# Programming For Biology 2025

programmingforbiology.org

**Instructors**
Simon Prochnik
Sofia Robb
Eric Ross
Jessen Bredeson

# Big Picture

## Why?

Why is it important for **biologists** to learn to program?

You might already know the answer to this question since you are here.

We firmly believe that knowing how to program is just as essential as knowing how to run a gel or set up a PCR reaction. The data we now get from a single experiment can be overwhelming. This data often needs to be reformatted, filtered, and analyzed in unique ways. Programming allows you to perform these tasks in an **efficient** and **reproducible** way.

## Helpful Tips

What are our tips for having a successful programming course?

1. Practice, practice, practice. Please spend as much time as possible actually coding.
2. Write only a line or two of code, then test it. If you write too many lines, it becomes more difficult to debug if there is an error.
3. Errors are not failures. Every error you get is a learning opportunity. Every single error you debug is a major success. Fixing errors is how you will cement what you have learned.
4. Don't spend too much time trying to figure out a problem. While it's a great learning experience to try to solve an issue on your own, it's not fun getting frustrated or spending a lot of time stuck. We are here to help you, so please ask us whenever you need help.
5. Lectures are important, but the practice is more important.
6. Review sessions are important, but practice is more important.
7. Our key goal is to slowly, but surely, teach you how to solve problems on your own.

---

# Unix 1

## Unix Overview

### What is the Command Line?

Underlying the pretty Mac OSX Graphical User Interface (GUI) is the operating system (OS). It's based on BSD (Berkeley Standard Distribution), which is a version of Unix. Linux is pretty similar and also a very common OS in bioinformatics and you'll run into dialects by Red Hat, Ubuntu and others.

The command line gives you access to the internals of the OS, and is also a convenient way to write custom software and scripts.

Most bioinformatics tools are written to run on the command line and have no Graphical User Interface. In many cases, a command-line tool is more versatile than a graphical tool because you can easily combine command line tools into automated scripts that accomplish custom tasks sequentially without human intervention.

In this course, we will be writing Python scripts and running them from the command line.

The Unix OS is complicated and there are always more things to learn about it. We'll just cover enough to get you going and show you

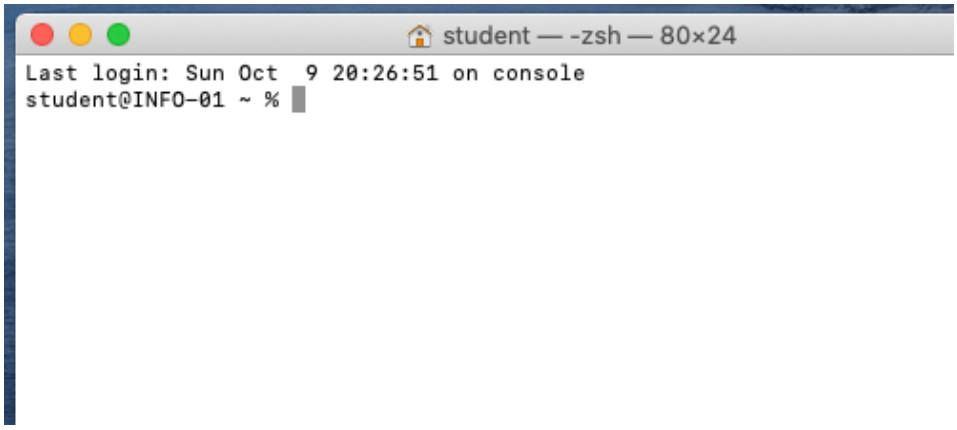
## The Basics

### Logging into Your Workstation

Your workstation is an iMac. To log into it, provide your user name and password. Your username is 'pfb' and the password is 'pfb2025'

### Bringing up the command line

To bring up the command line, use the Finder to navigate to *Applications->Utilities* and double-click on the *Terminal* application. This will bring up a window like the following:



You can open many Terminal windows at once. This is often helpful. A common way to work is to open your code editor in one window, run the code from another and view data and files in another.

You will be using the Terminal application a lot, so I suggest that you drag its icon into the shortcuts bar at the bottom of your screen.

## OK. I've Logged in.  What Now?

The terminal window is running **shell** called "zsh". (Mac recently changed from bash) The shell is a loop that:

1. Prints a prompt
2. Reads a line of input from the keyboard
3. Parses the line into one or more commands
4. Executes the commands (which usually print some output to the terminal)
5. Go back step 1.

There are many different shells with bizarre names like **bash**, **sh**, **csh**, **tcsh**, **ksh**, and **zsh**.  The "sh" part means shell.  Each shell has slightly different syntax and features. Your accounts are set up to use **zsh**.  It's very similar to **bash** which is  standard on linux systems.

## Command Line Prompt

Most of bioinformatics is done by running command line software in a shell, so you should take some time to learn to use the shell effectively.

This is a command line prompt:

```
bush202>
```

This is another:

```
(~) 51%
```

This is another:

```
srobb@bush202 1:12PM>
```

What you see depends on how the system administrator has customized your login.  You can customize it yourself (but we won't go into that here)

The prompt tells you the shell is ready for you to type a command.  Most commands run almost instantly, but if you run a long command, the prompt will not reappear until it is finished and the system is ready to accept your next request.

## Issuing Commands

You type in a command and press the <Enter> key to execute it.  If the command has output, it will appear on the screen.  Example:

```
(~) 53% ls -F
GNUstep/                cool_elegans.movies.txt  man/
INBOX                   docs/                    mtv/
INBOX~                  etc/                     nsmail/
Mail@                   games/                   pcod/
News/                   get_this_book.txt        projects/
axhome/                 jcod/                    public_html/
bin/                    lib/                     src/
build/                  linux/                   tmp/
ccod/
(~) 54%
```

The command here is `ls -F`, which produces a listing of files and directories in the current directory (more on that later).  Below its output, the command prompt appears again.

Some programs will take a long time to run. After you issue their command names, you won't recover the shell prompt until they're done. You can keep working in the meantime, either by launching a new shell (from Terminal's File menu), or running the command in the background by adding an ampersand at the end of the command like so

```
(~) 54% long_running_application &
(~) 55%
```

> The command will now run in the background until it is finished. If it has any output, the output will be printed to the terminal window. You may wish to capture the output in a file (called redirection). We'll describe this later.

## Command Line Editing

Most shells offer command line editing.  Up until the moment you press <Enter>, you can go back over the command line and edit it using the keyboard.  Here are the most useful keystrokes:

- *Backspace*: Delete the previous character and back up one.
- *Left arrow*, right arrow: Move the text insertion point (cursor) one character to the left or right.
- *control-a (^a)*: Move the cursor to the beginning of the line. (Mnemonic: A is first letter of alphabet)
- *control-e (^e)*: Move the cursor to the end of the line. (Mnemonic: E for the End) (^z was already used to interrupt a command).
- *control-d (^d)*: Delete the character currently under the cursor. D=Delete.
- *control-k (^k)*: Delete the entire line from the cursor to the end. k=kill.  The line isn't actually deleted, but put into a temporary holding place called the "kill buffer". This is like cutting text

- *control-y (^y)*: Paste the contents of the kill buffer onto the command line starting at the cursor.  y=yank. This is like paste.
- *Up arrow, down arrow*: Move up and down in the command history.  This lets you rerun previous commands, possibly modifying them before you do.

There are also some useful shell commands you can issue:

- `history` Show all the commands that you have issued recently, nicely numbered.
- `!<number>` Reissue an old command, based on its number (which you can get from `history`).
- `!!` Reissue the last command.
- `!<partial command string>`: Reissue the previous command that began with the indicated letters.  For example, `!l` (the letter el, not a number 1) would reissue the `ls -F` command from the earlier example.
- *control-r (^r)*: enter string and search through history for commands that match it. This is really useful.

**zsh** offers automatic command completion and spelling correction.  If you type part of a command and then the  key, it will prompt you with all the possible completions of the command.  For example:

```
(~) 51% fd<tab><tab>
(~) 51% fd
fd2ps    fdesign  fdformat fdlist   fdmount  fdmountd fdrawcmd fdumount
(~) 51%
```

> If you hit tab after typing a command, but before pressing <Enter>, **zsh** will prompt you with a list of file names. This is because many commands operate on files.

