Programming For Biology 2018

Instructors
Simon Prochnik
Sofia Robb

Table of Contents

Programming For Biology 2018

Instructors
Simon Prochnik
Sofia Robb

Big Picture

Why?

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

You probably 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

Unix 1

Unix Overview

What is the Command-Line?

Underlying the pretty Mac OSX GUI is a powerful command-line operating system. The command-line gives you access to the internals of the OS, and is also a convenient way to write custom software and scripts.

Many bioinformatics tools are written to run on the command-line and have no graphical 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 tasks without human intervention.

In this course, we will be writing Python scripts that are completely command-line based.

The Basics

Logging into Your Workstation

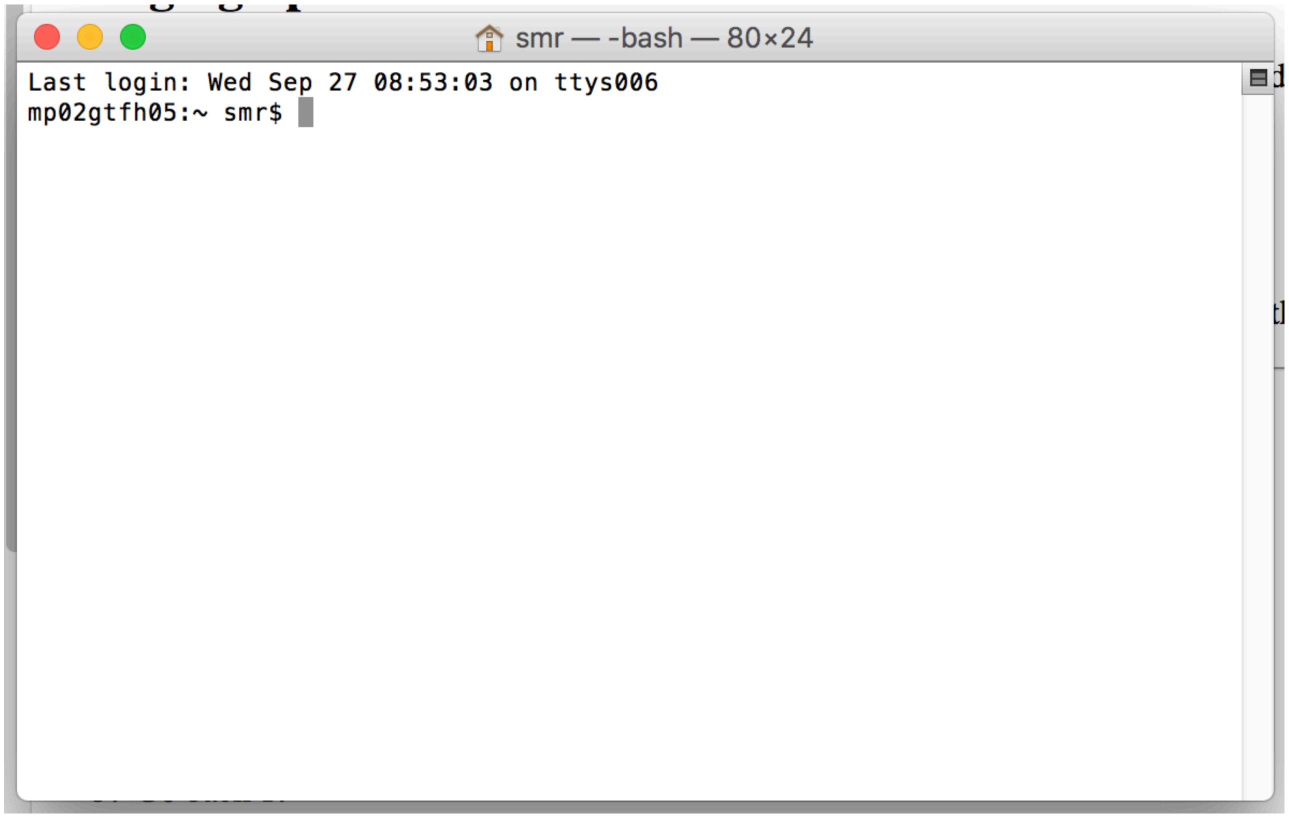
Your workstation is an iMac. To log into it, provide the following information:

Your username: admin

Your password: cshl

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:



```
Last login: Wed Sep 27 08:53:03 on ttys006
mp02gtfh05:~ smr$
```

You can open several Terminal windows at once. This is often helpful.

