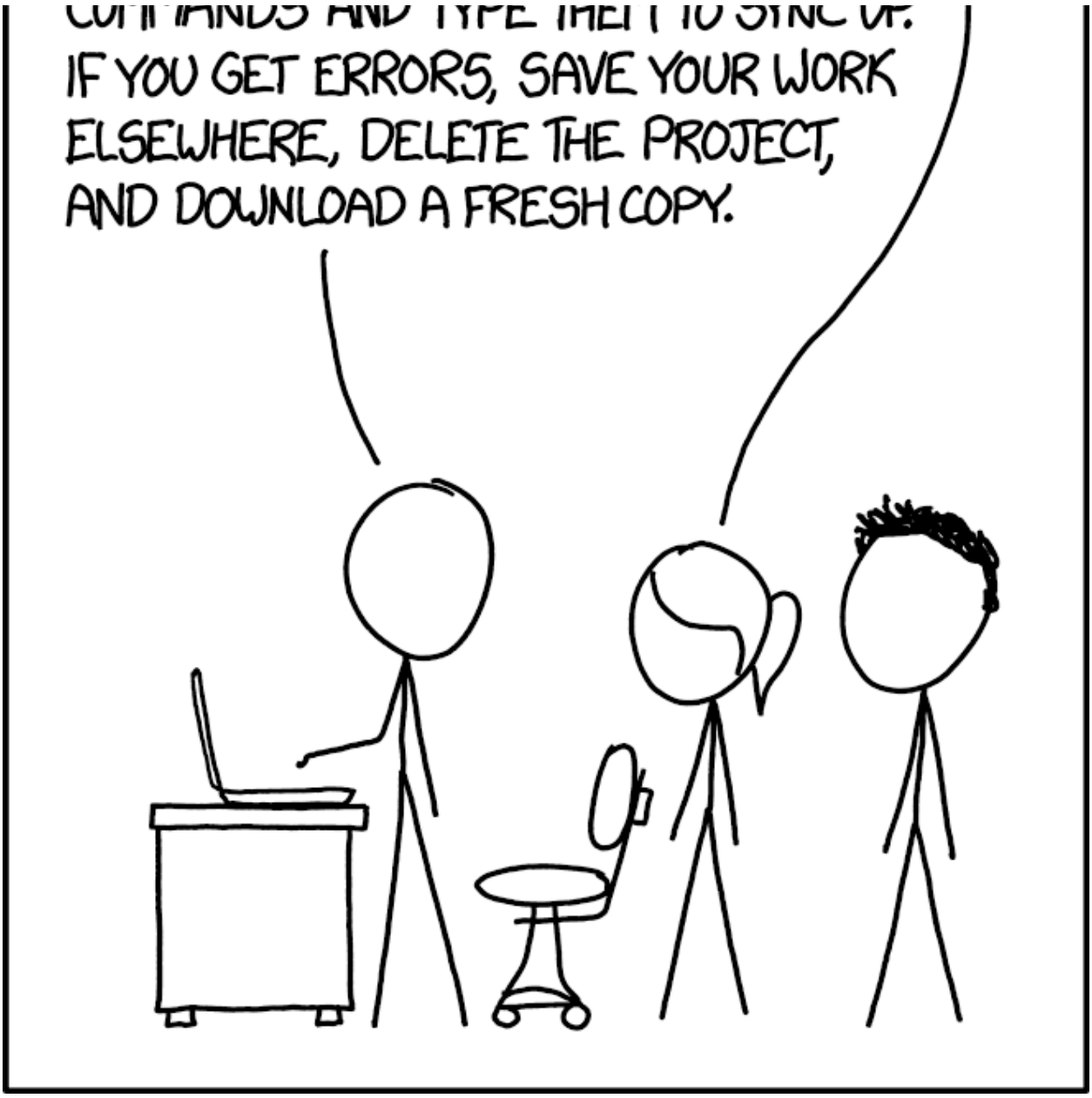
```
wc -l *out|sort -n|head -n 1 > shortest.file
```

You can see how these tools can be endlessly pieced together to do some really powerful manipulations!