## Wildcards

You can use wildcards when referring to files. `*` stands for zero or more characters. `?` stands for any single character.  For example, to list all files with the extension ".txt", run `ls` with the wildcard pattern "*.txt"

```
(~) 56% ls -F *.txt
final_exam_questions.txt  genomics_problem.txt
genebridge.txt            mapping_run.txt
```

There are several more advanced types of wildcard patterns that you can read about in the **zsh** manual page. For example, if you want to match files that begin with the characters "f" or "g" and end with ".txt", you can put both characters inside square brackets `[fg]` as part of the wildcard pattern. Here's an example

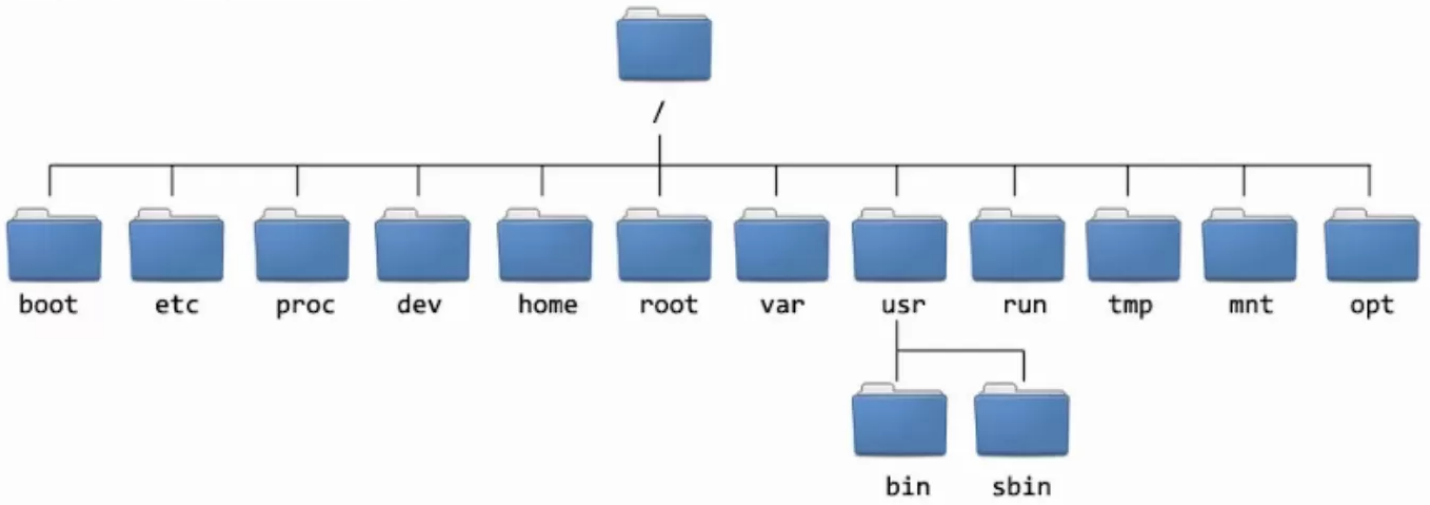
```
(~) 57% ls -F [fg]*.txt
final_exam_questions.txt  genebridge.txt            genomics_problem.txt
```

You can also search for a range of characters e.g. `[a-e]` or `[1-5]`.

# Directories and how they are organized

A computer comes with a place to store scripts, data, images, OS, and other files. It used to be floppy disks, then hard drives and these days it's often a solid state drive (SSD). Let's talk about how the storage is organized to help you find what you are working on. Directories or folders are created inside other directories. One directory is special. This is the **root directory** because it is not inside any other directories (it's written `/`). Files that go together are created inside a directory to keep them organized. This creates a structure that can be drawn like a branching tree. We find it clearer to turn it upside down to look like branching roots.

**Example diagram of a Linux directory structure starting from the root directory**




## Home Sweet Home (your home directory `~`)

When you first log in, you'll be in your personal directory (or folder), called the **home directory**. This directory has the same name as your login name, and on macOS is located inside the directory `/Users`. (On Linux, it's typically in `/home`). If your username is `dbrown`, your home directory would (probably) be `/Users/dbrown`. This is a filepath or a path. Unix if full of abbreviations to save on typing common things. The shell allows you to abbreviate it as `~username` (where "username" is your user name or someone elses), or simply as `~`. The weird character (called "tilde" or "twiddle") is at the upper left corner of your keyboard. The `$HOME` variable is usually set to your home directory.

In your home directory, you have permission to save, delete, open and move files and other directories. In general, you can read but not write or modify files elsewhere in the system.

To see what is in your home directory, issue the command `ls` for list directory contents

```
Desktop     Downloads   Movies      Pictures    expt1.countdata.tsv
Documents   Library     Music       Public      notes.txt
```

You can modify the way a command works with switches. These are single letters typed after a dash `-`. The case is important. So `ls -aF` lists all (`a`) files including hidden filenames that start with a `.` with fancy formatting turned on (`F`) such that a `/` is added to the end of directory names.

```
./                  .git/       Desktop/      Library/    Pictures/   notes.txt
../                 .lesshst    Documents/    Movies/     Public/
.CFUserTextEncoding .zshrc      Downloads/    Music/      expt1.countdata.tsv
```

> Don't go deleting the hidden files. Many of them are essential configuration files for commands and other programs. For example, the `.zshrc` file contains configuration information for the **zsh** shell. You can edit it (a bit later, when you know what you're doing) in order to change things like the command prompt and command search path.

## Changing to a New Directory

You can move around from directory to directory using the `cd` command (for change directory). Give the name of the directory you want to move to, or give no name to move back to your home directory. Use the `pwd` command (print working directory) to see where you are (or, if configured, this information will appear in the prompt):

```
dbrown@WideIsLove ~ % cd Documents
dbrown@WideIsLove Documents % pwd
/Users/dbrown/Documents
dbrown@WideIsLove Documents % cd
dbrown@WideIsLove ~ % cd /
dbrown@WideIsLove / % ls -F
Applications/ System/   Volumes/  cores/    etc@    opt/      sbin/   usr/
Library/      Users/    bin/      dev/      home@   private/  tmp@    var@
```

> Each directory contains two special hidden directories named `.` and `..`. The first, `.` refers always to the current directory. `..` refers to the parent directory. This lets you move upward in the directory hierarchy like this:

```
(~/docs) 64% cd ..
```

and to do cool things like this:

```
(~/docs) 65% cd ../../dbrown/Documents
```

> The latter command moves upward two levels, and then into a directory named `Documents` inside a directory called `dbrown`.

If you get lost, the `pwd` command prints out the full path to the current directory:

```
(~) 56% pwd
/Users/lstein
```

## Absolute and relative paths

We can type a path in two ways, an absolute path always starts with a `/` character and reads from the root directory. A relative path starts with another character and reads from the directory you are currently in. Here are two ways to get to a directory called `Music` in the home directory of the user `dbrown`. From `Documents` we can go up a directory and down into `Music` (relative path) or go from `/` to `Users` to `dbrown` to `Music` (absolute path).

```
dbrown@WideIsLove Documents % cd ../Music
dbrown@WideIsLove Music % cd
dbrown@WideIsLove ~ % cd /Users/dbrown/Music
dbrown@WideIsLove Music %
```

> Which of these paths would work for a different user as well as for `dbrown`?

# Essential Unix Commands

With the exception of a few commands that are built directly into the shell (e.g. `history`, `for`, `if`), all Unix commands are actually executable programs. When you type the name of a command, the shell will search through all the directories listed in the `$PATH` environment variable for an executable of the same name. If found, the shell will execute the command. Otherwise, it will give a "command not found" error.

Most commands live in `/bin`, `/sbin`, `/usr/bin`, `/usr/sbin/` or `/usr/local/bin`. You can use the `which` command to find a program's location, e.g., `which ls` may show `/bin/ls`.

## Getting Information About Commands

The `man` command will give a brief synopsis of a command. Let's get information about the command `wc`

```
(~) 76% man wc
Formatting page, please wait...
WC(1)                                                       WC(1)

NAME
       wc - print the number of bytes, words, and lines in files

SYNOPSIS
       wc [-clw] [--bytes] [--chars] [--lines] [--words] [--help]
       [--version] [file...]

DESCRIPTION
       This manual page documents the  GNU  version  of  wc.   wc
       counts  the  number  of bytes, whitespace-separated words,
```

```
...
```

## Finding Out What Commands are on Your Computer

The `apropos` command will search for commands matching a keyword or phrase. Here's an example that looks for commands related to 'column'

```
(~) 100% apropos column
showtable (1)        - Show data in nicely formatted columns
colrm (1)            - remove columns from a file
column (1)           - columnate lists
fix132x43 (1)        - fix problems with certain (132 column) graphics
modes
```

On some systems, you can also use `tldr` (too long didn't read). On macOS, you can use Homebrew (https://brew.sh/) `brew install tldr` to see help documentation:

```
$ tldr wc

wc

Count lines, words, or bytes.
More information: <https://keith.github.io/xcode-man-pages/wc.1.html>.

- Count lines in file:
    wc -l path/to/file

- Count words in file:
    wc -w path/to/file

- Count characters (bytes) in file:
    wc -c path/to/file

- Count characters in file (taking multi-byte character sets into account):
    wc -m path/to/file

- Use `stdin` to count lines, words and characters (bytes) in that order:
    find . | wc
```

## Arguments and Command Line Switches

Many commands take arguments.  Arguments are often the names of one or more files to operate on.  Most commands also take command line "switches" or "options", which fine-tune what the command does.  Some commands recognize "short switches" that consist of a minus sign `-` followed by one or more single characters, while others recognize "long switches" consisting of two minus signs `--` followed by a whole word.