You will be using this application a lot, so I suggest that you drag the Terminal 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 "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 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 different and somewhat confusing features. We have set up your accounts to use **bash**. Stay with **bash** and you'll get used to it, eventually.

Command-Line Prompt

Most of bioinformatics is done with command-line software, 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 get depends on how the system administrator has customized your login. You can customize it yourself when you know how.

The prompt tells you the shell is ready to accept a command. When a long-running command is going, the prompt will not reappear until the system is ready to deal with your next request.

Issuing Commands

Type in a command and press the <Enter> key. 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 either launch a new shell (from Terminal's File menu), or run the command in the background by adding an ampersand after the command

```
(~) 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 comment you press , 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 for 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 reissue previous commands, possibly after modifying them.

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 immediate previous 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.

bash offers automatic command completion and spelling correction. If you type part of a command and then the tab 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>, **bash** 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 **tcsh** manual page. For example, if you want to match files that begin with the characters "f" or "g" and end with ".txt", you can use a range of characters inside square brackets `[f-g]` as part of the wildcard pattern. Here's an example

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

Home Sweet Home

When you first log in, you'll be placed in a part of the system that is your personal directory, called the *home directory*. You are free to do with this area what you will: in particular you can create and delete files and other directories. In general, you cannot create files elsewhere in the system.

Your home directory lives somewhere in the filesystem. On our iMacs, it is a directory with the same name as your login name, located in `/Users`. The full directory path is therefore `/Users/username`. Since this is a pain to write, the shell allows you to abbreviate it as `~username` (where "username" is your user name), or simply as `~`. The weird character (called "tilde" or "twiddle") is usually hidden at the upper left corner of your keyboard.

To see what is in your home directory, issue the command `ls -F`:

```
(~) % ls -F
INBOX         Mail/         News/         nsmail/       public_html/
```

This shows one file "INBOX" and four directories ("Mail", "News") and so on. (The `-F` in the command turns on fancy mode, which appends special characters to directory listings to tell you more about what you're seeing. `/` at the end of a filename means that file is a directory.)

In addition to the files and directories shown with `ls -F`, there may be one or more hidden files. These are files and directories whose names start with a `.` (called the "dot" character). To see these hidden files, add an `a` to the options sent to the `ls` command:

```
(~) % ls -aF
./                  .cshrc              .login              Mail/
../                 .fetchhost          .netscape/          News/
.Xauthority         .fvwmrc             .xinitrc*           nsmail/
.Xdefaults          .history            .xsession@          public_html/
.bash_profile       .less               .xsession-errors
.bashrc             .lessrc             INBOX
```

> Whoa! There's a lot of hidden stuff there. But don't go deleting dot files. Many of them are essential configuration files for commands and other programs. For example, the `.profile` file contains configuration information for the **bash** shell. You can peek into it and see all of **bash**'s many options. You can edit it (when you know what you're doing) in order to change things like the command prompt and command search path.

Getting Around

You can move around from directory to directory using the `cd` command. 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 to see where you are (or rely on the prompt, if configured):

```
(~/docs/grad_course/i) 56% cd
(~) 57% cd /
(/) 58% ls -F
bin/          dosc/         gmon.out      mnt/          sbin/
boot/         etc/          home@         net/          tmp/
cdrom/        fastboot      lib/          proc/         usr/
dev/          floppy/       lost+found/   root/         var/
(/) 59% cd ~/docs/
(~/docs) 60% pwd
/usr/home/lstein/docs
(~/docs) 62% cd ../projects/
(~/projects) 63% ls
Ace-browser/                   bass.patch
Ace-perl/                      cgi/
Foo/                           cgi3/
Interface/                     computertalk/
Net-Interface-0.02/            crypt-cbc.patch
Net-Interface-0.02.tar.gz      fixer/
Pts/                           fixer.tcsh
Pts.bak/                       introspect.pl*
PubMed/                        introspection.pm
SNPdb/                         rhmap/
Tie-DBI/                       sbox/
ace/                           sbox-1.00/
atir/                          sbox-1.00.tgz
```

```
bass-1.30a/                zhmapper.tar.gz
bass-1.30a.tar.gz
(~/projects) 64%
```

> 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 arbitrarily weird things like this:

```
(~/docs) 65% cd ../../lstein/docs
```

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

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

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

Essential Unix Commands

With the exception of a few commands that are built directly into the shell, all Unix commands are standalone 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`, `/usr/bin`, or `/usr/local/bin`.

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
```

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 a single character, 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 `-c`, `-w` and/or `-l` options 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 short 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 and Funny Characters

The shell uses whitespace (spaces, tabs and other non-printing characters) to separate arguments. If you want to embed whitespace 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 contains 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 ASCII code is **nnn**

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.
`rm`	Remove (delete) a file.
`mkdir`	Make a directory
`rmdir`	Remove a directory
`ln`	Create a symbolic or hard link.
`chmod`	Change the permissions of a file or directory.

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.
`echo`	print a copy of some text to the screen. E.g. `echo 'Hello World!'`
`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.
`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.
`wc`	Count words, lines and/or characters in one or more files.
`tr`	Substitute one character for another. Also useful for deleting characters.
`sort`	Sort the lines in a file alphabetically or numerically.
`uniq`	Remove duplicated lines in a file.
`cut`	Remove columns from each line of a file or files.
`fold`	Wrap each input line to fit in a specified width.
`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.
`emacs`	Run the Emacs text editor (good for experts).
`vi`	Run the vi text editor (better for experts).

Networking

Command	Description
`ssh`	A secure (encrypted) way to log into machines.
`scp`	A secure way to copy (cp) files to and from remote machines.
`ping`	See if a remote host is up.
`ftp` / `sftp` (secure)	transfer files using the File Transfer Protocol.

Standard I/O and Redirection

Unix commands communicate via the command-line interface. They can print information out to the terminal 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 Type	Description
standard input	This is a communications stream 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 channel appears in your terminal window.
standard error	This stream is also initially attached to the terminal. It is a separate channel 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
      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, words and characters in the input.

More often, you'll want to count the number of lines in a big file; say a file filled with DNA sequences. You can do this by *redirecting* the contents of a file to the standard input of `wc`. This uses the `<` symbol:

```
(~) 63% wc < big_file.fasta
      2943    2998     419272
```

If you wanted to record these counts for posterity, you could redirect standard output as well using the `>` symbol:

```
(~) 64% wc < big_file.fasta > count.txt
```

Now if you `cat` the file *count.txt*, you'll see that the data has been recorded. `cat` works by taking its standard input and copying it to standard output. We redirect standard input from the *count.txt* file, and leave standard output at its default, attached to the terminal:

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

Redirection Meta-Characters

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

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 2>&1`	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
```

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:

```
(~) 66% grep 'gatttgc' < big_file.fasta
(~) 67% 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 one searches standard input, which happens to be redirected from the file. The second command is explicitly given the name of the file on the command line.

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:

```
(~) 68% 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 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 data flowing.

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 mzhang
```

Advanced 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 in the Internet Version (v3) of 'The Linux Command Line' by William Shotts.

- `awk`
- `sed` (p.295)
- `perl` one-liners
- `for` loops (p. 453)