> **Exercise Break:** Find a partner and try working through these exercises.
>
> - What is the difference in function between `sort` and `sort -n`. Try sorting one of the `.out` files to figure it out.
> - Write a command that will identify the index value (number in the first column) that has the largest value in the third column of `hiztory.out` . Return *only* the number in the first column and save the input to a file called big index. Hint: look at the man pages to find useful flags.
> - All these files were created with the `$RANDOM` value in bash. Write a command that will combine (one on top of the other) all the `.out` files. I want to know what the 10 most abundant numbers in the second column are and how many times they occur. Write a command that will do that and save the output to `10bignums.out`.

GitHub + Homework!





Cartoon from XKCD.

What is GitHub

I can't explain Git much better than Software Carpentry so let's flip over to them for a minute.
https://swcarpentry.github.io/git-novice/01-basics/index.html

Set up your Git on the HPC

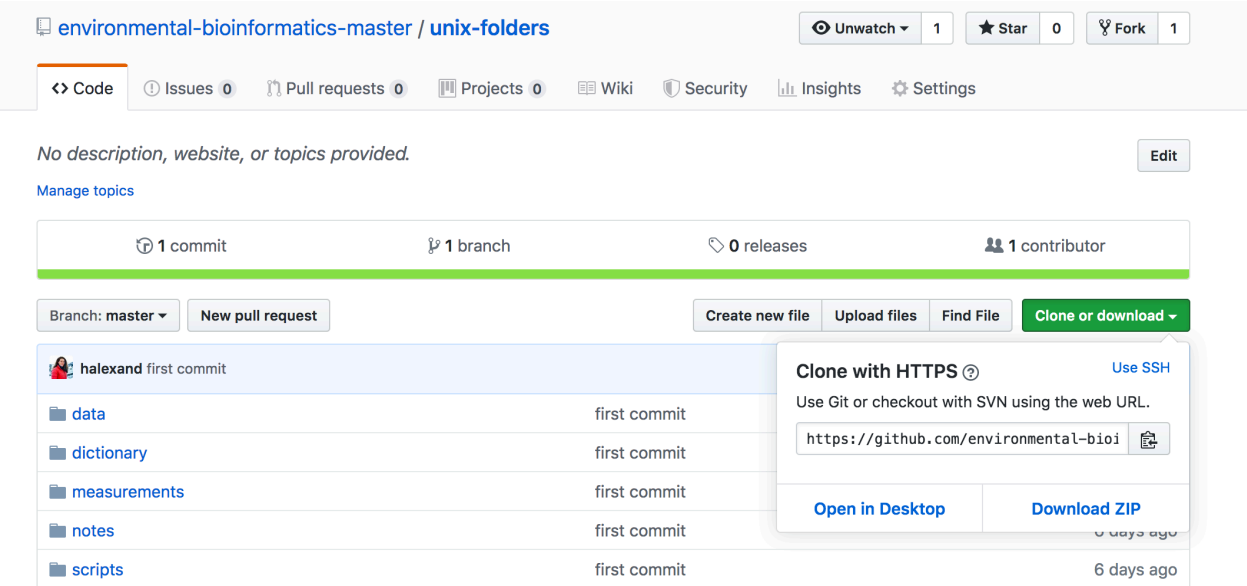
In a future class we will work through GitHub in much more detail. For today, we are just going to get through the basics of getting you set up on the HPC and downloading the homework.