The `wc` (word count) program is an example of a command that recognizes both long and short options. You can pass it the single `-c`, `-w` and/or `-l` options (or combinations of them) to count the characters, words, and lines in a text file, respectively. Or you can use the longer but more readable `--chars`, `--words` or `--lines` options. Both these examples count the number of characters and lines in the text file `/var/log/messages`:

```
(~) 102% wc -c -l /var/log/messages
     23     941 /var/log/messages
(~) 103% wc --chars --lines /var/log/messages
     23     941 /var/log/messages
```

You can cluster one-character switches by concatenating them together, as shown in this example:

```
(~) 104% wc -cl /var/log/messages
     23     941 /var/log/messages
```

Many commands will give a brief usage summary when you call them with the `-h` or `--help` switch.

## Spaces, tabs and newline Characters

The shell uses spaces to separate arguments. If you want to embed a space (see below for other whitespace or non-printing characters like a tab or newline etc) in an argument, put single quotes around it. For example:

```
mail -s 'An important message' 'Bob Ghost <bob@ghost.org>'
```

This will send an e-mail to the fictitious person Bob Ghost. The `-s` switch takes an argument, which is the subject line for the e-mail. Because the desired subject contains spaces, it has to have quotes around it. Likewise, my name and e-mail address, which contain embedded spaces, must also be quoted in this way.

Certain special non-printing characters have *escape codes* associated with them:

| Escape Code | Description |
|---|---|
| \n | new line character |
| \t | tab character |
| \r | carriage return character |
| \a | bell character (ding! ding!) |
| \NNN | the character whose octal ASCII code is **NNN** e.g. `printf '\101'` prints 'A' ASCII code tables |

# Useful Commands

Here are some commands that are used extremely frequently. Use `man` to learn more about them. Some of these commands may be useful for solving the problem set ;-)

## Manipulating Directories

| Command | Description |
|---|---|
| `ls` | Directory listing. Most frequently used as `ls -F` (decorated listing), `ls -l` (long listing), `ls -a` (list all files). |
| `mv` | Rename or move a file or directory. |
| `cp` | Copy a file. `cp -r` (recursively) to copy directories |
| `rm` | Remove (delete) a file. `rm -rf olddata.tsv` |
| `mkdir` | Make a directory |
| `ln` | Create a symbolic or hard link. `ln -s` makes a symbolic (sym) link |
| `chmod` | Change the permissions of a file or directory. |

## Reading files

| Command | Description |
|---|---|
| `cat` | Concatenate program. Can be used to concatenate multiple files together into a single file, or, much more frequently, to view the contents of a file or files in the terminal. |
| `more` | Scroll through a file page by page. Very useful when viewing large files. Works even with files that are too big to be opened by a text editor. |
| `less` | A version of `more` with more features. |
| `head` | View the first few lines of a file. You can control how many lines to view `head -3` prints the first three lines. |
| `tail` | View the end of a file. You can control how many lines to view. You can also use `tail -f` to view a file that you are writing to. |

## Analyzing, processing files

| Command | Description |
| --- | --- |
| `wc` | Count words, lines and/or characters in one or more files. |
| `sort` | Sort the lines in a file alphabetically or numerically (-g or -n) in reverse order (-r). |
| `uniq` | Remove duplicated lines in a file. |
| `cut` | Remove columns from each line of a file or files. |
| `grep` | Filter a file for lines matching a specified pattern. Can also be reversed to print out lines that don't match the specified pattern. |
| `gzip` (`gunzip`) | Compress (uncompress) a file. |
| `tar` | Archive or unarchive an entire directory into a single file. |

## Editing files

| Command | Description |
| --- | --- |
| `tr` | Substitute one character for another. Also useful for deleting characters. |
| `nano` | Very basic and easy to use text editor |
| `emacs` | Run the Emacs text editor (good for experts). |
| `vi` | Run the vi text editor (confusing even for experts). |
| `echo` | print text to the screen. E.g. `echo 'Hello World!'` |

## Connecting to other computers

| Command | Description |
| --- | --- |
| `ssh` | A secure (encrypted) way to log into machines. |
| `scp` | A secure way to copy (cp) files to and from remote machines. |

## Standard I/O and Redirection

Unix commands print output to the terminal (screen) for you to see, and accept input from the keyboard (that is, from *you*!)

Every Unix program starts out with three connections to the outside world.  These connections are called "streams", because they act like a stream of information (metaphorically speaking):

| Stream Name | Description |
|---|---|
| standard input | This is initially attached to the keyboard. When the program reads from standard input, it reads whatever text you type in. |
| standard output | This stream is initially attached to the terminal. Anything the program prints to this stream appears in your terminal window. |
| standard error | This stream is also initially attached to the terminal. It is a separate stream intended for printing error messages. |

The word "initially" might lead you to think that standard input, output, and error can somehow be detached from their starting places and reattached somewhere else. And you'd be right. You can attach one or more of these three streams to a file, a device, or even to another program. This sounds esoteric, but it is actually very useful.

## A Simple Example

The `wc` program counts lines, characters, and words in data sent to its standard input. You can use it interactively like this:

```
(~) 62% wc
Mary had a little lamb,
little lamb,
little lamb.

Mary had a little lamb,
whose fleece was white as snow.
^d                    # ** NOTE - this is ctrl-d which you type, but no output shows up
                      # in terminal window
      6      20     107
```

In this example, I ran the `wc` program. It waited for me to type in a little poem. When I was done, I typed the END-OF-FILE character, control-d (^d for short). `wc` then printed out three numbers indicating the number of lines (6), words (20), and characters (107) in the input.

More often, you'll want to count the number of lines in a big file and save the results in another file. We'll see this in two steps. First, we send the output of `wc` to stdout. In the second command, we *redirect* stdout from `wc` to a file with the `>` symbol.

```
(~) 63% wc big_file.fasta
      2943    2998     419272
(~) 64% wc big_file.fasta > count.txt
```

Now if you `cat` the file *count.txt*, you'll see that the data has been recorded.

```
(~) 65% cat count.txt
      2943    2998     419272
```

## Redirection Meta-Characters

Here's the complete list of redirection commands for `zsh`:

| Redirect command | Description |
|---|---|
| `< myfile.txt` | Redirect the contents of the file to standard input |
| `> myfile.txt` | Redirect standard output to file |
| `>> logfile.txt` | Append standard output to the end of the file |
| `1> myfile.txt` | Redirect just standard output to file (same as above) |
| `2> myfile.txt` | Redirect just standard error to file |
| `&> myfile.txt` | Redirect both stdout and stderr to file |

These can be combined.  For example, this command redirects standard input from the file named `/etc/passwd`, writes its results into the file `search.out`, and writes its error messages (if any) into a file named `search.err`.  What does it do?  It searches the password file for a user named "root" and returns all lines that refer to that user.

```
(~) 66% grep root /etc/passwd > search.out 2> search.err
```

For instance, the `find` command will look for files/directories/links.
When you try to look in a directory where you don't have permissions, you will get an error.
So, if I look from `/` for a file, I will get lots of error messages for directories that I can't read and that I'd like to ignore.
I can redirect to a file:

```
$ find / -name passwd.txt 2>err
```

That might actually produce a very large *err* file, and I don't actually care about the errors.
Instead, I can redirect them to a special filehandle called `/dev/null`, which throws them away entirely:

```
$ find / -name passwd.txt 2>/dev/null
```

## Filters, Filenames, and Standard Input

Many Unix commands act as filters, taking data from a file or standard input, transforming the data, and writing the results to standard output. Most filters are designed so that if they are called with one or more filenames on the command line, they will use those files as input. Otherwise they will act on standard input. For example, these two commands are equivalent:

```
(~) 67% grep 'gatttgc' big_file.fasta
(~) 68% grep 'gatttgc' < big_file.fasta
```

Both commands use the `grep` command to search for the string "gatttgc" in the file `big_file.fasta`. The first command is explicitly given the name of the file on the command line; the second one searches standard input, which is redirected from the file to stdin of `grep`.

Sometimes you want a filter to act on a series of files, one of which happens to be standard input. Many commands let you use `-` on the command line as an alias for standard input. Example:

```
(~) 70% grep 'gatttgc' big_file.fasta bigger_file.fasta -
```

This example searches for "gatttgc" in three places. First it looks in file `big_file.fasta`, then in `bigger_file.fasta`, and lastly in standard input (which, since it isn't redirected, will come from the keyboard).

## Standard I/O and Pipes

The coolest thing about the Unix shell is its ability to chain commands together into mini pipelines. Here's an example:

```
(~) 65% grep gatttgc big_file.fasta | wc -l
22
```

There are two commands here. `grep` searches a file or standard input for lines containing a particular string. Lines which contain the string are printed to standard output. `wc -l` is the familiar word count program, which counts words, lines, and characters in a file or standard input. The `-l` command line option instructs `wc` to print out just the line count. The `|` character, which is known as a "pipe", connects the two commands together so that the standard output of `grep` becomes the standard input of `wc`. Think of pipes connecting streams of flowing data.

