Third Unix Lesson

Goals for today:

Today we will:

1. Get GitHub set up on the HPC and get the homework downloaded
2. Learn about wildcards and regex
3. Learn how to deal with outputs & string commands things together
4. Learn about loops and automation
5. Repeat things with scripts
6. Finding things

GitHub + Homework!

What is Git / 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

Main takeaway: Git is a software that helps you version control your code. GitHub is a platform that integrates with git and helps you collaborate with others.

A few other points worth noting:

- Git is not an archival service
- Git is not a good place to store data or data products (checkout Zenodo or OSF.io)

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, downloading the homework, and making commits.

First, we are going to share a public ssh key between your poseidon account and GitHub. To do this navigate to `~/.ssh/`. Get the contents of the file to print to screen by typing:

```
cat id_rsa.pub
```

Select and copy the whole thing to your local clipboard (on a mac `ctrl+` click).

Now, we are going to provide this public ssh key to GitHub. Go here: https://github.com/settings/keys and click on `Add new SSH key` on the top left. Give the key a title that is informative for you (like poseidon) and paste your key into the key box. Now click `Add SSH key` at the bottom and you should be good to go!

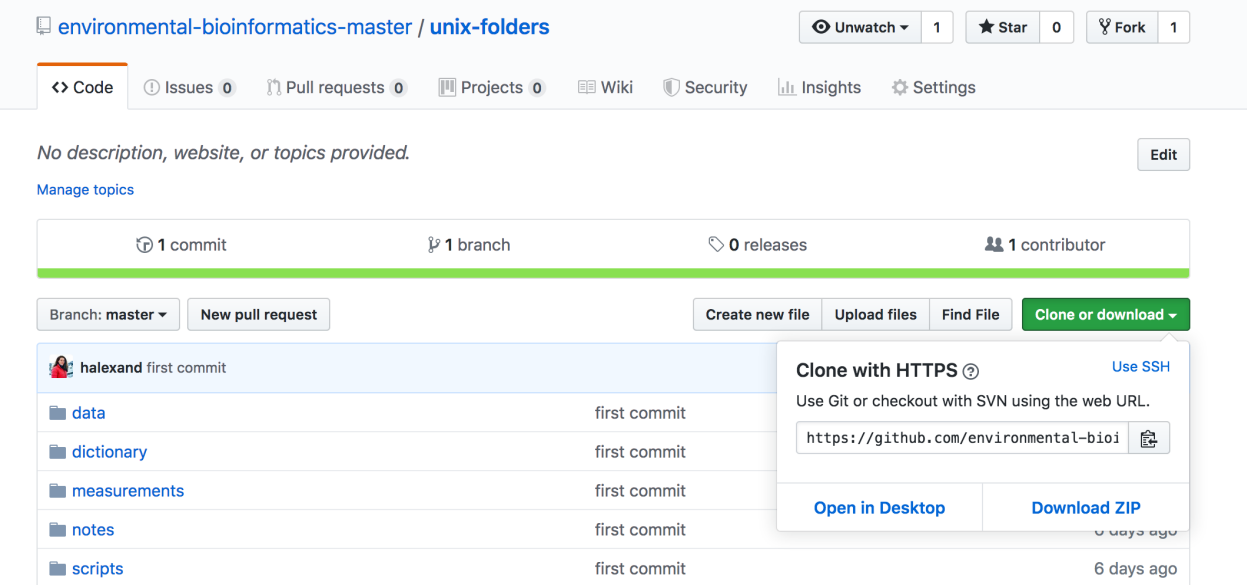
Now, switch back to terminal, we are going configure our GitHub information on poseidon. All `git` commands run as `git` and then the type of command you want to use. Here it will be `config`