Link to Unix 1 Problem Set

Unix 2

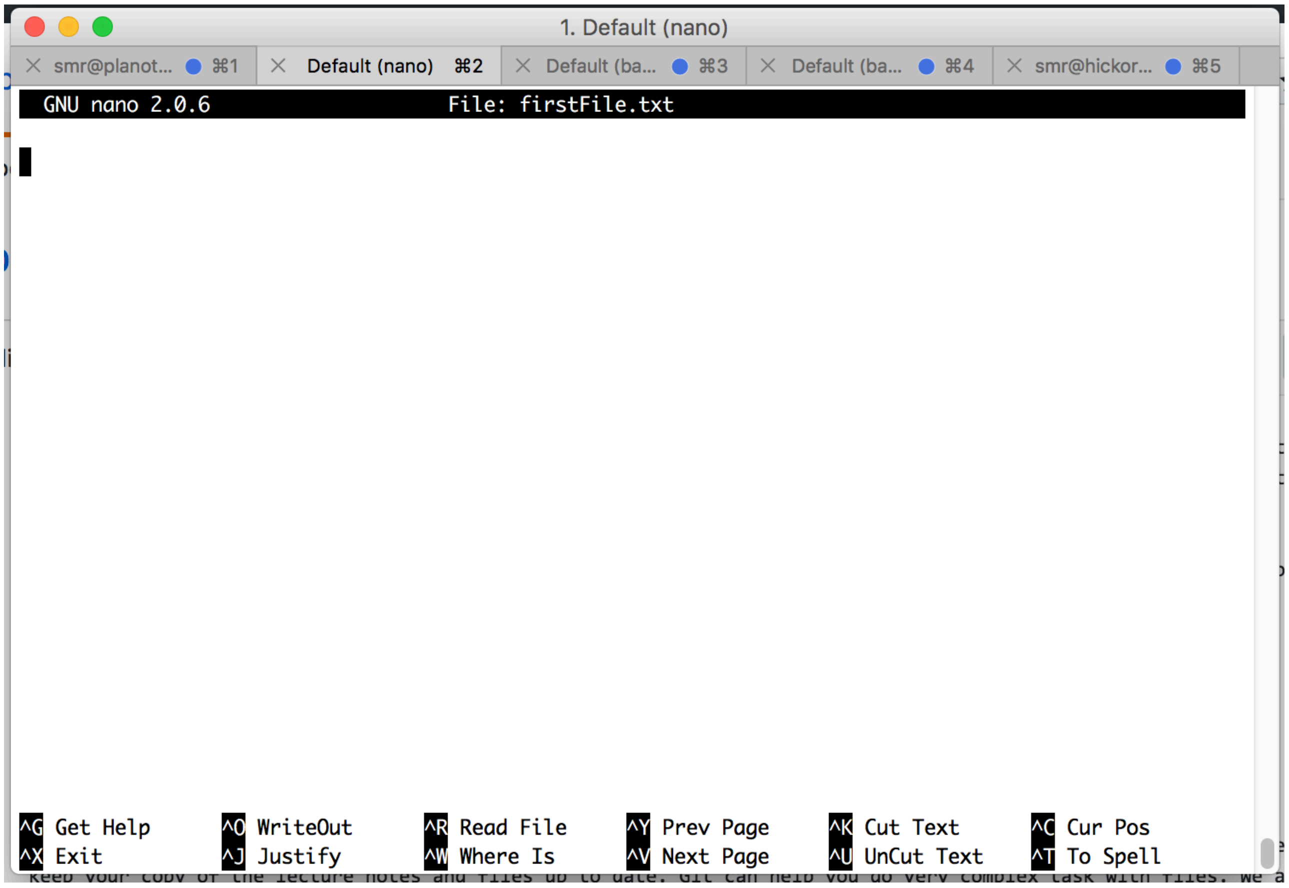
Text Editors

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 `nano`

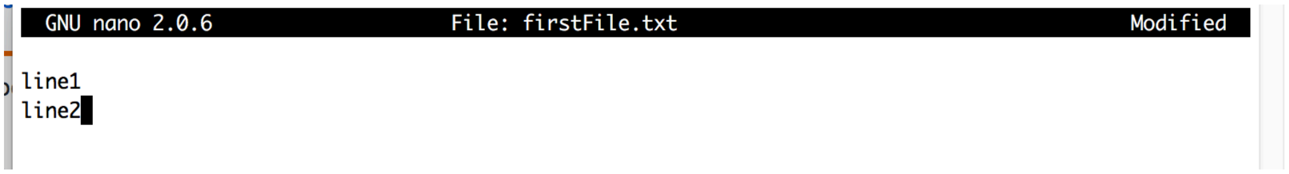
The way you use nano to create a file is simply by typing the command *nano* followed by the name of the file you wish to create.

```
(~) 71% nano firstFile.txt
```

This is what you will see:



```
  GNU nano 2.0.6                File: firstFile.txt






^G Get Help     ^O WriteOut     ^R Read File    ^Y Prev Page    ^K Cut Text     ^C Cur Pos
^X Exit         ^J Justify      ^W Where Is     ^V Next Page    ^U UnCut Text   ^T To Spell
```