What does this pipe do? It prints out the number of lines in which the string "gatttgc" appears in the file `big_file.fasta`.

## More Pipe Idioms

Pipes are very powerful. Here are some common command line idioms.

**Count the Number of Times a Pattern does NOT Appear in a File**

The example at the top of this section showed you how to count the number of lines in which a particular string pattern appears in a file. What if you want to count the number of lines in which a pattern does **not** appear?

Simple.  Reverse the test with the `-v` switch:

```
(~) 65% grep -v gatttgc big_file.fasta | wc -l
2921
```

### Uniquify Lines in a File

If you have a long list of names in a text file, and you want to weed out the duplicates:

```
(~) 66% sort long_file.txt | uniq > unique.out
```

This works by sorting all the lines alphabetically and piping the result to the `uniq` program, which removes duplicate lines that occur one after another.  That's why you need to sort first. The output is placed in a file named `unique.out`.

### Concatenate Several Lists and Remove Duplicates

If you have several lists that might contain repeated entries among them, you can combine them into a single unique list by concatenating them together, then sorting and uniquifying them as before:

```
(~) 67% cat file1 file2 file3 file4 | sort | uniq
```

### Count Unique Lines in a File

If you just want to know how many unique lines there are in the file, add a `wc` to the end of the pipe:

```
(~) 68% sort long_file.txt | uniq | wc -l
```

### Page Through a Really Long Directory Listing

Pipe the output of `ls` to the `more` program, which shows a page at a time.  If you have it, the `less` program is even better:

```
(~) 69% ls -l | more
```

### Monitor a Growing File for a Pattern

Pipe the output of `tail -f` (which monitors a growing file and prints out the new lines) to `grep`.  For example, this will monitor the `/var/log/syslog` file for the appearance of e-mails addressed to 'mzhang':

```
(~) 70% tail -f /var/log/syslog | grep dbrown
```

# Permissions

Often several different users work on the same computer. To help them work without disrupting each other, a set of permissions is attached to files and directories. These permissions allow or prevent reading, writing or executing a file. There are permissions for the user, group and other.  You can view permissions with an `ls -l` command.

```
dbrown@WideIsLove ~ % ls -l
total 16
drwx------+  3 dbrown  staff   96 Oct 15 14:09 Desktop
drwx------+  3 dbrown  staff   96 Oct 15 14:09 Documents
...
-rw-r--r--   1 dbrown  staff    9 Oct 15 14:12 expt1.countdata.tsv
-rw-r--r--   1 dbrown  staff   20 Oct 15 14:11 notes.txt
-rw-r--r--   1 dbrown  staff   23 Oct 16 13:37 print.py
```

The series of characters at the beginning of the line describe the permissions. The first character is

- `d`: directory
- `l`: link
- `-`: file

Then there are three triples of characters (- means not allowed, a letter means you have read, write, exectute permission). x for a directory means you can access the directory

```
rwx
---
```

for user, group, other, like so

```
 UsrGrpOth
drwxrwxrwx+
```

Users can belong to groups (who collaborate) and other is everyone else.

The `chmod` command (change mode) alters permissions. You add permissions with `+` and take them away with `-`

You can alter permissions for user `u`, group `g`, other `o` or all `a`

To make the print.py file executable (i.e. runnable like a program) you write

```
dbrown@WideIsLove ~ % ls -l print.py
-rw-r--r--  1 dbrown  staff  23 Oct 16 13:37 print.py
dbrown@WideIsLove ~ % chmod a+x print.py
dbrown@WideIsLove ~ % ls -l print.py
-rwxr-xr-x  1 dbrown  staff  23 Oct 16 13:37 print.py
dbrown@WideIsLove ~ %
```

To stop others from reading or executing the Python script, you write

```
dbrown@WideIsLove ~ % chmod o-rx print.py
dbrown@WideIsLove ~ % ls -l print.py
-rwxr-x---  1 dbrown  staff  23 Oct 16 13:37 print.py
```

To convert a script into an exectuable program, you'll need to make it executable with `chmod` and then add the current directory to your `$PATH` (we'll see how to do this later) for you, you can type the command in your terminal window.

```
export PATH=".:${PATH}" # add '.', the current dir to the list of paths
                        # the OS looks for programs in
rehash    # this rebuilds set of paths for the shell (zsh, tcsh, not bash)
```

NOTE: You must use the `$` when you want the shell to interpolate the value of the variable called `PATH`, but you do *not* use it when setting the value.

You can use `echo` to inspect your `$PATH` variable, which is a colon-separated list of directories where the OS will search for executables.
You can pipe this output to `tr` to change the colon (`:`) to newlines to make it readable:

```
$ echo $PATH | tr : '\n'
/Library/Frameworks/Python.framework/Versions/3.12/bin
/opt/homebrew/bin
/opt/homebrew/sbin
/usr/local/bin
/usr/bin
/bin
/usr/sbin
/sbin
/Users/kyclark/.local/bin
```

# More Unix

Here are a few more advanced Unix commands that are very useful, and when you have time you should investigate further. We list the page numbers for the Linux Phrasebook Second Edition by Scott Granneman or links to online tutorials.

- `awk` (Linux Phrasebook p.194-198) online tutorial
- `sed` (Linux Phrasebook p.188-194) online tutorial
- `perl` one-liners online tutoral
- `for` loops online tutorial

---

# Link to Unix 1 Problem Set

# Unix 2

## Text Editors

It is often necessary to create and write to a file while using the terminal. This makes it essential to use a terminal text editor. There are many text editors out there. Some of our favorite are Emacs and vim. We are going to start you out with a simple text editor called `vi`

## Introduction to vi

What is **vi**?

> **vi** is a command line text editor. vi is included in every Linux installation. You don't have to install it, ever.

What is a command line text editor?

> A command line text editor is an text editor that you use from the command line. In most command line text editors, don't expect to be able to point and click. You will need to naviage with keyboard key strokes. The two most popular text editors are **vi** and **emacs**. You are free to use either, but we will start with **vi** since the keystrokes are less complex than in **emacs**.

Why do I care about command line text editors?

> If you are logged into a remote machine, a command line text editor is the fastest, easiest, most efficient way to write text files.

### Getting Started with vi

#### Opening a file

On the command line, type `vi` followed by a file name.

```
srobb% vi <file>
```

Let's try it:

```
srobb% vi first_vi_file.txt
```

You will see this in your terminal.

```
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"first_vi_file.txt" [New File]
```

Notice the file name at the bottom.

If you **do not** include a file name you will see something similar to this:

```
~
~
~
~
~                                          VIM - Vi IMproved
~
~                                          version 8.0.1283
~                                     by Bram Moolenaar et al.
~                                   Vim is open source and freely distributable
~
~                                          Become a registered Vim user!
~                                   type  :help register<Enter>   for information
~
~                                   type  :q<Enter>               to exit
~                                   type  :help<Enter>  or  <F1>  for on-line help
```

```
~                                   type  :help version8<Enter>   for version info
~
~
~
```

Read what the message says and type `:q<Enter>` to **Q**uit or exit.

**vi has two modes.**

1. **Insert Mode**
2. **Command Mode**

**Insert Mode** is for typing your file contents. All keyboard strokes will be interpreted as characters you want to see in your file.

**Command Mode** is for using commands. All keyboard stokes will be interpreted as commands and ***not*** as part of your file. Common commands are for deleting, copying, searching, replacing, and saving.

## Creating, Writing, And Saving a File Walk through

**Create**

From the command line open a new file by typing

`vi first_vi_file.txt`

**Write**

Start typing content. To do this we need to enter **Insert Mode**.

To do this type `i`.

Your vi session will now look like this:

```
~
~
~
~
~
~
-- INSERT (paste) --
```

Notice the `INSERT` at the bottom of the screen.

Start typing your file contents. Remember that all keystrokes are ones you want to see in your file and that your mouse will not work.

#### Save

Now that the file contains some content let's enter **Command Mode** so that we can save our file.

1. Press the `<ESC>` key to enter **Command Mode**.
2. type `:w` (colon followed by a w) to **Save (Write)**

If you want to type some more content, enter **Insert Mode** (`i`).

If, instead you want to exit, since you are already in Command Mode you can use the quit keystrokes `:q`

## Common Activities and vi Commands

Enter into **Command Mode** for all commands. If you are unsure that you are in **command mode**, just press the `<esc>` key. It will not hurt if you are already in **Command Mode**

#### Saving and Exiting

Remember to enter into **Command Mode** with `<esc>` key

| key stroke | meaning |
| --- | --- |
| `:wq` | Save (**W**rite) and **Q**uit |
| `:q!` | **Q**uit without Saving!!! |
| `:w` | Save (**W**rite) Only |

Most commands within vi are executed as soon as you press a sequence of keys. Any command beginning with a colon ( : ) requires you to hit `<enter>` to complete the command.