First, we are going configure our GitHub information on poseidon. All `git` commands run as `git` and then the type of command you want to use. Here it will be `config`

```
git config --global user.name "Firstname Lastname"
git config --global user.email "email@youremail.com"
git config --global core.editor "nano -w"
```

Accepting the homework

Now let's initiate your homework. Click this link: https://classroom.github.com/a/bWVrG4b-. This link is going to take you to GitHub classroom and walk you through the creation of how to create a new repository (repo) that is specific to you. Here, you should be taken to a new repository called `Homework1a-Unix-username` within our group (`2019-Environmental-Bioinformatics`). Once you are there, click on the green `Clone or download` buttom at the top. This will lead to a drop down menu which provides the address for the repo. If you are doing this on a local computer you want to use the `Clone with HTTPS` option. However, most of us are running it on an HPC. As such, you will want to click `Use SSH` to copy the address for the repo.



Once you have copied the address switch back to your terminal window. Navigate to your user directory in posiedon: `/vortexfs1/omics/env-bio/users/yourusername`. We are now going to use the `clone` function of `git`. This is used to clone (or copy) repositories from GitHub to your local machine. Cloning is more than downloading as a zipped file as it will carry with it all the information and metadata that `git` needs to maintain version control. To clone we will type the command:

```
git clone [PASTE THE THING YOU COPIED]
```

You can then hit enter and you should see some printout like this:

```
Cloning into 'REPO NAME'...
remote: Enumerating objects: 256, done.
remote: Counting objects: 100% (256/256), done.
remote: Compressing objects: 100% (40/40), done.
remote: Total 256 (delta 0), reused 256 (delta 0), pack-reused 0
Receiving objects: 100% (256/256), 29.52 KiB | 0 bytes/s, done.
```

If you now run `ls` you should see a new folder called `Homework1a-Unix-username/`. Go into this folder and type `ls -a`. The `-a` flag shows all files (including those that are hidden). Many programs create hidden files (files or folders that start with `.` to prevent users from

messing with them).

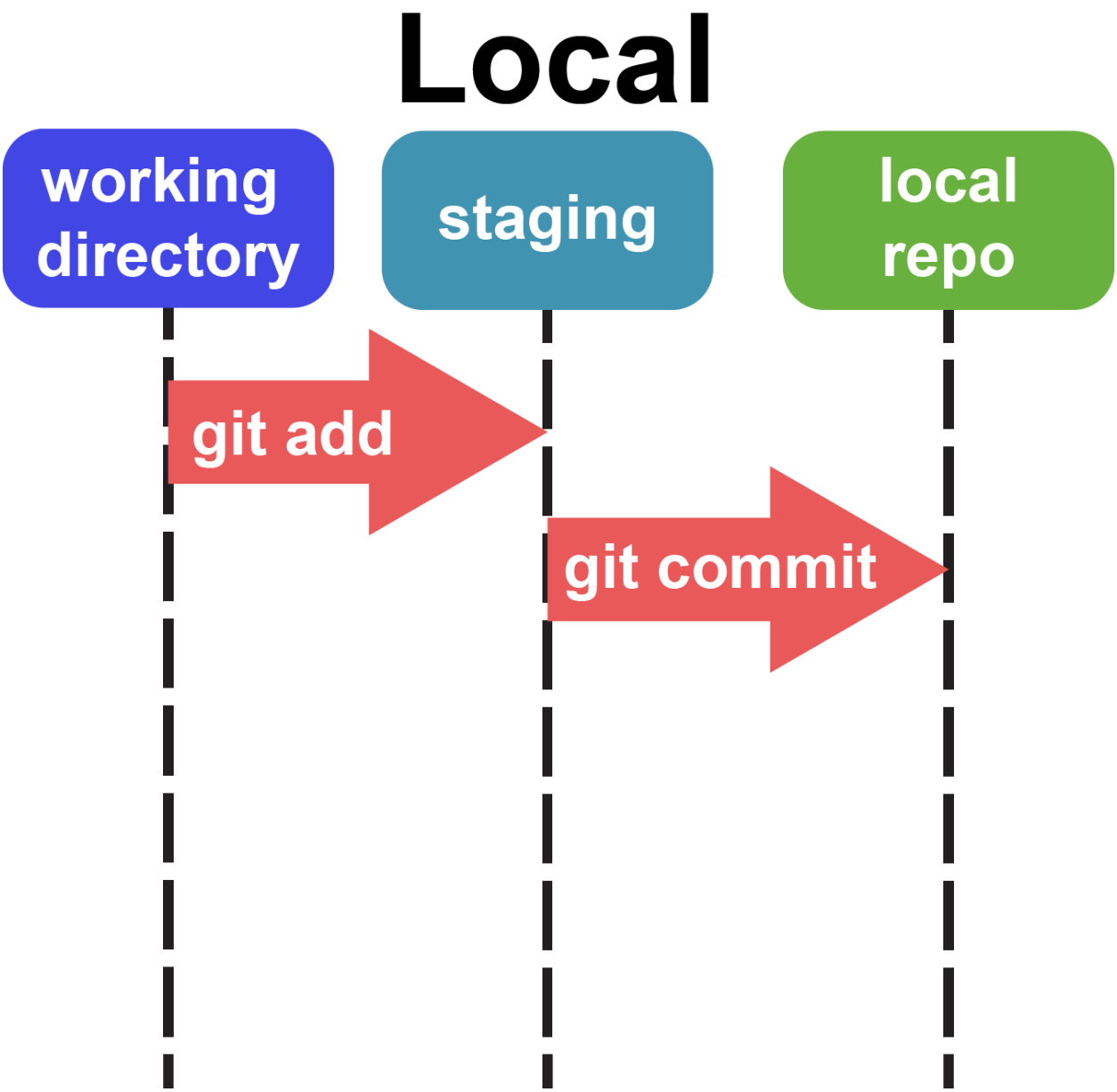
What do you see? You should see the homework assignment (`README.pdf` and `README.md`). The `.md` is markdown file which is a common and relatively easy to use formatting language. You can learn more about it here.