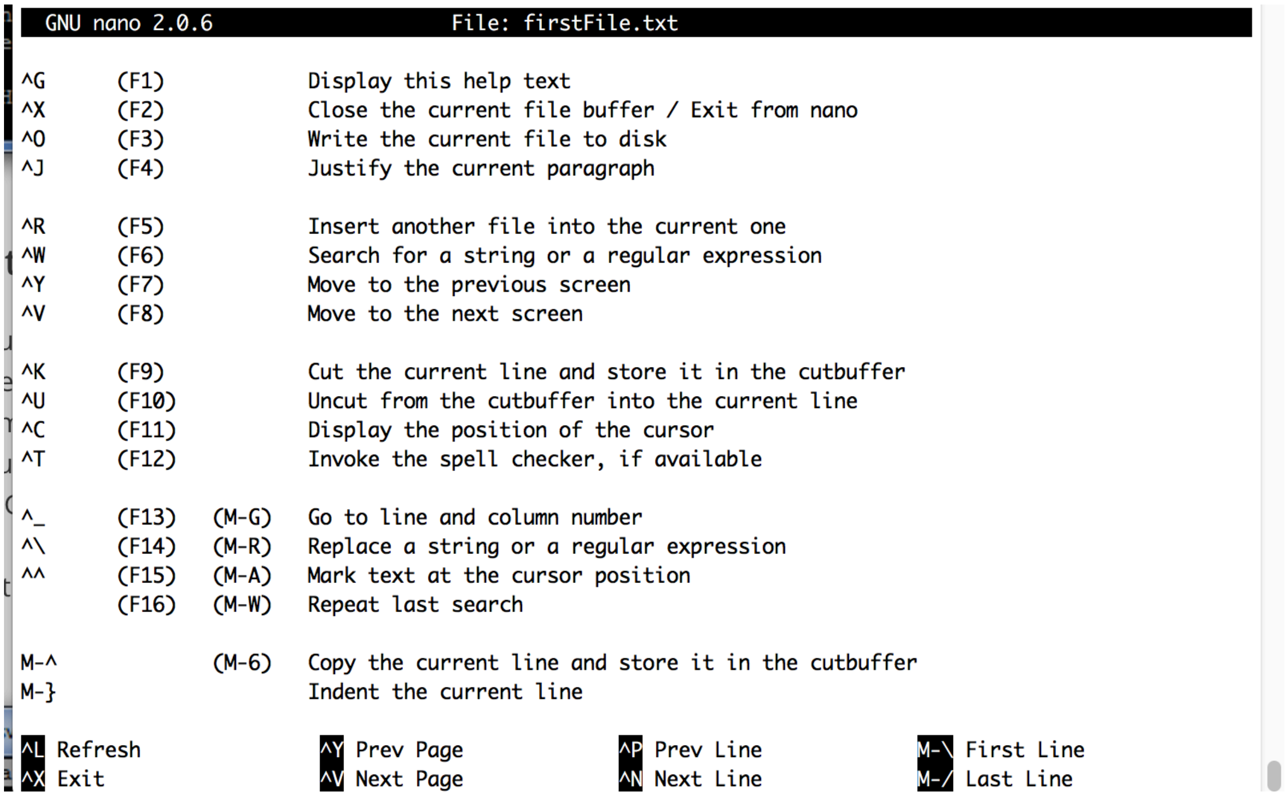
```
  GNU nano 2.0.6                File: firstFile.txt                          Modified

line1
line2
```

Things to notice:

- At the top
    - the name of the program (nano) and its version number
    - the name of the file you're editing
    - and whether the file has been modified since it was last saved.
- In the middle
    - you will see either a blank area or text you have typed
- At the bottom
    - A listing of keyboard commands such as Save (control + o) and Exit (control + x)

Keyboard commands are the only way to interact with the editor. You cannot use your mouse or trackpad.

Find more commands by using `control g`:



```
  GNU nano 2.0.6                    File: firstFile.txt

^G      (F1)            Display this help text
^X      (F2)            Close the current file buffer / Exit from nano
^O      (F3)            Write the current file to disk
^J      (F4)            Justify the current paragraph

^R      (F5)            Insert another file into the current one
^W      (F6)            Search for a string or a regular expression
^Y      (F7)            Move to the previous screen
^V      (F8)            Move to the next screen

^K      (F9)            Cut the current line and store it in the cutbuffer
^U      (F10)           Uncut from the cutbuffer into the current line
^C      (F11)           Display the position of the cursor
^T      (F12)           Invoke the spell checker, if available

^_      (F13)   (M-G)   Go to line and column number
^\      (F14)   (M-R)   Replace a string or a regular expression
^^      (F15)   (M-A)   Mark text at the cursor position
        (F16)   (M-W)   Repeat last search

M-^             (M-6)   Copy the current line and store it in the cutbuffer
M-}                     Indent the current line

^L Refresh                ^Y Prev Page              ^P Prev Line              M-\ First Line
^X Exit                   ^V Next Page              ^N Next Line              M-/ Last Line
```

The Meta key is <esc>. To use the Meta+key, hit <esc>, release, then hit the following key

Helpful commands:

- Jump to a specific line:
    - control + _ then line number
- Copy a block of highlighted text
    - control + ^ then move your cursor to start to highlight a block for copying
    - Meta + ^ to end your highlight block

- Paste
    - control + u

Nano is a beginner's text editor. vi and Emacs are better choices once you become a bit more comfortable using the terminal. These editors do cool stuff like syntax highlighting.