#### Getting around

Remember to enter into **Command Mode** with `<esc>` key

| key stroke | meaning |
| --- | --- |
| Arrow keys | move the cursor around |
| `j`, `k`, `h`, `l` | move the cursor down, up, left and right (similar to the arrow keys) |
| `0` (zero) | move cursor to beginning of current line |
| `^` (caret) | move cursor to beginning of current line |
| `$` | move cursor to end of the current line |
| `:n` | move to the nth line in the file |
| `nG` | move to the **n**th line (eg 5G moves to 5th line) |
| `G` | move to the last line |
| `w` | move to the beginning of the next word |
| `nw` | move forward n word (eg 2w moves two words forwards) |
| `b` | move to the beginning of the previous word |
| `nb` | move back n word |

#### Deleting content

Remember to enter into **Command Mode** with `<esc>` key

| key stroke | meaning |
|---|---|
| `x` | delete a single character |
| `nx` | delete n characters (eg 5x deletes five characters) |
| `dd` | delete the current line |
| `dn` | d followed by a movement command. Delete to where the movement command would have taken you. (eg d5w means delete 5 words) |
| `yy` | "yank" (copy) a line |
| `p` | "paste" a line that was deleted or yanked |

**Undoing**

Remember to enter into **Command Mode** with `<esc>` key

| key stroke | meaning |
|---|---|
| `u` | Undo the last action (you may keep pressing u to keep undoing) |
| `U` | (Note: capital)** - Undo all changes to the current line |

## Other Useful Tips

| key stroke | meaning |
|---|---|
| `:set number` | display line numbers |
| `:set nonumber` | turn off line numbers |
| `:/[search text]` | find [this text] in your file |
| `n` | go to next occurance of your search result |

## Mug of vi


## FILE COMMANDS

| | |
|---|---|
| `vi` *filename(s)* | edit a file or files |
| `vi -r` *filename* | retrieve saved file after crash |
| `ZZ`, `:wq`, `:x` | save and exit |
| `:q`, `:q!` | quit; quit without saving |
| `:w`, `:w` *fn* | save file, save file as *fn* |
| `:e` *filename* | edit *filename* |
| `:r` *filename* | insert *filename* |
| `:sh` | drop to shell |
| `:!`*cmd* | run command *cmd* |
| `:r !`*cmd* | execute *cmd* and insert output |
| `!`*movement cmd* | pipe lines in *movement* through *cmd* |

## SEARCH AND REPLACE

| | |
|---|---|
| `/`*txt*, `?`*txt* | find *txt* forward or backward |
| `/^`*txt* | find next line that starts with *txt* |
| `n`, `N` | repeat last search forward, backward |
| `R` | replace text from current character |

## DELETING/INSERTING TEXT

| | |
|---|---|
| `dw`, `dd`, `x` | delete word, line, character |
| *n*`dd`, *n*`x` | delete *n* lines, *n* characters |
| `x`, `X` | delete character forward, backward |
| `D`, `d$` | delete to end of line |
| `d`*motion* | delete from cursor to *motion* ($, 0, etc.) |
| `:>`, `:<` | indent, outdent line |
| `S` | replace text with blank line |
| `o`, `O` | insert new line below, above current line |
| `u` | undo last change |
| `.` | repeat last change |

## CUT/COPY/PASTE

| | |
|---|---|
| *n*`yy`, *n*`Y` | copy *n* lines |
| `yw`, `yy` | copy word, line |
| `p`, `P` | paste text after, before cursor |
| `a`, `i` | insert text after, before cursor |
| `A`, `I` | insert text end, beginning of line |

## MOVING AROUND

| | |
|---|---|
| *n*`G` | move to line *n* |
| `h`, `l`, `k`, `j` | left, right, up, down one character |
| *n*`b`, *n*`w` | left or right *n* words |
| `CTRL-B`, `F` | backward, forward one screen |
| `CTRL-U`, `D` | up, down one screen |
| `$`, `G` | go to end of line, end of file |
| `0` | go to beginning of line (zero) |
| `)`, `(` | move to next, previous sentence |
| `}`, `{` | move to next, previous paragraph |
| `w`, `b` | move forward, back one word |
| `e` | go to end of current or next word |

## WICKED COOL STUFF

| | |
|---|---|
| `~` | change case |
| `xp` | transpose characters |
| `J` | combine current line with next |
| `m`*p* | create a mark called *p* |
| `` ``*p* | return to *p* |
| `d`` `x`, `y`` `x` | del, copy text from mark to cursor |
| `:>` *n* | indent *n* lines |

THE MUG OF VI
no starch press

# Git for Beginners

Git is a tool for managing files and versions of files. It is a *Version Control System*. It allows you to keep track of changes. You are going to be using Git to manage your course work and keep your copy of the lecture notes and files up to date. Git can help you do very complex task with files. We are going to keep it simple.

## The Big Picture.

A Version Control System is good for Collaborations, Storing Versions, Restoring Previous Versions, and Managing Backups.

### Collaboration

Using a Version Control System makes it possible to edit a document with others without the fear of overwriting someone's changes, even if more than one person is working on the same part of the document. All the changes can be merged into one document. These documents are all stored one place.

### Storing Versions

A Version Control System allows you to save versions of your files and to attach notes to each version. Each save will contain information about the lines that were added or altered.

### Restoring Previous Versions

Since you are keeping track of versions, it is possible to revert all the files in a project or just one file to a previous version.

### Backup

A Version Control System makes it so that you work locally and sync your work remotely. This means you will have a copy of your project on your computer and the Version Control System Server you are using.

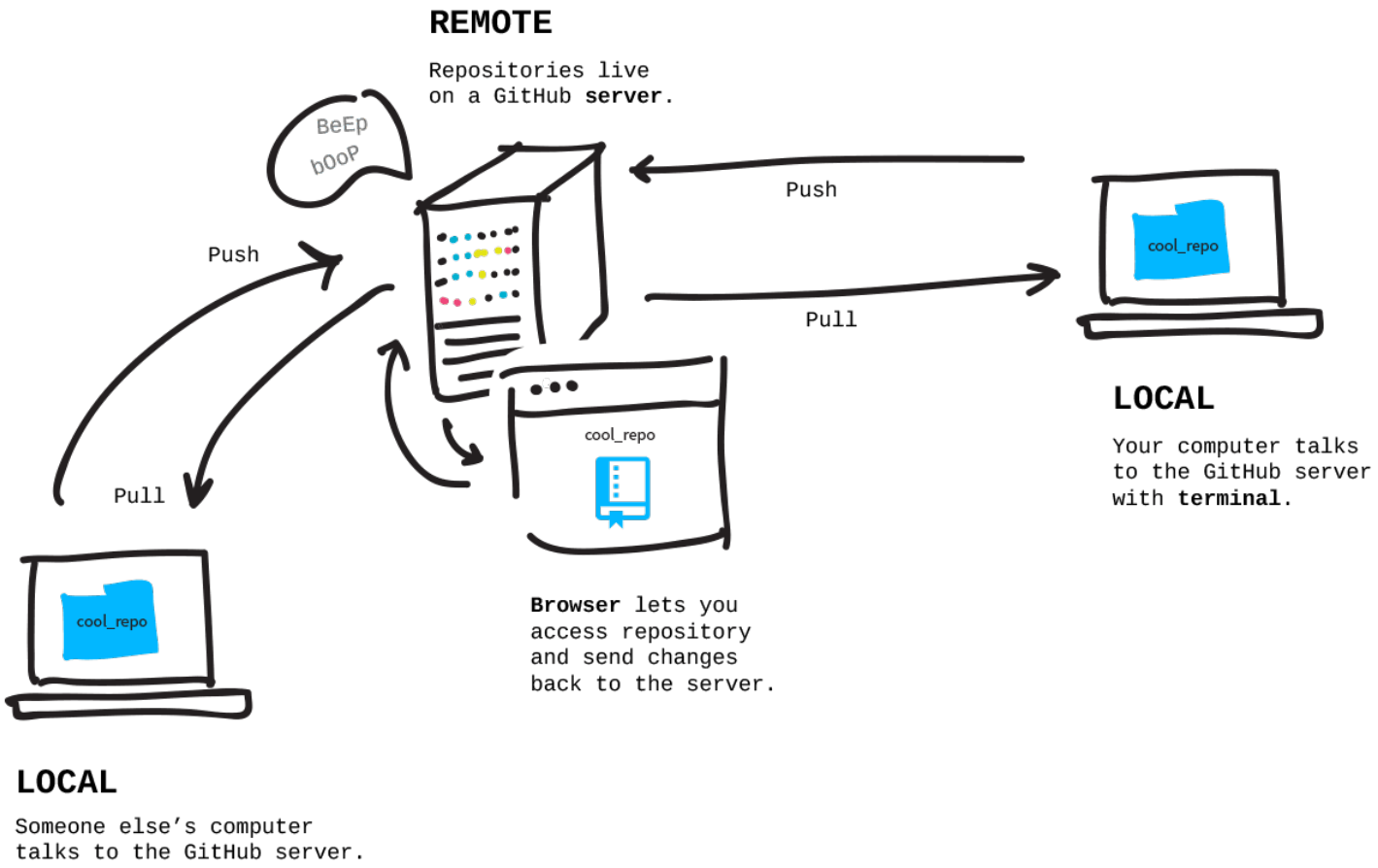
### The Details

git is the Version Control System we will be using for tracking changes in our files.