You should also see a folder called `sequences/` that has some data you will need in it. You will also see a `.git/` folder. This folder contains all the history associated with this repo. Basically, anything that has been committed to GitHub's memory is encoded within the `.git/` folder.

How to use Git to track changes and make commits

There are many levels of expertise that one can have with GitHub. For right now we are going to just cover the basics of the workflow that you might use if you are working on a project on your own.

After a git has been initiated (either via `git init` or `git clone`, as done here) you can start adding files to be tracked. First off, we have our local repo. Fundamentally, git does not automatically track any files. Files within your working directory are not being followed by git.



Local repository schematic

Let's make a new file in our homework directory called `temp`. Using `nano` write something in `temp`, save and the quit out of nano.

To check the status of what files are being followed we can use the command `git status`. This tells us what files are being followed, which aren't, and which have been changed. We can see at the bottom that `temp` is listed as a file that isn't being tracked. To tell git that you want it to follow a file and move it to the staging area you use the command `git add temp`.

This will add a file from our working directory to the staging area.

> Run `git status` again. What changed?

Git isn't really tracking this file yet. To finalize our changes we need to commit them. This will take our `temp` file that is in the staging area and commit the changes moving it into our local repo. These changes are now remembered by git.

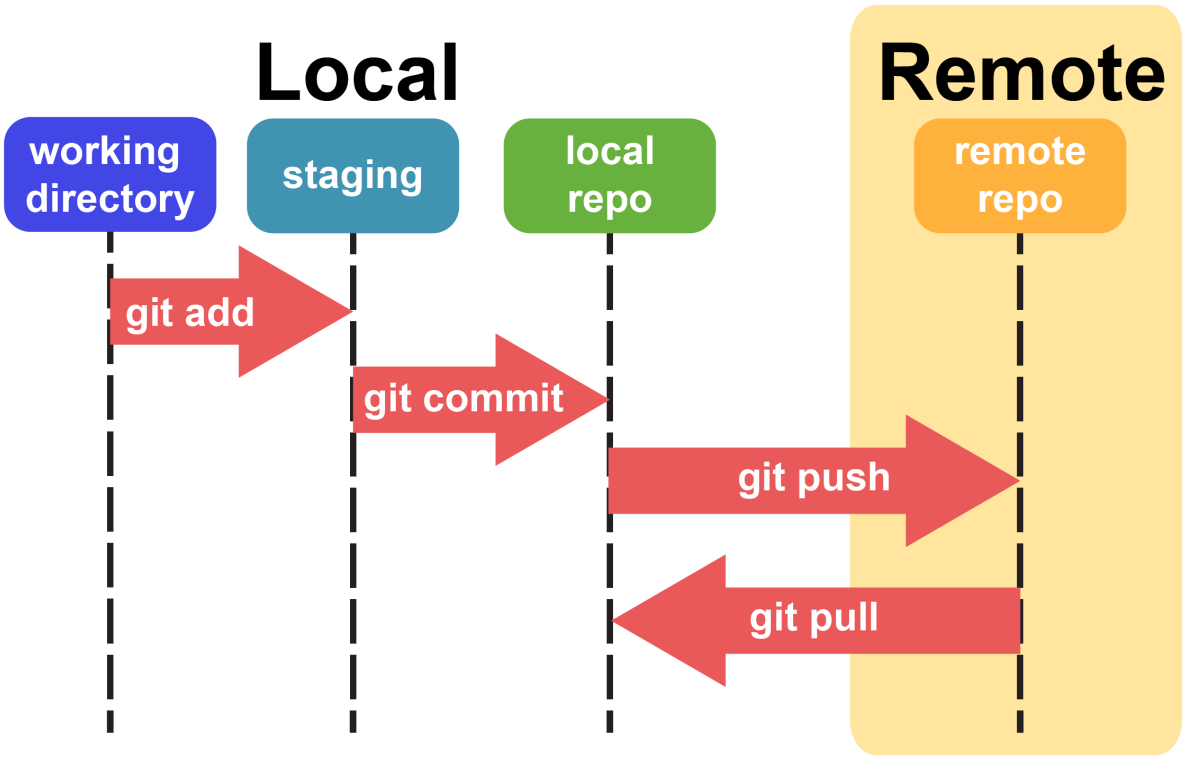
> Type `git status` again and see what has changed.

Now, what if make a change to temp. Use `nano` to add some text to `temp`.

> What happens when you type `git status` ?

As you can see it says that `temp` is tracked– but it isn't staged staged. In order to stage the file you have to run `git add` and `git commit` again.

Next time, we are going to talk about linking these changes to our remote repository: GitHub!



Setup on your own time

Password-less login

If you are like me, your password is **long** and **annoying**. It is likely that you are going to want to `ssh` on to `poseidon` frequently. Your life can be made much easier by setting up an SSH key-based authentication. This is a form of public-key cryptography (more here). The basic idea is that you generate a key pair (public and private) on your local machine and share the public key with your remote machine. That way, any time you try to log on to the remote machine, it automatically recognizes you. Meaning… no more typing your password. There are *many* only resources describing how to do this. Here is one tutorial that I found with a quick Google. Try to follow along (or find a different tutorial that makes more sense to you). Hint: client machine = your local computer and remote server = … remote server.

Password-less GitHub

Additionally, you might want to follow the instructions for seeing up an SSH key with your GitHub account. This will make it so that you no longer need to type your username/password when you push commits to GitHub. This again relies on the same concept as the above – using SSH keys to help computers recognize each other. However this time you will generating a public key on `poseidon` and sharing that key with GitHub. Instructions can be found here: https://help.github.com/en/articles/connecting-to-github-with-ssh. Again, you don't *have* to do this, but it will make your life just a little bit easier.

Next time

For loops, shell scripts, and slurm! Oh my!