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 alted.

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

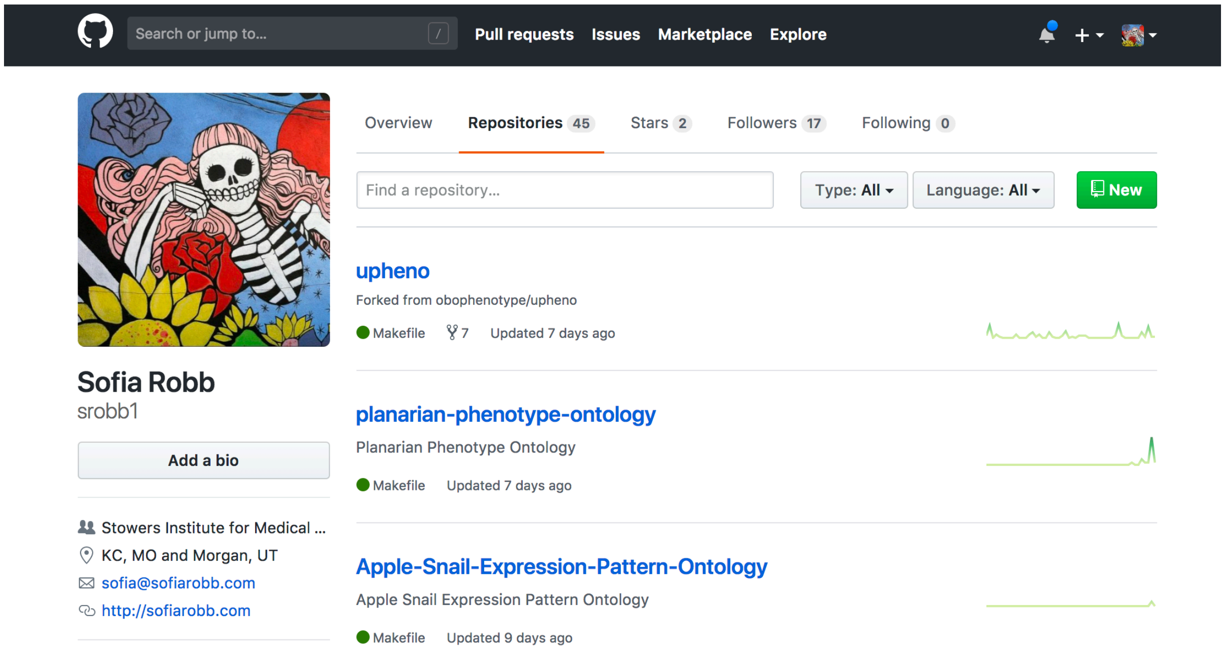
Creating a new repository

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.

Let's Do It!

Follow Steps 1 and 2 to create the remote repository. Follow Step 3 to create your local repository and link it to the remote.