GitHub is the Version Control System Server we will be using. They provide free account for all public projects.

## The Basics

Usually you have a local copy of your project/repository and a remote copy. The **local** repository is stored on your computer and the **remote** is on a online service like GitHub.




You can use a web browser to interact with the remote server (gitHub) and the terminal to interact with the local repository.

## SSH Keys

SSH Keys are needed to write to a personal repository. Lets set this up first

Github requires authentication with the use of ssh keys. Essentially, our github repos are LOCKED, we need a KEY to write to them. We will generate the a key (private) and a lock (public). Then We tell github about our lock and keep our key to ourselves.

Here is a great GitHub Tutoral.

## Adding a new SSH key to your GitHub account

Here is a summary of the steps:

## Generating a new SSH KEY

Make your key and add your email address.

```
$ ssh-keygen -t ed25519 -C "your_email@example.com"
```

Save your key and lock in your `.ssh` directory. It will prompt you for a path, the default is usually a good place to store your SSH key files

```
> Enter a file in which to save the key (/Users/YOU/.ssh/id_ALGORITHM: [Press enter]
```

Next, pick a passphrase, something easy. It is a password, so that not just anyone on your physical computer can access your GitHub account. You CAN leave it blank.

```
> Enter passphrase (empty for no passphrase): [Type a passphrase]
> Enter same passphrase again: [Type passphrase again]
```

## Adding your SSH key to the ssh-agent

```
 $ eval "$(ssh-agent -s)"
> Agent pid 59566
```

First, check to see if your ~/.ssh/config file exists in the default location.

```
  $ open ~/.ssh/config
> The file /Users/YOU/.ssh/config does not exist.
```

If the file doesn't exist, create the file.

```
$ touch ~/.ssh/config
```

Open your ~/.ssh/config file using `vi` and add the following content:

```
Host *
  IdentityFile ~/.ssh/id_ed25519
```

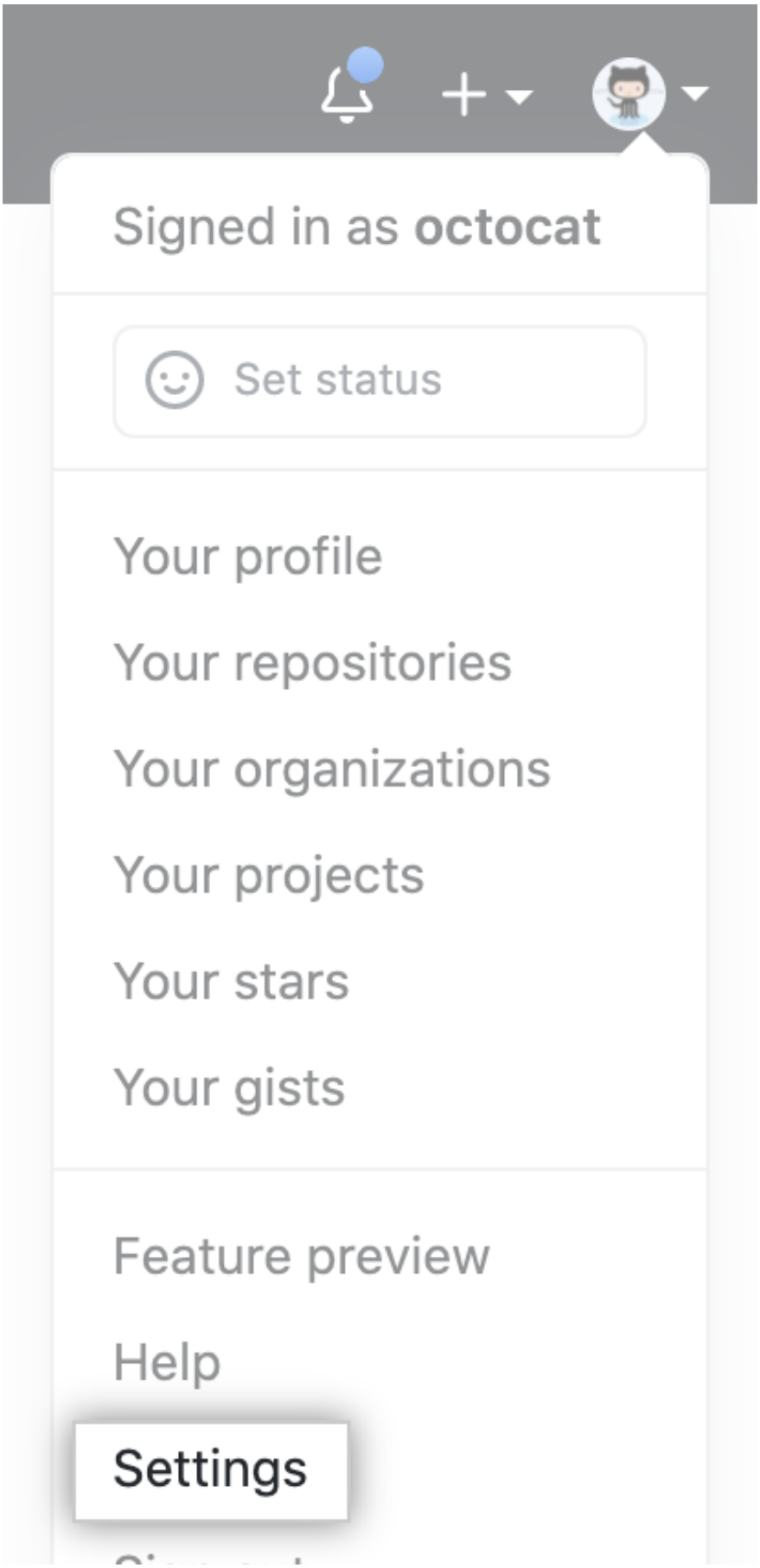
# Adding a new SSH key to your GitHub account

Print the contents of your PUBLIC ssh key file (our lock) and paste them into your github account

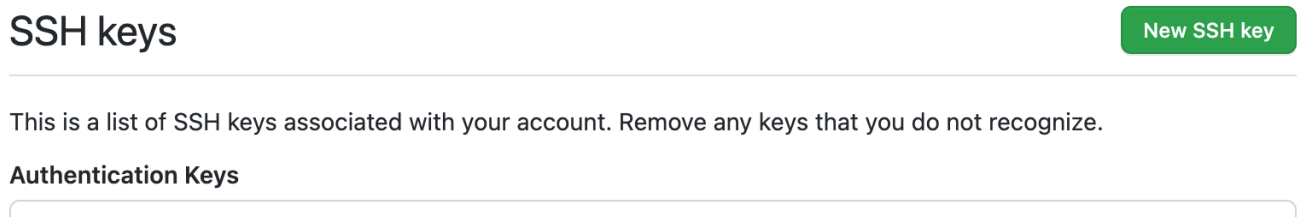
```
$ cat  ~/.ssh/id_ed25519.pub
```

# Paste into your GitHub account

1. Go to Settings




2. In the "Access" section of the sidebar, click  "SSH and GPG keys".
3. Click New SSH key or Add SSH key. https://github.com/settings/keys




4. Add a title, for example "PFB CSHL KEY"


.

5. Click "Add SSH Key"


## Do this to ensure that you do not encounter errors by attemping to add files that are too large

```
cd ~/PFB_problemsets/.git/hooks/
curl -O https://raw.githubusercontent.com/prog4biol/pfb2025/master/setup/pre-commit
```