```
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

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).

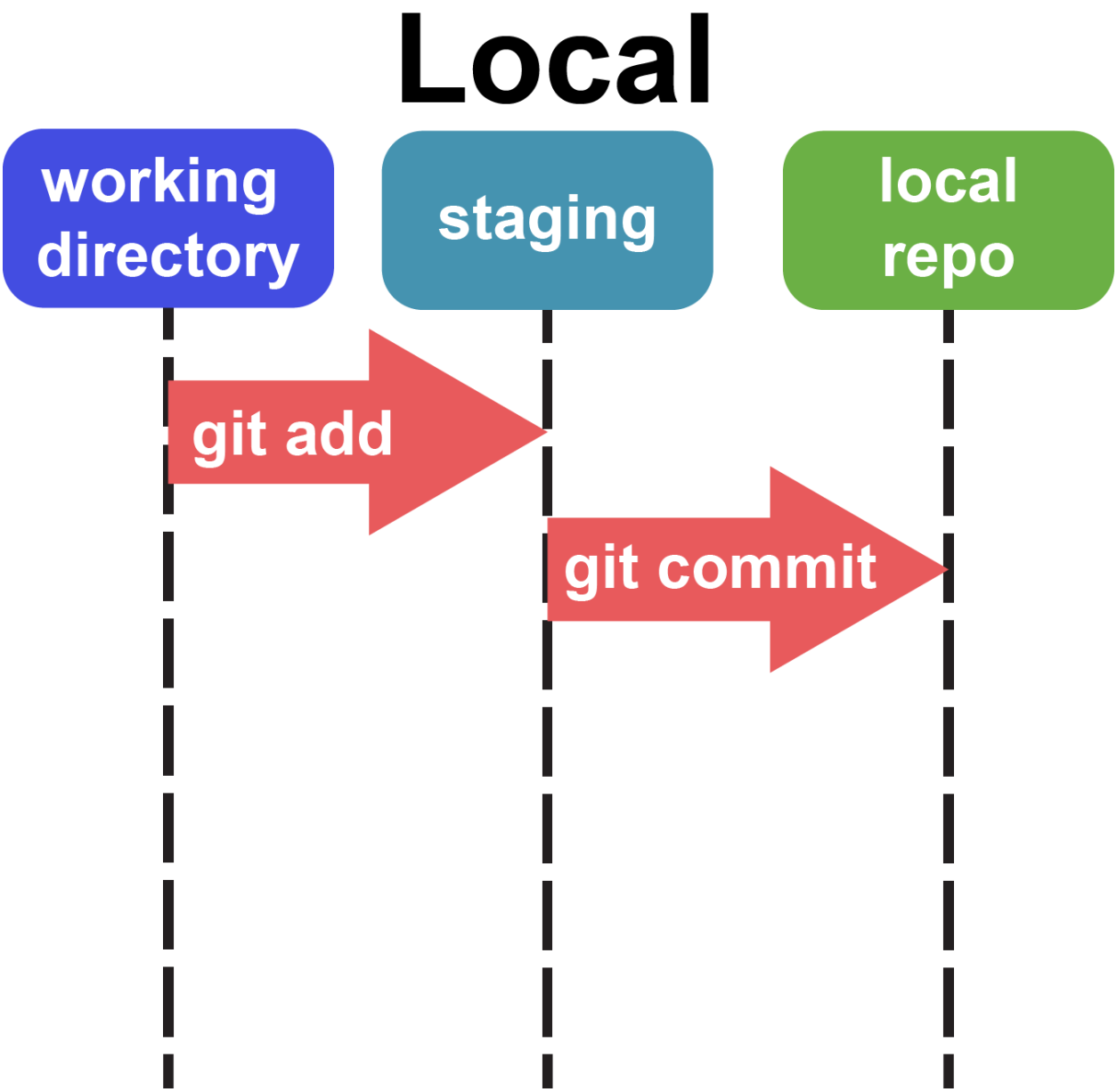
What do you see? You should see the homework assignment (`README.pdf` and `README.md`). The `.md` is markdown file which is a common and relatively easy to use formatting language. You can learn more about it here.

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`.

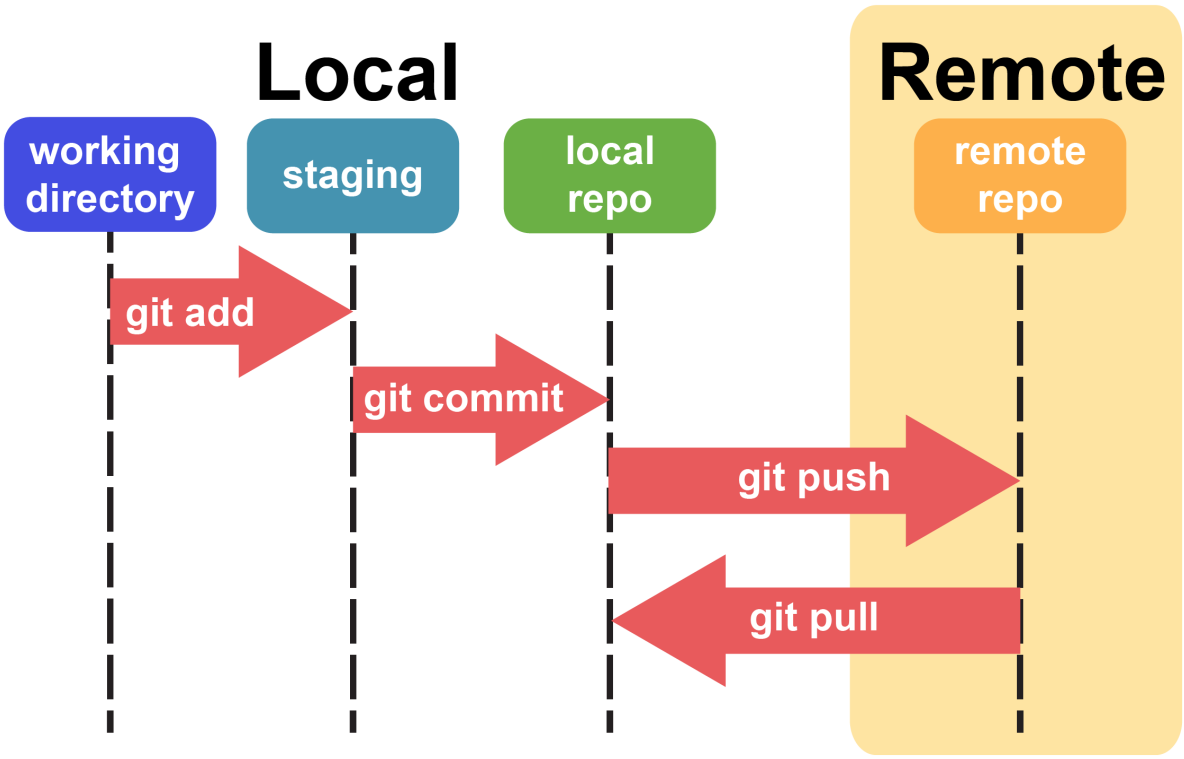
> What happens when you type `git status` ?

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.

Linking local Git to a remote (like GitHub)

What if we now wanted to share our changes that we made on our local repo with a remote system like GitHub? There are three main ways of interacting with the remote:

1. `git clone`: we already used this. This will copy a version of the repo to your local system.
2. `git pull`: copies changes from a remote repository to a local repository.
3. `git push`: copies changes from a local repository to a remote repository.



We have all made some changes to our local repositories. Let's try pushing them to the remote. Generally, I find it is good practice to run `git pull` first just in case there have been any changes. This will help you avoid `conflicts`.

Run `git status`. You can see we are on our `master` branch. We will dive into branches in a later session– but for right now you can think of `master` as the name of the local repo. By convention, the remote is typically called `origin`. So, to push our current commits to master we will type:

```
git push master origin
```

Go and take a look at your remote– you should see that the file `temp` has been added to your online repository. You are now ready to go and work on the homework. Once you are done with the homework you will want to `commit` your changes and `push` them to the remote repository.

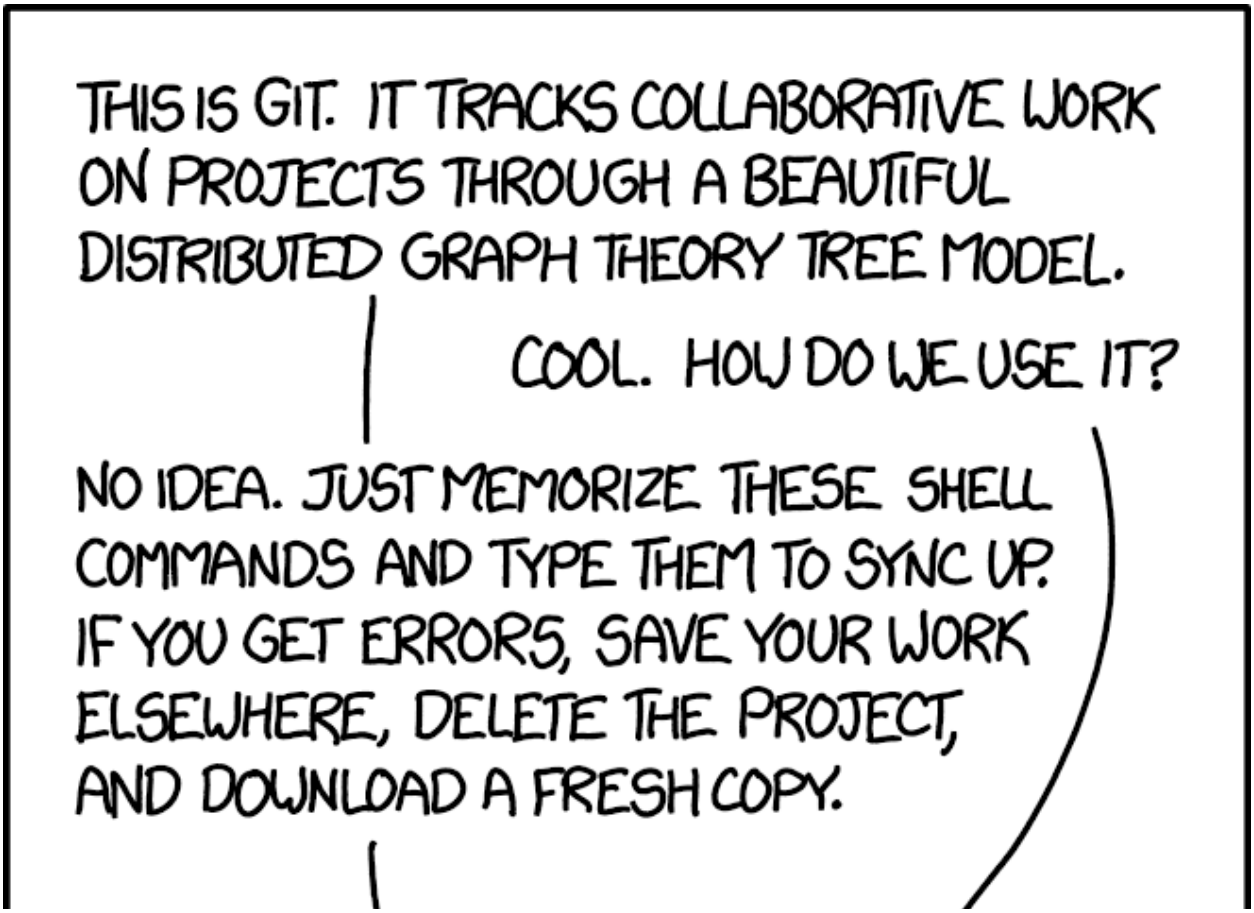
A parting thought on Git

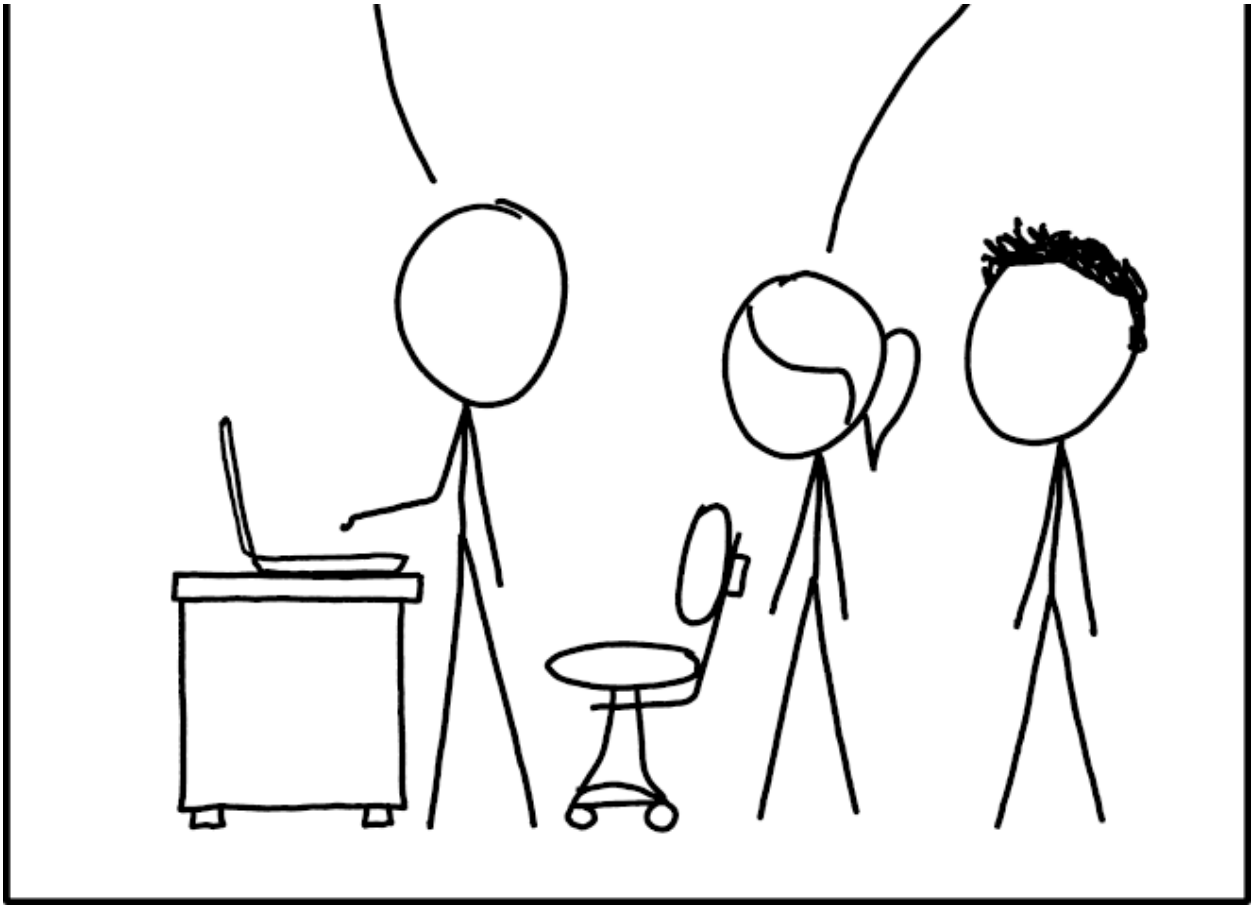
Starting out with `git` can be a bit overwhelming at times. It is easy to mess up your repo or the like.

Two common problems:

1. Accidentally adding files that are over the 100 MB size limit and trying to push them to GitHub.