1. Navigate to GitHub --> Create Account / Log In --> Go To Repositories --> Click 'New'



Sofia Robb
srobb1

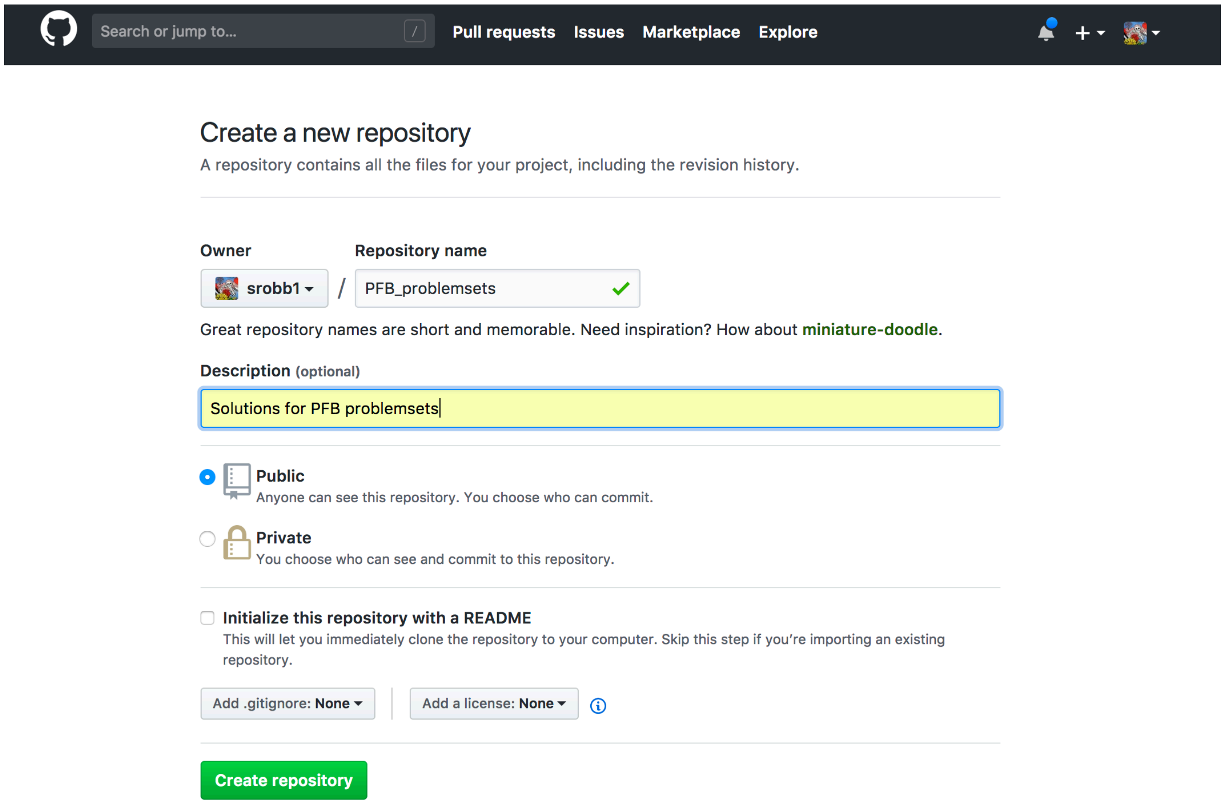
Stowers Institute for Medical ...
KC, MO and Morgan, UT
sofia@sofiarobb.com
http://sofiarobb.com

upheno
Forked from obophenotype/upheno
Makefile 7 Updated 7 days ago

planarian-phenotype-ontology
Planarian Phenotype Ontology
Makefile Updated 7 days ago

Apple-Snail-Expression-Pattern-Ontology
Apple Snail Expression Pattern Ontology
Makefile Updated 9 days ago

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 the files for your project, including the revision history.

Owner: srobb1 / Repository name: PFB_problemsets

Great repository names are short and memorable. Need inspiration? How about miniature-doodle.

Description (optional): Solutions for PFB problemsets

[✓ Public] Anyone can see this repository. You choose who can commit.