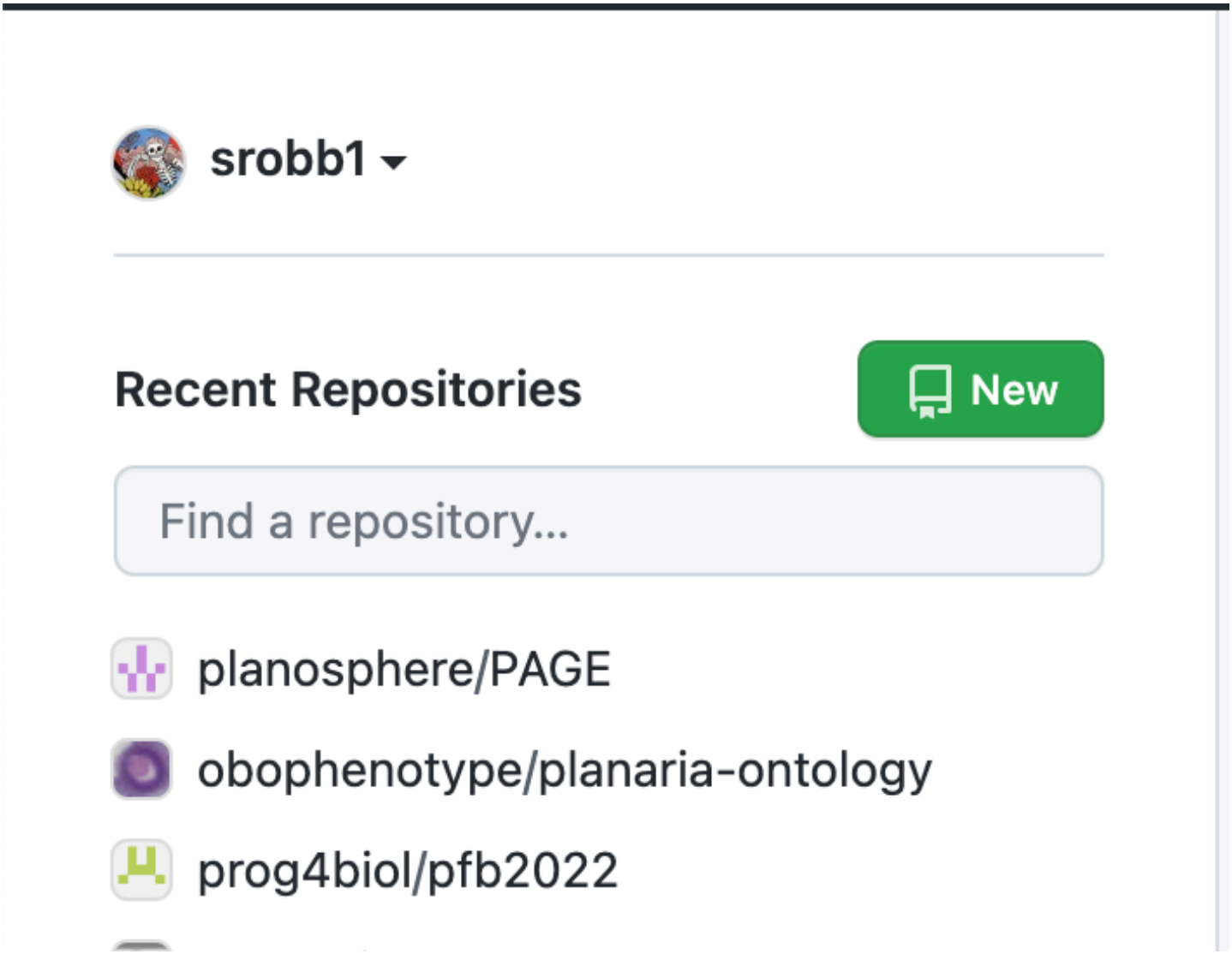
## Creating a new repository

A repository is a project that contains all of the project files, and stores each file's revision history. Repositories can have multiple collaborators. Repositories usually have two components, one **remote** and one **local**.

Use Steps 1 and 2 below to create the **remote repository**.

Then, use Step 3 to create your **local repository** and **link it** to the **remote repository**.

1. Navigate to GitHub --> Create Account / Log In --> Click on your icon --> Click "Your Repositories" --> Click 'New'




2. Add a name (i.e., PFB_problemsets) and a description (i.e., Solutions for PFB Problem Sets) and click "Create Repository"

# Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

**Repository template**

Start your repository with a template repository's contents.

No template

**Owner** * **Repository name** *

srobb1 / PFB_problemsets

Great repository names are short and memorable. Need inspiration? How about **effective-octo-waffle**?

**Description** (optional)

Solutions for PFB Problemsets

**Public**
Anyone on the internet can see this repository. You choose who can commit.

**Private**
You choose who can see and commit to this repository.

**Initialize this repository with:**

Skip this step if you're importing an existing repository.

**Add a README file**
This is where you can write a long description for your project. Learn more.

**Add .gitignore**

Choose which files not to track from a list of templates. Learn more.

.gitignore template: None

**Choose a license**

A license tells others what they can and can't do with your code. Learn more.

License: None

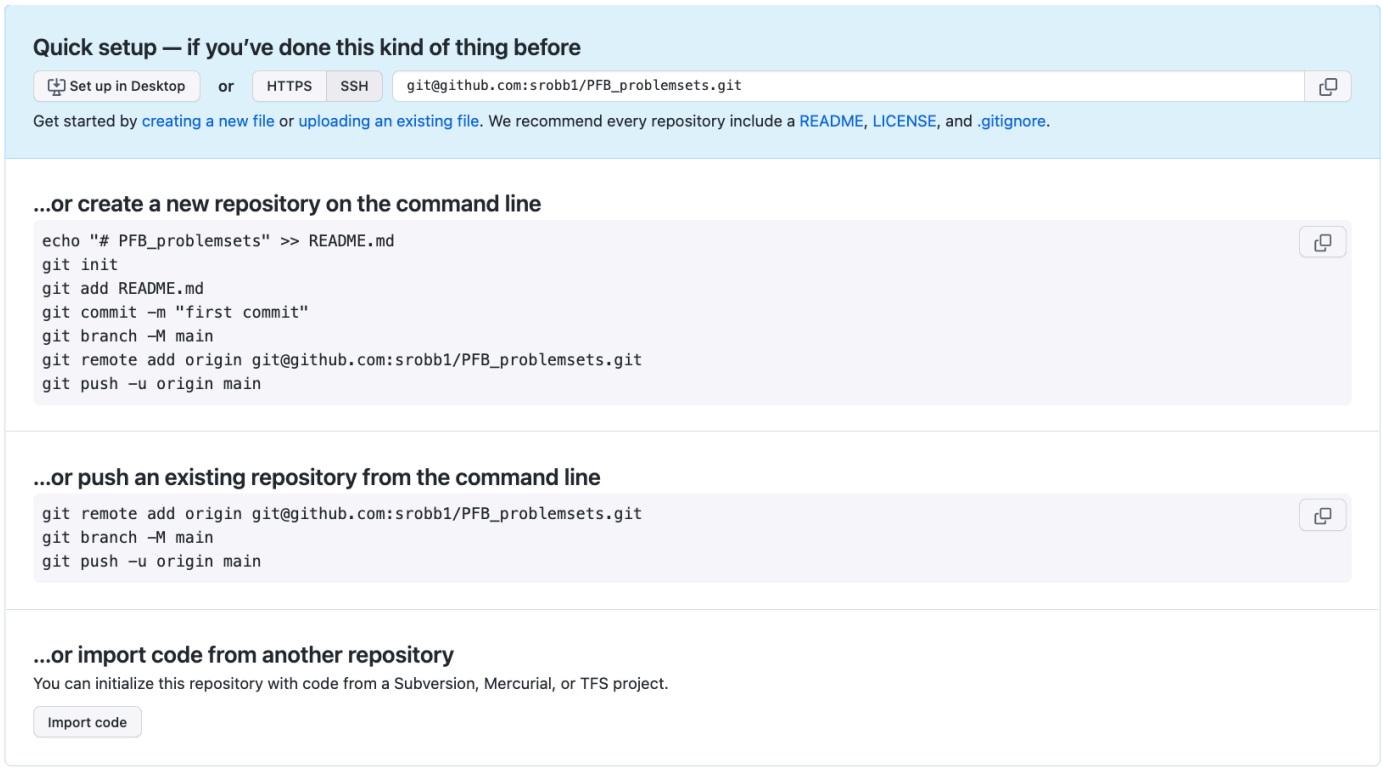
You are creating a public repository in your personal account.

Create repository

3. Create a directory on your computer in your home directory

```
cd ~
mkdir PFB_problemsets
```

4. Make sure to select SSH in the GitHub new repository window.
5. Now follow the directions provided by github.



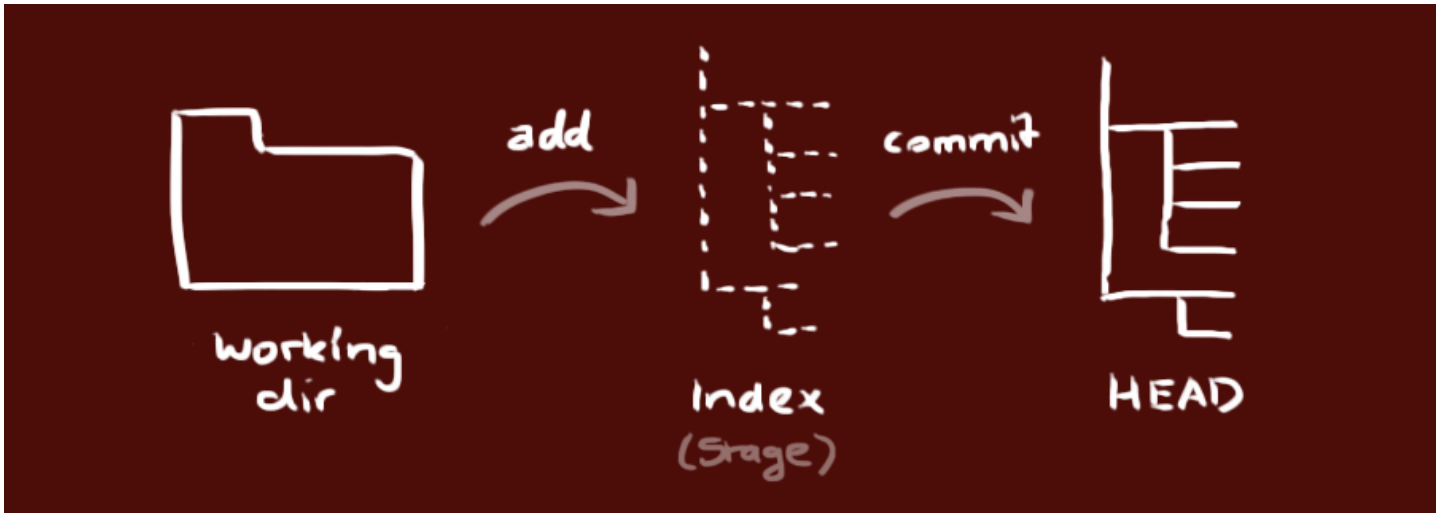

6. If you did not select SSH and use the SSH URL, do a `git remote set-url 'git@blalabbalalal'` with the SSH URL

The new local repository consists of three "trees" maintained by git. The first one is your "Working Directory" which holds the actual files. the second one is the "Index" which acts as a staging area and finally the "HEAD" which points to the last commit you've made.