2. Conflicting pushes: if you made any changes to your remote Git and forgot while trying to push your local. Thus, always try to pull and then push.

And honestly, sometimes you just need to start over. Re-clone, move files around, pull and push. However, the simple fact that you can *re-download* a version of your project from the internet is very powerful!





Cartoon from XKCD.

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.

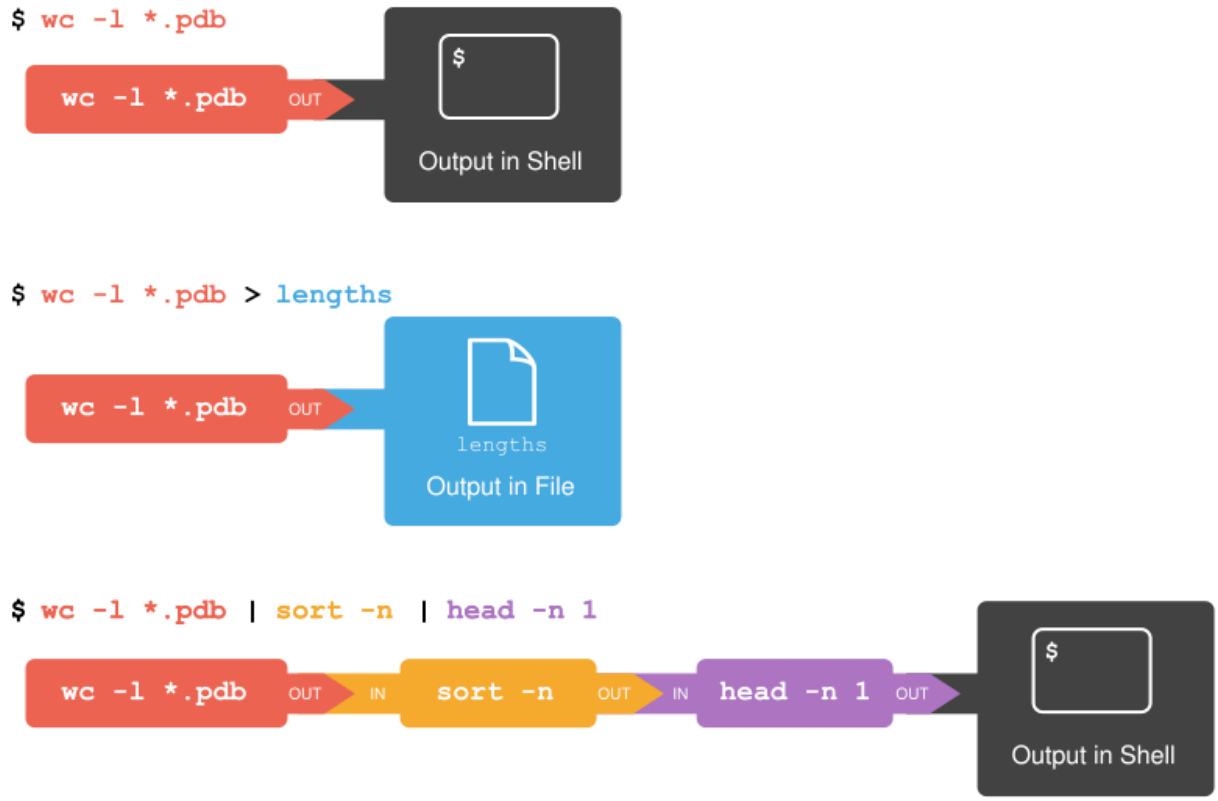


Image from Software Carpentry.

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.
>
> - Write a command that will print the 34th line (exactly) of the file `compartment.out`.
> - 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`.

Automation with for loops

Loops are a programming construct which allows us to perform a series of commands in the same way for each item in a list. Loops facilitate automation and save you time– moreover loops reduce the amount of typing required (and mistakes made). So, if you ever find yourself thinking: gosh– I want to do this one thing to all 10502395 of my files. You should think: `for loop`.

Loops are common across programming languages. Today we will be writing one in `bash` but we will also learn how to write them in `python` later on. Regardless of the language the general format is the same:

```
### PSEUDO CODE
FOR item in LIST
    DO_SOME_STUFF
DONE
```

In `bash` for loops look like this:

```
for item in list
do
    DO_SOME_STUFF $item
done
```

When the shell sees the word `for` it knows that a loop is coming. It

The `$` in `bash` is used to designate variables. We have already seen variables (e.g. `$HOME` and `$PATH`). A variable name is a name whose value can be changed (rather than a text string or command that is static and cannot be changed. In a for loop the `item` in a list becomes a variable and changes as it moves through the loop.

Navigate to `data/`. Let's pretend we wanted to retrieve the 25th value from `lion.out` and `secret.out` data files. If we tried to do this with our wildcard command and pipes it wouldn't work very well. We can encode this in a for loop.

```
for file in lion.out secret.out
do
    head -n 25 $file | tail -n1
done
```

> What if we wanted to save this out to a file? Modify this to create a file that has the

> output. This can be done in more than one way.

Now let's say that we want actually copy each of our `*out` files. If we tried to do this with our wildcard command and pipes it wouldn't work very well. We can encode this in a for loop.

> Note: The shell prompt changes to `>` as we were typing in our loop. This indicates that the terminal is waiting for something to tell it that the command is finished. In this case `done`.

`echo` is a very useful command– it is effectively `print` and will report the value of any variable. For example, we just defined the variable file in the above for loop.

If we now type:

```
echo file #prints the name file
echo $file #prints the value of file
```

It will print the value of the variable `$file`.

Variables are powerful because we can easily manipulate them to do things like find and replace values. Let's make a variable called `santa` and assign it the value `hohoho`.

```
santa=hohoho
echo $santa
```

For example, if we wanted to change the letter `o` to an `a` we do this easily with string manipulations.