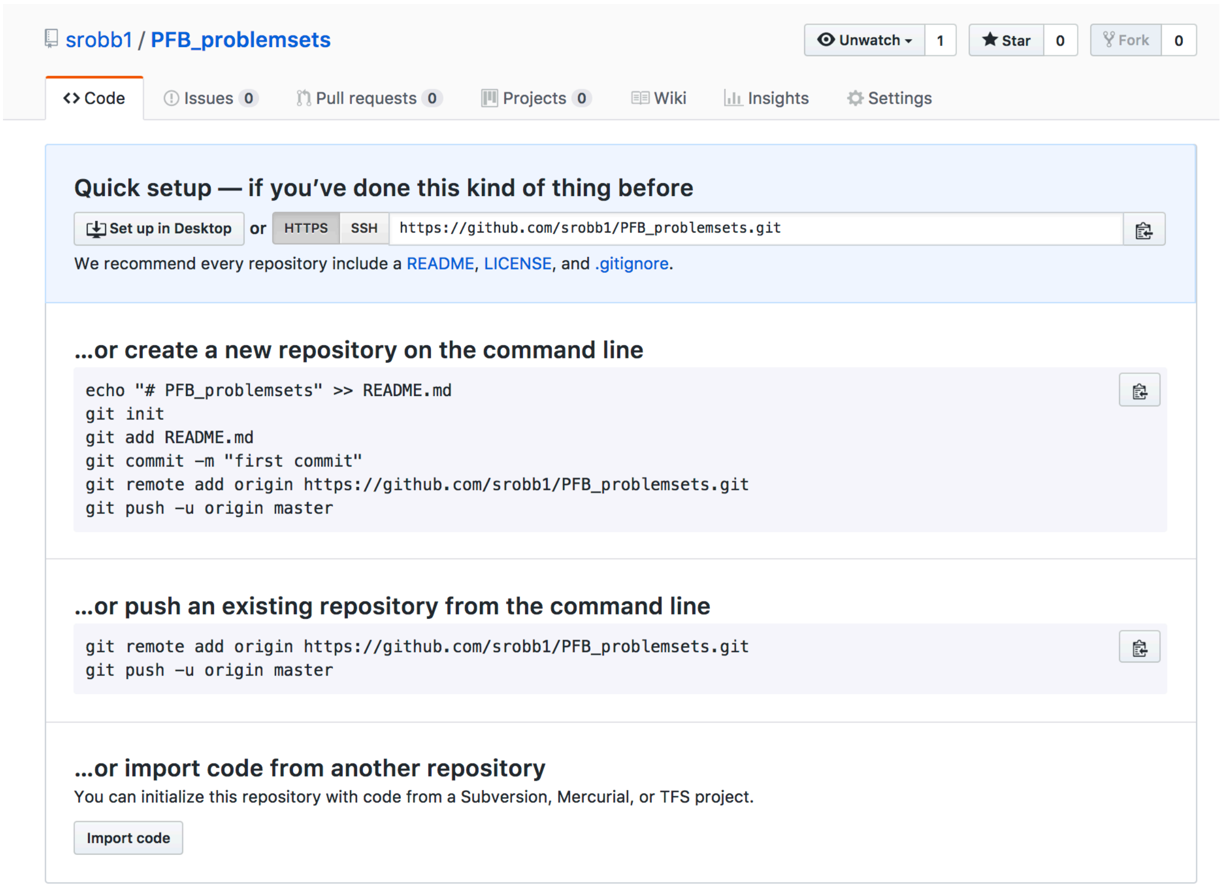
Private: You choose who can see and commit to this repository.

[ ] Initialize this repository with a README

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None | Add a license: None

3. Create a directory on your computer and follow the instructions provided.


srobb1 / PFB_problemsets

Quick setup — if you've done this kind of thing before

Set up in Desktop or HTTPS SSH https://github.com/srobb1/PFB_problemsets.git

We recommend every repository include a README, LICENSE, and .gitignore.

...or create a new repository on the command line

```
echo "# PFB_problemsets" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/srobb1/PFB_problemsets.git
git push -u origin master
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/srobb1/PFB_problemsets.git
git push -u origin master
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

- Open your terminal and navigate to the location you want to put a directory for your problem

sets
- Create a new directory directory (i.e., PFB_problemsets)
- Follow the instructions provided when you created your repository. These are my instructions, yours will be different.

```
echo "# PFB_problemsets" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/srobb1/PFB_problemsets.git
git push -u origin master
```

You now have a repository!

Let's back up a bit and talk more about git and about these commands. For basic git use, these are almost all the command you will need to know.

Every git 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	**description**
`git init`	Creates your new local repository with the three trees on (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

Let's Do it!

1. Make sure you are in your local repository
2. Create a new file with nano: `nano git_exercises.txt`

3. Add a line of text to the new file.
4. Save (control + o) and Exit (control + x)
5. (Add) Stage your changes. `git add git_exercises.txt`
6. (Commit) Become sure you want your changes your changes. `git commit -m 'added a line of text'`
7. (Push) Sync/Upload your changes to the **remote** repository. `git push origin master`

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.

Cloning a Repository

Sometimes you want to download and use someone else's repository.

Let's clone the course material.

Let's do it!

1. Go to our PFB2018 GitHub Repository
2. Click the 'Clone or Download' Button
3. Copy the URL
   ~Clone PFB2018
4. *Clone* the repository to your local machine
   `git clone https://github.com/prog4biol/pfb2018.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 copy you can *pull* the changes. `git pull`

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

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 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

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