Every git local repository has three main elements called *trees*:

1. The *Working Directory* contains your files
2. The *Index* is the staging area
3. The *HEAD* points to the last commit you made.

> There are a few new words here. We will explain them as we go




**Command Review**

| command | description |
|---|---|
| `git init` | Creates your new local repository with the three trees (local machine) |
| `git remote add remote-name URL` | Links your local repository to a **remote** repository that is often named *origin* and is found at the given URL |
| `git add filename` | Propose changes and add file(s) with changes to the index or staging area (local machine) |
| `git commit -m 'message'` | Confirm or commit that you really want to add your changes to the HEAD (local machine) |
| `git push -u remote-name remote-branch` | Upload your committed changes in the HEAD to the specified remote repository to the specified branch |

**Using your new repository**

You created a **local** repository above. It is linked to a **remote**. The `git remote add` command connects your local to the remote. Before this command the local will not know anything about your remote and vice versa.

**Review:**

1. You created a new remote repository on the github website.
2. You followed all the instructions given to you on github when you created your repository.

**Now we will add some files to your new repository:**

1. Change directory to your local repository
2. Create a new file with vi: `vi git_exercises.txt`
3. Add a line of text to the new file.
4. Save `:w` and Exit `:q`
5. (Add) Stage your changes. `git add git_exercises.txt`
6. (Commit) Become sure you want your changes. `git commit -m 'added a line of text'`
7. (Push) Sync/Upload your changes to the **remote** repository. `git push origin main`

That is all there is to it! There are more complicated things you can do, but we won't get into those. You will know when you are ready to learn more about git when you figure out there is something you want to do but don't know how. There are thousands of online tutorials for you to search and follow.

## Keeping track of differences between local and remote repositories

If you are ever wondering what do you need to add to your remote repository use the `git status` command. This will provide you with a list of files that have been modified, deleted, and those that are untracked. Untracked files are those that have never been added to the staging area with `git add`

| command | description |
|---|---|
| `git status` | To see a list of files that have been modified, deleted, and those that are untracked |

## Deleting and moving files

| command | description |
|---|---|
| `git rm` | Remove files from the index, or from the working tree and from the index |
| `git mv` | Move or rename a file, a directory, or a symlink |

> these two commands will update your index as well as change your local files. If you use just `rm` or `mv` you will have to update the index with add/commit.

## Get a copy of file on your remote

Sometimes you really really mess up a file, or you delete it by mistake. You have a small heart attack then you remember that you have a good copy in your remote github repo. How do you get it in your local repo?

```
git checkout <filename>
```

> Whew, what a life saver!

## Tips

1. Adding files over 50M will break your git repo. Don't add large files. Don't blindly use `git add -A` when there might be large files present. You will be very sad if you do.
2. Don't clone a git repository into another git repository. This makes git really unhappy.
3. Don't be afraid to ask your questions on Google. git can be complicated and a lot of people ask a lot of questions that get answered in online forums, or GitHub will have a tutorial

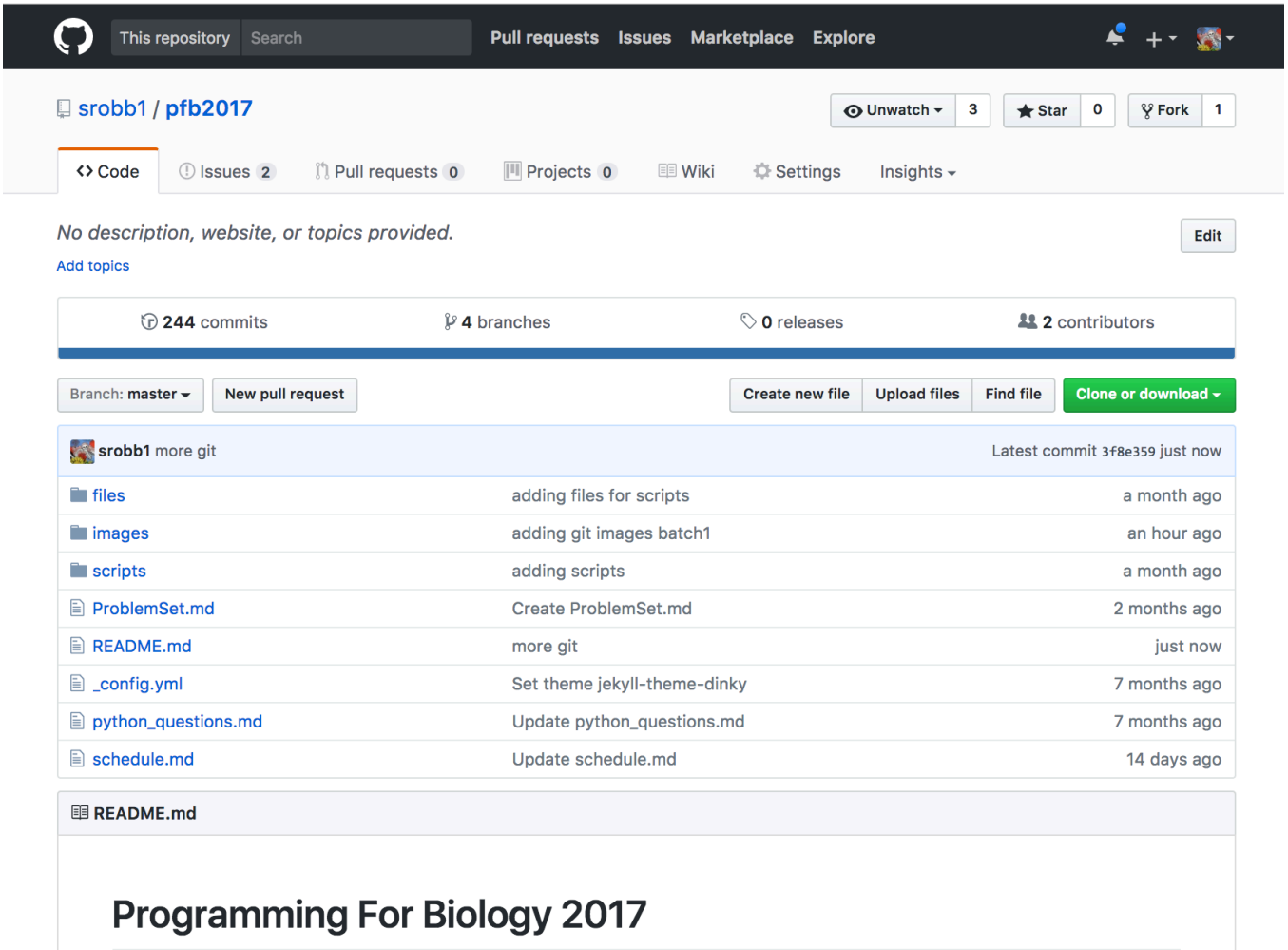
# Cloning a Repository

Sometimes you want to download and use someone else's repository. This is different from above where we created our own repository. This is just a copy of someone else's repository

Let's clone the course material.

Let's do it!

1. Go to our PFB GitHub Repository
2. Click the 'Code' or 'Download' Button




3. Copy the URL: git@github.com:prog4biol/pfb2025.git
4. *Clone* the repository to your local machine. On the command line use this command:
   `git clone git@github.com:prog4biol/pfb2025.git`

Now you have a copy of the course material on your computer!

### Bringing Changes in from the Remote Repository to your Local Repository

If changes are made to any of these files in the online, remote repository, and you want to update your local copy, you can *pull* the changes.
`git pull`

| command | description |
| --- | --- |
| `git pull` | To get changes from the remote into your local copy |

## Links to *slightly* less basic topics

You will KNOW if you need to use these features of git.

1. View Commit History
2. Resolving Merge Conflicts
3. Undoing Previous Commits

---

## Link To Unix 2 Problem Set

---