```
echo $santa//o/a
```

> Now, what if you wanted to change the name of all the `.out` to `.data`. Write a for

> loop to do that.

Repeat it with scripts

Ultimately, the thing that makes shell so powerful is your ability to save things you want to do more than once. Navigate into the `scripts` folder.

Here are some ready made scripts that we are going to take a look at. First type `ls -l`. You should see that all of these scripts are executable.

Let's take a look at `hello.sh`.

As you can see it is pretty straight forward. This is a file that contains one command `echo Hello, world.`. To the screen.

To execute a bash script, you can use the command `bash`.

```
bash hello.sh
```

This is nice– but not really the most useful thing in the world. Let's edit this script to allow the user to pass an input to it. There are a series of special variables in shell `$1`, `$2`, etc. These variables correspond to order of the input following the name of the script. The variable `$@` refers to all the variables passed to the script.

Let's change `hello.sh` so that it will say hello to whatever the input is. Use `nano` and change the input to:

```
echo Hello $1!
```

We can now execute this by typing:

```
bash hello.sh Elmo
```

What is the output of the above?

> Edit `hello.sh` again to make it print a hello statement to two people. Note what happens when two inputs aren't provided.

Now let's look at the script I used to make all the fake data files we have been looking at. Use less to look at `script.sh`.

```
#!/bin/bash
for s in $(seq $1)
do
        echo $s $RANDOM $RANDOM
done
```

This script introduces a new useful command `seq`. `seq` will automatically generate a number line from any start position to any end position. The other new thing is the variable `$RANDOM`. What does this do?

As you can see at the top of the file there is a funny string `#!/bin/bash`. This is called the shebang or bang. It is a common feature of shell scripts– it tells the computer the path to the interpreter that should be used. This ensures that however you execute your script it will be interpreted with the correct interpreter (i.e. `bash` and `csh`).

Now, try running this script– what do you think you would pass the script?

Now let's take a look at an even more complex script. Open up `make-data.sh`.

```
#!/bin/bash

for name in $(cat $1)
do
        bash script.sh $RANDOM > ${name}.out
done
```

What does this script do?

> Now, let's take one of the commands wrote earlier and turn it into a script. Write a script to return the 25th line of a file provided by the user.
>
> > Can you generalize this to make it so that it returns the nth line of a file (as specified by a user)?

Finding things

Finding words in text files

Global regular expression print `grep` is a command that lets you search for words or regular expressions within files. Navigate to `unix-folders-master/writing/fables-poems` and let's give it a try. We are going to search for the word Cat in the file `LaFontaine.txt`.

```
grep Cat LaFontaine.txt
```

As you can see this is printing all the lines that contain the word `Cat` within the file. Now try searching for `cat`. What is the difference? `grep` is a very powerful tool– use man to take a look at its abilities. Can you figure out how to make it ignore case?

You can also use `grep` regular expressions. To be safe it is good to append the flag `-E` to the command to force it to read the string passed as a regular expression.

```
grep -E "nose|ring" lear.txt
```

> Write a command to find all occurrences of two capital letters followed by a space.

Find and replace?

Often times you will have a file that you want to fix so that it doesn't contain certain character or the like. One option for finding and replacing things within a text file is the command `sed`.

`sed` is a powerful tool and has had books written about it. Today, we are going to just cover the simplest functionality. Genearlly it takes the form `'s/old_word/new_word/'`. This will replace an old_word with new_word. Let's check this out. Navigate to: `unix-folders-master/writing/fortunes`.

Let's try replacing `a` with `A` in this file.

```
sed -e 's/a/A/' fortune5
```

What happened?

As you can see – only the first occurrence of `a` was replaced. To replace all the instances of an occurring character in the text file you can add `g` after the search pattern.

```
sed -e 's/a/A/g' fortune5
```

> Write a `sed` command to find and replace the letters `a` or `s` with `!`.

Finding files

Sometimes, even with the best organization skills you forget where you put things. While `grep` searches a particular file– `find` is used to search your file system. Go back to `unix-folders-master/`.

Try typing `find .`. This is going to to list all the files that exist at this level and below. But, of course, find is much more powerful than that. To find all the directories:

```
find . -type d
```

Find can also be used to find files that match a certain name:

```
find . -name *.txt
```

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.